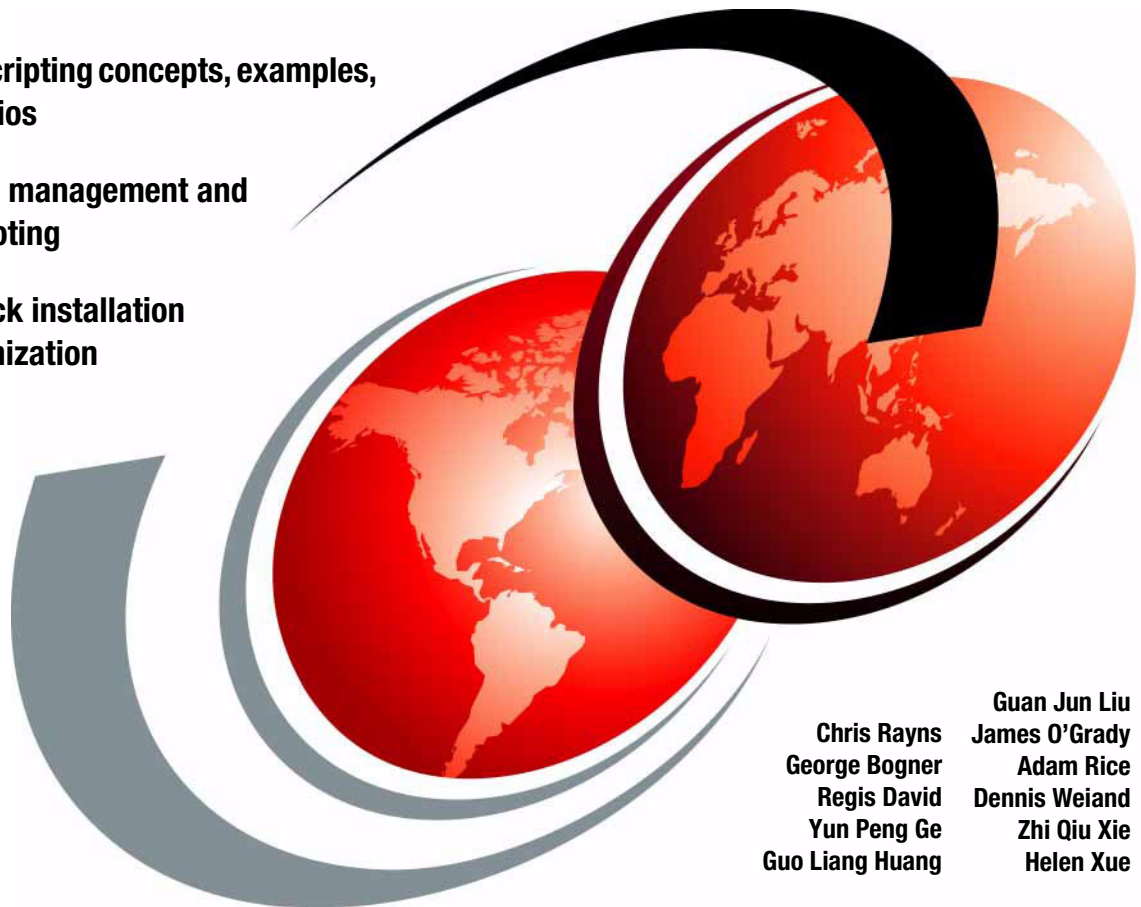


# Introduction to CLCS Dynamic Scripting

Dynamic Scripting concepts, examples,  
and scenarios

Application management and  
troubleshooting

Feature pack installation  
and customization



Chris Rayns	Guan Jun Liu
George Bogner	James O'Grady
Regis David	Adam Rice
Yun Peng Ge	Dennis Weiand
Guo Liang Huang	Zhi Qiu Xie
	Helen Xue





International Technical Support Organization

## **Introduction to CICS Dynamic Scripting**

March 2011

**Note:** Before using this information and the product it supports, read the information in “Notices” on page ix.

**First Edition (March 2011)**

This edition applies to Version 4, Release 1, CICS Transaction Server.

**© Copyright International Business Machines Corporation 2011. All rights reserved.**

Note to U.S. Government Users Restricted Rights -- Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

# Contents

<b>Notices</b> .....	ix
Trademarks .....	x
<b>Preface</b> .....	xi
The team who wrote this book .....	xii
Now you can become a published author, too! .....	xiv
Comments welcome .....	xiv
Stay connected to IBM Redbooks .....	xv
<b>Part 1. Technology introduction</b> .....	1
<b>Chapter 1. Dynamic Scripting overview</b> .....	3
1.1 Introduction .....	4
1.2 Project Zero .....	5
1.3 Purpose of Dynamic Scripting .....	6
1.4 Typical uses or applications .....	8
1.5 Advantages of Dynamic Scripting .....	10
<b>Chapter 2. Project Zero and WebSphere sMash features and capabilities overview</b> .....	11
2.1 Introduction .....	12
2.2 WebSphere sMash components .....	12
2.2.1 Application builder .....	12
2.2.2 Core .....	13
2.2.3 Assemble .....	13
2.2.4 Reliable Transport Extension .....	14
2.2.5 Deployment .....	14
2.3 Project Zero concepts .....	14
2.3.1 Application-centric versus server-centric design .....	14
2.3.2 Application creation and administration .....	15
2.3.3 A Project Zero application .....	15
2.3.4 Project Zero application development .....	16
2.3.5 WebSphere sMash is an event-based system .....	17
2.3.6 PHP Scripting Language .....	18
2.3.7 Groovy .....	19
2.3.8 Groovy templates .....	19
2.3.9 Global context .....	20
2.3.10 Modules, dependencies, and virtualized directories .....	21
2.3.11 Data model .....	22

2.3.12 Configuration files . . . . .	24
2.3.13 CICS concepts . . . . .	24
2.3.14 Tutorials, samples, and demos . . . . .	27
<b>Chapter 3. Web 2.0 . . . . .</b>	<b>29</b>
3.1 Web 2.0 concepts . . . . .	30
3.2 iWidgets. . . . .	30
3.3 Mashups . . . . .	31
3.4 REST. . . . .	32
3.5 Atom . . . . .	35
<b>Chapter 4. Web 2.0 technologies . . . . .</b>	<b>37</b>
4.1 Interaction pattern . . . . .	38
4.2 PHP . . . . .	39
4.3 Groovy . . . . .	42
4.4 JavaScript. . . . .	43
4.5 AJAX . . . . .	44
4.6 JSON. . . . .	45
4.7 Dojo . . . . .	46
<b>Part 2. Systems management . . . . .</b>	<b>49</b>
<b>Chapter 5. Installation . . . . .</b>	<b>51</b>
5.1 Software prerequisites. . . . .	52
5.2 Choosing an installation location. . . . .	53
5.2.1 Space requirements . . . . .	53
5.2.2 Installation location options . . . . .	54
5.3 Downloading the ZIP file to the mainframe . . . . .	60
5.4 Inflating the ZIP file . . . . .	61
5.4.1 Inflating the ZIP file using TSO OMVS . . . . .	61
5.4.2 Inflating the ZIP file using TSO ishell . . . . .	62
5.5 Configuring the UNIX environment . . . . .	62
5.5.1 Specifying the environment variables for all users . . . . .	63
5.5.2 Specifying the environment variables for one user . . . . .	63
5.5.3 Specifying the environment variables in a shell script. . . . .	64
5.6 Creating the CICS load library for Dynamic Scripting . . . . .	64
5.6.1 Copying the file from UNIX System Services using TSO OMVS . . . . .	64
5.6.2 Copying the file from UNIX System Services using TSO ishell. . . . .	65
5.6.3 Issuing the RECEIVE command . . . . .	65
5.7 Repository access zerocics.config . . . . .	66
5.8 UNIX System Services security. . . . .	67
5.8.1 Creating a new RACF UNIX group . . . . .	68
5.8.2 Running the install command . . . . .	71

<b>Chapter 6. Customizing CICS Dynamic Scripting Feature Pack</b>	73
6.1 Configuring CICS for dynamic scripting	74
6.2 Customizing the default configuration file	75
6.3 Configuring CICS security	79
6.4 Test your first dynamic scripting application	81
6.5 Dynamically created resources and their naming conventions	83
6.6 Editing an ASCII file on an EBCDIC system	85
<b>Chapter 7. Administration</b>	87
7.1 Application management	88
7.1.1 Sample applications overview	88
7.1.2 Sample 1: Hardcoded shell scripts	90
7.1.3 Sample 2: Variable shell scripts	98
7.1.4 Sample 3: Command-line interface control applications	106
7.2 JVMServer tuning	127
7.2.1 Use of the JVMServer with CICS Dynamic Scripting	127
7.2.2 How CICS Dynamic Scripting Feature Pack uses JVMServer	132
7.2.3 Zeroing in on JVMServer usage	133
7.2.4 Share Class Cache	136
7.2.5 Just-in-time compiler	136
7.2.6 Garbage collection	138
7.2.7 JVMServer Tuning	139
7.2.8 Use of zAAP by CICS Dynamic Scripting	140
7.3 Common zero commands for an administrator	141
7.3.1 Module commands	141
7.3.2 Module group commands	146
7.3.3 Repository commands	147
7.4 Security (file systems, applications,CICS), Active Content Filtering	148
7.4.1 z/OS UNIX System Services security	150
7.4.2 Client Access Security	154
7.4.3 Application Execution Security	157
7.4.4 Active content filtering	158
7.5 Local and remote repositories	175
7.5.1 Setting up a local shared repository	176
7.6 Packaging, publishing, sharing CICS Dynamic Scripting applications	177
7.7 Movement from development to production	178
<b>Chapter 8. Troubleshooting</b>	181
8.1 Repository access problems	182
8.2 Common errors (commands, code page, coding)	183
8.2.1 Installation problems	183
8.2.2 Command-line interface problems	185
8.2.3 Dynamic scripting encoding consideration	190

8.2.4	Setting up JVMSERVER . . . . .	191
8.3	Support considerations . . . . .	194
8.3.1	Collecting documentation required by the support team. . . . .	194
8.3.2	Submitting documentation to the support team. . . . .	197
8.3.3	Troubleshooting tips . . . . .	197
8.3.4	Learning more about the problem or component . . . . .	197
8.4	Health checker . . . . .	198
8.4.1	IBM Monitoring and Diagnostic Tools for Java - Health Center . . .	198
8.4.2	Features and benefits . . . . .	203
<b>Chapter 9. Dynamic Scripting for unit testing of CICS programs . . . . .</b>		<b>207</b>
9.1	Introduction to application testing . . . . .	208
9.1.1	Test phases. . . . .	208
9.1.2	Benefits of unit and component testing. . . . .	209
9.1.3	Benefits of using the test framework. . . . .	211
9.2	JUnit overview. . . . .	211
9.2.1	Setting up zero.test . . . . .	212
9.2.2	Writing tests . . . . .	212
9.2.3	Running tests . . . . .	216
9.3	Unit testing dynamic script application . . . . .	217
9.4	Unit testing traditional CICS programs . . . . .	226
<b>Part 3. Scenarios . . . . .</b>		<b>227</b>
<b>Chapter 10. Development options . . . . .</b>		<b>229</b>
10.1	Overview of the development environment. . . . .	230
10.2	What is an application . . . . .	231
10.2.1	Common directories . . . . .	231
10.2.2	Common tasks . . . . .	233
10.2.3	What is necessary to develop an application . . . . .	233
10.3	Development options. . . . .	234
10.3.1	The command-line interface (CLI). . . . .	234
10.3.2	The App Builder . . . . .	236
10.3.3	Eclipse. . . . .	241
10.4	Dependencies . . . . .	251
10.4.1	Configuration of dependencies . . . . .	252
10.4.2	Frequently used dependencies . . . . .	253
10.5	Module groups. . . . .	254
10.5.1	Overview of module groups . . . . .	254
10.5.2	The management of module groups . . . . .	255
10.6	Dependency management commands . . . . .	259
10.6.1	The zero resolve command. . . . .	259
10.6.2	The zero version command. . . . .	260
10.6.3	The zero update command . . . . .	261



10.6.4 The zero switch command . . . . .	262
10.7 Moving applications to Dynamic Scripting . . . . .	262
10.8 Dynamic scripting encoding considerations . . . . .	266
10.8.1 Converting between Java Strings and byte arrays . . . . .	267
10.8.2 Working with character data in PHP . . . . .	268
10.8.3 PHP strings and the PHP-Java bridge . . . . .	268
10.8.4 Character sets of HTTP responses . . . . .	270
<b>Chapter 11. Derby and DB2, create easy example . . . . .</b>	<b>273</b>
11.1 Database access . . . . .	274
11.2 Configuring a connection . . . . .	274
11.2.1 Configuring a DB2 connection . . . . .	274
11.2.2 Validating the connection . . . . .	276
11.2.3 Connection limitation . . . . .	276
11.3 Using ZRM for data access . . . . .	278
11.3.1 Obtaining ZRM functionality . . . . .	278
11.3.2 Creating a model definition . . . . .	278
11.3.3 Creating a resource handler . . . . .	282
11.3.4 CLI commands . . . . .	282
11.3.5 ZRM and existing data . . . . .	285
11.3.6 Testing ZRM . . . . .	286
11.4 Using the zero.data API for data access . . . . .	287
11.4.1 Obtaining the zero.data functionality . . . . .	287
11.4.2 Configuring the connection . . . . .	287
11.4.3 Using zero.data in Groovy . . . . .	288
11.4.4 Using zero.data in PHP . . . . .	292
11.5 Running SQL directly against the database . . . . .	296
<b>Chapter 12. Dynamic scripting scenarios . . . . .</b>	<b>297</b>
12.1 Exposing CICS resources in a RESTful style . . . . .	298
12.1.1 A simple do-nothing RESTful dynamic script . . . . .	298
12.1.2 A simple VSAM file resource access . . . . .	301
12.1.3 Returning the VSAM record as an Atom feed . . . . .	303
12.1.4 The zero.cics.tsq.demo sample . . . . .	307
12.1.5 More about JCICS within dynamic scripts . . . . .	309
12.1.6 More complex message interface support using JZOS . . . . .	321
12.1.7 More complex message interface support using Rational Developer for System z . . . . .	326
12.1.8 The CICS catalog application tutorial . . . . .	329
12.2 Summary . . . . .	330
<b>Chapter 13. Command-line sample . . . . .</b>	<b>331</b>
13.1 Command-line commands . . . . .	332
13.2 Command-line basics . . . . .	332

13.3 JVM servers, applications and commands .....	333
13.4 Creating a simple command in Groovy .....	334
13.5 Creating a simple command in PHP .....	335
13.6 Creating a simple command in Java .....	336
13.7 Providing help for commands .....	337
13.8 Creating subtasks .....	339
13.8.1 Updating zero.config for the main command .....	339
13.8.2 Creating a subcommand in Groovy. ....	340
13.8.3 Creating a subcommand in PHP .....	347
13.8.4 Creating a subcommand in Java. ....	355
13.9 Copying zerocics.config to the config directory .....	363
13.9.1 Where is the default zerocics.config file .....	364
13.9.2 Is the application running .....	364
13.9.3 Does a zerocics.config file already exist .....	365
13.9.4 Reading zerocics.config file .....	366
13.9.5 Writing zerocics.config .....	367
13.9.6 Extending the config command to specify the CICS region .....	367
13.10 Making commands available across applications .....	371
13.10.1 A useful file in public/redb/ .....	371
13.10.2 Packaging and publishing .....	374
13.11 Combining existing commands .....	375
<b>Related publications</b> .....	379
IBM Redbooks .....	379
Online resources .....	379
Help from IBM .....	380

# Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

*IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785 U.S.A.*

**The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:** INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.


## COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

# Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at <http://www.ibm.com/legal/copytrade.shtml>

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

CICS Explorer™	Language Environment®	Redbooks (logo)  ®
CICSplex®	Lotus Notes®	System z®
CICS®	Lotus®	VTAM®
DB2®	MVS™	WebSphere®
developerWorks®	Notes®	z/OS®
Dynamic Infrastructure®	RACF®	zSeries®
IBM®	Rational®	
IMS™	Redbooks®	

The following terms are trademarks of other companies:

Java, and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

# Preface

IBM® CICS® Transaction Server Feature Pack for Dynamic Scripting embeds and integrates technology from WebSphere® sMash into the CICS TS V4.1 run time, helping to reduce the time and cost of CICS application development. The Feature Pack provides a robust, managed environment for a wide range of situational applications allowing PHP and Groovy developers to create reports, dashboards, and widgets, and integrate CICS assets into mash-ups, and much more.

The CICS Dynamic Scripting Feature Pack combines the benefits of scripted, Web 2.0 applications with easy and secure access to CICS application and data resources. The Feature Pack includes a PHP 5.2 run time implemented in Java™ and with Groovy language support, support for native Java code and access to many additional libraries and connectors to enhance the development and user experience of rich Internet applications. Access to CICS resources is achieved by using the JCICS APIs.

The inclusion of Dynamic Scripting and WebSphere sMash technologies into the CICS Transaction Server V4.1 environment opens CICS, making it accessible to millions of PHP and Groovy developers. Using the agile dynamic scripting technologies can reduce the time and cost of CICS application development, and still providing a robust, managed environment for situational applications written in PHP and Groovy.

In this IBM Redbooks® publication, we introduce the Dynamic Scripting Feature Pack, show how to install and customize it, and provide examples for using it.

## The team who wrote this book

This book was produced by a team of specialists from around the world working at the International Technical Support Organization (ITSO), Poughkeepsie Center.

**Chris Rayns** is an IT Specialist and the CICS Project Leader at the International Technical Support Organization, Poughkeepsie Center. He writes extensively about all areas of CICS. Before joining the ITSO, Chris worked in IBM Global Services in the United Kingdom as a CICS IT Specialist.

**George Bogner** is a Software IT Specialist working in IBM Sales and Distribution supporting the CICS Transaction Server product suite. George worked at IBM for 25 years, specializing in the DB/DC area, working with CICS, IMS™, and DB2®, supporting client accounts in IBM Global Services. He currently works out of Raleigh, North Carolina, supporting North American clients by providing CICS seminars, proofs of technology (POT), proofs of concept (POC), and consulting services for CICS-related topics.

**Regis David** is a senior I/T Product Services professional in France. He has 30 years of experience working on the full scope of the CICS ecosystem. His areas of expertise include advanced client/server implementations such as web services, Java Connector Architecture, and RESTful style, including messaging. He is an expert of pragmatic service-oriented architecture implementations. He runs multiple presentations in France, focussed on new technology adoption within CICS and System z®.

**Yun Peng Ge** is a Client Technical Professional on the mainframe team with the IBM China Software Group. He spends most of his time supporting mainframe customers of mainland China and Hong Kong. He received a Bachelor of Sciences degree in Computer Science from Fudan University in 2005. His areas of expertise includes enterprise integration and connectivity, application servers, and mainframe application development.

**Guo Liang Huang** is a Software Engineer with the WebSphere Business Process Management (BPM) team at the IBM China Development Lab. He has very rich experience with Java EE and service-oriented architecture (SOA), and is also very familiar with the Web 2.0. In addition, he has expertise in problem determination with WebSphere Process Server production.

**Guan Jun Liu** is a Software Engineer on the CICS TS Development Team at the IBM China Development Lab. He tests and develops Java and web technologies for CICS Transaction Server, and has more than two years experience with web services and CICS Transaction Server.

**James O’Grady** is a CICS Systems Tester in England. He has 10 years of experience in CICS, four of them at Hursley. He holds a degree in History from the University of Warwick. His areas of expertise include CICS web support, CICSplex® SM Workload Management and CICS Dynamic Scripting. He worked on a Redbooks publication about WS-Security with CICS.

**Adam Rice** is a trainee Software Engineer working at IBM Hursley, United Kingdom. He graduated from Aston University, Birmingham with a BSc Computer Science degree and has worked in CICS System Test since joining IBM in 2009. In that time he has worked on the CICS Dynamic Scripting Feature Pack and has written a developerWorks® article on the subject of resource handlers in dynamic scripting applications.

**Dennis Weiland** is a Technical Sales Specialist at the IBM Dallas Systems Center. Currently, Dennis works primarily with web services, Web 2.0, Events, and Java as they relate to CICS, plus the CICS Transaction Gateway. He holds a Masters degree in Computer Science from Tarleton State University in central Texas.

**Zhi Qiu Xie** is a Software Engineer with the WebSphere BPM team at the IBM China Development Lab. He has in-depth knowledge of SOA, BPM, and Java EE. His areas of expertise include application design and development and also problem determination based on the IBM WebSphere stack products.

**Helen Xue** is a Software Engineer at the IBM China Development Lab. She has three years of experience in Java application development, and works as Developer and Testing Lead with the mainframe team. She also has experience in IBM Rational® Developer for z/OS®.

Thanks to the following contributor for his contribution to this project:

Richard M Conway  
International Technical Support Organization, Poughkeepsie Center

## Now you can become a published author, too!

Here's an opportunity to spotlight your skills, grow your career, and become a published author—all at the same time! Join an ITSO residency project and help write a book in your area of expertise, while honing your experience using leading-edge technologies. Your efforts will help to increase product acceptance and customer satisfaction, as you expand your network of technical contacts and relationships. Residencies run from two to six weeks in length, and you can participate either in person or as a remote resident working from your home base.

Learn more about the residency program, browse the residency index, and apply online at:

[ibm.com/redbooks/residencies.html](http://ibm.com/redbooks/residencies.html)

## Comments welcome

Your comments are important to us!

We want our books to be as helpful as possible. Send us your comments about this book or other IBM Redbooks publications in one of the following ways:

- Use the online **Contact us** review Redbooks form found at:

[ibm.com/redbooks](http://ibm.com/redbooks)

- Send your comments in an email to:

[redbooks@us.ibm.com](mailto:redbooks@us.ibm.com)

- Mail your comments to:

IBM Corporation, International Technical Support Organization  
Dept. HYTD Mail Station P099  
2455 South Road  
Poughkeepsie, NY 12601-5400



## Stay connected to IBM Redbooks

- ▶ Find us on Facebook:  
<http://www.facebook.com/IBMRedbooks>
- ▶ Follow us on Twitter:  
<http://twitter.com/ibmredbooks>
- ▶ Look for us on LinkedIn:  
<http://www.linkedin.com/groups?home=&gid=2130806>
- ▶ Explore new Redbooks publications, residencies, and workshops with the IBM Redbooks weekly newsletter:  
<https://www.redbooks.ibm.com/Redbooks.nsf/subscribe?OpenForm>
- ▶ Stay current on recent Redbooks publications with RSS Feeds:  
<http://www.redbooks.ibm.com/rss.html>





# Part 1

## Technology introduction

In this part of the book, we introduce the following technologies:

- ▶ Dynamic Scripting
- ▶ Project Zero
- ▶ WebSphere sMash
- ▶ Web 2.0 and technologies





# Dynamic Scripting overview

In this chapter, we give an overview of Dynamic Scripting Feature Pack running in the CICS Transaction Server environment.

# 1.1 Introduction

IBM CICS Transaction Server Feature Pack for Dynamic Scripting embeds and integrates technology from WebSphere sMash into the CICS TS V4.1 run time, helping to reduce the time and cost of CICS application development. The Feature Pack provides a robust, managed environment for a wide range of situational applications allowing millions of PHP and Groovy developers to create reports, dashboards, and widgets, and integrate CICS assets into mash-ups, and much more.

The CICS Dynamic Scripting Feature Pack combines the benefits of scripted, Web 2.0 applications with easy and secure access to CICS application and data resources. The Feature Pack includes a PHP 5.2 run time that is implemented in Java along with Groovy language support, support for native Java code and access to many additional libraries and connectors to enhance the development and user experience of Rich Internet Applications. Access to CICS resources is achieved using the JCICS APIs.

The inclusion of Dynamic Scripting and WebSphere sMash technologies into the CICS Transaction Server V4.1 environment enables CICS to be more accessible to PHP and Groovy developers. Using the agile dynamic scripting technologies can reduce the time and cost of CICS application development, and still provide a robust, managed environment for situational applications written in PHP and Groovy.

CICS Transaction Server Feature Pack for Dynamic Scripting is built on scalable CICS TS V4 JVMSERVER technology to provide the capacity needed for tactical applications without impacting strategic production workload. The built-in Zero Resource Model (ZRM) provides a simplified way to create RESTful resource handlers with a data store that can be mapped to IBM DB2 for z/OS for improved resilience, security, and performance. CICS scripting regions can be easily isolated from production CICS workload and still provide access to CICS resources. CICS Dynamic Scripting workload is zAAP eligible, and so can be off-loaded to available specialty processors.

CICS Dynamic Scripting allows you to quickly try new business ideas, build productivity applications such as reports, dashboards, and widgets, or application front-ends for IT and Line of Business users

You can introduce new IT staff to CICS through PHP and port many existing unmanaged PHP and WebSphere sMash applications into CICS.

Figure 1-1 shows the architecture for Project Zero on CICS showing how new Web 2.0 style applications can run in a CICS Transaction Server V4.1 environment and gain access to existing CICS TS applications and resources through the JCICS API.

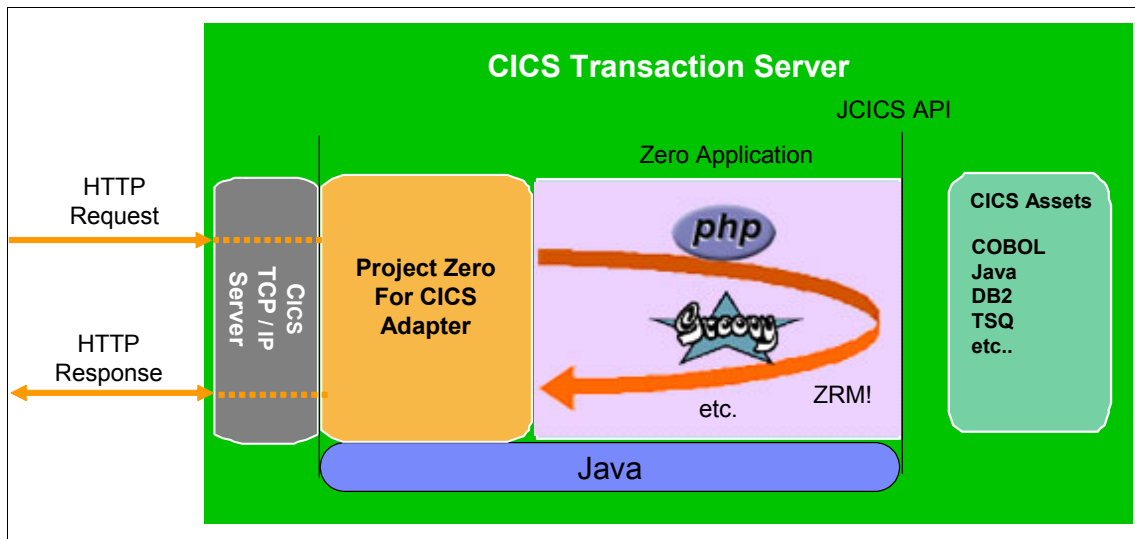


Figure 1-1 Architecture for Project Zero on CICS

## 1.2 Project Zero

Project Zero began life as an incubator project to explore a new idea that we believed had promise. That idea was of a development and runtime environment that could revolutionize creation of dynamic web applications, providing a powerful development and execution platform for modern web applications, and at the same time, having the overall experience of being radically simple. We started this incubator because we wanted to address the complexities of modern web applications without the chains of previous architectures, technologies, or decisions.

We use ProjectZero.org to incubate the technology, offering an open window into the development process, the code and the developers. You can be as involved as you like, from simply looking to being fully engaged, and from anonymous download to registered users, trying the code, reviewing the source, giving feedback, sharing experiences, making recommendations and providing requirements. We deliver multiple versions of IBM WebSphere sMash based on the work here and we have active code in development.

## 1.3 Purpose of Dynamic Scripting

A Smart SOA Application Foundation can help you to deliver on business objectives and contain or even reduce costs across mainframe and distributed applications and all interaction modes (for example, Web 2.0). As part of the Smart SOA Application Foundation, three foundational attributes allow both IBM CICS Transaction Server for z/OS (CICS TS) and IBM WebSphere Application Server to run robust and agile business applications, leading to the following advantages:

- ▶ Efficient development and management
- ▶ Highly effective performance
- ▶ Application innovation

The CICS Transaction Server for z/OS (CICS TS) and WebSphere Application Server Feature Packs for Dynamic Scripting promote efficient development and application innovation by enabling enterprise application developers with an agile dynamic scripting programming model. This innovative programming model gives developers the power to rapidly create, assemble, and run situational applications to address departmental, project, and team requirements.

Figure 1-2 on page 7 shows how CICS Dynamic Scripting technologies can be used to reveal *long tail* opportunities that simply were not possible using existing programming tools.



## CICS Dynamic Scripting can be used to develop and deploy...

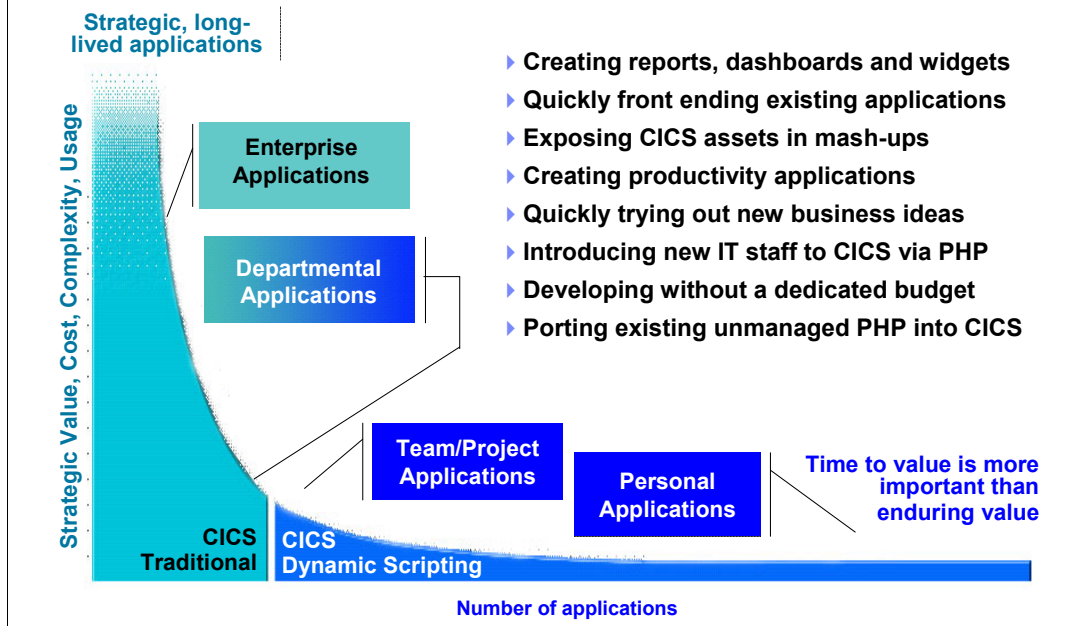


Figure 1-2 CICS Dynamic Scripting applications

Based on technology from WebSphere sMash V1.1.1, CICS and WebSphere Application Server Feature Packs for Dynamic Scripting provide support for dynamic scripting languages: PHP Hypertext Preprocessor (PHP) and Groovy. With Feature Packs, developers can exploit the programming languages and use Web 2.0 technologies, such as Asynchronous JavaScript and XML (AJAX), Representational State Transfer (REST), Atom, JavaScript Object Notation (JSON), and Really Simple Syndication (RSS) to unleash and reuse enterprise content. The built-in Zero Resource Model (ZRM) provides a simplified way to create RESTful resource handlers with a data store that can be mapped to both IBM DB2 for z/OS and multiplatforms for improved resilience, security, and performance. Feature Packs for Dynamic Scripting can be used to perform the following tasks:

- ▶ Rapidly develop, evolve, and deploy rich web applications, using dynamic scripting languages.
- ▶ Expose RESTful web services and widgets for use in mashups and consoles.
- ▶ Compose composite services, widgets, and applications that combine Web 2.0- style public and enterprise data feeds.

The Feature Packs for Dynamic Scripting are optional product extensions for IBM CICS Transaction Server for z/OS, V4.1, and IBM WebSphere Application Server V6.1 and V7.0.

## 1.4 Typical uses or applications

You can develop CICS dynamic scripting applications using the WebSphere sMash programming model, and you can integrate your applications with CICS resources and assets through the JCICS API.

IBM WebSphere sMash provides an event-driven architecture and uses REST concepts in its programming model. Using this model, you can quickly develop scripting applications that use Groovy, PHP, or Java handlers. You can also invoke the JCICS API to access data in CICS assets.

Each application uses a global context, which provides a uniform mechanism to store and retrieve state information. The global context has a number of zones that contain different information. For example, the request zone provides access to the HTTP request data. IBM WebSphere sMash provides APIs in Java, Groovy, and PHP to access the global context.

Each application also has a configuration zone that contains the configuration information, including port numbers, directory locations, and dependencies.

Settings in the configuration zone are made using the `zero.config` configuration file that contains properties that are specific to a dynamic scripting application. These properties are processed and applied when you start the application.

For more information about the programming model, including using the global context and list of zones, see the Developer Guide at the community website for WebSphere sMash:

<http://www.projectzero.org/>

Dynamic scripting applications consist of a well-defined directory structure, containing files that you can edit by using the tools of your choice. You can develop your dynamic scripting applications using the IBM WebSphere sMash App Builder, Eclipse, or Eclipse-based tools such as Rational Developer for IBM System z.

Figure 1-3 shows a comparison of the development cycles of dynamic scripting applications versus traditional applications.

The two methods for developing dynamic scripting applications using Eclipse are as follows:

- ▶ You may develop the application on your workstation and then deploy it to z/OS.
- ▶ You may use Eclipse tooling on your workstation to remotely develop the application, so that it remains on the z/OS platform throughout development.

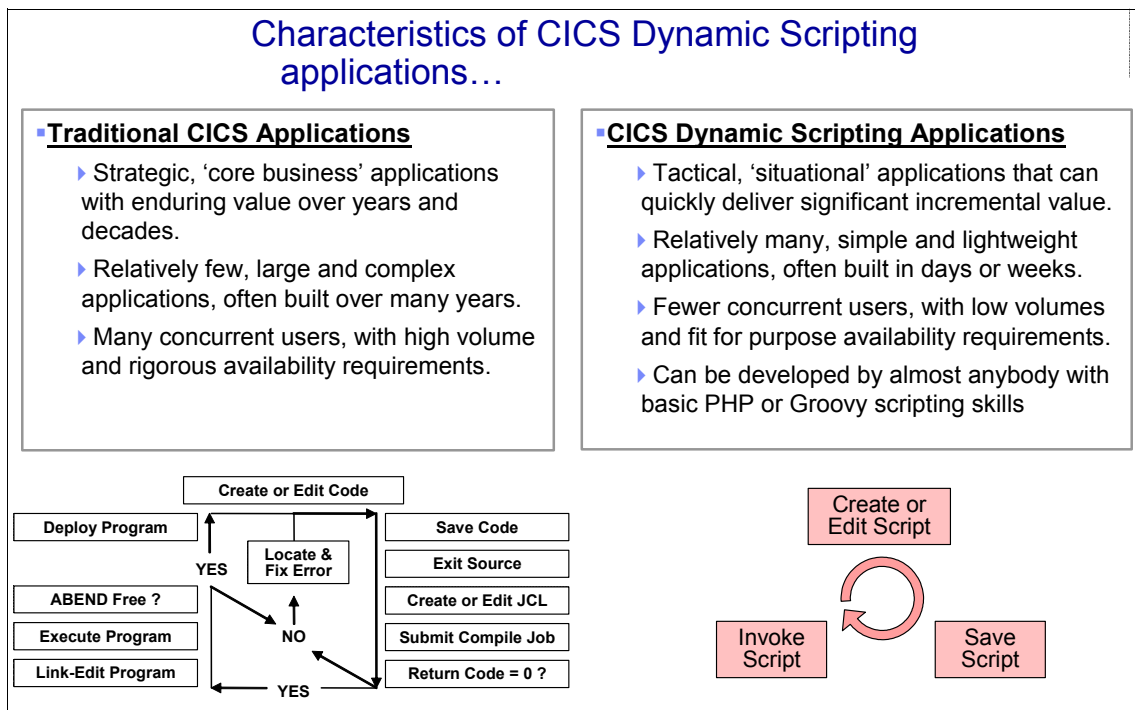


Figure 1-3 Dynamic scripting applications versus traditional applications

## 1.5 Advantages of Dynamic Scripting

The CICS Dynamic Scripting Feature Pack provides functions that can speed up, simplify, and promote agile application development.

### Speed

Productivity is increased by using reusable components and situational applications that require less time, fewer lines of code, and less specialized skill to produce.

The advantages are as follows:

- ▶ By using Groovy or PHP, developers can quickly and easily develop web applications.
- ▶ Deployment is simplified by a runtime environment which interfaces with CICS.
- ▶ Management systems such as CICSplex System Manager can help run and manage these agile applications in an efficient and cost-effective way.

### Simplicity

Dynamic scripting simplifies the task of creating applications using the REST architectural style. The advantages are as follows:

- ▶ REST services can be used to expose feeds (RSS) and use pre-existing content (for example HTTP and JMS).
- ▶ REST-style architecture maintains SOA principles and can be reused in a scalable but simplistic way.
- ▶ For extreme efficiency, you can build applications by assembling existing services and feeds (from both internal and external sources) into composite situational applications.

### Agility

The advantages are as follows:

- ▶ You can build reusable building blocks, content, templates, and patterns, and reuse any dynamic scripting content in the form of front-end widgets.
- ▶ An integrated environment can help you cost-effectively develop and manage web 2.0 applications.



# **Project Zero and WebSphere sMash features and capabilities overview**

This chapter first describes the components of WebSphere sMash, then describes various capabilities and features. Later in this book, each of the various capabilities are described in more detail.

## 2.1 Introduction

WebSphere sMash is a development and execution platform for quickly building agile, web-based applications using less costly development skills by leveraging dynamic scripting languages. This is accomplished by basing WebSphere sMash on the highly -acclaimed public incubator and developer community, Project Zero. WebSphere sMash Developers Edition runs on Linux®, Windows®, and Mac, and is available for download and limited deployment. WebSphere sMash licenses are purchased when deploying it into production.

The Project Zero website contains the newest function not yet available in the IBM WebSphere sMash production version. The Project Zero website also contains additional tooling for WebSphere sMash available at the following address:

<http://www.projectzero.org>

From this community site, users can provide feedback, ask questions, and steer the development effort of IBM WebSphere sMash.

IBM CICS Transaction Server for Dynamic Scripting V1.0 embeds and integrates technology from WebSphere sMash into the CICS TS V4.1 run time. All components of WebSphere sMash are not integrated into CICS Dynamic Scripting at this time.

## 2.2 WebSphere sMash components

WebSphere sMash consists of five components. Not all components are currently implemented by the CICS Dynamic Scripting Feature Pack. All WebSphere sMash components are listed in this section. The core and deployment components are the main components and are supported by the CICS Dynamic Scripting Feature Pack with the restrictions listed in the CICS information center and also in later sections of this book.

### 2.2.1 Application builder

The WebSphere sMash application builder is a web-based tool for developing WebSphere sMash applications. The application builder is itself a WebSphere sMash application, which means you can acquire and manage it the same way you do other WebSphere sMash applications.

CICS Dynamic Scripting does not currently support this component.

## 2.2.2 Core

The WebSphere sMash core provides a simple programming model as a guideline for building Web 2.0 applications using RESTful principles. With the programming model, developers can create RESTful resource handlers by using either the Groovy scripting language (available by default) or the PHP programming language (available by adding its dependency in a configuration file).

The scripting support in WebSphere sMash allows developers to build applications quickly by following a few basic conventions. For basic database backed applications that have a CRUD-based (Create, Retrieve, Update, Delete) interaction, the programming model includes the Zero Resource Model (ZRM) technology to easily model and access data as RESTful resources. Using the core engine, you can create the database tables and run the appropriate SQL queries for retrieving the resource representations using either JavaScript Object Notation (JSON) encoding or as Atom feeds for easier integration with AJAX based clients. The engine also supports building rich user interfaces using AJAX technology provided by the Dojo toolkit.

WebSphere sMash includes a data abstraction layer to allow developers to work directly with SQL to query and update data associated with the application. This works with an API to update or retrieve remote resources using, for example HTTP(S) or using email. WebSphere sMash also provides a host of features to secure the application. These features include the standard authentication methods.

The core engine of WebSphere sMash delivers simplicity by providing a command-line utility so you can manage and create applications and their dependencies. The application structure builds on the convention over configuration theme by using a simple directory and artifact naming convention.

CICS Dynamic Scripting supports the core component of WebSphere sMash, with the restrictions listed in the CICS information center.

## 2.2.3 Assemble

The WebSphere sMash Assemble component provides the capability to access different services and assemble them into a WebSphere sMash application.

CICS Dynamic Scripting does not currently support this component.

## **2.2.4 Reliable Transport Extension**

The reliable transport extension allows WebSphere sMash applications to communicate with each other using asynchronous messages.

CICS Dynamic Scripting does not currently support this component. CICS has support for WebSphere MQ which provides for asynchronous messaging.

## **2.2.5 Deployment**

Each WebSphere sMash application is a self-contained entity that contains all the components necessary to run the application. This application-centric approach removes the error-prone task of deploying an application into a server. By adopting an application-centric model, WebSphere sMash can both simplify deployment and maximize isolation.

CICS Dynamic Scripting also uses this approach to deployment.

## **2.3 Project Zero concepts**

The main Project Zero concepts are discussed in this section at an overview level. More details are described later in this book, and are also discussed in the WebSphere sMash information center.

### **2.3.1 Application-centric versus server-centric design**

From a Project Zero developer's perspective, the application is the server. This is in contrast to the normal thinking where you have a server and run multiple applications under that server. When a Project Zero application is started, it listens on its own port and takes care of all HTTP and database interactions. It is like the infrastructure is an extension of the application (versus the normal thinking of the application being an extension of the infrastructure).

How this is physically applied is that you create an application, add application code, then start the application. That is all there is to it. The application listens on a specified port, and responds to HTTP requests as appropriate. The capabilities such as listening and responding to HTTP, interacting with a database, using email, and so on are added to the application by adding dependencies.



## 2.3.2 Application creation and administration

The interface to the Project Zero technology, both in WebSphere sMash and in CICS Dynamic Scripting, is through a command-line interface (CLI).

The available ZERO commands differ, depending on whether you are in an application directory or not. If you are in your application directory you can use commands such as **zero start**, **zero stop**, and **zero resolve**, which operate against or with your Zero Application. When outside your application's directory you can use a command such **zero create** to create a new application.

The CLI, in a CICS environment, communicates to CICS using EXCI. After the CLI request arrives in CICS, the command is executed in a CICS JVMServer.

The exceptions to the Project Zero command-line interface is when using the WebSphere sMash Eclipse plug-in. This plug-in allows you to start and stop your WebSphere sMash application by selecting options from the context menu presented by right-clicking your WebSphere sMash project. When using the App Builder (a web browser-based development environment) you can access the command-line interface through the web browser interface. You cannot currently access your z/OS-based Zero Application data through the WebSphere sMash plug-in. CICS currently does not support the App Builder. You may, however, in many situations, develop your application on a workstation using WebSphere sMash and the Eclipse-based WebSphere sMash plug-in or the App Builder, and then migrate your application to CICS Dynamic Scripting.

Zero Application development and migration is discussed in section Chapter 10, "Development options" on page 229.

## 2.3.3 A Project Zero application

Each dynamic scripting application is a standard (well-known) directory structure containing content within that structure. There are specific directories available for specific types of artifacts. For example, the default location for HTML page is in your application's *public* directory. All the directories (standard or optional) are documented in the Project Zero documentation.

## 2.3.4 Project Zero application development

Because a Project Zero Application is a set of directories and their content, you can use a variety of tools and editors to manipulate the contents of the directories. You may use the Interactive System Productivity Facility (ISPF) or the vi editor on z/OS, or use Microsoft® Notepad or similar tools on the workstation. However, higher-level tools and editors specifically designed to work with the various types of files that make up your application are available.

WebSphere sMash contains a component called the App Builder. The App Builder allows you to use a web browser interface to create, display, and modify file contents. Currently, CICS Dynamic Scripting does not support the App Builder.

Multiple Eclipse plug-ins are available for editing and debugging Project Zero Applications. You may use these plug-ins in a base Eclipse environment, or you may add these tools to other Eclipse-based environments such as Rational Application Developer, or Rational Developer for System z. A WebSphere sMash plug-in is available so you can perform application administration (for example start, stop, resolve) from context menus instead of the command line. Also, editing and debugging plug-ins exist for PHP and Groovy.

When interacting with a CICS resource such as a VSAM file, Rational Application Developer can be very useful as it provides the ability to analyze a COBOL copybook and generate a corresponding Java object containing getters and setters for each field in the COBOL-like data structure. This makes interacting with your CICS simple.

Similarly, you can generate data objects to interface with CICS resources using the JZOS toolkit. JZOS tools are additional Java support classes that are useful in a z/OS environment and are included with the z/OS distribution of Java. For more information about using JZOS to generate Java data objects that correspond to your CICS resource data structures, see 12.1, “Exposing CICS resources in a RESTful style” on page 298.

Rational Application Developer and Rational Developer for System z contain editors for many of the resources you will be manipulating in your Project Zero application. They contain an editor and debugger for Java and editors for JavaScript, HTML, and Cascading Style Sheets (CSS). Because Rational Application Developer and Rational Developer for System z are Eclipse-based tools, you can add plug-ins for editing and debugging PHP programs and editing and debugging Groovy programs. A list of Eclipse plug-ins that make Project Zero application easier to develop, test, and debug are listed in Chapter 10, “Development options” on page 229.

Rational Developer for System z provides an additional ease of use by allowing you to access your z/OS based resources directly from your workstation. Rational Developer for System z will also help you will code page conversion as necessary. For more information about the use of Rational Developer for System z for your Project Zero applications, see the following topics:

- ▶ 6.6, “Editing an ASCII file on an EBCDIC system” on page 85
- ▶ 8.2.3, “Dynamic scripting encoding consideration” on page 190

Also, an Eclipse Target Management plug-in is available for editing files on your workstation that reside on z/OS UNIX® System Services. See the CICS information center for more details and a hyperlink to the plug-in.

If you decide to develop your application on a workstation and later migrate your application to CICS Dynamic Scripting, you can create a module group and associate it with your application. This way ensures that during development you only add dependencies that CICS allows. See the following resources:

- ▶ CICS information center, which describes how to specify the module group:  
[http://publib.boulder.ibm.com/infocenter/cicsts/v4r1/index.jsp?topic=/com.ibm.cics.ts.smash.doc/smash\\_devapps\\_eclipse\\_workstation.html](http://publib.boulder.ibm.com/infocenter/cicsts/v4r1/index.jsp?topic=/com.ibm.cics.ts.smash.doc/smash_devapps_eclipse_workstation.html)
- ▶ Section 7.3.2, “Module group commands” on page 146

**Note:** Most files are expected to be encoded as UTF-8. Consider this information when choosing your development environment.

### 2.3.5 WebSphere sMash is an event-based system

All of the key behavior of WebSphere sMash is exposed to your Zero Application as a set of events, along with the appropriate event state. Application developers mainly provide a set of handlers that hook into the well-known system events to achieve application behavior as needed. The standard sMash events are meaningful activities of interest to applications.

Common events that most application developers use, such as providing a response to a GET request to a URI or when an application is started, are provided. Event handlers are stateless blocks of function that handle the events. Events are identified by an event name, such as GET or LOG. Event handlers indicate an interest in handling a particular event under a particular condition.

Events in the Zero platform are ways to orchestrate behavior in the form of loosely-coupled event handlers. Zero *fires* a fixed set of events for HTTP request processing and stages of the application life cycle. Developers may add other

event types. Firing an event, which equates to an API invocation, causes Zero to invoke the associated handlers through an event dispatcher.

The set of associated handlers is determined by two mechanisms:

- ▶ Explicit registration: Evaluate registration rules.
- ▶ Implicit registration: Identify scripts as handlers through convention.

With the WebSphere sMash environment, you may write event handlers in Groovy, PHP, or Java.

## 2.3.6 PHP Scripting Language

PHP, is a recursive acronym that stands for PHP Hypertext Preprocessor. PHP was written in approximately 1994 by Rasmus Lerdorf, and at that time, PHP stood for “personal home page.” The use of PHP expanded past home use and around 1997 the acronym was changed to stand for what it does today (PHP Hypertext Preprocessor). Over the years, several people have contributed to PHP so its syntax was influenced by C, Perl, Java, C++, TCL, and other languages.

Worldwide, millions of developers know PHP, and the number of PHP programmers, including corporate PHP programmers, is expected to grow.

The PHP interpreter used with CICS Dynamic Scripting runs in Java. The PHP code is compiled, at run time, into Java bytecode. Because the WebSphere sMash PHP interpreter is written in Java, there is a bridge to Java, where you can directly instantiate Java objects and invoke methods on those objects. This technique allows CICS’s Java-based API (the JCICS classes) to be available to PHP scripts. Limitations on invoking Java classes, in general, are discussed in the WebSphere sMash documentation, and the restrictions or limitations on invoking the JCICS API from a PHP script is discussed in the CICS information center.

For more information about PHP and its capabilities, see 4.2, “PHP” and Chapter 10, “Development options” on page 229.

### 2.3.7 Groovy

Groovy is a dynamic scripting language written by James Strachan and Bob McWhirter in 2003. Groovy 1.0 was released January 2007.

Groovy is a dynamic, object-oriented scripting language with features inspired by Python, Ruby, and Smalltalk. Groovy provides a scripting language to Java developers with almost no learning curve. It can be easy to read and maintain, integrates with all existing Java classes and libraries, and compiles into Java bytecode.

Groovy contains *well-known* methods that are invoked for various WebSphere sMash events. For example, for RESTful interfaces, an `onList()` method is available for GET requests against a collection, `onCreate()` for a POST of a member, `onRetrieve()` for a GET of a member, `onUpdate()` for a PUT of a member, and `onDelete()` for a DELETE of a member.

### 2.3.8 Groovy templates

With Groovy templates you build reusable static documents that include well defined place holders to insert dynamic information. For pages with mostly static content, building the user interface (UI) on the server side using Groovy templates can yield results similar to embedding PHP in regular HTML.

WebSphere sMash and CICS Dynamic Scripting treat files with a `.gt` suffix as Groovy templates, using Groovy's `SimpleTemplateEngine` to translate statements that are enclosed with open and closed angle brackets (`<%` and `%>`) and evaluating Groovy expressions placed within `<%=` and `=%>`, to strings.

Figure 2-1 shows a simple HTML example that is then copied and modified to show embedded a PHP and Groovy template examples:

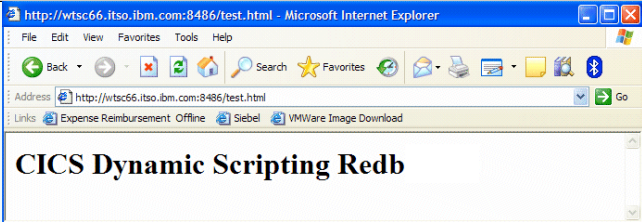
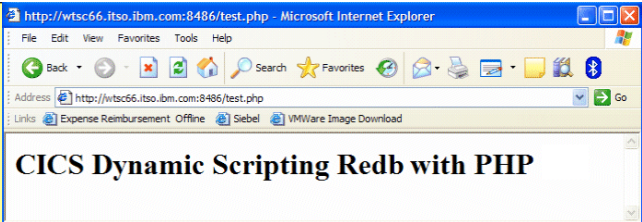

HTML only Example	Execution Time Results
<pre>&lt;html&gt; &lt;body&gt;   &lt;h1&gt; CICS Dynamic Scripting Redb &lt;/h1&gt; &lt;/body&gt; &lt;/html&gt;</pre>	
HTML with embeded PHP Example	Execution Time Results
<pre>&lt;html&gt; &lt;body&gt;   &lt;h1&gt; CICS Dynamic Scripting Redb   &lt;?php     echo 'with PHP';   ?&gt; &lt;/h1&gt; &lt;/body&gt; &lt;/html&gt;</pre>	
HTML with Groovy Template Example	Execution Time Results
<pre>&lt;html&gt; &lt;body&gt;   &lt;h1&gt; CICS Dynamic Scripting Redb   &lt;%     println("with Groovy");   %&gt; &lt;/h1&gt; &lt;/body&gt; &lt;/html&gt;</pre>	

Figure 2-1 HTML example showing embedded PHP and Groovy templates

## 2.3.9 Global context

The global context is areas where your program can perform tasks:

- ▶ Access information about the current environment
- ▶ Store/access information that is shared between all requests
- ▶ Store/access information that is private to a request
- ▶ Store/access information that persists between requests
- ▶ Store/access information that exists only during the request

The two general types of zones in the global context are those that are non-persistent and those that are persistent.

All zones can be accessed from PHP, Groovy, and Java so you can use a combination of these languages in your Zero Application. An example of the use of the global context is the following statement:

```
$route = zget("/request/params/custID");
```

This statement might be used to get the value from a web browser form field named custID, and place that value in a variable named \$route.

The Project Zero documentation contains a list of all the zones and how they can be used.

## 2.3.10 Modules, dependencies, and virtualized directories

In CICS Dynamic Scripting, all applications depend on *zero.cics.core* module. The *zero.cics.core* module provides much of the base functionality for a CICS-based Zero Application. Your application might need more than just the base functionality and may have a dependency on additional features or capabilities that are related to HTTP, database interactions, Dojo support, email, and more. The application's dependencies are specified in the application's *ivy.config* file in the application's *config* directory. The *ivy.xml* file defines the name and version of the current module, and also any dependencies the module has. Version ranges can be enforced on dependencies.

As soon as the application's dependencies are specified, the dependent modules are incorporated into the application by issuing a **zero resolve** or a **zero update**. The **zero resolve** command, if a dependency was previously resolved, uses the same module version from the last resolve. The **zero update** checks for the most current version of the module (within the specified version range) and uses that module level.

From the previous two paragraphs, you might think that application dependencies are only used for adding features such as email capabilities. However, applications themselves are also *modules*. Modules inherit all assets (scripts, static files, Java classes, and so on) from their dependencies. This means that Project Zero applications can enjoy a type of inheritance model at the application level.

For example, you could have base application A. You could then specify that application B has a dependency of application A. Application B would then inherit all of application A's functionality, scripts, HTML pages, and so on. Although in this case application A's artifacts would not physically reside in application B's

directory structure, for all practical purposes, application B and A are virtually a single application.

To display the resources of a module, the modules dependencies, and the dependency's dependencies, as though they are in a single directory, a Virtualized Directory viewer is available. This viewer is useful to visualize the resources made available to your application by its dependencies. The Virtualized Directory viewer shows all of the artifacts of your application and its dependencies, and also indicates which module contributed that artifact.

For more information about modules, dependencies, the repositories where dependencies are retrieved, the `ivy.config` file, and the Virtualized Directory viewer, see section Chapter 7, “Administration” on page 87 and 10.8, “Dynamic scripting encoding considerations” on page 266.

### 2.3.11 Data model

Project Zero has a resource model called the Zero Resource Model (ZRM). The ZRM provides a simplified way to create a RESTful resource handler with a data store. Developers provide only simple “model” definitions of resources; ZRM uses the model definitions to create the data store and support full CRUD semantics. In addition, ZRM supports a variety of content formats, including JSON and Atom Publishing Protocols. ZRM in the CICS Dynamic Scripting environment supports DB2 and Derby.

The application programmer has to create only a simple file, placed in the proper directory, with the list of desired database columns and their format. Using a **zero model sync** command, a corresponding table will be created. To load initial values into a table, again, the application programmer creates only a simple file with desired column values, and the **zero model sync** command adds the values to the database table.

For more information about ZRM, JSON, other related terms and concepts, see section Chapter 11, “Derby and DB2, create easy example” on page 273.

#### REST services

Representational State Transfer (REST) is an architectural style that applies the approach we use to access web pages to access our business data. Similar to the way we use a URL to access the current state of a web page, you use a URL to access the current state of business data. We can specify a specific web page on a URL, we can also specify a specific account number on a URL.

We normally need to perform LCRUD (List, Create, Read, Update, and Delete) functions on our business data. The HTTP *methods* that flow with the request



indicate the action to be performed on the data. Whereas we normally use only a GET or a POST method when accessing a web page, for data, a GET method indicates a list of a data collection or a read of a member of a collection, a DELETE method for a delete, POST for an add, and a PUT for an update.

REST results in lightweight interactions with a minimal amount of characters transferred.

REST does not require a specific format for the returned data, although most people return data from a REST request in XML or JavaScript Object Notation (JSON) format.

REST is documented in Roy Fielding's year 2000 doctoral thesis<sup>1</sup>. It indicates that REST started in 1994 and was iteratively refined. Because many people were not aware that REST started in 1994, they think it is a follow-on to web services, however web services came after REST.

For program-to-program communications, where you want interfaces documented with WSDL, transactionality, and more security options, web services are the best choice. Where you simply need lightweight data access that uses the HTTP protocol, REST is probably the preferred choice.

One of the primary uses of REST is for requests from web browsers. JavaScript running in a web browser can use Asynchronous JavaScript and XML (AJAX) to make RESTful requests to back-end data and business logic systems such as Zero Applications.

## Atom feeds

Project Zero provides you the ability to read and write XML documents in the Atom Syndication Format through Atom Renderer and Atom APIs along with some configuration options.

For more information and an example of using Atom with ZRM in a Dynamic Scripting environment, see the following sections:

- ▶ 3.5, "Atom" on page 35
- ▶ 12.1.3, "Returning the VSAM record as an Atom feed" on page 303

---

<sup>1</sup> University Of California, Irvine, *Architectural Styles and the Design of Network-based Software Architectures Dissertation*, by Roy Thomas Fielding, 2000

## 2.3.12 Configuration files

The two main configuration files that you will probably need to customize in a CICS Dynamic Scripting environment are as follows:

- ▶ `zero.config`

The `zero.config` file occurs in every Project Zero module (application) and is in the module's config directory. This configuration file is not specific to CICS, but occurs in every module in every implementation of the Project Zero technology. The `zero.config` file contains the port your application will listen. When your application is created, this port is set to 8080 so it is likely that you will want to edit the `zero.config` file to change the port.

- ▶ `zerocics.config`

The `zerocics.config` file is in the CICS Dynamic Scripting installation's config directory. The contents of the installation `config/zerocics.config` file can be overridden by adding a `config/zerocics.config` file to your application's directory.

The contents of the `zerocics.config` file is briefly described in this chapter, and also in more detail in 6.2, "Customizing the default configuration file" on page 75.

## 2.3.13 CICS concepts

To effectively write and implement a Project Zero application using the CICS Dynamic Scripting Feature Pack, be aware of several CICS concepts.

### **Traditional CICS-based data structures and objects**

Data passed to or from CICS is in byte arrays. Although this type of data is native for COBOL, PL/I, Assembler, and other languages, a Java program wants access to the data in an object-oriented fashion using getters and setters.

Because the PHP interpreter used in CICS Dynamic Scripting is implemented in Java, there is a 'bridge' from PHP to Java. This allows you to access many Java functions, for example, some of the JCICS classes and Java-based data objects.

In the case of invoking a CICS program using a COMMAREA interface, for example, we need a way to construct the series of bytes that make up a COMMAREA, but also be able to access the individual fields in the COMMAREA using getters and setters. To do this, we can use Java data objects generated using the JZOS product, or wizards in Rational Application Developer.

JZOS is supplied with the System z Java implementations. The JZOS Java classes allow you to perform many z/OS specific functions such as reading a

PDS or a VSAM file. A function in JZOS can generate a Java data object that corresponds to a data structure in a high-level language such as COBOL.

You can compile your COBOL program with the ADATA compiler option. The results of the ADATA compiler option can be input to JZOS classes that generate a Java equivalent of the data structure. Getter and setter methods exist that correspond to the fields in the data structure plus a `getByteBuffer()` method that allows you to get the series of bytes that correspond to the data structure.

After the Java classes that correspond to the data structure that represents the COMMAREA are generated, you can instantiate the Java class in your PHP program, invoke setters on the Java object to provide data such as `setEmployeeId()`, then pass the data object to the JCICS LINK request.

Likewise, with wizards in Rational Application Developer, you can have a COBOL program be input to the wizard, indicate the data structure in question, and have the wizard generate data classes similar to those data objects generated by JZOS discussed in the previous paragraphs. Data object generation using JZOS is performed in a batch job, whereas Rational Application Developer's wizards are interactive.

## Invoking CICS Commands from Scripts

The equivalent of invoking EXEC CICS commands can be performed using the PHP to Java bridge and invoking the appropriate JCICS commands. Consider the PHP script in Example 2-1.

### *Example 2-1 PHP script*

---

```
<?php
// Instantiate a COMMAREA representation
// The com.mycompany.EMPLOYEE_CommArea class created from
// a COBOL data layout using JZOS classes supplied with z/OS Java
$commArea = new Java('com.mycompany.EMPLOYEE_CommArea');
// Set some data in the commarea by calling method on the class
$commArea->setEmployeeNumber('115');
// Use the JCICS class to call a CICS program
$program = new Java('com.ibm.cics.server.Program');
$program->setName('EMPLOYEE');
try {
    $program->link($commArea->getByteBuffer());
} catch (CICSEException $e) {
    echo $e->getMessage();
    exit;
}
echo "Return value is " . $commArea->getReturnValue();
?>
```

---

For this code example, we compiled the target CICS program (EMPLOYEE in this case) with the ADATA compiler option. We used the ADATA information representing the COMMAREA of the EMPLOYEE program as input to the JZOS classes to generate a Java object that represents the COMMAREA (which was called `com.mycompany.EMPLOYEE_Commarea` for this example).

In the code example, we use a new `Java()` request to get an instance of the class that represents the EMPLOYEE program's COMMAREA. We then invoke methods on the class to set values; the example invokes the `setEmployeeNumber()` method.

After data values are set in the object that represents the COMMAREA, we create a new CICS Program object and use the `setName()` method to indicate the program we are referring to has a name of 'EMPLOYEE' (because EMPLOYEE is the name of the target CICS program). We then invoke the `link` method of the CICS Program object, passing the byte array that represents the COMMAREA.

In the code example, you can see that after the program invocation, we are accessing getters in the data object to obtain the information returned by the EMPLOYEE program in the COMMAREA.

This slide illustrates a link to a program using a COMMAREA, but channels and containers may also be used, plus many other CICS APIs are supported.

For more details about using the JCICS classes and associated restrictions (along with a more extensive example), see 12.1.6, "More complex message interface support using JZOS" on page 321.

## **Running ZERO commands and Zero Applications in CICS**

Like all other Project Zero implementations, CICS provides a CLI for application creation and administration. The CICS-provided CLI communicates to an associated CICS region using EXCI.

Each ZERO command issued from the CLI, and each dynamic scripting application runs in its own CICS-based JVMServer resource.

After a Zero Application is running, HTTP requests are routed to the Zero Application running in the CICS environment using a TCPIPService definition to tell CICS to listen on a specified port, a URIMAP definition to direct HTTP requests to a specified PIPELINE, a PIPELINE resource whose pipeline configuration file specifies a handler that turns the HTTP request over to a JVMServer (which is running your Zero Application).

The TCPIPService, URIMAP, PIPELINE, and JVMServer resources, along with the pipeline configuration file and the `JVMProfile` file are dynamically created to run each CLI command and to run each Zero Application. When the

CLI command is complete or the Zero Application is stopped, CICS dynamically removes these resources. Although CICS dynamically creates these resources, you can tell CICS the desired characteristics of these resources in the `zero.config` and the `zerocics.config` files.

### **2.3.14 Tutorials, samples, and demos**

Several tutorials, samples, and demos are available on the Project Zero website to accelerate learning and enhance your ability to use Project Zero capabilities.

More information is provided on these tutorials, samples, and demos in 12.1.8, “The CICS catalog application tutorial” on page 329.





# Web 2.0

In this chapter, we introduce the Web 2.0 concepts and related Web 2.0 applications such as iWidgets, Mashups, REST, and Atom.

## 3.1 Web 2.0 concepts

Web 2.0 is not a single product or a single piece of technology. It is a combination of behaviors, patterns, and uses of existing technologies available for the Internet that allow for the creation of innovative solutions that bring communities and networks of people together.

Web 2.0 represents a new wave in business innovation, exploiting the maturation of the Internet as a new medium for communication and commerce. Although Web 2.0 is not a new trend, having existed since at least 2003, its adoption by business is in a relatively early stage, and its overall impact is still growing.

## 3.2 iWidgets

A *widget* is a simple and extensible browser-based component model for frameworks presenting such components to a user. To be brief, widgets are simple, and useful applications that can be embedded on a web page, blog, or social media page.

The iWidget specification is an IBM specification that defines a standard way to wrap web content so that it can participate in a mashup environment. The browser-oriented components provide either a logical service to a web page or a visualization for users.

Note the following terminology:

- iWidget

An iWidget is a browser-oriented component, potentially extending a server-side component, that provides either a logical service to the page or a visualization for the user (normally related to a server-side component or a configured data source).

- page

A page is the larger entity being composed for presentation to a user and might include both display and non-display items, each of which can come from a separate source. The composition can be changed while it is being presented (for example new items added to the composition) such that the exact definition of when a transition to a new page occurs is a policy decision left to the implementation that is managing the composition.



- ▶ iContext

The iContext components provide the overall management of the page, or some portion of it. This includes any page level controls, page layout, coordination between the various items on the page and providing the services defined in this specification to items on the page.

- ▶ Encapsulation wrapper

The wrapper portion of the overall iContext encapsulates a particular iWidget. Whether or not this conceptual portion of the overall model results in any actual implementation depends in a large measure on the implementation choices of the particular iContext.

## 3.3 Mashups

Mashups are lightweight web applications created by combining information or capabilities from more than one existing source to deliver new functions and insights. Mashups typically mash data either to create a new data source or a new application that presents data in a single graphical interface. In a business environment, a mashup typically combines enterprise and web-based data from an assembly of widgets into a single, dynamic application to address a specific situation or problem.

Prominent mashup genres are as follows:

- ▶ Mapping mashups

Humans collect prodigious amounts of data about things and activities, both of which are apt to be annotated with locations. All these diverse data sets that contain location data tend to be presented graphically using maps. One of the big catalysts for the advent of mashups was Google's introduction of its Google Maps API. This API opened the floodgates, allowing web developers to mash all sorts of data onto maps.

- ▶ Video and photo mashups

The emergence of photo hosting and social networking sites such as Flickr with APIs that expose photo sharing has led to a variety of interesting mashups. Because these content providers have metadata associated with the images they host (such as who took the picture, what it is a picture of, where and when it was taken, and more), mashup designers can mash photos with other information that can be associated with the metadata.

- ▶ Search and shopping mashups

Search and shopping mashups have existed long before the term mashup was coined. Before the days of web APIs, comparative shopping tools such as BizRate, Google's Froogle used combinations of business-to-business (B2B) technologies or screen-scraping to aggregate comparative price data.

- ▶ News mashups

News sources (such as the BBC) have used syndication technologies like RSS and Atom since 2002 to disseminate news feeds related to various topics. Syndication feed mashups can aggregate a user's feeds and present them over the web, creating a personalized newspaper that caters to the reader's particular interests.

## 3.4 REST

Representational State Transfer (REST) is a style of software architecture for distributed hypermedia systems such as the World Wide Web. The terms *Representational State Transfer* and *REST* were introduced in 2000 in the doctoral dissertation of Roy Fielding, one of the principal authors of the Hypertext Transfer Protocol (HTTP) specification. The terms have since come into widespread use in the networking community.

REST strictly refers to a collection of network architecture principles that outline how resources are defined and addressed. The term is often used in a looser sense to describe any simple interface that transmits domain-specific data over HTTP without an additional messaging layer such as SOAP or session tracking through HTTP cookies.

### Principles and benefits

REST is receiving significant interest from the current generation of Internet architects because of its ability to handle the web's scalability and growth requirements in a simple and easy to understand way. This is a direct result of a few key design principles:

- ▶ Application state and functionality are divided into resources.
- ▶ Every resource is uniquely addressable using a universal syntax for use in hypermedia links.
- ▶ All resources share a uniform interface for the transfer of state between client and resource, consisting of the following sets:
  - A constrained set of well-defined operations
  - A constrained set of content types, optionally supporting code on demand

- ▶ Internet browsers have the mechanics to invoke and consume asynchronous REST requests with no additional coding.
- ▶ REST payloads can contain various MIME types ranging from XML to JSON to TEXT.
- ▶ Offers an ability to move back to simplicity rather than being drowned in the world of enterprise standards.
- ▶ Provides improved response times and server loading characteristics because of support for caching.
- ▶ Improves server scalability by reducing the need to maintain communication state. This means that different servers can be used to handle initial and subsequent requests.
- ▶ Requires less client-side software to be written than other approaches because a single browser can access any application and any resource.
- ▶ Depends less on vendor software than mechanisms that layer additional messaging frameworks on top of HTTP.
- ▶ Provides equivalent functionality when compared to alternative approaches to communication.
- ▶ Does not require a separate resource discovery mechanism because of the use of hyperlinks in content.
- ▶ Provides better long-term compatibility and evolvability characteristics than RPC for the following reasons:
  - The capability of document types such as HTML to evolve without breaking backwards-compatibility or forwards-compatibility
  - The ability of resources to add support for new content types as they are defined without dropping or reducing support for older content types

## **Additional concepts**

The REST client-server separation of concerns simplifies component implementation, reduces the complexity of connector semantics, improves the effectiveness of performance tuning, and increases the scalability of pure server components. Layered system constraints allow intermediaries (proxies, gateways, and firewalls) to be introduced at various points in the communication without changing the interfaces between components, thus allowing them to assist in communication translation or improve performance through large-scale, shared caching.

REST enables intermediate processing by constraining messages to be self-descriptive:

- ▶ Interaction is stateless between requests.
- ▶ Standard methods and media types are used to indicate semantics and exchange information.
- ▶ Responses explicitly indicate cacheability.

An important concept in REST is the existence of resources (sources of specific information), each of which can be referred to using a global identifier (a URI). In order to manipulate these resources, components of the network (clients and servers) communicate through a standardized interface (for example, HTTP) and exchange representations of these resources (the actual documents conveying the information).

## 3.5 Atom

Atom is the name of an XML-based web content and metadata syndication format, and an application-level protocol for publishing and editing web resources belonging to periodically updated websites. All Atom feeds must be well-formed XML documents, and are identified with the application/atom+xml media type. Feeds are composed of a number of items, known as entries, each with an extensible set of attached metadata. For example, each entry has a title. See Example 3-1.

### *Example 3-1 Sample feed*

---

```
<?xml version="1.0" encoding="utf-8"?>
<feed xmlns="http://www.w3.org/2005/Atom">

  <title>Example Feed</title>
  <link href="http://example.org/" />
  <updated>2003-12-13T18:30:02Z</updated>
  <author>
    <name>John Doe</name>
  </author>
  <id>urn:uuid:60a76c80-d399-11d9-b93C-0003939e0af6</id>

  <entry>
    <title>Atom-Powered Robots Run Amok</title>
    <link href="http://example.org/2003/12/13/atom03" />
    <id>urn:uuid:1225c695-cfb8-4ebb-aaaa-80da344efa6a</id>
    <updated>2003-12-13T18:30:02Z</updated>
    <summary>Some text.</summary>
  </entry>

</feed>
```

---





# Web 2.0 technologies

This chapter introduces the central technologies for Web 2.0 and their interaction:

- ▶ PHP
- ▶ Groovy
- ▶ JavaScript
- ▶ AJAX
- ▶ JSON
- ▶ Dojo

## 4.1 Interaction pattern

In a Web 1.0 application, users fill out form fields and click a Submit button. Then, the entire form is sent to the server, the server passes on processing to a script (usually PHP or Java or maybe a CGI process or something similar), and when the script is done, it sends back a completely new page. While the script or program on the server is processing and returning a new form, users have to wait. The user's screen goes blank and then is redrawn as data returns from the server.

This means that the entire web page is reloaded in a request/response fashion. It is where low interactivity comes into play: users do not receive instant feedback and they certainly do not feel like they are working on a desktop application.

Web 2.0 combines a group of related web development technologies available for the Internet such as JavaScript, AJAX, JSON, Dojo, and so on, which can be used to create more responsive and interactive web applications. This is achieved by exchanging small amounts of data with the server asynchronously using JavaScript to communicate with the server. And the entire web page is not reloaded in a request/response fashion. Instead, the data is requested from the server and loaded in the background without interfering with the display and behavior of the existing page.

Web 2.0 essentially puts JavaScript technology and the XMLHttpRequest object between the web form and the server. When users fill out forms, the data is sent to some JavaScript code and not directly to the server. Instead, the JavaScript code grabs the form data and sends a request to the server. While this is happening, the form on the user's screen does not flash, blink, disappear, or stall. In other words, the JavaScript code sends the request behind the scenes; the user does not realize that the request is being made. Even better, the request is sent *asynchronously*, which means that your JavaScript code (and the user) does not wait around on the server to respond. Therefore, users can continue entering data, scrolling, and using the application.

Then, the server sends data back to the JavaScript code (still standing in for the web form) which decides what to do with that data. It can update form fields on the fly, giving that immediate feeling to your application: the user receives new data without the user's form being submitted or refreshed. The JavaScript code could even get the data, perform some calculations, and send another request, all without user intervention! The result is a dynamic, responsive, highly-interactive experience like a desktop application, but with all the power of the Internet behind it.



## 4.2 PHP

The first release of PHP in 1994 was developed by Rasmus Lerdorf under the name PHP/FI (Personal Home Page/Forms Interpreter). Now, PHP is a recursive acronym *PHP: Hypertext Preprocessor*.

PHP is a powerful server-side scripting language that was invented and designed for creating dynamic web applications with non-static content. The PHP code can be a stand-alone program and also an insert inside HTML (Hypertext Markup Language) or XHTML (Extensible Hypertext Markup Language). The PHP syntax is based mostly on and similar to C, Java, and Perl. You can use PHP based on an open-source license. You can run the PHP program directly from command line. PHP's modular design also allows you to build an application using the graphical user interface and an extension named PHP-GTK.

Example 4-1 shows a simple “Hello World!” PHP program.

*Example 4-1 Sample PHP program*

---

```
<html>
  <head>
    <title>Example 1</title>
  </head>
  <body>
    <?php
      echo 'Hello World !';
      $HTTP_SERVER_VARS['SERVER_SIGNATURE'];
      echo phpinfo();
    ?>
  </body>
</html>
```

---

In the HTML file, the PHP statements are included inside the `<?php` tag. The `<?php` tag is the signal to the HTML processor that PHP processing is necessary. The `echo` statement is a PHP statement.

Object-oriented programming (OOP) in PHP is introduced in Version 3 with limited functionality. In PHP 5, many OOP components, such as interfaces, access control, and abstract class have been added. OOP model is mostly based on C++ and Java. Example 4-2 on page 40 shows a simple OOP program with *interface* and *class*.

```
<?php
    interface Book {
        public function author();
        public function writeDescription();
    }

    class Description implements Book {
        private $author;

        public function __construct($author){
            $this->author = $author;
        }

        public function author(){
            return 'Book written by: ' .
                $this->author;
        }

        public function writeDescription(){
            echo 'Stories for kids mostly';
        }
    }

    $book = new Description('Hans Ch. Andersen');
    echo $book->author();
    $book->writeDescription();
?>
```

---

PHP has a modular-based design, but not all the modules are installed by default. For example, separate activation and configuration are required for all database drivers and connectors, including the newly announced PDO (PHP Data Objects) extension. PDO is an interface (some abstraction layer) for accessing databases. Each database driver that implements the PDO interface can expose database-specific features as regular extension functions. Some other official modules that you must activate separately are as follows:

- ▶ Bzip2
- ▶ GTK+
- ▶ Iconv
- ▶ Image
- ▶ IMAP, POP3, and NNTP
- ▶ IRC
- ▶ MCrypt
- ▶ Ncurses

- ▶ ODBC
- ▶ OpenSSL
- ▶ PDF
- ▶ Service Data Objects
- ▶ SOAP
- ▶ Sockets

The base PHP consists of four main modules. The most interesting modules to programmers are the PEAR (PHP Extension and Application Repository) and PECL (PHP Extension Community Library) repositories. PEAR is a framework and distribution system for reusable PHP components. PECL is an extension which contains many useful free functions based on open source licensing. These modules are created by programmers from around the world.

More detailed information is located at the following web addresses:

- ▶ <http://www.php.net/docs.php>
- ▶ <http://pecl.php.net/>
- ▶ <http://pear.php.net/>

## Why PHP

PHP is a popular web application development language. Several reasons to use PHP are as follows:

- ▶ Easy to use

PHP is a scripting language included directly in HTML. This means that getting started is easy. There is no need to compile PHP programs or spend time learning tools to create PHP. You can simply insert statements and get quick turnaround as you make changes.

- ▶ Fully functional

The PHP language has built-in functions to access your favorite database. With PHP, your HTML pages can reflect current information from databases. You can use information of the user viewing your HTML web page to customize the page specifically for that user. You can create classes for object-oriented programming, or use flat file or Lightweight Directory Access Protocol (LDAP) databases. It also includes a spell checker, XML functions, image generation functions, and more.

- ▶ Compatible and quick

PHP is compatible with all web browsers, because PHP generates plain HTML.

- ▶ Secure

Although PHP is open source, it is a secure environment. One of its advantages is that the web clients can only see the pure HTML code. The logic of the PHP program is never exposed to the client, therefore, reducing security exposures.

- ▶ Open source

PHP is an open-source programming language. It is easy to get started and find examples from websites, such as:

<http://www.sourceforge.net>

## 4.3 Groovy

Groovy is an alternate language for the JVM (*alternate* means that you can use Groovy for Java programming on the Java platform in much the same way you use Java code). Groovy offers the following features:

- ▶ Is an agile and *dynamic language* for the Java Virtual Machine.
- ▶ Builds upon the strengths of Java but has additional power features inspired by languages like Python, Ruby and Smalltalk.
- ▶ Makes modern programming features available to Java developers with almost-zero learning curve.
- ▶ Supports Domain-Specific Languages (DSL) and other compact syntax so your code becomes easy to read and maintain.
- ▶ Makes writing shell and build scripts easy with its powerful processing primitives, OO abilities and an Ant DSL.
- ▶ Increases developer productivity by reducing scaffolding code when developing web, GUI, database or console applications.
- ▶ Simplifies testing by supporting unit testing and are ready for immediate use, with little or no setup.
- ▶ Seamlessly integrates with all existing Java classes and libraries.
- ▶ Compiles straight to Java bytecode so you can use it anywhere you can use Java.

One useful feature about Groovy is that its syntax is similar to the syntax in the Java language. Although Groovy's syntax was inspired by languages like Smalltalk and Ruby, you can think of it as a simpler, more expressive variation on the Java language.

Many Java developers take comfort in the similarity between Groovy code and Java code. From a learning standpoint, if you know how to write Java code, you are already on your way to knowing Groovy. The main difference between Groovy and the Java language is that Groovy lets you write less code (sometimes far less!) to accomplish the same tasks you might labor over in your Java code.

Groovy is an open source language managed by a community of passionate Java developers. Because Groovy is licensed under the Apache Software License, Version 2.0, you are free to use it for the development of free, and also proprietary, software. Groovy 1.0 was released on January 2, 2007. The latest major and stable version is 1.7.

The Groovy website is available at:

<http://groovy.codehaus.org>

### Why Groovy

Even with all of Groovy's similarities to the Java language, it is a different language. You might be wondering why you should take the time to learn it. The short answer is that Groovy is a more *productive* language. It offers a relaxed syntax with some special features that enable you to code things more quickly.

For example, when you see how easy it is to navigate collections using Groovy, you might never work with them in Java again. Being able to code quickly in Groovy also means receiving feedback sooner, not to mention the satisfaction of crossing tasks off of your to-do list. At a high level, if you can put code in front of stake-holders more quickly, you can give them more releases in a shorter time. In essence, Groovy lends itself more to agile development than Java does.

## 4.4 JavaScript

JavaScript is a scripting language most often used for client-side web development. It was the originating dialect of the ECMAScript standard. It is a language that is dynamic, weakly typed, and prototype-based. It uses first-class functions. Although the word "Java" is in the name, JavaScript is basically unrelated to the Java programming language. Java and JavaScript, however, have a common C syntax, and JavaScript uses many Java naming conventions.

JavaScript was influenced by many languages and was designed to appear like Java, but be easier for non-programmers to work with. The language is best known for its use in websites (as client-side JavaScript).

## 4.5 AJAX

Asynchronous JavaScript and XML (AJAX) is a group of related web development techniques that can be used to create interactive web applications. A primary characteristic is the increased responsiveness and interactivity of web pages achieved by exchanging small amounts of data with the server *behind the scenes* so that entire web pages do not have to be reloaded each time that there is a need to fetch data from the server. This is intended to increase the web page's interactivity, speed, functionality, and usability.

AJAX is asynchronous, in that extra data is requested from the server and loaded in the background without interfering with the display and behavior of the existing page. JavaScript is the scripting language in which AJAX function calls are usually made. Data is retrieved using the XMLHttpRequest object that is available to scripting languages run in most current browsers, or, alternatively, through the use of Remote Scripting in browsers that do not support XMLHttpRequest. In any case, it is not required that the asynchronous content be formatted in XML.

AJAX is a cross-platform technique usable on many different operating systems, computer architectures, and web browsers, as it is based on open standards such as JavaScript and the DOM. There are free and open source implementations of suitable frameworks and libraries.

### Why AJAX

The main justification for AJAX-style programming is to overcome the page loading requirements of HTML/HTTP-mediated web pages. AJAX creates the necessary initial conditions for the evolution of complex, intuitive, dynamic, data-centric user interfaces in web pages. Indeed, much of the web's innovation and evolution during the Web 2.0 era has relied upon and benefited immensely from the capabilities of an AJAX platform.

Web pages, unlike native applications, are loosely coupled, meaning that the data that they display are not tightly bound to data sources and must be first marshaled (set out in proper order) into an HTML page format before they can be presented to a user agent on the client machine. For this reason, web pages have to be re-loaded each time a user needs to view different data sets. By using the XMLHttpRequest object to request and return data without a re-load, a developer bypasses this requirement and makes the loosely coupled web page behave much like a tightly coupled application, however, with a more variable lag time for the data to pass through a longer wire to the remote web browser.

For example, in a classic desktop application, a web developer has the option of populating a tree view control with all the data needed when the form initially

loads, or with just the top-most level of data, which would load more quickly, especially when the data set is very large. In the second case, the application would fetch additional data into the tree control depending on which item the user selects. This functionality is difficult to achieve in a web page without AJAX. To update the tree based on a user's selection would require the entire page to reload, leading to a very jerky, non-intuitive feel for the web user who is browsing the data in the tree.

## 4.6 JSON

JavaScript Object Notation (JSON) is a lightweight computer data interchange format used to represent data in the business logic running on browsers. It is a text-based, human-readable format for representing simple data structures and associative arrays (called objects).

The JSON format was specified in RFC 4627 by Douglas Crockford. The official Internet media type for JSON is `application/json`. The JSON file extension is `.json`.

The JSON format is often used for transmitting structured data over a network connection in a process called serialization. It is primarily used in AJAX web application programming, where it serves as an alternative to the traditional use of the XML format.

Although JSON was based on a subset of the JavaScript programming language (specifically, Standard ECMA-262 3rd Edition, December 1999) and is commonly used with that language, it is considered to be a language-independent data format. Code for parsing and generating JSON data is readily available for a large variety of programming languages. The JSON website provides a comprehensive listing of existing JSON bindings, organized by programming language:

<http://json.org/>

## 4.7 Dojo

Dojo is a multi-platform, open source JavaScript toolkit backed by IBM, Oracle, SitePen, AOL, and other companies. It is not tied to any specific server-side toolkit. Therefore, it can be used in conjunction with PHP, Java servlets, Java Server Pages, ASP, and so on, or even with mobile devices.

See the Dojo website for the latest information regarding the Dojo Toolkit and APIs:

<http://www.dojotoolkit.org/>

This chapter describes the main concepts of Dojo, and therefore does not provide an API description of the toolkit.

The Dojo toolkit has a modular design:

- ▶ Base: The base is the kernel of the toolkit. Dojo Base bootstraps the toolkit and includes the following features:
  - A module loader
  - Language utilities, for example, array functions, functions for declaring classes, and inheritance
  - Query, node, and CSS utilities
  - Functions for cross-browser communication, for example, XHR
  - JSON serialization/deserialization
  - A cross-browser event and a publish/subscribe system
  - Color functions
  - Browser type detection
  - URL functions
  - Document load/unload hooks
  - Effect functions for fading, sliding, and animating different style properties of an element
- ▶ Core: The Core provides additional facilities on top of Dojo Base:
  - Drag and drop support
  - Back-button support
  - String manipulation functions
  - RPC, IFrame communication
  - Internationalization (i18n)



- Date
- Number
- Currency
- Colors
- Strings
- Math
- Data access
- Regular expressions
- Debug facilities (through Firebug lite)
- Build system
- Markup parser
- OpenAJAX hub 1.0
- ▶ Dijit: This is a set of interaction rich widgets and themes for use when developing AJAX applications. Dojo widget features are as follows:
  - Accessible (keyboard support, support for high-contrast mode, panel reader support)
  - High quality, neutral default theme (replaceable)
  - Extensive layout and form capabilities
  - Data-bound widgets
  - Grid and charts
  - Fully internationalized for many languages
  - Bidirectional (BiDi) support
- ▶ DojoX (Dojo eXtensions): This contains features that may in the future migrate into Core, Dijit, or even a new module. Many of these extensions are community provided.
- ▶ Util: The AJAX client run time provides the Dojo Objective Harness (DOH) in the Util module. DOH is a client testing framework.





## Part 2

# Systems management

This part of the book focuses on systems management tasks:

- ▶ Dynamic Scripting Feature Pack installation
- ▶ Customization of the Dynamic Scripting Feature Pack
- ▶ Administration
- ▶ Troubleshooting
- ▶ Development options





# Installation

In this chapter, we discuss installation of the Dynamic Scripting Feature Pack. We cover software prerequisites, installation options, repository access and security settings.

## 5.1 Software prerequisites

Prerequisites are as follows:

- ▶ z/OS 1.9 or later
- ▶ CICS Transaction Server for z/OS 4.1 with APARs PM08649, PM08661, PM11157, and PM11791 applied

With all required maintenance, you see the following information:

- SDFHLOAD contains module DFHSJJI.
- Program DFHSJJI is defined to CICS.
- The CEMT INQUIRE JVMPOOL command shows the value of the JVMPROFILEDIR System Initialization Parameter.

- ▶ Java 6, service release 8 or later

Use the **java -version** command to determine the level (Example 5-1).

*Example 5-1 Example output from java -version command*

---

```
java version "1.6.0"
Java(TM) SE Runtime Environment (build pmz3160sr8fp1-20100624_01(SR8
FP1))
IBM J9 VM (build 2.4, JRE 1.6.0 IBM J9 2.4 z/OS s390-31
jvmmz3160sr8ifx-20100609_59383 (JIT enabled, AOT enabled)
J9VM - 20100609_059383
JIT - r9_20100401_15339ifx2
GC - 20100308_AA)
JCL - 20100624_01
```

---

- ▶ DB2 Version 9.1 for z/OS with APAR PK71020 applied

This prerequisite applies only when you want to write dynamic scripting applications that connect to DB2.

## 5.2 Choosing an installation location

The Dynamic Scripting Feature Pack is downloadable as a ZIP file. This file must be saved onto the mainframe and then extracted to begin the installation process.

Where you save the ZIP file is where the Feature Pack will be installed. In this section, we discuss downloading the ZIP file to the mainframe, and then where the Feature Pack may be installed. We also list advantages and disadvantages of each.

The ZIP file must be inflated using the Java **jar** command. The UNIX **tar** command is not supported. The **jar** command does not support inflation to a separate directory, therefore you must place the ZIP file in the desired directory first.

The file location might be accessible across the sysplex, or might be accessible only on one image. Separate file systems have separate mount policies. The UNIX System Services command **df filename** can be used to see which data set backs the UNIX System Services file. Data sets or directory structures with system names or z/OS version numbers may be unique to one image.

### 5.2.1 Space requirements

The initial ZIP file is 16 MB in size.

After the file is inflated by using the **jar** command, the initial Dynamic Scripting Feature Pack is approximately 47 MB in size.

If the Dynamic Scripting repository is downloaded and installed on the local file system, it is approximately 343 MB in size.

The XDFHRPL file that contains the load modules for the DFHRPL concatenation is 16 tracks in size in XMIT format. After it has been expanded with TSO RECEIVE, the resulting load library is 12 tracks in size.

## 5.2.2 Installation location options

Options are as follows:

- ▶ Create a new system file system.

Creating a new file system for Dynamic Scripting is described in “Creating a new system file system” on page 54.

- ▶ Use an existing user file system.

Use an existing user’s file system to install Dynamic Scripting. This is described in “Using an existing user directory structure” on page 57.

- ▶ Use an existing system file system such as /usr/lpp.

IBM installs products into /usr/lpp location. See “Use an existing system directory structure: /usr/lpp” on page 58.

- ▶ Use an existing system file system such as /etc.

The /etc directory is for configuration of the system. Using this file system is described in “Use an existing system directory structure: /etc” on page 59.

### Creating a new system file system

A new file system is created and used to install Dynamic Scripting. This process is in four stages:

1. Create a new zFS file.

zFS files are VSAM Linear data sets. They are allocated using IDCAMS function. Example 5-2 shows the allocation of a zFS file that has the following name:

CICSSEM.ZFS.REDBOOK

We chose zFS instead of HFS, because zFS is the preferred file system. *UNIX Systems Services Planning*, GA22-7800 states that new file systems should be created as zFS file systems.

*Example 5-2 IDCAMS command to allocate a new zFS file*

---

```
//DEFINE EXEC PGM=IDCAMS
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//AMSDUMP DD SYSOUT=*
//SYSIN DD *
DELETE CICSSEM.ZFS.REDBOOK
DEFINE CLUSTER (NAME(CICSSEM.ZFS.REDBOOK) -
LINEAR CYL(50 50) SHAREOPTIONS(3))
/*
```

---



2. Format the zFS file.

VSAM Linear data sets must be formatted before use as a zFS file. The **zfsadm format** command, documented in *z/OS Distributed File Service zSeries File System Implementation z/OS V1R11*, SG24-6580 must be issued, as shown in Example 5-3.

*Example 5-3 The zfsadm format command to format VSAM data set as zFS file*

---

```
//FORMAT EXEC PGM=IKJEFT01,REGION=0M,COND=(0,LT)
//SYSTSPRT DD SYSOUT=*
//STDOUT DD SYSOUT=*
//STDERR DD SYSOUT=*
//SYSTSIN DD *
BPXBATCH SH +
  zfsadm format -aggregate +
  CICSSEM.ZFS.REDBOOK +
  -compat
/*
```

---

3. Create an empty directory to serve as the mount point.

The new zFS must be mounted within the existing file structure. The mount point must be an empty directory, otherwise the contents of the directory is hidden until the zFS is unmounted.

Example 5-4 shows an example of the UNIX System Services **mkdir** command to create a new directory. This is also possible using the **ishell** command, or the Time Sharing Option (TSO) command **MKDIR**. In Example 5-4, we create the directory in the root directory, **/**.

*Example 5-4 Command to create a new directory to use as the mountpoint*

---

```
//MKDIR EXEC PGM=IKJEFT01,REGION=0M,COND=(0,LT)
//SYSTSPRT DD SYSOUT=*
//STDOUT DD SYSOUT=*
//STDERR DD SYSOUT=*
//SYSTSIN DD *
BPXBATCH SH +
  mkdir -p /dynamicScript
/*
```

---

4. Mount the zFS file at the empty directory. See Example 5-5.

The TSO MOUNT command, the UNIX System Services **mount** command, or entries in the PARMLIB member **BPXPRMxx** can be used to mount the file system. If **automount** command is not being used, then adding entries to **BPXPRMxx** can ensure that the mounting persists over an initial program load (IPL).

*Example 5-5 Command to mount the new zFS file in read-write mode*

---

```
//MOUNT      EXEC      PGM=IKJEFT01,REGION=0M,COND=(0,LT)
//SYSTSPRT DD  SYSOUT=*
//SYSTSIN DD  *
      MOUNT FILESYSTEM('CICSSEM.ZFS.REDBOOK') -
          TYPE(ZFS) MODE(RDWR) -
          MOUNTPPOINT('/dynamicScript')
/*
```

---

### **Advantages**

Advantages are as follows:

- ▶ A new file system has no existing security rules or files so can be managed as required.
- ▶ Mounting the file system at the root is useful in a sysplex environment as the files are available to all images in the sysplex.

### **Disadvantages**

Disadvantages are as follows:

- ▶ A new file system must be created, which requires disk space and issuing commands.
- ▶ The sysplex root may be mounted as read-only, in which case it must be remounted as read-write, a process that can be disruptive.
- ▶ In a non-sysplex environment, the version file system may be loaded at the root. This means that any maintenance applied to the mainframe may remove the mount point.
- ▶ As noted in z/OS UNIX System Services Planning, there is an upper limit to the number of HFS or zFS file systems that can be mounted at one time in your system. Each file system consumes approximately 1 KB of storage below the 16M line when mounted. At z/OS 1.12, the SWA(ABOVE) option in BPXPRMxx can be used to force this storage to be allocated above the 16M line. See the BPXPRMxx topic in *z/OS MVS Initialization and Tuning Reference*, SA22-7592 for more details about the SWA parmlib statement.

## Using an existing user directory structure

Each user with an OMVS Segment has a home directory. This can be seen with the **echo \$HOME** command. Output is listed in Example 5-6.

*Example 5-6 Example output for echo \$HOME for user ID CICSR2*

---

```
/u/cicrs2
```

---

Using RACF®, issuing the command **LISTUSER CICSR2 OMVS NORACF** can also show the home directory (Example 5-7).

*Example 5-7 Output for LISTUSER command for user ID CICSR2*

---

```
USER=CICSR2
```

```
OMVS INFORMATION
-----
UID= 0000000680
HOME= /u/cicrs2
PROGRAM= /bin/sh
CPUTIMEMAX= NONE
ASSIZEMAX= NONE
FILEPROCMAX= NONE
PROCUSERMAX= NONE
THREADSMAX= NONE
MMAPAREAMAX= NONE
```

---

We install Dynamic Scripting into this file system, which is backed by zFS file CICSR2.HFS.

### **Advantages**

Advantages are as follows:

- ▶ Convenience  
The user directory already exists and the user already has access to create files.
- ▶ Cross LPAR access  
User file systems are typically accessible across the sysplex.

## ***Disadvantages***

Disadvantages are as follows:

- ▶ **Security**

User directories might not be set up to be shared across multiple users. The CICS region user ID might not have security access to user directories.

- ▶ **Space**

Initial installation of Dynamic Scripting requires 47 MB of space. This can grow as additional resources are added to the repository. Enough space might not be allocated for user zFS file systems, which can cause errors. See Example 5-8.

---

*Example 5-8 System Log messages showing HFS file system growth*

---

```
IOEZ00551I Aggregate CICSRS2.HFS ran out of space.
IOEZ00312I Dynamic growth of aggregate CICSRS2.HFS in progress, (by
user CICSRS2).
IOEZ00329I Attempting to extend CICSRS2.HFS by a secondary extent.
IOEZ00324I Formatting to 8K block number 11700 for secondary extents
of CICSRS2.HFS
IOEZ00309I Aggregate CICSRS2.HFS successfully dynamically grown (by
user CICSRS2).
```

---

## **Use an existing system directory structure: /usr/lpp**

IBM products with files in UNIX Systems Services are installed into /usr/lpp, which is where IBM installs Licensed Program Products.

## ***Advantages***

Advantages are as follows:

- ▶ **Convenience**

The system directory already exists, and is configured for access from multiple users.

- ▶ **Convention**

IBM installed products are found in /usr/lpp by convention, so it would be a logical place to install Dynamic Scripting.

**Disadvantages**

Disadvantages are as follows:

- Read-only  
System file systems might be mounted as read-only, especially if they are populated from SMP/E, which can prevent the creation of new directories.  
To determine whether a file system is mounted as read-only, issue the UNIX System Services command **df filename** for any file in the file system. See Example 5-9.

*Example 5-9 Sample df command and response*

---

```
df /usr/lpp
```

Mounted on	Filesystem	Avail/Total	Files
/Z1BRE1	(OMVS.ZOSR1B.Z1BRE1.ROOT)	623694/5726880	4294948687

---

Searching for the matching MOUNT command in PARMLIB showed that the file system was mounted in MODE(READ).

- Possibility of loss  
If the Feature Pack is installed next to SMP/E data, the Systems Programmer might delete and recreate the file system when applying maintenance to z/OS or CICS.
- Space  
The file system might not be allocated with sufficient space to cope with expansion, which can cause errors.

**Use an existing system directory structure: /etc**

The /etc file system is the location for your own customization data for products. You set up the /etc files and you maintain their content. IBM products create directories under /etc during installation, but does not create files under /etc during SMP/E installation. Because IBM products do not create files into /etc, there is no possibility that SMP/E installation of an IBM product or service will overlay your own files within /etc.

### ***Advantages***

Advantages are as follows:

- ▶ Convenience  
The system directory already exists, and is configured for access from multiple users.
- ▶ Safety  
IBM installations do not overwrite files in /etc.
- ▶ Locality  
The /etc is typically mounted separately on each image in the sysplex, which means that installations of Dynamic Scripting might be kept distinct.

### ***Disadvantages***

Disadvantages are as follows:

- ▶ Locality  
If /etc is mounted separately on each image in the sysplex, separate installations of Dynamic Scripting will need to be performed.
- ▶ Portability of applications and CICS regions  
Dynamic scripting applications run within a CICS region. If /etc is mounted locally, then changing the image on which the application runs, by either changing the CICS region in `zerocics.config`, or restarting the CICS region on another image, may cause the application to fail.
- ▶ Space  
The file system might not be allocated with sufficient space to cope with expansion, which can cause errors.

## **5.3 Downloading the ZIP file to the mainframe**

Perform the following steps, using Rational Developer for System z:

1. For Windows, open a Command Prompt.
2. Change directory to where you saved the ZIP file from IBM.
3. Enter the **ftp** command to start an FTP session.
4. Enter the command **open wtsc66.itso.ibm.com** to open an FTP session to host `wtsc66.itso.ibm.com`.
5. The file must be transferred to the mainframe as binary. To set the transfer mode, issue the command **binary**.

6. Change to the desired directory. z/OS Communications Server may be configured to one of two modes. The default is for STARTDIRECTORY (see Example 5-10) to be set to MVS™. In this case, you must change the directory into UNIX Systems Services to the dynamicScript directory:

```
cd /dynamicScript
```

*Example 5-10 Logon message with STARTDIRECTORY MVS*

---

```
230-220-FTPMVS1 IBM FTP CS V1R11 at wtsc66.itso.ibm.com.  
230-CICSR2 is logged on. Working directory is "CICSR2.".
```

---

7. Enter the following command:

```
put cics_dynamic_scripting_v1000.zip
```

## 5.4 Inflating the ZIP file

As noted in the CICS Transaction Server information center for installing the Feature Pack, the downloaded ZIP file must be inflated using the Java **jar** command.

So that the **jar** command works correctly, Java must be installed and within the PATH search order.

Alternatively, the **jar** command can be issued specifying the fully qualified location in the file system of the **jar** executable.

The **jar** command inflates the ZIP file in the current directory. If you want to inflate the ZIP file to a different directory, you must move or copy it first.

### 5.4.1 Inflating the ZIP file using TSO OMVS

Perform the following steps:

1. Issue the **TSO OMVS** command to start a UNIX System Services command shell.
2. Issue change directory (**cd**) command to navigate to the correct directory.
3. Issue the **jar -xf cics\_dynamic\_scripting\_v1000.zip** command to inflate the ZIP file. (Using the option **-xvf** instead gives verbose mode.)

## 5.4.2 Inflating the ZIP file using TSO ishell

Perform the following steps:

1. Issue the command **TSO ishell** to start a UNIX System Services command shell.

Navigating to the correct directory in ishell (ISPF shell for UNIX) has no effect because all commands start a new shell session in the user's home directory. Therefore, the command must include the **cd** command to change to the directory, followed by the **jar** command.

2. Issue the following command to inflate the ZIP file (using the option **-xvf** instead gives verbose mode):

```
sh cd /dynamicScript && jar -xf cics_dynamic_scripting_v1000.zip
```

This step changes directory to **/dynamicScript** and issues the **jar inflate** command.

## 5.5 Configuring the UNIX environment

For a user to be able to successfully issue Dynamic Scripting commands, it is necessary to specify four environment variables.

### ► ZERO\_HOME

This is the installation directory of the Dynamic Scripting Feature Pack. In our installation, this is **/dynamicScript/zero**.

### ► PATH

This is the search order for commands. We need to add **ZERO\_HOME** to the **PATH**, so the command interpreter can find the **zero** command.

### ► STEPLIB

This is Dynamic Scripting using the External Call Interface, EXCI, to communicate with CICS Transaction Server. For this to work, the CICS load library **SDFHLOAD** and the EXCI library **SDFHEXCI** must be allocated to the **STEPLIB** environment variable. In our installation, these data sets are **CICSTS41.CICS.SDFHEXCI** and **CICSTS41.CICS.SDFHLOAD**.

### ► JAVA\_HOME

This is the installation directory of Java. As noted in 5.1, “Software prerequisites” on page 52, the version of Java must be Java 6, service release 8 or later. In our installation, Java is installed at **/usr/lpp/java/J6.0**.



These options are set by the export command, as Example 5-11 shows.

*Example 5-11 UNIX System Services export commands to set environment variables.*

---

```
export ZERO_HOME=/dynamicScript/zero
export PATH=$ZERO_HOME:$PATH
export STEPLIB=CICSTS41.CICS.SDFHEXCI:CICSTS41.CICS.SDFHLOAD
export JAVA_HOME=/usr/lpp/java/J6.0
```

---

### 5.5.1 Specifying the environment variables for all users

The `/etc/profile` file provides a default system-wide user environment. You might want all users of UNIX System Services to be able to issue Dynamic Scripting commands.

If so, add the lines in Example 5-11 to the `/etc/profile` file.

#### ***Advantage***

All users use the same default environment.

#### ***Disadvantage***

All users of UNIX System Services allocate CICS libraries for the duration of their session, no matter what they are doing, which can inhibit the application of maintenance to CICS systems.

### 5.5.2 Specifying the environment variables for one user

A `.profile` file can be created in the user's home directory, and can contain the commands in Example 5-11. This `.profile` file is executed every time this user starts a session with UNIX System Services, and any values in `.profile` override those in `/etc/profile`.

#### ***Advantages***

Advantages are as follows:

- ▶ Each user may use a separate environment.
- ▶ Only some users will allocate CICS libraries while logged on.

#### ***Disadvantages***

Disadvantages are as follows:

- ▶ A separate `.profile` file must be created and maintained for each user.
- ▶ Users with these commands will still allocate CICS libraries, no matter what they are doing.

### 5.5.3 Specifying the environment variables in a shell script

A shell script is an executable file in UNIX System Services. It can be executed by users as required. A file of any name can be created and populated with the commands in Example 5-11 on page 63.

The file must be marked as executable. We create a file `/dynamicScript/setup` and mark it as executable.

#### ***Advantages***

Advantages are as follows:

- ▶ One shell script can be maintained centrally for each installation. Each user may use a different environment.
- ▶ Only users actively using Dynamic Scripting will allocate CICS libraries while logged on.

#### ***Disadvantage***

Users must remember to execute the shell script every time they log on.

## 5.6 Creating the CICS load library for Dynamic Scripting

Within the `/dynamicScript/cics` directory is the `XDFHRPL` file, which is the result of a TSO TRANSMIT command. Before the file can be used, it must be inflated with the TSO RECEIVE command.

TSO RECEIVE is an authorized command and cannot be used from UNIX System Services. TSO RECEIVE cannot receive files from UNIX System Services.

You must copy the file from UNIX System Services to a data set of the correct format. The correct format in this case is Fixed Block, with a Logical Record Length of 80 bytes.

### 5.6.1 Copying the file from UNIX System Services using TSO OMVS

The UNIX System Services command `cp` can be used with the following options to copy the file to a data set in one command:

- ▶ The `-P` option allows the specification of parameters, within quotes, which are used to create a new sequential data set.
- ▶ `XDFHRPL` is the name of the file to be copied.

- ▶ The final parameter is the name of the target file.
  - The double slashes indicate an MVS file.
  - The single quotation marks are required to prevent the current user ID being added to the data set name as a prefix.
  - The double quotation marks are required to prevent the single quotation marks from being interpreted by the shell, which can cause an error.

See Example 5-12. The first command copies XDFHRPL to a new data set called DYNAMIC.SCRIPT.LOADLIB.XMIT. The second command copies XDFHRPL to a new data set which will begin with the current user ID.

*Example 5-12 Two examples of the cp command.*

---

```
cp -P "RECFM=FB,LRECL=80" XDFHRPL "'/'DYNAMIC.SCRIPT.LOADLIB.XMIT'"

cp -P "RECFM=FB,LRECL=80" XDFHRPL "'/'DYNAMIC.SCRIPT.LOADLIB.XMIT"
```

---

## 5.6.2 Copying the file from UNIX System Services using TSO ishell

The ISPF shell for UNIX, ishell, allows easy copying of files to data sets. However, it does not offer the opportunity of specifying data set attributes. Therefore, you must preallocate the data set before copying into it.

This step can be done by using the ISPF Data Set Utility, or the TSO ALLOCATE command. Example 5-13 shows the command without quotation marks, which allocates a data set prefixed with the current user ID.

*Example 5-13 TSO ALLOCATE command to allocate a data set*

---

```
ALLOCATE DATASET(DYNAMIC.SCRIPT.LOADLIB.XMIT) DSORG(PS) LRECL(80)
RECFM(f,b) BLKSIZE(3120)
```

---

You must specify the **Binary copy** option otherwise the file will become corrupted.

## 5.6.3 Issuing the RECEIVE command

The TSO RECEIVE command must be issued to expand the data set. The TSO profile must be set to PROMPT if you want to select a data set name other than the default. See Example 5-14 on page 66.

*Example 5-14 Example of TSO RECEIVE to create the CICS load library.*

---

TSO PROFILE PROMPT

TSO RECEIVE INDSNAME(DYNAMIC.SCRIPT.LOADLIB.XMIT)

INMR901I Dataset P8BUILD.P8BUILDI.BUILD.LOAD from P8BUILD on WINMVS28

INMR154I The incoming data set is a 'PROGRAM LIBRARY'.

INMR906A Enter restore parameters or 'DELETE' or 'END' +

DSNAME(DYNAMIC.SCRIPT.LOADLIB)

IEBCOPY MESSAGES AND CONTROL STATEMENTS

PAGE 1

IEB1135I IEBCOPY FMID HDZ1A10 SERVICE LEVEL UA53210 DATED 20100315

DFSMS 01.11.00 z/OS 01.11.00 HBB7760 CPU 2097

IEB1035I CICSRS2 IKJACCT IKJACCFK03:32:57 TUE 19 OCT 2010 PARM=' '

COPY INDD=((SYS00086,R)),OUTDD=SYS00085

IEB1013I COPYING FROM PDSU INDD=SYS00086 VOL=

DSN=SYS10292.T033257.RA000.CICSRS2.ROF00070

IEB1014I TO PDSE OUTDD=SYS00085 VOL=TST015

DSN=CICSRS2.DYNAMIC.SCRIPT.LOADLIB

IGW01551I MEMBER ZERODISC HAS BEEN LOADED

IGW01551I MEMBER ZEROINQJ HAS BEEN LOADED

IGW01551I MEMBER ZEROLIFE HAS BEEN LOADED

IGW01551I MEMBER ZEROQUIE HAS BEEN LOADED

IGW01551I MEMBER ZEROSERV HAS BEEN LOADED

IGW01550I 5 OF 5 MEMBERS WERE LOADED

IEB147I END OF JOB - 0 WAS HIGHEST SEVERITY CODE

INMR001I Restore successful to dataset 'CICSRS2.DYNAMIC.SCRIPT.LOADLIB'

---

## 5.7 Repository access zerocics.config

Dynamic Scripting Feature Pack downloads and installs resources as and when they are required. This means that at initial installation time, the LPAR on which it is installed must have access to a correctly configured repository.

The default repository can be found at the following address:

<http://download.boulder.ibm.com/ibmdl/pub/software/http/cics/updates/ds/1.0/repo/base/>

If this repository cannot be accessed, the initial **zero** command fails.

## 5.8 UNIX System Services security

Files within UNIX System Services are owned by a UID (user ID) and have an owning group of a GID (group ID). UID and GID are numbers that are associated with the OMVS Segment of a user ID and a RACF Group respectively.

Permissions are set on a file or directory giving access to the owner, the group and other. Each permission can include write access, read access, and execute or search access. These permissions are commonly represented as octal numbers:

- ▶ 4 is equal to write
- ▶ 2 is equal to read
- ▶ 1 is equal to execute or search

To determine which permission to use, determine the following information:

1. Is the UID accessing the file or directory the same as that of the owner? If so, use the owner permissions.
2. If not, does the UID belong to the same GID as the file? If so, use the group permissions.
3. If not, then use the other permissions.

A UID may be unique to one user, or may be shared between many users. A user must be in one group, but may be in more than one group. One of these groups will be the default group used when creating a new file.

The UNIX System Services command `id` shows the UID, GID and groups for the currently signed-on user, as Example 5-15 shows.

*Example 5-15 Response from the UNIX System Services ID command*

---

```
uid=680(CICSR2) gid=0(SYS1)
groups=2300(CBCFG1),87679(CICS),87837(R4003)
```

---

Membership of a RACF Group confers membership of a GID only if the RACF Group has a GID defined. This GID can be specified, shared or automatically assigned.

When the Dynamic Scripting Feature Pack runs commands, files and directories may be created. These files may be created by the user running the command, or they may be created by the CICS region user ID:

- ▶ The CICS region user ID must have a UID.

As noted in *CICS Transaction Server for z/OS Installation Guide*, in the section *Authorizing access to z/OS UNIX System Services*, CICS requires access to z/OS UNIX System Services. The user ID associated with the CICS region must have an OMVS Segment in its RACF user profile. This is required for access to CICS web support, TCP/IP and Java.

- ▶ The user issuing command-line commands must have a UID.
- ▶ The CICS region user ID and the user issuing commands must both be members of the same group.

This is because files are created and accessed by two separate user IDs while running any **zero** command, the command-line user and the CICS region user ID. Because UNIX files can only be owned by one user, the group access is used to allow the other user ID to access the file. If the users and the CICS regions did not share a group, then file accesses may fail unexpectedly with unpredictable results.

### 5.8.1 Creating a new RACF UNIX group

The easiest way to ensure that command-line users and CICS region user IDs have a group in common is to create one.

#### ***Advantages***

Advantages are as follows:

- ▶ The new group has no existing members or authorities, so no unexpected access are granted by adding members to it.
- ▶ The new group can exist for the sole purpose of developing dynamic scripting applications, and access to the group can be used to control access to Dynamic Scripting.
- ▶ Command-line users must have READ access to the CICS SDFHLOAD and SDFHEXCI load libraries. The group can be granted access instead.

#### ***Disadvantages***

Disadvantages are as follows:

- ▶ An additional RACF group will need to be created. This may be considered an overhead by security administrators.
- ▶ An additional UNIX System Services group ID or GID must be created.

## Creating a new RACF UNIX Group using ISPF panels

Navigate to the RACF - SERVICES OPTION MENU, as shown in Example 5-16. The exact method to start the menu will vary with your installation of z/OS.

### *Example 5-16 RACF ISPF Menu*

---

```
RACF - SERVICES OPTION MENU
OPTION ===>
```

SELECT ONE OF THE FOLLOWING:

- 1 DATA SET PROFILES
  - 2 GENERAL RESOURCE PROFILES
  - 3 GROUP PROFILES AND USER-TO-GROUP CONNECTIONS
  - 4 USER PROFILES AND YOUR OWN PASSWORD
  - 5 SYSTEM OPTIONS
  - 6 REMOTE SHARING FACILITY
  - 7 DIGITAL CERTIFICATES, KEY RINGS, AND TOKENS
  - 99 EXIT
- 

Select option **3** to start the panel shown in Example 5-17.

### *Example 5-17 RACF - Group Profile Services panel*

---

SELECT ONE OF THE FOLLOWING.

- |        |         |                                        |
|--------|---------|----------------------------------------|
| 1      | ADD     | Add a group profile                    |
| 2      | CHANGE  | Change a group profile                 |
| 3      | DELETE  | Delete a group profile                 |
| 4      | CONNECT | Add or change a user connection        |
| 5      | REMOVE  | Remove users from the group            |
|        |         |                                        |
| D or 8 | DISPLAY | Display profile contents               |
| S or 9 | SEARCH  | Search the RACF data base for profiles |

ENTER THE FOLLOWING INFORMATION.

```
GROUP NAME      ===>
```

---

Select option **1**, and at the bottom of the screen type the name of the new group. The new group we create is named ZERORED.

Press Enter to open the panel, shown in Example 5-18 on page 70. Be sure to select OMVS Parameters in the optional information section.

*Example 5-18 RACF- ADD GROUP panel*

---

Enter the following information:

OWNER	====> CICSRS2	Userid or group name
SUPERIOR GROUP	====> SYS1	
USE TERMINAL UACC	====> YES	YES or NO
UNIVERSAL ATTRIBUTE	====> NO	YES or NO

Identify a model profile for group datasets (optional):

PROFILE NAME                   ====>

To ADD the following optional information, enter any character:

\_ INSTALLATION DATA  
\_ DFP PARAMETERS  
/ OMVS PARAMETERS  
\_ OVM PARAMETERS  
\_ CSDATA PARAMETERS

---

The OMVS Parameters panel opens, as shown in Example 5-19.

*Example 5-19 OMVS Parameters panel of RACF Add Group panel*

---

Enter OMVS segment information:

Specify Group Identifier       (GID) \_\_\_\_\_ 0 - 2147483647  
Allow shared use of this GID (SHARED) \_           Enter any  
character  
  
-- OR --

Assign a unique GID           (AUTOGID) /           Enter any  
character

---

You may specify a Group Identifier directly, or to let RACF select a currently unused Group Identifier by using the AUTOGID option. The results of this choice is shown in Example 5-20.

*Example 5-20 Results of creating a group with AUTOGID option*

---

IRR52177I Group ZEROED was assigned an OMVS GID value of 383.

---



## 5.8.2 Running the install command

The final stage of installation is to run the **install** command from within the zero installation directory.

The **install** command takes an optional argument, which is the name of the RACF Group you wish to share between CICS regions and command-line users.

When run, the **install** command sets the owning group for all the installation directories and sub-directories to this group.

However, you must still do this for any application directories that you want to set up and use.





## Customizing CICS Dynamic Scripting Feature Pack

The CICS Dynamic Scripting Feature Pack provides an agile web application platform for developing and running modern web applications. You install the feature pack and create applications in z/OS UNIX, and then start and run your dynamic scripting applications in CICS. The feature pack uses EXCI to connect to the CICS region which allows you to manage and administer your dynamic scripting applications.

This section takes you through the steps needed to customize a single CICS region to run a dynamic scripting application.

Before continuing with this chapter, you should have already installed the CICS Dynamic Scripting Feature Pack by following the steps outlined in Chapter 5, “Installation” on page 51, then follow the instructions in this chapter once for each CICS region in which you want dynamic scripting applications to run.

## 6.1 Configuring CICS for dynamic scripting

Language Environment® is a prerequisite for Java programs that run in JVMs in CICS. To run dynamic scripting applications in your CICS regions, you must ensure that the Language Environment libraries are available. You must also ensure that specific system initialization parameters are set correctly.

Perform the following steps for each CICS region that you want to support dynamic scripting applications:

1. Add the following Language Environment libraries to the STEPLIB concatenation:

```
SCEERUN  
SCEERUN2
```

2. Add the following libraries to the DFHRPL concatenation:

```
SCEECICS  
SCEERUN  
SCEERUN2
```

3. Define all the CSD resources in the following members of the SCEESAMP library:

```
SCEESAMP(CEECCSD)  
SCEESAMP(CEECCSDX)
```

Ensure that these members are up to date.

4. Specify the following system initialization parameters and values:
  - Specify ISC=YES and IRCSTRT=YES to enable the feature to connect to the CICS region using the EXCI interface.
  - Specify TCPIP=YES to enable web support in the CICS region.
  - Specify JVMPROFILEDIR=directory, where directory is the name of a directory to which your CICS region has write access.

The dynamic scripting application dynamically creates a JVM profile for the JVM server in this directory. Each CICS region must specify a unique location for the JVMPROFILEDIR system initialization parameter. Sharing a JVM profile directory between CICS regions is not supported for dynamic scripting applications.

5. Copy the CICS-supplied JVM profiles from the CICS installation directory to the directory that you specified for the JVMPROFILEDIR system initialization parameter.

6. Create a group that contains all the required CICS resources to support the feature, including programs, transactions, and connections:
  - a. Copy the `/install_directory/cics/zerocics.csd` file from z/OS UNIX to a new data set. You can use the following command:

```
cp -P "RECFM=FB,LRECL=80" zerocics.csd "'/'CTS.ZEROCICS.DATA'"
```

This command copies the file in the correct format to the CTS.ZEROCICS.DATA data set.
  - b. Use the data set as input to DFHCSDUP to create the DFH\$ZERO group in the CSD.
  - c. Add the group to a list that is included in the GRPLIST system initialization parameter.
7. Optional: If you want to edit the resource definitions, copy the definitions in DFH\$ZERO to a new group and edit the values. Do not change the attributes on the PROGRAM definitions.
8. Install the DFH\$ZERO group, or your own copy, in the CICS region.

## 6.2 Customizing the default configuration file

Dynamic scripting applications use configuration files to control the system settings and behavior of an application. Before you can run any applications, you must customize the global configuration file.

The configuration of a dynamic scripting application is set in two configuration files:

- ▶ The `zerocics.config` global configuration file defines how the feature pack interacts with a CICS region and contains properties that control common settings across more than one application; for example, resource attribute values, the location and heap size of the JVM, and the CICS region in which the applications run. You can override values for particular applications by copying the global configuration file into the application config directory and modifying the values.
- ▶ The `zero.config` configuration file contains properties that are specific to an application; for example, it contains lists of application extensions, map sets, and event handlers. Each application must have this configuration file in its config directory.

These configuration files are processed when you start the application. If a `zerocics.config` file exists in the config directory of the application, the values in this file override the default `zerocics.config` file.

The global configuration file is located in the `/install_directory/zero/config` directory.

Perform the following steps:

1. Open the `/install_directory/zero/config/zerocics.config` file to customize the default values for your CICS region. The configuration file is split into sections. You must edit each section to correctly configure the feature pack to work with CICS and Java. The file is in ASCII format, so use one of the methods described in 6.6, “Editing an ASCII file on an EBCDIC system” on page 85.
2. Set the options for the command-line interface. You use the command-line interface to stop and start applications and perform other operations. The interface uses EXCI to access CICS. Perform the following steps:
  - a. Set the `CICS_APPLID` parameter to the APPLID of the CICS region where you want to run an application. This CICS region becomes the default for running all applications, unless you override it with another configuration file.
  - b. Set the `CICS_NETNAME` parameter to the net name that is specified on the EXCI CONNECTION resource in the supplied DFH\$ZERO group. The default is ZEROCLI.
  - c. Set the `CICS_CLIUSER` to a user ID that you want to use to run all command-line interface commands. This user ID must have authority to log on to DFHIRP, connect to the target region, and create CICS resources. If you want to use the z/OS UNIX user ID instead, delete this line from the configuration file.
  - d. Set the `CICS_TRANSID` to the transaction ID that is the mirror for EXCI requests. If you delete this line, the ZERO transaction is used. The ZERO transaction is a copy of the CSMI transaction with command security enabled.
3. Set the options to dynamically create a JVM profile:
  - a. Set the `CICS_HOME` parameter to the path of the root directory for the CICS files on z/OS UNIX. This value is the same as the USSHOME system initialization parameter.
  - b. Set the `JAVA_DUMP_OPTS` parameter to obtain diagnostic information for an abend in the JVM. Information about Java dump options can be found in the *IBM Developer Kit and Runtime Environment, Java Technology Edition Diagnostics Guide*, which is available to download from the following address:

<http://www.ibm.com/developerworks/java/jdk/diagnosis/>

- c. Set the LIBPATH\_PREFIX parameter to include additional directories in the library path. If you want to use DB2 or another database, add the DB2 path to this parameter.
  - d. Set a maximum size for the JVM heap using the -Xmx parameter. The default value is 64M. You can specify other JVM options as appropriate.
4. Set the options that apply to all TCPIP SERVICE resources for HTTP and HTTPS requests. Use the syntax `attribute(value)`. These values are used to dynamically create a TCPIP SERVICE resource when you start an application. For the full list of attributes, see TCPIP SERVICE in Example 6-1. You cannot set the following attributes:
    - IPADDRESS
    - PORTNUMBER

You specify these values in the `zero.config` application configuration file.
  5. Set the options that apply to all URIMAP resource definitions for HTTP and HTTPS requests. If you set a value for the USERID attribute, this user ID is used to run all the inbound HTTP requests if no other user ID is present on the request. For a full list of attributes, see URIMAP attributes in Example 6-1. You cannot set the following attributes:
    - PIPELINE
    - SCHEME
    - TCPIP SERVICE
    - USAGE
  6. Set the options that apply to all PIPELINE resource definitions. For a full list of attributes, see PIPELINE attributes in Example 6-1. You cannot set a value for the CONFIGFILE attribute.
  7. Set the options that apply to all JVM SERVER resource definitions. For a full list of attributes, see JVM SERVER attributes in Example 6-1. You cannot set a value for the JVM PROFILE attribute.

Example 6-1 shows a global configuration file, `zerocics.config`, where the applications use HTTP and basic authentication.

*Example 6-1 Global configuration file*

---

```
[ZEROCLI]
# Options for CICS EXCI configuration. Update for your environment.
ICS_APPLID=EPRED5
CICS_NETNAME=ZEROCLI
CICS_CLIUSER=CICSR5
CICS_TRANSID=ZERO
```

```

[JVMPROFILE]
# Options of CICS JVM server configuration. The contents of this
section are copied to all JVMSERVER profiles. Update for your
environment.
CICS_HOME=/usr/lpp/cicsts/cicsts41
JAVA_DUMP_OPTS="ONANYSIGNAL(JAVADUMP,CEEDUMP,SYSDUMP),ONINTERRUPT(NONE)
"

LIBPATH_PREFIX=/usr/lpp/java/J6.0/bin/classic:/usr/lpp/java/J6.0/bin
-Xmx64M
-Xms128K
-Xss64K
-Xss256K

[TCPIPService-HTTP]
# Options to be applied to all TCPIPService resource definitions for
HTTP connections. Update for your environment
AUTHENTICATE(BASIC)
SOCKETCLOSE(10)
MAXDATALEN(524288)
BACKLOG(50)

[TCPIPService-HTTPS]
# Options to be applied to all TCPIPService resource definitions for
HTTPS connections. Update for your environment

[URIMAP-HTTP]
# Options to be applied to all URIMAP resource definitions for HTTP
connections. Update for your environment
TRANSACTION(ZPIH)
USERID(CICSR5)

[URIMAP-HTTPS]
# Options to be applied to all URIMAP resource definitions for HTTPS
connections. Update for your environment

[PIPELINE]
# Options to be applied to all PIPELINE resource definitions. Update
for your environment

[JVMSERVER]
# Options to be applied to all JVMSERVER resource definitions. Update
for your environment

```

---



## 6.3 Configuring CICS security

When you develop and run dynamic scripting applications in CICS, the commands and CICS resources might be subject to CICS security. You must configure CICS to provide the right levels of access.

The command-line interface uses EXCI to connect to CICS. It connects using the user ID that is set in the CICS\_CLIUSER parameter of the `zerocics.config` file. If you remove the parameter from the configuration file, the user ID that issues commands in z/OS UNIX is used instead. This user ID must have authority to log on CICS and run DFHIRP and connect to the target region.

The feature pack has CICS resources that are configured for security and it uses SPI commands to create CICS resources for applications. The user ID must also have access to create and access these resources.

Perform the following steps:

1. Ensure that the user ID that is set in the CICS\_CLIUSER parameter of the `zerocics.config` file has the correct authority for the DFHAPPL profile in the FACILITY class. The user ID requires UPDATE authority to DFHAPPL.cics\_cliuser and READ authority to DFHAPPL.cics\_applid. For example, you can update RACF by using the following command:

```
RDEFINE FACILITY DFHAPPL.cics_cliuser UACC(NONE)
PERMIT DFHAPPL.cics_cliuser CLASS(FACILITY) ID(cics_cliuser)
ACCESS(UPDATE)
RDEFINE FACILITY DFHAPPL.cics_applid UACC(NONE)
PERMIT DFHAPPL.cics_applid CLASS(FACILITY) ID(cics_cliuser)
ACCESS(READ)
```

2. If various users will be using the command-line interface, each user ID must have surrogate authority to run commands using the user ID that is set in the CICS\_CLIUSER parameter. If you remove this parameter from the configuration file, the current user ID that issues ZERO commands is used instead. For more information about surrogate authority, see Surrogate user security in the CICS information center.
3. If transaction security is active in CICS, the CICS\_CLIUSER user ID must have READ authority to access the feature pack transactions. The following example uses the default transaction-attach security classes TCICSTRN and GCICSTRN:

```
RDEFINE GCICSTRN (ZEROCLI) ADDMEM(ZERO, ZLIF, ZPIH) UACC(NONE)
PERMIT ZEROCLI CLASS(GCICSTRN) ID(cics_cliuser) ACCESS(READ)
```

For more information about transaction security, see Transaction security in the CICS information center.

4. If command security is active in CICS, the CICS\_CLIUSER user ID must have ALTER authority to dynamically create resources when the application starts. The following example uses the default command security classes CCICSCMD and VCICSCMD:

```
RDEFINE VCICSCMD (ZERO) ADDMEM(JVMSEVER, URIMAP, TCPIPSERVICE,  
PIPELINE) UACC(NONE)  
PERMIT ZERO CLASS(VCICSCMD) ID(cics_cliuser) ACCESS(ALTER)
```

For more information about CICS command security, see CICS command security in the CICS information center.

5. If program security is active in CICS, the CICS\_CLIUSER user ID must have READ authority to the feature pack programs. The following example uses the default program resource security classes MCICSPPT and NCICSPPT:

```
RDEFINE NCICSPPT (ZERO) ADDMEM(ZEROSERV, ZERODISC, ZEROQUEIE,  
ZEROINQJ, ZEROLIFE) UACC(NONE)  
PERMIT ZERO CLASS(NCICSPPT) ID(cics_cliuser) ACCESS(READ)
```

For more information about resource security, see Resource security.

6. If temporary storage security is active in CICS, the CICS\_CLIUSER user ID must have UPDATE authority to enable the feature pack to write to the ZEROTSM temporary storage queue. The following example uses the default temporary storage queue security classes SCICSTST and UCICSTST:

```
RDEFINE SCICSTST (ZEROTSM) UACC(NONE)  
PERMIT ZEROTSM CLASS(SCICSTST) ID(cics_cliuser) ACCESS(UPDATE)
```

7. If the XRES system initialization parameter is set to YES, the CICS\_CLIUSER user ID must have ALTER authority to create JVMSEVER resources that begin with a JC prefix. The following example uses the default XRES resource security classes RCICSRES and WCICSRES:

```
RDEFINE RCICSRES (JVMSEVER.JC*) UACC(NONE)  
PERMIT JVMSEVER.JC* CLASS(RCICSRES) ID(cics_cliuser) ACCESS(ALTER)
```

8. The CICS default user ID is used to handle HTTP requests if no other authentication is present. If you want to specify a different user ID on the USERID attribute of URIMAP resources, you must authorize the following access:

- The user ID specified in the URIMAP must have READ authority the ZPIH transaction. The following example uses the default transaction-attach security classes:

```
RDEFINE GCICSTRN (SMASHUSR) ADDMEM(ZPIH) UACC(NONE)  
PERMIT SMASHUSR CLASS(GCICSTRN) ID(cics_httpuser) ACCESS(READ)
```

- If program security is active, the user ID must have READ authority to the same programs as the CICS\_CLIUSER user ID.

9. Use the SETR RACLIST(class\_name) REFRESH command to update the RACF profiles.

After completing these steps, you have given the user IDs the correct authorities to use the command-line interface, create CICS resources in CICS, and process HTTP requests.

## 6.4 Test your first dynamic scripting application

In this section we create a simple application and run it to verify that the feature pack is installed correctly and the connection with CICS has no problem.

**Note:** Because the creation of the application requires connection to the IBM website repository, to solve the dependency and download additional packages for the first time, be sure to have the Internet connection ready for your z/OS. If reaching the external website in your environment is not possible, see 7.5, “Local and remote repositories” on page 175 to build a local repository before proceeding.

1. Go to the z/OS UNIX directory called apps that you created to contain your dynamic scripting applications.
2. Enter the command **zero create CICS\_Dynamic\_Scripting**. Do not use spaces when creating the application. The feature creates a directory for the application called CICS\_Dynamic\_Scripting. This directory contains a number of subdirectories, including a config directory. The config directory contains a simple zero.config file that sets a default port and a runtime mode for the application.
3. Enter the following command to change into the application directory:  

```
cd CICS_Dynamic_Scripting
```

4. Enter the command **zero start** to start the application. The feature starts the application and dynamically creates the CICS resources based on the values in the `zerocics.config` file.

If port 8080 is already used, you get a CWPZI8847E error message (Example 6-2).

---

*Example 6-2 Port conflict error message*

---

```
CWPZI8847E: Open of CICS TCPIP SERVICE TP000028 on port number 08080
failed. Previously installed resources will be discarded.
CWPZI8830E: Failure during install of CICS resources for JVM SERVER
JC000028 <28>.
CWPZI8403E: Bad response code from ZEROSERV.
fifoDir=</u/cicsrs5/cicsds/apps/CICS_Dynamic_Scripting/.zero/private
/cics>
workdir=</u/cicsrs5/cicsds/apps/CICS_Dynamic_Scripting/.zero/private
/cics> command=<4> response=<213>
CWPZT0601E: Error: Command start failed
CWPZI8503I: CICS unit of work rolled back after a CLI task returned
a non-zero return code.
```

---

You can change the port number in the application configuration file as follows (see Example 6-3):

`apps/CICS_Dynamic_Scripting/config/zero.config`

---

*Example 6-3 Changing the default port*

---

```
# HTTP port (default is 8080)
/config/http/port = 8090

# Runtime mode (default is "production")
/config/runtime/mode="development"
```

---

5. In your web browser, enter the following URL:

`http://machine_name:8090/`

The default home page for the application opens (Figure 6-1).

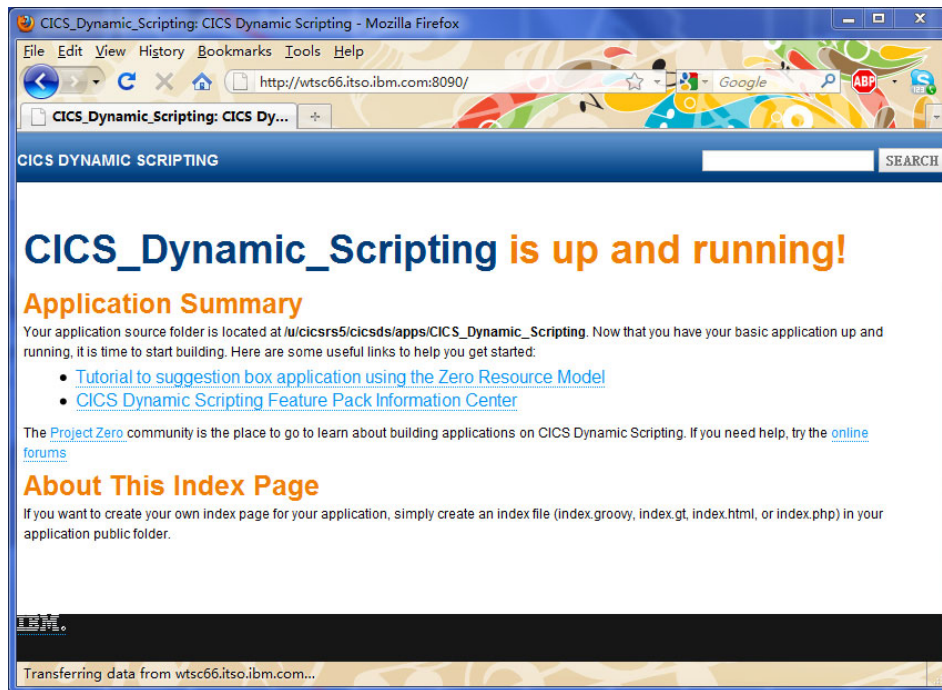


Figure 6-1 Welcome page of the sample application

## 6.5 Dynamically created resources and their naming conventions

After you start your application, the feature pack automatically creates resources in the CICS region to support the application. Those dynamic resources are not written into the CSD file but can survive during a recovery.

Resources that are created dynamically are named using a prefix corresponding to the resource type and purpose, followed by a six-digit ID that is obtained from the temporary storage queue ZEROJID. This ID ensures that the name of each resource is unique to the CICS region. Each command invocation creates and installs a JVM server that starts JC. The attributes of the JVM server are obtained from the JVMSERVER section of `zerocics.config`. The JVM server is automatically discarded when the command completes. See Table 6-1 on page 84.

Usually the following resources are defined for each application:

- ▶ TCIPSERVICE resource
- ▶ URIMAP resource
- ▶ PIPELINE resource
- ▶ JVMSERVER resource

*Table 6-1 Dynamically created resources for each application*

Resource name	Resource type	Description
TPxxxxxx	TCIPSERVICE	Installed if HTTP is enabled. The port and IP address are obtained from the zero.config file. The other attributes are obtained from the zerocics.config.
TSxxxxxx	TCIPSERVICE	Installed if HTTPS is enabled. The port and IP address are obtained from the zero.config file. The other attributes are obtained from the zerocics.config.
UPxxxxxx	URIMAP	Installed if HTTP is enabled. Associated with the TCIPSERVICE and PIPELINE resources. Other attributes are obtained from zerocics.config.
USxxxxxx	URIMAP	Installed if HTTPS is enabled. Associated with the TCIPSERVICE and PIPELINE resources. Other attributes are obtained from zerocics.config.
Plxxxxxx	PIPELINE	The shelf and configuration file are in application_home/.zero/private/cics. Uses the JVMSERVER as a terminal handler. Other attributes are obtained from zerocics.config.
JCxxxxxx	JVMSERVER	Attributes are obtained from zerocics.config. JVM options are obtained from zerocics.config and zero.config.

You can find the generated configuration files for pipelines in application\_home/.zero/private/cics and the profile for JVMSERVER in the directory specified in CICS System Initialization Parameter JVMPROFILEDIR.

If you enter zero stop in an application directory, the feature pack disables and discards all the resources installed by the zero start command. The feature pack also deletes the pipeline shelf and configuration file.

## 6.6 Editing an ASCII file on an EBCDIC system

All configuration files, log files, and scripts are encoded in ASCII and you must use an appropriate method to view and edit them:

- ▶ Use the z/OS UNIX directory list utility in ISPF (option 3.17). Enter the **ea** command to edit an ASCII file.
- ▶ Use the Remote Systems Explorer in Rational Developer for System z to edit an ASCII file in z/OS UNIX. The tool defaults to EBCDIC.
- ▶ Use the Target Management plug-in in Eclipse to modify an ASCII file. This tool defaults to ASCII.
- ▶ Enter the **iconv** command in z/OS UNIX to modify an ASCII file:
  - a. Convert the file from ASCII to EBCDIC:

```
iconv -f IS08859-1 -t IBM-1047 zero.config >zero.config.EBCDIC
```
  - b. Use the vi editor to view and modify the contents of the zero.config.EBCDIC file:

```
vi zero.config.EBCDIC
```
  - c. Convert the zero.config.EBCDIC file to ASCII and replace the original version of the zero.config file:

```
iconv -f IBM-1047 -t IS08859-1 zero.config.EBCDIC >zero.config
```
- ▶ Use FTP to transfer an ASCII file, in binary mode, to your workstation for editing. When you finish editing the file, transfer the file back in binary mode to the correct directory.
- ▶ Use file tagging to mark certain files as ASCII. The files are converted by z/OS when you access them. For more information about file tagging, see File tagging in the z/OS Information Center.
  - a. To convert tagged files automatically, add `export _BPXK_AUTOCVT=ON` to your login shell script.
  - b. To tag a file as ASCII, enter `chtag`. The following example tags the zero.config file as ASCII:

```
chtag -tc IS08859-1 zero.config
```

After you tag the file, you can access it using a z/OS UNIX tool such as vi. To list files and the associated tags, use the following command:

```
ls -T filename.
```







# Administration

This chapter covers administration topics such as application management, tuning JVMServer, common administrator commands, security, and code movement.

## 7.1 Application management

In a CICS Dynamic Scripting environment your dynamic scripting applications act in a server role, thus they must be started and available during the applications availability windows. As the number of dynamic scripting applications grow, you might find it helpful to have some tools or scripts that can assist with controlling your dynamic scripting applications.

In this section, we discuss how to automate the starting and stopping of your CICS Dynamic Scripting applications by creating scripts to issue ZERO commands to one or more CICS Dynamic Scripting applications. For instance suppose you want to start all CICS Dynamic Scripting applications that run in a specific CICS region or perhaps you want to display a status of all your CICS Dynamic Scripting applications running in every CICS region.

We present you with several solutions and ideas ranging from simple hardcoded shell scripts to more flexible solutions using variable substitution:

- ▶ Simple hardcoded shell scripts
- ▶ Variable or smart shell scripts
- ▶ Command-line interface extension applications

Building on the ideas listed, as the number and scope of your applications grow, you might want to generate scripts that perform actions on groups of applications that are related. The following list presents examples of how you can combine the scripts, grouping them to work on subsets of your applications:

- ▶ Start the HR Apps
- ▶ Stop the Payroll Apps
- ▶ Start All Apps
- ▶ Display a Status of all Apps running in Region EPRED4

Additionally you can perform these same functions with your automation platform or you could create some mixture of shell scripts and automations. For our samples we will not be using an automations tool.

### 7.1.1 Sample applications overview

For our first sample, Sample 1, we create several single-purpose shell scripts, one for each command we want to perform, that executes against several CICS Dynamic Scripting applications that are hard coded into the script.

The Sample 2 shell scripts use variables to allow the execution of any ZERO command against the applications.

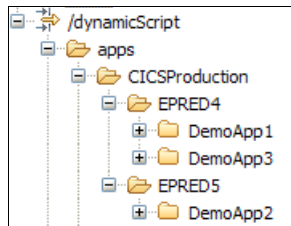
For Sample 3, we use Groovy scripts that use `zero.management.appconfig` APIs to control our demonstration applications.

The reason for several types of samples that perform the same or similar function is to give you ideas. There is no right solution, so choose any of these solutions or use your own ideas; these are simply several samples to show you what is possible and how they work.

To test our sample scripts, we need applications to test against. We have three applications, DemoApp1, DemoApp2 and DemoApp3 spread across two CICS regions, EPRED4 and EPRED5. Figure 7-1 shows the applications and their directory structure.

For our example, we assume these are production applications, therefore they are all located in a production directory and the applications are stored in a directory that is named after the CICS region where they are designated to run.

Figure 7-1 shows DemoApp1 and DemoApp3 both running in CICS region EPRED4 and DemoApp2 running in CICS region EPRED5. There are no rules of how you store your applications; this is simply an example, but the idea is to keep the applications organized and in some ways, self-documenting.



*Figure 7-1 Demo Applications Used to Test the Sample Control Scripts*

In our example, the shell scripts are designed so that they can be stored and run from any directory. It was set up this way so that someone running the scripts does not have to know what directories to change to in order to run the scripts, and that person does not have to know where all the resources are. However, when the scripts need to be updated, locating them all is more difficult for the programmer.

## 7.1.2 Sample 1: Hardcoded shell scripts

We create several simple shell scripts that use hardcoded values, both the commands to issue and the target applications to control, to manage multiple applications through one command. To do this task, we create multiple shell scripts that are almost identical, with the exception of the command being executed.

Naturally this sample presents the simplest solution but it does have the disadvantage of having to edit them when system changes or updates are made.

All the shell scripts in the Sample 1 group have the Dynamic Scripting environment *setup* script included in them as the first part of the script. Example 7-1 shows the contents of the setup script:

*Example 7-1 Contents of /dynamicScript/setup showing the environment variables*

---

```
export PATH=$PATH:/usr/lpp/java/J6.0/bin/
export ZERO_HOME=/dynamicScript/zero/
export PATH=$ZERO_HOME:$PATH
export STEPLIB=CICSTS41.CICS.SDFHEXCI:CICSTS41.CICS.SDFHLOAD
export JAVA_HOME=/usr/lpp/java/J6.0/
```

---

If you want to issue ZERO commands against an application, normally you run the setup script to set your CICS Dynamic Scripting environment variables, then you change to the applications directory and issue ZERO commands. With the scripts in Sample 1 and Sample 2, you do not need to do this step; the scripts can be placed in any directory and are able to access your target applications. Therefore, you can put all the scripts in your user ID's root directory and not have to know where the applications are.

### **Start.sh shell script**

This script issues ZERO START commands against the three demonstration applications as described in 7.1.1, "Sample applications overview" on page 88.

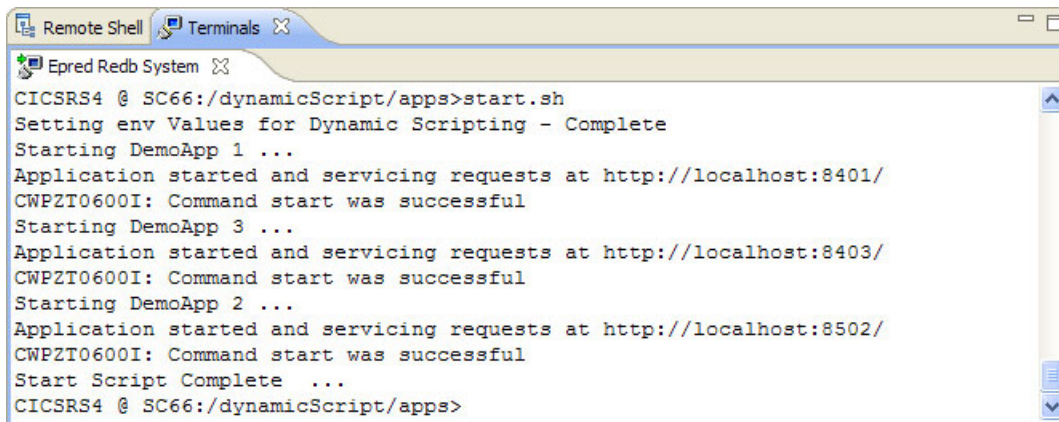
See the script in Example 7-2 on page 91. The first part of the script executes the setup script that sets the Dynamic Scripting environment variables.

Next, the script changes to each applications directory to issue the ZERO START command against that application.

*Example 7-2 Sample 1: start.sh shell script*

```
#####  
# Dynamic Scripting Redb Sample Shell Script #  
# START.SH - Issue ZERO START commands against preset Applications #  
#####  
  
./dynamicScript/setup  
echo "Setting env Values for Dynamic Scripting - Complete"  
  
echo "Starting DemoApp 1 ..."  
cd /dynamicScript/apps/CICSProduction/EPRED4/DemoApp1  
zero start  
echo "Starting DemoApp 3 ..."  
cd /dynamicScript/apps/CICSProduction/EPRED4/DemoApp3  
zero start  
echo "Starting DemoApp 2 ..."  
cd /dynamicScript/apps/CICSProduction/EPRED5/DemoApp2  
zero start  
echo "Start Script Complete ..."
```

Figure 7-2 shows the start.sh script being run; notice that the status messages from the script are intermixed with the output from each ZERO command as it is being executed.



```
Remote Shell  Terminals  
Epred Redb System  
CICSR4 @ SC66:/dynamicScript/apps>start.sh  
Setting env Values for Dynamic Scripting - Complete  
Starting DemoApp 1 ...  
Application started and servicing requests at http://localhost:8401/  
CWP2T0600I: Command start was successful  
Starting DemoApp 3 ...  
Application started and servicing requests at http://localhost:8403/  
CWP2T0600I: Command start was successful  
Starting DemoApp 2 ...  
Application started and servicing requests at http://localhost:8502/  
CWP2T0600I: Command start was successful  
Start Script Complete ...  
CICSR4 @ SC66:/dynamicScript/apps>
```

*Figure 7-2 Start.sh shell script execution output*

The START command causes the application to be loaded into CICS and several system resources to be dynamically created based on the values in the applications zerocics.config file. See 6.5, "Dynamically created resources and

their naming conventions” on page 83 for more information about the dynamically created resources.

Based on the output in Figure 7-2 on page 91, availability is as follows:

- ▶ DemoApp1 is now available through port 8401 from the EPRED4 region.
- ▶ DemoApp3 is now available through port 8403 also from the EPRED4 region.
- ▶ DemoApp2 is now available through port 8502 from the EPRED5 region.

## Stop.sh shell script

This script issues ZERO STOP commands against the three demonstration applications as described in 7.1.1, “Sample applications overview” on page 88.

See the script in Example 7-3. The first part of the script executes the setup script that sets the Dynamic Scripting environment variables.

Next, the script changes to each applications directory to issue the ZERO STOP command against that application.

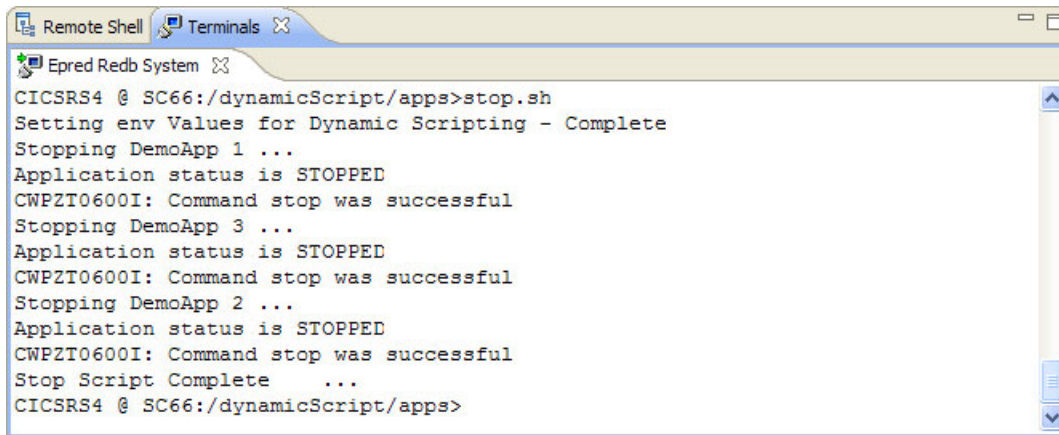
### *Example 7-3 Sample 1: stop.sh shell script*

---

```
#####  
# Dynamic Scripting Redb Sample Shell Script                                     #  
# STOP.SH - Issue ZERO STOP commands against preset Applications             #  
#####  
  
. /dynamicScript/setup  
echo "Setting env Values for Dynamic Scripting - Complete"  
  
echo "Stopping DemoApp 1 ..."  
cd /dynamicScript/apps/CICSProduction/EPRED4/DemoApp1  
zero stop  
echo "Stopping DemoApp 3 ..."  
cd /dynamicScript/apps/CICSProduction/EPRED4/DemoApp3  
zero stop  
echo "Stopping DemoApp 2 ..."  
cd /dynamicScript/apps/CICSProduction/EPRED5/DemoApp2  
zero stop  
echo "Stop Script Complete      ..."
```

---

Figure 7-3 shows the stop.sh script being run; notice that the status messages from the script are intermixed with the output from each ZERO command as it is being executed.



```
CICSR54 @ SC66:/dynamicScript/apps>stop.sh
Setting env Values for Dynamic Scripting - Complete
Stopping DemoApp 1 ...
Application status is STOPPED
CWPZT0600I: Command stop was successful
Stopping DemoApp 3 ...
Application status is STOPPED
CWPZT0600I: Command stop was successful
Stopping DemoApp 2 ...
Application status is STOPPED
CWPZT0600I: Command stop was successful
Stop Script Complete ...
CICSR54 @ SC66:/dynamicScript/apps>
```

Figure 7-3 STOP.SH shell script execution output

The STOP command causes the application to be terminated from CICS, and the dynamic resources that are created when the application was stopped are discarded. See section 6.5, “Dynamically created resources and their naming conventions” on page 83 for more information about the dynamically created resources.

### Status.sh shell script

This script issues ZERO STATUS commands against the three demonstration applications, as described in 7.1.1, “Sample applications overview” on page 88.

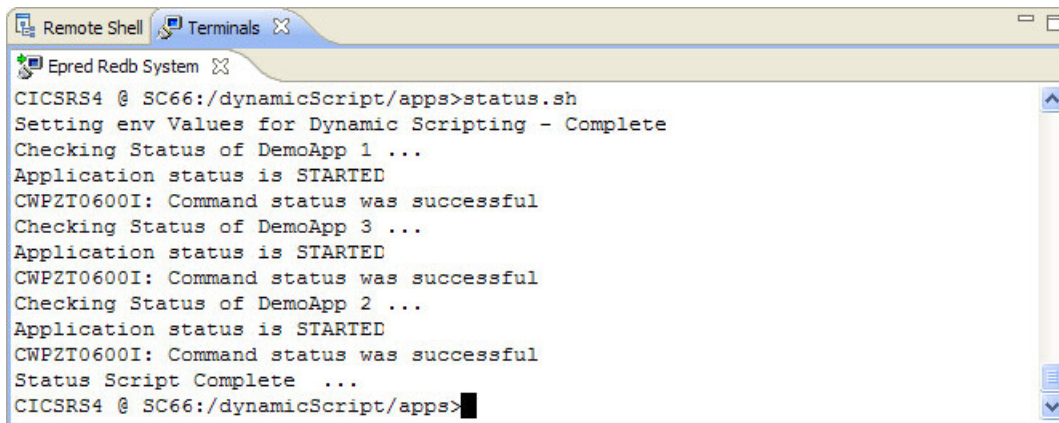
See the script in Example 7-4 on page 94. The first part of the script executes the setup script that sets the Dynamic Scripting environment variables.

Next, the script changes to each applications directory to issue the ZERO STATUS command against that application.

#### Example 7-4 Sample 1: status.sh shell script

```
#####  
# Dynamic Scripting Redb Sample Shell Script #  
# STATUS.SH - Issue ZERO STATUS commands against preset Applications #  
#####  
./dynamicScript/setup  
echo "Setting env Values for Dynamic Scripting - Complete"  
  
echo "Checking Status of DemoApp 1 ..."  
cd /dynamicScript/apps/CICSProduction/EPRED4/DemoApp1  
zero status  
echo "Checking Status of DemoApp 3 ..."  
cd /dynamicScript/apps/CICSProduction/EPRED4/DemoApp3  
zero status  
echo "Checking Status of DemoApp 2 ..."  
cd /dynamicScript/apps/CICSProduction/EPRED5/DemoApp2  
zero status  
echo "Status Script Complete ..."
```

Figure 7-4 shows the status.sh script being run; notice that the status messages from the script are intermixed with the output from each ZERO command as it is being executed.



```
Remote Shell  Terminals  
Epred Redb System  
CICSRs4 @ SC66:/dynamicScript/apps>status.sh  
Setting env Values for Dynamic Scripting - Complete  
Checking Status of DemoApp 1 ...  
Application status is STARTED  
CWPZT0600I: Command status was successful  
Checking Status of DemoApp 3 ...  
Application status is STARTED  
CWPZT0600I: Command status was successful  
Checking Status of DemoApp 2 ...  
Application status is STARTED  
CWPZT0600I: Command status was successful  
Status Script Complete ...  
CICSRs4 @ SC66:/dynamicScript/apps>
```

Figure 7-4 STATUS.SH shell script execution output

The STATUS command displays the current state of the application. Valid states are STOPPED, STARTED, and UNKNOWN. UNKNOWN can occur if the application has a running process associated with it but it cannot be reached using its base URL, as reported by the **info** command. The UNKNOWN



condition can also occur if the application's directory structure has been tampered with after the application has been started.

### The resolve.sh shell script

This script issues ZERO RESOLVE commands against the three demonstration applications as described in 7.1.1, "Sample applications overview" on page 88.

See the script in Example 7-5. The first part of the script executes the setup script that sets the Dynamic Scripting environment variables.

Next, the script changes to each applications directory to issue the ZERO RESOLVE command against that application.

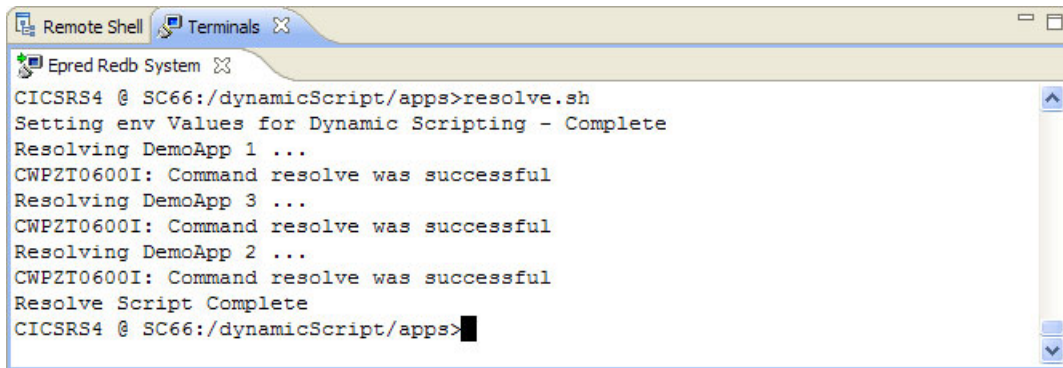
*Example 7-5 Sample 1: resolve.sh shell script*

---

```
#####  
# Dynamic Scripting Redb Sample Shell Script                                     #  
# RESOLVE.SH - Issue ZERO RESOLVE commands against preset Applications#  
#####  
  
./dynamicScript/setup  
echo "Setting env Values for Dynamic Scripting - Complete"  
  
echo "Resolving DemoApp 1 ..."  
cd /dynamicScript/apps/CICSProduction/EPRED4/DemoApp1  
zero resolve  
echo "Resolving DemoApp 3 ..."  
cd /dynamicScript/apps/CICSProduction/EPRED4/DemoApp3  
zero resolve  
echo "Resolving DemoApp 2 ..."  
cd /dynamicScript/apps/CICSProduction/EPRED5/DemoApp2  
zero resolve  
echo "Resolve Script Complete"
```

---

Figure 7-5 shows the `resolve.sh` script being run; notice that the status messages from the script are intermixed with the output from each ZERO command as it is being executed.



```
Remote Shell  Terminals X
Epred Redb System X
CICSR4 @ SC66:/dynamicScript/apps>resolve.sh
Setting env Values for Dynamic Scripting - Complete
Resolving DemoApp 1 ...
CWPZT0600I: Command resolve was successful
Resolving DemoApp 3 ...
CWPZT0600I: Command resolve was successful
Resolving DemoApp 2 ...
CWPZT0600I: Command resolve was successful
Resolve Script Complete
CICSR4 @ SC66:/dynamicScript/apps>
```

Figure 7-5 *RESOLVE.SH* shell script execution output

The RESOLVE command verifies that all application dependencies are available based on the applications `config/ivy.xml` file. The results of the resolve are in the `.zero/private/resolved.properties` file.

### Update.sh shell script

This script issues ZERO UPDATE commands against the three demonstration applications as described in 7.1.1, “Sample applications overview” on page 88.

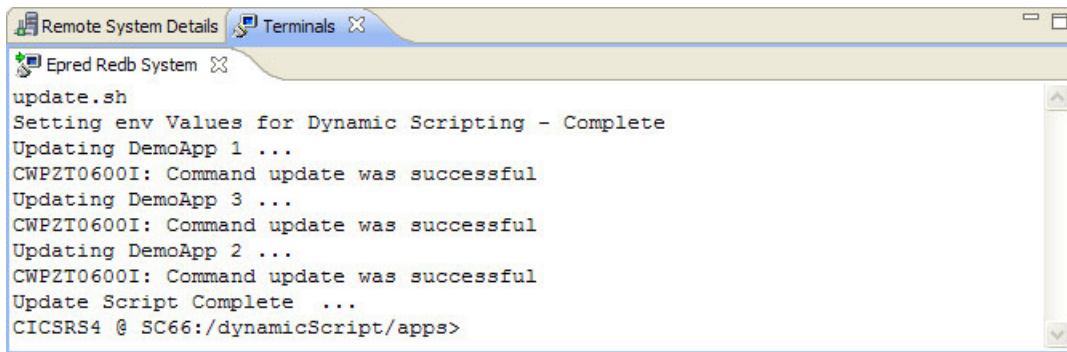
See the script in Example 7-6 on page 97. The first part of the script executes the setup script that sets the Dynamic Scripting environment variables.

Next the script changes into each applications directory to issue the ZERO UPDATE command against that application.

*Example 7-6 Sample 1: update.sh shell script*

```
#####  
# Dynamic Scripting Redb Sample Shell Script #  
# UPDATE.SH - Issue ZERO UPDATE commands against preset Applications #  
#####  
  
./dynamicScript/setup  
echo "Setting env Values for Dynamic Scripting - Complete"  
  
echo "Updating DemoApp 1 ..."  
cd /dynamicScript/apps/CICSProduction/EPRED4/DemoApp1  
zero update  
echo "Updating DemoApp 3 ..."  
cd /dynamicScript/apps/CICSProduction/EPRED4/DemoApp3  
zero update  
echo "Updating DemoApp 2 ..."  
cd /dynamicScript/apps/CICSProduction/EPRED5/DemoApp2  
zero update  
echo "Update Script Complete ..."
```

Figure 7-6 shows the `update.sh` script being run; notice that the status messages from the script are intermixed with the output from each ZERO command as it is being executed.



```
update.sh  
Setting env Values for Dynamic Scripting - Complete  
Updating DemoApp 1 ...  
CWPZT0600I: Command update was successful  
Updating DemoApp 3 ...  
CWPZT0600I: Command update was successful  
Updating DemoApp 2 ...  
CWPZT0600I: Command update was successful  
Update Script Complete ...  
CICSR54 @ SC66:/dynamicScript/apps>
```

*Figure 7-6 UPDATE.SH shell script execution output*

The UPDATE command performs a resolve, ignoring the locked revision levels, and resolves against all revisions that are available in the local repository.

### 7.1.3 Sample 2: Variable shell scripts

We continue with the theme from Sample 1 but combine our scripts into a single shell script that allows you to enter the ZERO command as a parameter and run it against multiple applications. This solution results in fewer scripts to manage.

Because we are building on Sample 1, we are still hard coding the applications that we want to act on, you may want to create several versions that work against various groupings of your applications.

The first script, DEMOAPP, is a shell script that acts on a group of applications, and for the second example, ALLAPPS, it works on every CICS Dynamic Scripting application in the apps directory. Therefore, many options are available; you could create a script for each CICS region and then create an ALLAPPS script that runs only each of the CICS Region scripts. For our example we have one application group script and one ALL script.

#### **DEMOAPP shell script**

The idea behind this script is that we have three related applications, DemoApp1, DemoApp2, and DemoApp3 that run in potentially separate CICS regions; because they are related, we want to start and stop them at the same time, thus the script name DEMOAPP. If perhaps you have several payroll applications and a couple of HR applications, you could make two scripts, one to control all the payroll applications and another to control your HR applications.

This script accepts a variable as input, the command to process, and then issues a ZERO command using that parameter as the command parameter, executing it against the three demonstration applications as described in 7.1.1, “Sample applications overview” on page 88.

Because DEMOAPP is accepting parameters, it performs a check to determine whether a valid parameter has been entered, and if not, displays the command syntax, as shown in Figure 7-7 on page 99. Note that stop it is not a valid parameter.

```

CICSR4 @ SC66:/dynamicScript/apps>demoapp.sh stopit
Invalid input, you typed in 'stopit'
Enter one of the following parameters:
-h | start | stop | status | update | resolve
Example: demoapp.sh stop
CICSR4 @ SC66:/dynamicScript/apps>

```

Figure 7-7 DEMOAPP.SH verifying the input parameters

Based on Figure 7-7, valid input to demoapp.sh are start, stop, status, update or resolve.

See the script in Example 7-7 The first part of the script is the usage() function, which displays the input syntax.

Next is the processCommand() function that issues the ZERO commands, so this section is pretty much identical to the scripts from Sample 1 with the exception that it uses variable input, but it still has hardcoded applications.

The next section is actually the beginning of the shell script. First, it verifies that one and only one parameter is entered. If not, it executes the usage() function to display the syntax message.

Finally, the last section verifies the parameter against a list of valid values, and if good, executes the processCommand() function or displays an error message along with the usage syntax.

#### Example 7-7 Sample 2: DemoApp shell script

```

#!/bin/ksh
#####
#
# Dynamic Scripting Redb Sample Shell Script
# DEMOAPP.SH - Issue ZERO commands against preset Applications
#
# Script to submit ZERO commands against multiple applications
# at the same time. This shell script has the location of each
# application hard coded into this shell script thus it can be
# run from any directory.
#
#####

```

```

typeset -u request

# Display Command Syntax

usage() {
    echo "Enter one of the following parameters:"
    echo "  -h | start | stop | status | update | resolve"
    echo "Example: demoapp.sh stop"
}

# Process ZERO Command against each application

processCommand() {

    echo "Zero Command to be processed: $request";
    . /dynamicScript/setup
    echo "Setting env Values for Dynamic Scripting - Complete"

    echo "$request DemoApp 1 ..."
    cd /dynamicScript/apps/CICSProduction/EPRED4/DemoApp1
    zero $1
    echo "$request DemoApp 3 ..."
    cd /dynamicScript/apps/CICSProduction/EPRED4/DemoApp3
    zero $1
    echo "$request DemoApp 2 ..."
    cd /dynamicScript/apps/CICSProduction/EPRED5/DemoApp2
    zero $1
    echo "DemoAPP $request Script Complete"
}

# Verify the parameter count is one

if test $# != 1 ; then
    echo "You entered $# parameters, only 1 parameter is allowed"
    usage
    exit 1
fi

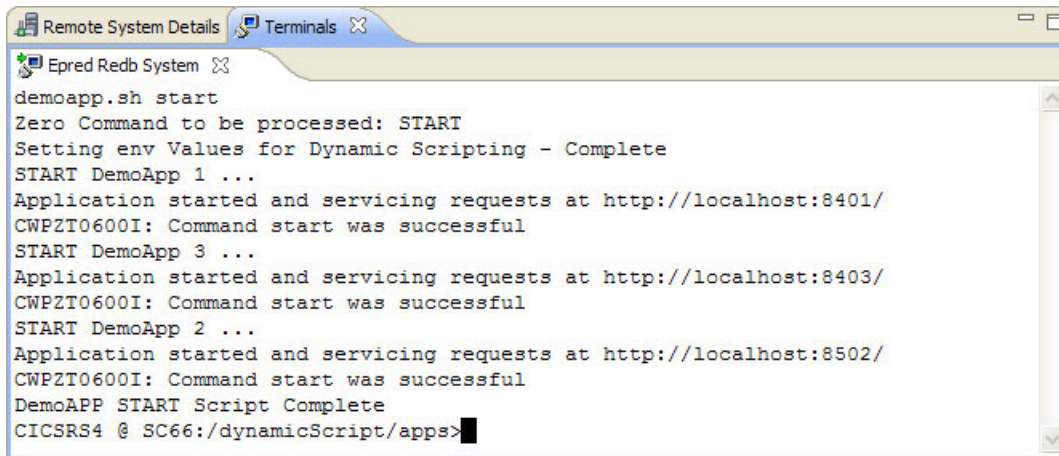
# Verify and Process the Input Command

request=$1
case $request in
    -H) usage; exit 1;;
    START|STOP|STATUS|UPDATE|RESOLVE) processCommand $1;;
    *) echo "Invalid input, you typed in '$1'"; usage; exit 1;;
esac

```

---

Figure 7-8 shows DEMOAPP.SH being run with START as input; notice that the status messages from the script are intermixed with the output from each ZERO command as it is being executed.



```
demoapp.sh start
Zero Command to be processed: START
Setting env Values for Dynamic Scripting - Complete
START DemoApp 1 ...
Application started and servicing requests at http://localhost:8401/
CWPZT0600I: Command start was successful
START DemoApp 3 ...
Application started and servicing requests at http://localhost:8403/
CWPZT0600I: Command start was successful
START DemoApp 2 ...
Application started and servicing requests at http://localhost:8502/
CWPZT0600I: Command start was successful
DemoAPP START Script Complete
CICSR4 @ SC66:/dynamicScript/apps>
```

Figure 7-8 DEMOAPP shell script execution output

The START command causes the application to be loaded into CICS and several system resources are dynamically created based on the values in the applications `zerocics.config` file. See section 6.5, “Dynamically created resources and their naming conventions” on page 83 for more information about the dynamically created resources.

Based on the output in Figure 7-8, availability is as follows:

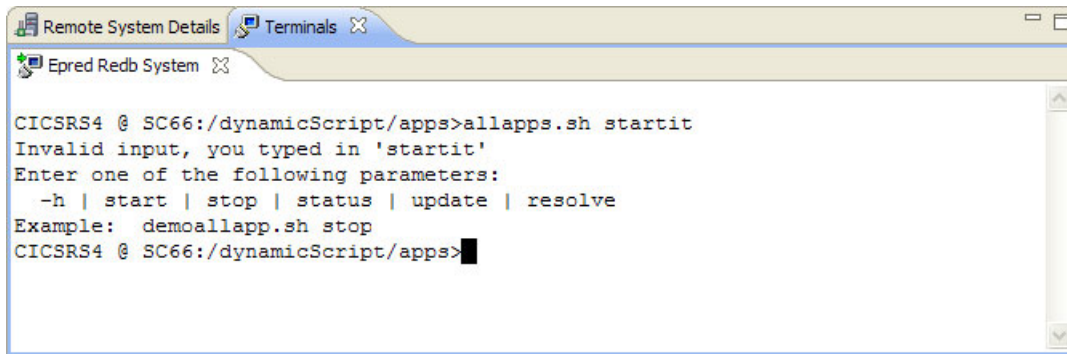
- ▶ DemoApp1 is now available through port 8401 from the EPRED4 region.
- ▶ DemoApp3 is now available through port 8403 also from the EPRED4 region.
- ▶ DemoApp2 is now available through port 8502 from the EPRED5 region.

## ALLAPPS shell script

This script is almost identical to DEMOAPP.SH with the difference being what applications it acts on. The purpose of DEMOAPP.SH is that it processes a group of related applications, and ALLAPPS.SH acts on every application in the main dynamic scripting apps directory. Other than that they are identical.

This script accepts a variable as input, the command to process, and then issues a ZERO command using that parameter as the command parameter, executing it against all CICS Dynamic Scripting applications found under the main apps directory; It also finds non-application directories, which generate failure messages. This, however, is not a problem; if you want, you may remove this part of the code. The idea here is to keep things simple.

Because ALLAPPS is accepting parameters, it performs a check to verify whether a valid parameter has been entered, and if not displays the command syntax, as shown in Figure 7-9. Note that startit is not a valid parameter.



```
Remote System Details  Terminals X
Epred Redb System X

CICSR54 @ SC66:/dynamicScript/apps>allapps.sh startit
Invalid input, you typed in 'startit'
Enter one of the following parameters:
-h | start | stop | status | update | resolve
Example: demoallapp.sh stop
CICSR54 @ SC66:/dynamicScript/apps>
```

Figure 7-9 ALLAPPS.SH verifying the input parameters

Based on Figure 7-9, valid input to ALLAPPS.SH are start, stop, status, update or resolve.

See the script in Example 7-8 on page 103.



```
#!/bin/ksh
#####
#
# Dynamic Scripting Redb Sample Shell Script
# ALLAPPS.SH - Issue ZERO commands against ALL Applications
#
# Script to execute ZERO commands against ALL applications in the
# ZERO app directory. This shell script change into each sub
# directory found issuing the ZERO command passed in as a
# parameter.
#
#####

typeset -u request

# Display Command Syntax

usage() {
    echo "Enter one of the following parameters:"
    echo "  -h | start | stop | status | update | resolve"
    echo "Example: demoallapp.sh stop"
}

# Process ZERO Command against each application sub directory

processCommand() {

    echo "Zero Command to be processed: $request";
    . /dynamicScript/setup
    echo "Setting env Values for Dynamic Scripting - Complete"

    for i in */;
    do
        echo "Processing ZERO $request for Sub-Directory: $i ..."
        cd "$i"
        zero $1
        cd ..
    done

}

# Verify the parameter count is one
```

```

if test $# != 1 ; then
    echo "You entered $# parameters, only 1 parameter is allowed"
    usage
    exit 1
fi

# Verify and Process the Input Command

request=$1
case $request in
    -H) usage; exit 1;;
    START|STOP|STATUS|UPDATE|RESOLVE) processCommand $1;;
    *) echo "Invalid input, you typed in '$1'"; usage; exit 1;;
esac

```

---

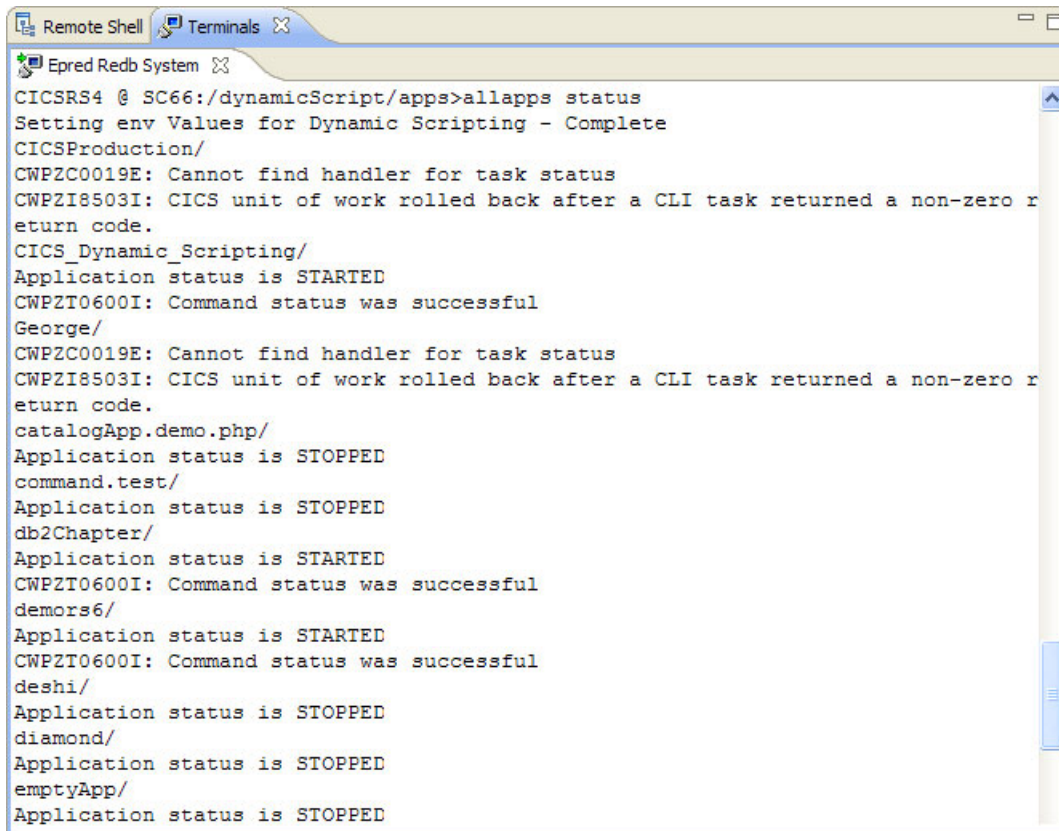
In the example, the first part of the script is the usage() function, which displays the input syntax.

Next is the processCommand() function that issues the ZERO commands, this is the difference between DEMOAPP.SH, instead of hard coding the location of each application, it will go through each directory under the apps directory and process the ZERO command. Several directories are not applications, so when the script finds them it issues the command, which generates an error message, and the script moves on. One other problem is that it assumes all applications are one directory deep, therefore any application that is more than one directory deep gets bypassed. See 7.1.1, “Sample applications overview” on page 88, where we talk about directory structure; each script might need tailoring to work with your environment setup.

The next section is actually the beginning of the shell script. First, it verifies that one and only one parameter is entered, if not it executes the usage() function to display the syntax message.

Finally, the last section verifies the parameter against a list of valid values, and if good, it executes the processCommand() function or displays an error message along with the usage syntax.

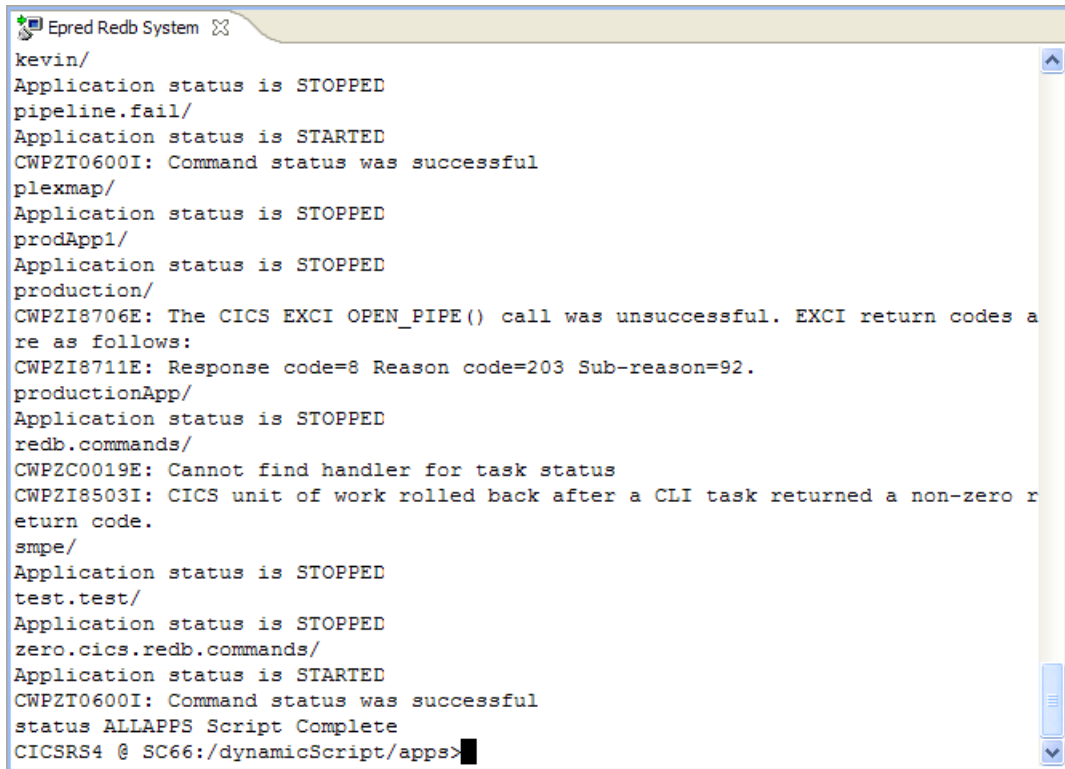
Figure 7-10 and Figure 7-11 on page 106 show ALLAPPS.SH being run with STATUS as input. Notice that the status messages from the script are intermixed with the output from each ZERO command as it is being executed. Also notice that when a directory that is not an application is found, an error message (Cannot find handler...) is displayed.



```
Remote Shell  Terminals X
Epred Redb System X
CICSR54 @ SC66:/dynamicScript/apps>allapps status
Setting env Values for Dynamic Scripting - Complete
CICSProduction/
CWPZC0019E: Cannot find handler for task status
CWPZI8503I: CICS unit of work rolled back after a CLI task returned a non-zero r
return code.
CICS_Dynamic_Scripting/
Application status is STARTED
CWPZT0600I: Command status was successful
George/
CWPZC0019E: Cannot find handler for task status
CWPZI8503I: CICS unit of work rolled back after a CLI task returned a non-zero r
return code.
catalogApp.demo.php/
Application status is STOPPED
command.test/
Application status is STOPPED
db2Chapter/
Application status is STARTED
CWPZT0600I: Command status was successful
demors6/
Application status is STARTED
CWPZT0600I: Command status was successful
deshi/
Application status is STOPPED
diamond/
Application status is STOPPED
emptyApp/
Application status is STOPPED
```

Figure 7-10 ALLAPPS shell script execution output part 1

The output is continued in Figure 7-11.



```
kevin/  
Application status is STOPPED  
pipeline.fail/  
Application status is STARTED  
CWPZT0600I: Command status was successful  
plexmap/  
Application status is STOPPED  
prodApp1/  
Application status is STOPPED  
production/  
CWPZI8706E: The CICS EXCI OPEN_PIPE() call was unsuccessful. EXCI return codes are as follows:  
CWPZI8711E: Response code=8 Reason code=203 Sub-reason=92.  
productionApp/  
Application status is STOPPED  
redb.commands/  
CWPZC0019E: Cannot find handler for task status  
CWPZI8503I: CICS unit of work rolled back after a CLI task returned a non-zero return code.  
smpe/  
Application status is STOPPED  
test.test/  
Application status is STOPPED  
zero.cics.redb.commands/  
Application status is STARTED  
CWPZT0600I: Command status was successful  
status ALLAPPS Script Complete  
CICSR4 @ SC66:/dynamicScript/apps>
```

Figure 7-11 ALLAPPS shell script execution output part 2

The STATUS command displays the current state of the application. Valid states are STOPPED, STARTED, and UNKNOWN. The UNKNOWN condition can occur if the application has a running process associated with it but it cannot be reached using its base URL, as reported by the INFO command. The UNKNOWN condition can also occur if the application's directory structure has been tampered with after the application has been started.

### 7.1.4 Sample 3: Command-line interface control applications

With CICS Dynamic Scripting, you can extend the command-line interface by writing your own command scripts and an API that includes the following management class that allows you to manage applications using methods such as start(), stop(), getStatus(), and so on:

```
zero.management.appconfig.Application
```

For Samples 1 and 2, we use z/OS UNIX shell scripts to issue ZERO commands to control target applications. In this section, we explore using a Groovy script as a command-line interface extension that uses the Application class to manage the CICS Dynamic Scripting applications in a target CICS region.

In Sample 3, the first script, APIAPP, again, uses hardcoded applications, similar to previous samples. The second script, APICFG, uses a configuration file to read a list of applications to be processed. Both scripts are CICS Dynamic Scripting applications and thus are running inside a CICS region. Therefore, we can only control other applications that are running inside the same region where our APIAPP and APICFG applications are running.

Using the Application class allows us to get a finer level of control over our applications. For instance with ZERO STOP, you can issue a STOP command and you also have a **-force** parameter that kills any remaining zero processes that related to the application. Using the Application management methods, you have a stop() method and you also have getStatus() and waitForStatusToEqual() methods for more detailed control. The Application class also provides two varieties of kill(), using a boolean immediate parameter you can kill the application immediately or have the system attempt a stop(); after maxStopTime() method, the stop() method transitions to a kill() method immediately.

You can find more detailed information about methods that are used by APIAPP and APICFG by either of the following ways:

- ▶ Search the web for `zero.management.appconfig`.
- ▶ Search for “API Documentation” at the IBM WebSphere sMash information center:

<http://publib.boulder.ibm.com/infocenter/wsmashin/v1r1m1/index.jsp>

In our previous samples, we allowed resolve and update parameters. With Sample 3, we are removing them and adding INFO, STOPKILL, and KILL parameters. Any of the sample scripts can issue any of the commands, so neither are restricted to a specific sample, they differ simply to show alternate ways of doing these tasks. Any script you build could incorporate elements from any or all of the sample scripts.

**Note:** As we progress from Sample 1 to Sample 3, more error checking and status information is being added.

## APIAPP Groovy CLI script

APIAPP is a CICS Dynamic Scripting application written in Groovy and is used to control other CICS Dynamic Scripting applications running in the same CICS

region, which for this example is EPRED4. Figure 7-1 on page 89 identified the layout of our “example” CICS Production environment, under the EPRED4 CICS region are the DemoApp1 and DemoApp3 applications, which are the targets of APIAPP.

APIAPP.groovy is known as an extension to the command-line interface and is located in the app/tasks directory under the APIAPP directory, as shown in Figure 7-12.

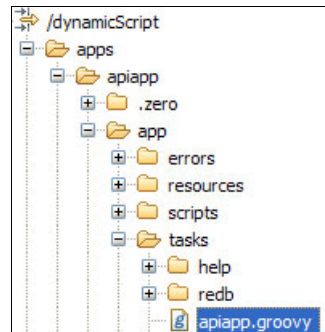


Figure 7-12 Location of APIAPP

APIAPP accepts a parameter as input, the command to process, and then issues the appropriate Application methods, executing them against the two demonstration applications running in EPRED4, as described in 7.1.1, “Sample applications overview” on page 88.

Figure 7-13 shows APIAPP being executed from its application directory. It performs a check to verify whether a valid parameter has been entered, and if not, it displays the command syntax, as shown. Because badinput is not one of the valid parameters APIAPP accepts, it is rejected and the command syntax is displayed.

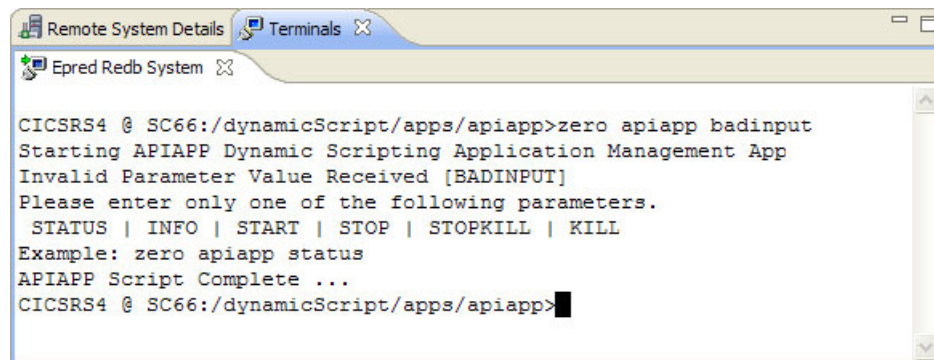


Figure 7-13 APIAPP verifying the input parameters

APIAPP accepts one of the following valid input parameters: Status, Info, Start, Stop, StopKill, or Kill. All are Application class methods except for StopKill, which is the kill() method with a false value for the immediate parameter.

See the script in Example 7-9 on page 111. The first part of the script is several import commands to ensure our script has access to the various classes we use:

- ▶ The first one, com.ibm.cics.server.Region gives the ability to get the runtime region APPLID
- ▶ The second one, zero.management.appconfig.\*, gives us access to the Application classes that are used to manage the target applications (using methods such as stop(), start(), kill(), and so on).

Next, is our Groovy script's implementation of the onCliTask method that is invoked when a ZERO APIAPP CLI command is entered. You can create command-line commands by registering a handler for the cliTask event. With Groovy the way you do this task is simply to place a Groovy script, which includes a definition for onCliTask, into the applications app/tasks directory. The location of APIAPP.groovy is shown in Figure 7-12 on page 108.

Next is the *try* block, which is the top of our code, first up is the validParmCount() function, which is used to verify that one and only one parameter is entered; if not an error message and the command syntax are displayed.

Next, the validParmValue() function checks the command parameter that is entered. Similar to validParmCount(), if one of the valid parameters is not entered, an error message and the command syntax are displayed.

Next is validParameterReceived(), which is the driver routine for both the previous functions. It converts the input parameter to uppercase and then checks for both, validParmCount() and validParmValue().

The next section is the beginning of the Groovy script. First, it gets the parameter entered by using the zget() method, which is part of the Global content APIs, and then wrappers the main processing within a call to validParameterReceived(), passing in the input parameter.

If the input parameter passes all the checks, the script defines the hardcoded applications and then processes the input parameter against each hardcoded application. To add or change the applications that APIAPP can process, edit an existing or add a new apphome variable definition pointing to the applications full path.

The getApplication() method is used to get an application representation of each hardcoded application to allow usage of the various Application class methods, such as getInfo(), getStatus(), getName(), and so on.

From this point on, the script has an *if* block for each possible input parameter that we allow as input:

- ▶ Status

This section is simple; all it does is output the results of `getName()` and `getStatus()`.

- ▶ Info

This section is even simpler, it outputs the results of `getInfo()`.

- ▶ Start

This section first checks to see if the application is already started, and if not issues a `start()` method and then waits until the status is either `STARTED` or if the timer pops, it issues an error message.

- ▶ Stop

This section first checks to see if the application is started, and if true will issue a `stop()` method and then wait until the status is either `STOPPED` or if the timer pops it issues an error message.

- ▶ StopKill

This section first sets the immediate boolean variable to false and then checks to see if the application is started; if true, it issues a `kill(immediate)` method and then waits until the status is either `STOPPED`, or if the timer pops, it issues an error message. By setting immediate to false, the system issues a `stop()` method and then after `maxStopTime()`, the command becomes a `kill()` command.

Figure 7-19 on page 126 shows the output of a Groovy Template named `testGet.gt` that displays the output from several various `getxxx()` methods in the `Application` class. With the `StopKill` command, you can see that `maxStopTime()` is set to 99 seconds for `DemoApp3`. The same section shows an example of how to change configuration settings such as `maxStopTime()`. See “Testing Various Application Class get Methods” on page 124.

- ▶ Kill

This section first sets the immediate boolean variable to true and then checks to see if the application is started, and if true, it issues a `kill(immediate)` method and then waits until the status is either `STOPPED`, or if the timer pops, it issues an error message. By setting immediate to true, the system issues a `kill()` method that kills any remaining zero processes that are related to the application.

After the `kill()` method, a message is displayed stating `APIAPP` is complete.

Example 7-9 on page 111 shows the complete source code for `APIAPP.groovy`.



### Example 7-9 Sample 3: APIAPP Groovy CLI script

---

```
import com.ibm.cics.server.Region;
import zero.management.appconfig.*;
import zero.cli.tasks.commands.*;
import zero.cli.tasks.CliException;

def onCliTask() {
    try {
        //
        // One and Only One parameter is allowed
        //
        def validParmCount = {
            List<String> parmIn ->
            switch (parmIn.size()) {
                case 0:
                    System.out.println("No Parameter Received.");
                    System.out.println("Please enter one of the following parameters.");
                    System.out.println(" STATUS | INFO | START | STOP | STOPKILL | KILL");
                    System.out.println("Example: zero apiapp status");
                    //throw new CliException("Incorrect number of input parameters.");
                    return false;
                    break;
                case 1:
                    return true;
                    break;
                default:
                    System.out.println("Invalid Parameter Count Received [" + parmIn.size() + "]");
                    System.out.println("Please enter Only One of the following parameters.");
                    System.out.println(" STATUS | INFO | START | STOP | STOPKILL | KILL");
                    System.out.println("Example: zero apiapp status");
            }
        }
        //
        // Validate the single input parameter to the valid input listing
        //
        def validParmValue = {
            String parmValue ->
            if (parmValue in ["STATUS", "INFO", "START", "STOP", "STOPKILL", "KILL"]) {
                return true;
            }
            else {
                System.out.println("Invalid Parameter Value Received [" + parmValue + "]");
                System.out.println("Please enter only one of the following parameters.");
                System.out.println(" STATUS | INFO | START | STOP | STOPKILL | KILL");
                System.out.println("Example: zero apiapp status");
                return false;
            }
        }
        //
        // Verify that the proper number and value of input parameters hsa been received
        // before continuing with processing the input command
        //
        def validParameterReceived = {
            List<String> parmIn ->
            if (validParmCount(parmIn)) {
                parmValue=parmIn.first().toUpperCase()
                if (validParmValue(parmValue)) {
                    return true;
                }
                else {
                    return false;
                }
            }
            else {
                return false;
            }
        }
    }
}
```

```

    }
}
//
// Main processing routine
//
System.out.println("Starting APIAPP Dynamic Scripting Application Management App");
List<String> args = zget("/event/args");
if (validParameterReceived(args)) {
    String apphome = zget("/config/root");
    String[] apphomes = new String[2];
    apphomes[0] = "/dynamicScript/apps/CICSProduction/EPRED4/DemoApp1";
    apphomes[1] = "/dynamicScript/apps/CICSProduction/EPRED4/DemoApp3";
    for(int i = 0; i < apphomes.size(); i++) {
        apphome = apphomes[i];
        File appHome = new File(apphome);
        Application myApp = Application.getApplication(appHome);
        int exitCode = zget("/config/exitCode");
        if (parmValue == 'STATUS') {
            System.out.println("Application " + myApp.getName() + " is " + myApp.getStatus());
        }
        if (parmValue == 'INFO') {
            System.out.println(myApp.getInfo());
        }
        if (parmValue == 'START') {
            if (!myApp.isAppStarted()) {
                myApp.start();
                cliResponse = myApp.waitForStatusToEqual(Application.Status.STARTED,10000);
                if (cliResponse) {
                    System.out.println("Application " + myApp.getName() + " is now " + myApp.getStatus());
                }
                else {
                    System.out.println("Application " + myApp.getName() + " Failed to START: " +
myApp.getStatus());
                }
            }
            else {
                System.out.println("Application " + myApp.getName() + " is already " + myApp.getStatus());
            }
        }
        if (parmValue == 'STOP') {
            if (myApp.isAppStarted()) {
                myApp.stop();
                cliResponse = myApp.waitForStatusToEqual(Application.Status.STOPPED,10000);
                if (cliResponse) {
                    System.out.println("Application " + myApp.getName() + " is now " + myApp.getStatus());
                }
                else {
                    System.out.println("Application " + myApp.getName() + " Failed to STOP: " +
myApp.getStatus());
                }
            }
            else {
                System.out.println("Application " + myApp.getName() + " is already " + myApp.getStatus());
            }
        }
        if (parmValue == 'STOPKILL') {
            boolean immediate = false;
            if (myApp.isAppStarted()) {
                myApp.kill(immediate);
                cliResponse = myApp.waitForStatusToEqual(Application.Status.STOPPED,65000);
                if (cliResponse) {
                    System.out.println("Application " + myApp.getName() + " is now " + myApp.getStatus());
                }
                else {
                    System.out.println("Application " + myApp.getName() + " Failed to STOP: " +
myApp.getStatus());
                }
            }
        }
    }
}

```

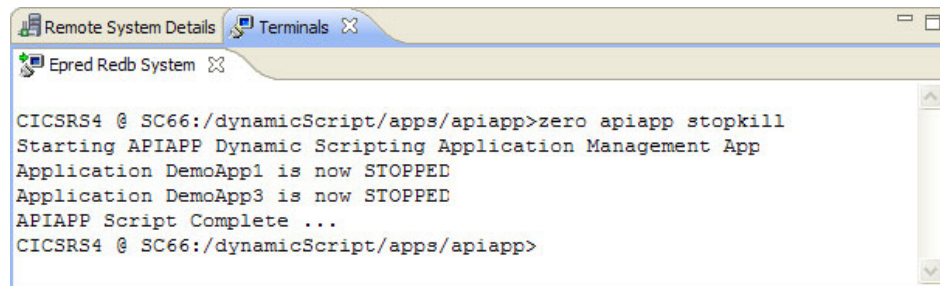
```

    }
    else {
        System.out.println("Application " + myApp.getName() + " is already " + myApp.getStatus());
    }
}
if (parmValue == 'KILL') {
    boolean immediate = true;
    if (myApp.isAppStarted()) {
        myApp.kill(immediate);
        cliResponse = myApp.waitForStatusToEqual(Application.Status.STOPPED, 65000);
        if (cliResponse) {
            System.out.println("Application " + myApp.getName() + " is now " + myApp.getStatus());
        }
        else {
            System.out.println("Application " + myApp.getName() + " Failed to STOP: " +
myApp.getStatus());
        }
    }
    else {
        System.out.println("Application " + myApp.getName() + " is already " + myApp.getStatus());
    }
}
}
}
}
System.out.println("APIAPP Script Complete ...");
}
catch(Exception e) {
    System.out.println(e);
}
}

```

---

Figure 7-8 on page 101 shows APIAPP being run with STOPKILL as input. Unlike the Sample 1 and Sample 2 shell scripts, you do not see any output from the ZERO commands being intermixed with the status messages, simply because APIAPP is controlling all commands.



```

CICSR54 @ SC66:/dynamicScript/apps/apiapp>zero apiapp stopkill
Starting APIAPP Dynamic Scripting Application Management App
Application DemoApp1 is now STOPPED
Application DemoApp3 is now STOPPED
APIAPP Script Complete ...
CICSR54 @ SC66:/dynamicScript/apps/apiapp>

```

Figure 7-14 APIAPP Groovy CLI script execution output

The STOPKILL command in this example results in the system issuing a STOP command; if the application fails to stop in 60 seconds a KILL command is issued.

## APICFG Groovy CLI script

APICFG is a CICS Dynamic Scripting application written in Groovy and is used to control other CICS Dynamic Scripting applications running in the same CICS region, which for this example is EPRED4. Figure 7-1 on page 89 identified the layout of our “example” CICS production environment, under the EPRED4 CICS region are the DemoApp1 and DemoApp3 applications, which are the targets of APICFG.

APICFG.groovy is known as an extension to the command-line interface (CLI) and is located in the app/tasks directory under the apicfg directory, as shown in Figure 7-15.

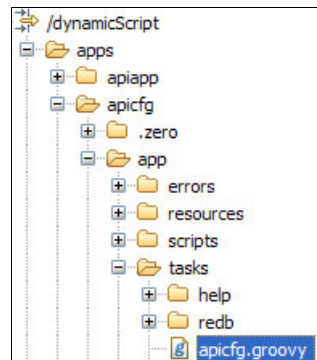
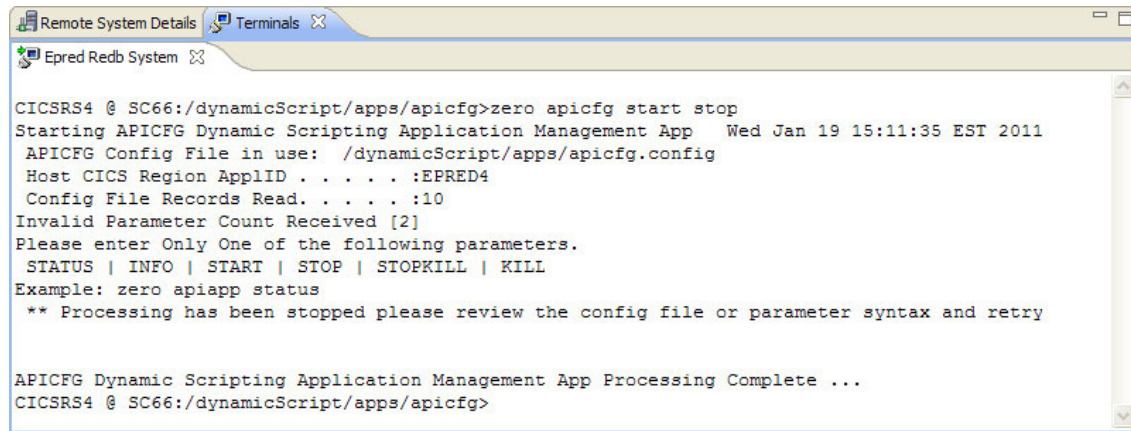


Figure 7-15 Location of APICFG

APICFG accepts a parameter as input, the command to process, and then issues the appropriate application methods, executing **apicfg** against the two demonstration applications running in EPRED4, as described in 7.1.1, “Sample applications overview” on page 88.

Figure 7-16 on page 115 shows APICFG being executed from its application directory. It performs a check to see if a valid parameter has been entered, and if not, displays the command syntax, as shown. APICFG allows only a single parameter to be processed, thus a check is made to ensure that there is always a single valid input parameter entered. In this example, `start stop` is invalid; both are valid commands but only one parameter is allowed per execution, thus it is rejected.



```
CICSRS4 @ SC66:/dynamicScript/apps/apicfg>zero apicfg start stop
Starting APICFG Dynamic Scripting Application Management App Wed Jan 19 15:11:35 EST 2011
APICFG Config File in use: /dynamicScript/apps/apicfg.config
Host CICS Region ApplID . . . . :EPRED4
Config File Records Read. . . . :10
Invalid Parameter Count Received [2]
Please enter Only One of the following parameters.
STATUS | INFO | START | STOP | STOPKILL | KILL
Example: zero apiapp status
** Processing has been stopped please review the config file or parameter syntax and retry

APICFG Dynamic Scripting Application Management App Processing Complete ...
CICSRS4 @ SC66:/dynamicScript/apps/apicfg>
```

Figure 7-16 APICFG verifying the input parameters

APICFG accepts only one of the following valid input parameters: Status, Info, Start, Stop, StopKill, or Kill. All are Application class methods except for StopKill, which is the kill() method with a false value for the immediate parameter.

The first part of the script, shown in Example 7-11 on page 118, has several import commands to ensure our script has access to the various classes we use:

- ▶ The first one, `com.ibm.cics.server.Region` gives the ability to get the runtime region APPLID.
- ▶ The second one, `zero.management.appconfig.*`, gives us access to the Application classes that are used to manage the target applications using methods such as `stop()`, `start()`, `kill()`, and so on.

Next, is our Groovy script's implementation of the `onCliTask` method that is invoked when a ZERO APICFG CLI command is entered. You can create command-line commands by registering a handler for the `cliTask` event. With Groovy the way you do this is simply to place a Groovy script, that includes a definition for `onCliTask`, into the applications `app/tasks` directory. The location of `APICFG.groovy` is shown in Figure 7-15 on page 114.

Next is the `try` block, which is the top of our code. First are several global variable definitions, and the code that gets the runtime CICS regions APPLID for use by several of the functions that follow.

Next, is the `validParmCount()` function, which is used to verify that one and only one parameter is entered, if not an error message and the command syntax is displayed.

Next, is the `validParmValue()` function that checks the command parameter that is entered, and similar to `validParmCount()`; if one of the valid parameters is not entered, an error message and the command syntax are displayed.

Next, is `validParameterReceived()`, which is the driver routine for both the previous functions, it converts the input parameter to uppercase and then checks for both, `validParmCount()` and `validParmValue()`.

APICFG is the first script that does not hard code its input pointing to the target applications. By providing application information in a simple configuration file, you can quickly change the flow of APICFG without touching the code and the same config file can be use by all your scripts.

**Note:** You could have used a Derby or DB2 database to store your target application information.

Function `validConfigFile()` is used to validate the contents of `apicfg.config`. First, it gets the current runtime region's Applid, which will be used to select records from the configuration file. It checks to make sure that the config file contains valid input that contain both the Applid and Application Path parameters that are required for processing. Example 7-10 shows the contents of the `apicfg.config` file.

*Example 7-10 APICFG input configuration file: apicfg.config*

---

```
#
# CICS Dynamic Scripting Control Application APICFG Configuration file
#
# Use the following spacing for each line, # of lines is iimited to 100
# 1 - 8=Target CICS Applid 9=space 10 > =Full Zero App path
#
EPRED4 /dynamicScript/apps/CICSProduction/EPRED4/DemoApp3
EPRED5 /dynamicScript/apps/CICSProduction/EPRED5/DemoApp2
EPRED4 /dynamicScript/apps/CICSProduction/EPRED4/DemoApp1
EPRED6 /dynamicScript/apps/CICSProduction/EPRED6/DemoApp8
#EPRED4 /dynamicScript/apps/CICSProduction/EPRED4/DemoApp4
EPRED9 /dynamicScript/apps/CICSProduction/EPRED9/DemoApp9
EPRED1 /dynamicScript/apps/CICSProduction/EPRED1/DemoApp10
EPRED2 /dynamicScript/apps/CICSProduction/EPRED2/DemoApp11
EPRED3 /dynamicScript/apps/CICSProduction/EPRED3/DemoApp12
```

---

The format of the `config` file is as follows:

- ▶ Columns 1 - 8 contain the target regions ApplID.
- ▶ Leave a space.
- ▶ Then, include the target application's full application path.

See the script in Example 7-11 on page 118. In our example, we show input for many various CICS regions and CICS Dynamic Scripting applications. You can use this `config` file to control many other functions, thus making it easier to control your environment.

The next section is the beginning of the main processing code. First, it gets the parameter entered by using the `zget()` method, which is part of the Global content APIs, and then sets access to the `apicfg.config` configuration file, checking to see if it exists, sets several variables and displays status information. It then wrappers the main processing within a call to `validParameterReceived()` and `validConfigFile()` passing in the input parameter and the contents of `apicfg.config` file respectively.

If the input parameter passes all the checks and if the `config` file passes all of its checks, the script processes the input parameter against each application that has a matching ApplID parameter.

The `getApplication()` method is used to get an application representation of each matching config file app, to allow usage of the various Application class methods, such as `getInfo()`, `getStatus()`, `getName()`, and so on.

From this point on, the script has an *if* block for each possible input parameter that we allow as input:

▶ Status

This section is simple code. Actually it does nothing because on the previous line of code, we get and save the application's current status and then at the bottom of the *if* blocks, we get the status again. Although it is empty, it was left in for flow and documentation purposes; basically each command gets its own if block, even if it does nothing.

▶ Info

This section gets and saves the results of `getInfo()`.

▶ Start

This section first checks to see if the application is already started, and if not issues a `start()` method and then waits until the status is either `STARTED` or if the timer pops, it stores an error indicator/counter.

► Stop

This section first checks to see if the application is started, and if true issues a stop() method and then waits until the status is either STOPPED or if the timer pops, it stores an error indicator/counter.

► StopKill

This section first sets the immediate boolean variable to false and then checks to see if the application is started, and if true, it issues a kill(immediate) method and then waits until the status is either STOPPED, or if the timer pops it stores an error indicator/counter. By setting immediate to false, the system issues a stop() method and then after maxStopTime(), the command becomes a kill() command.

Figure 7-19 on page 126 shows the output of a Groovy Template called testGet.gt that displays the output from a few of the various getxxx() methods in the Application class. With the StopKill command, maxStopTime() is set to 99 seconds for DemoApp3. The same section shows an example of how to change configuration settings like maxStopTime(). See “Testing Various Application Class get Methods” on page 124.

► Kill

This section first sets the immediate boolean variable to true and then checks to see if the application is started, and if true, it issues a kill(immediate) method, and then waits until the status is either STOPPED or, if the timer pops, it stores an error indicator/counter. By setting immediate to true, the system issues a kill() method that will kill any remaining zero processes related to the application.

After the kill() method, a final status is saved along with updating some counters for the report.

The final section uses all the saved status and counters to display a detailed report.

Example 7-11 shows the complete source code for APICFG.groovy:

*Example 7-11 Sample 3: APICFG Groovy CLI script*

---

```
import com.ibm.cics.server.Region;
import zero.management.appconfig.*;
import zero.cli.tasks.commands.*;
import zero.cli.tasks.CliException;

def onCliTask() {
    try {
        int cfgRecordErrors = 0;
        int inputErrors = 0;
        Region myRegion = new Region();
```



```

String myApplId = myRegion.getAPPLID();
//
// One and Only One parameter is allowed
//
def validParmCount = {
    List<String> parmIn ->
    switch (parmIn.size()) {
        case 0:
            System.out.println("No Parameter Received.");
            System.out.println("Please enter one of the following parameters.");
            System.out.println(" STATUS | INFO | START | STOP | STOPKILL | KILL");
            System.out.println("Example: zero apicfg status");
            //throw new CliException("Incorrect number of input parameters.");
            inputErrors++;
            return false;
            break;
        case 1:
            return true;
            break;
        default:
            System.out.println("Invalid Parameter Count Received [" + parmIn.size() + "]");
            System.out.println("Please enter Only One of the following parameters.");
            System.out.println(" STATUS | INFO | START | STOP | STOPKILL | KILL");
            System.out.println("Example: zero apicfg status");
            inputErrors++;
    }
}
//
// Validate the single input parameter against list of allowable input commands
//
def validParmValue = {
    String parmValue ->
    if (parmValue in ["STATUS", "INFO", "START", "STOP", "STOPKILL", "KILL"]) {
        return true;
    }
    else {
        System.out.println("Invalid Parameter Value Received [" + parmValue + "]");
        System.out.println("Please enter only one of the following parameters.");
        System.out.println(" STATUS | INFO | START | STOP | STOPKILL | KILL");
        System.out.println("Example: zero apicfg status");
        inputErrors++;
        return false;
    }
}
//
// Verify that the proper number and value of input parameters hsa
// been received before processing the input command
//
def validParameterReceived = {
    List<String> parmIn ->
    if (validParmCount(parmIn)) {
        parmValue=parmIn.first().toUpperCase()
        if (validParmValue(parmValue)) {

```

```

        return true;
    }
    else {
        return false;
    }
}
else {
    return false;
}
}
//
// Verify that the config file is valid and contains valid syntax free applications
// before attempting to process ZERO commands against the applications
//
def validConfigFile = {
    String[] lineArray ->
    boolean response = true;
    int lineCount = 0;
    lineCount = 0;
    for (line in lineArray) {
        lineCount++;
        if ( line.size() > 0 && ( ! line.substring(0,1).equals("#"))) {
            String[] cfgFileArray = line.split();
            if (cfgFileArray[0] == myApplid) {
                if (cfgFileArray.length < 2 ) {
                    System.out.println(" ** Config File Record Error1 ** -> Line: "+lineCount);
                    System.out.println(" - There must be at least two arguments on each non-comment
line. The first argument is the applid and the second is the command to be executed.");
                    System.out.println(" - No commands were executed, processing has been
stopped.");
                    cfgRecordErrors++;
                    response = false;
                }
                if (cfgFileArray[0].size() > 8) {
                    System.out.println(" ** Config File Record Error2 ** -> Line: "+lineCount);
                    System.out.println(" - The first argument is the CICS applid and must be 8
characters or less. You specified \""+cfgFileArray[0]);
                    System.out.println(" - No commands were executed, processing has been
stopped.");
                    cfgRecordErrors++;
                    response = false;
                }
                File theCmd = new File((String)cfgFileArray[1]);
                if ( ! theCmd.exists() ) {
                    System.out.println(" ** Config File Record Error3 ** -> Line: "+lineCount);
                    System.out.println(" - The specified command (the second argument) doesn't
exist. You specified \""+cfgFileArray[1]);
                    System.out.println(" - No commands were executed, processing has been
stopped.");
                    cfgRecordErrors++;
                    response = false;
                }
                if ( ! theCmd.canExecute() ) {
                    System.out.println(" ** Config File Record Error4 ** -> Line: "+lineCount);

```

```

        System.out.println(" - The specified command (the second argument) is not
executable. You specified \""+cfgFileArray[1]);
        System.out.println(" - No commands were executed, processing has been
stopped.");
        cfgRecordErrors++;
        response = false;
    }
}
}
}
return response;
}
//
// Main processing routine
//
def today= new Date();
System.out.println("Starting APICFG Dynamic Scripting Application Management App "+today);
List<String> args = zget("/event/args");
File cfgFile = new File("/dynamicScript/apps/apicfg.config");
if (cfgFile.exists()) {
    String[] lineArray = cfgFile.readlines().toArray();
    System.out.println(" APICFG Config File in use: /dynamicScript/apps/apicfg.config");
    System.out.println(" Host CICS Region ApplID . . . . :"+myApplid);
    System.out.println(" Config File Records Read. . . . :"+lineArray.size());
    def appName= new String[100];
    def oldStatus= new Application.Status[100];
    def newStatus= new Application.Status[100];
    def appInfo= new Object[100];
    int cfgRecordsProcessed = 0;
    int cfgRecordsMatched = 0;
    int cmdErrors = 0;
    if (validParameterReceived(args) && validConfigFile(lineArray)) {
        int lineCount = 0;
        for (line in lineArray) {
            String[] cfgFileArray = line.split();
            if (cfgFileArray[0] == myApplid) {
                File appHome = new File(cfgFileArray[1]);
                Application myApp = Application.getApplication(appHome);
                int exitCode = zget("/config/exitCode");
                oldStatus[cfgRecordsMatched]=myApp.getStatus();
                if (parmValue == 'STATUS') {
                }
                if (parmValue == 'INFO') {
                    appInfo[cfgRecordsMatched]=myApp.getInfo().toMapString();
                }
                else {
                    appInfo[cfgRecordsMatched]='No Info';
                }
            }
            if (parmValue == 'START') {
                if (!myApp.isAppStarted()) {
                    myApp.start();
                    cliResponse = myApp.waitForStatusToEqual(Application.Status.STARTED,10000);
                    if (cliResponse) {
                        cmdErrors++;
                    }
                }
            }
        }
    }
}

```

```

    }
    }
}
if (parmValue == 'STOP') {
    if (myApp.isAppStarted()) {
        myApp.stop();
        cliResponse = myApp.waitForStatusToEqual(Application.Status.STOPPED,10000);
        if (cliResponse) {
            cmdErrors++;
        }
    }
}
if (parmValue == 'STOPKILL') {
    boolean immediate = false;
    if (myApp.isAppStarted()) {
        myApp.kill(immediate);
        cliResponse = myApp.waitForStatusToEqual(Application.Status.STOPPED,65000);
        if (cliResponse) {
            cmdErrors++;
        }
    }
}
if (parmValue == 'KILL') {
    boolean immediate = true;
    if (myApp.isAppStarted()) {
        myApp.kill(immediate);
        cliResponse = myApp.waitForStatusToEqual(Application.Status.STOPPED,65000);
        if (cliResponse) {
            cmdErrors++;
        }
    }
}
appName[cfgRecordsMatched]=myApp.getName();
newStatus[cfgRecordsMatched]=myApp.getStatus();
cfgRecordsMatched++;
}
cfgRecordsProcessed++;
lineCount++;
}
}
if (inputErrors == 0) {
System.out.println(" Config File Record Errors . . . . :"+cfgRecordErrors);
System.out.println(" Config File Records Processed . . :"+cfgRecordsProcessed);
System.out.println(" Config File Records Matched . . . :"+cfgRecordsMatched);
System.out.println(" Commands Processed with Errors. . :"+cmdErrors);
System.out.println(" ");
System.out.println(" ");
}
if (cfgRecordErrors == 0 & inputErrors == 0) {
    System.out.println("CICS Dynamic Scripting Application Summary for CICS Region: "+myApplid);
    System.out.println(" ");
    System.out.println(" Application      Initial Status      ZERO Command      Final Status");
    System.out.println(" -----      -----      -----      -----");
    for (int i = 0; i < cfgRecordsMatched; i++) {

```

```

        System.out.println(" "+appName[i]+" "+oldStatus[i]+" "+parmValue+"
"+newStatus[i]);
        if (appInfo[i].length() > 7) {
            System.out.println(" "+appInfo[i]);
        }
    }
}
else {
    System.out.println(" ** Processing has been stopped please review the config file or
parameter syntax and retry");
}
System.out.println(" ");
System.out.println(" ");
System.out.println("APICFG Dynamic Scripting Application Management App Processing Complete
...");
}
else {
    System.out.println(" APICFG Config File NOT FOUND: /dynamicScript/apps/apicfg.config");
}
}
catch(Exception e) {
    System.out.println(e);
}
}
}

```

Figure 7-17 shows APICFG being run with START as input. Notice that the output from this sample looks vastly different from any of the previous scripts.

```

CICSRS4 @ SC66:/dynamicScript/apps/apicfg>zero apicfg start
Starting APICFG Dynamic Scripting Application Management App  Thu Jan 20 08:30:39 EST 2011
APICFG Config File in use:  /dynamicScript/apps/apicfg.config
Host CICS Region ApplID . . . . . :EPRED4
Config File Records Read. . . . . :15
Config File Record Errors . . . . . :0
Config File Records Processed . . :15
Config File Records Matched . . . :2
Commands Processed with Errors. . :2

CICS Dynamic Scripting Application Summary for CICS Region: EPRED4

Application  Initial Status  ZERO Command  Final Status
-----
DemoApp3     STOPPED          START         STARTED
DemoApp1     STOPPED          START         STARTED

APICFG Dynamic Scripting Application Management App Processing Complete ...
CICSRS4 @ SC66:/dynamicScript/apps/apicfg>

```

Figure 7-17 APICFG Groovy CLI script execution output: START

The START command causes the application to be loaded into CICS and several system resources dynamically created based on the values in the applications `zerocics.config` file. See 6.5, “Dynamically created resources and their naming conventions” on page 83 for more information about dynamically created resources.

Based on the output in Figure 7-17 on page 123, DemoApp3 and DemoApp1 were originally STOPPED and are now STARTED. The order of the applications in the listing has to do with how they are placed in the input configuration file, therefore if you want your applications to appear in a particular order, sort them in the configuration file.

Figure 7-18 shows APICFG being run with INFO as an input parameter.

```

CICSR54 @ SC66:/dynamicScript/apps/apicfg>zero apicfg info
Starting APICFG Dynamic Scripting Application Management App   Thu Jan 20 01:13:02 EST 2011
APICFG Config File in use:  /dynamicScript/apps/apicfg.config
Host CICS Region ApplID . . . . . :EPRED4
Config File Records Read. . . . . :15
Config File Record Errors . . . . . :0
Config File Records Processed . . :15
Config File Records Matched . . . :2
Commands Processed with Errors. . :0

CICS Dynamic Scripting Application Summary for CICS Region: EPRED4

Application   Initial Status   ZERO Command   Final Status
-----
DemoApp3      STARTED         INFO          STARTED
[apphome:/dynamicScript/apps/CICSProduction/EPRED4/DemoApp3, baseUrl:http://localhost:8403/, managed:true]
DemoApp1      STARTED         INFO          STARTED
[apphome:/dynamicScript/apps/CICSProduction/EPRED4/DemoApp1, baseUrl:http://localhost:8401/, managed:true]

APICFG Dynamic Scripting Application Management App Processing Complete ...
CICSR54 @ SC66:/dynamicScript/apps/apicfg>

```

Figure 7-18 APICFG Groovy CLI script execution output: INFO

The APICFG implementation of the INFO command displays the current status of the applications, both are STARTED, along with the directory path of the application, known as its *apphome*, and the port number and managed attribute.

## Testing Various Application Class get Methods

In Sample 3 applications, the `zero.management.appconfig.Application` class methods are used to control target applications. Example 7-12 on page 125 shows a short Groovy template named `testGet.gt` that shows how to use several of the commands.

```
<html>
  <%
    import com.ibm.cics.server.Region;
    import zero.management.appconfig.Application;
  %>
<head>
<body bgcolor='lightyellow'>
<title>testGET</title>
</head>
<body>
<h1>Testing various Application Class Get methods:</h1>
  <%
    String cfgAppPath =
"/dynamicScript/apps/CICSProduction/EPRED4/DemoApp3";
    File appHome = new File(cfgAppPath);
    Application myApp = Application.getApplication(appHome);
    println "<br/>getAppHome {" + myApp.getAppHome()+"}";
    println "<br/>getBaseUrl {" + myApp.getBaseUrl()+"}";
    println "<br/>getConfigFileList {" +
myApp.getConfigFileList()+"}";
    println "<br/>getContextRoot {" + myApp.getContextRoot()+"}";
    println "<br/>getHostname {" + myApp.getHostname()+"}";
    println "<br/>getHttpPort {" + myApp.getHttpPort()+"}";
    println "<br/>getIPAddress {" + myApp.getIPAddress()+"}";
    println "<br/>getLocalBaseUrl {" +
myApp.getLocalBaseUrl()+"}";
    println "<br/>getLocalIPAddress {" +
myApp.getLocalIPAddress()+"}";
    println "<br/>getMaxStopTime {" + myApp.getMaxStopTime()+"}";
    println "<br/>getNativesPath {" + myApp.getNativesPath()+"}";
  %>
</body>
</html>
```

---

Again, the target application is hard coded into the script; this sample displays information for the DemoApp3 application.

You can put testGet.gt in any application's public/ directory and execute it from a browser using that applications URL and Port, as shown in Figure 7-19 on page 126.

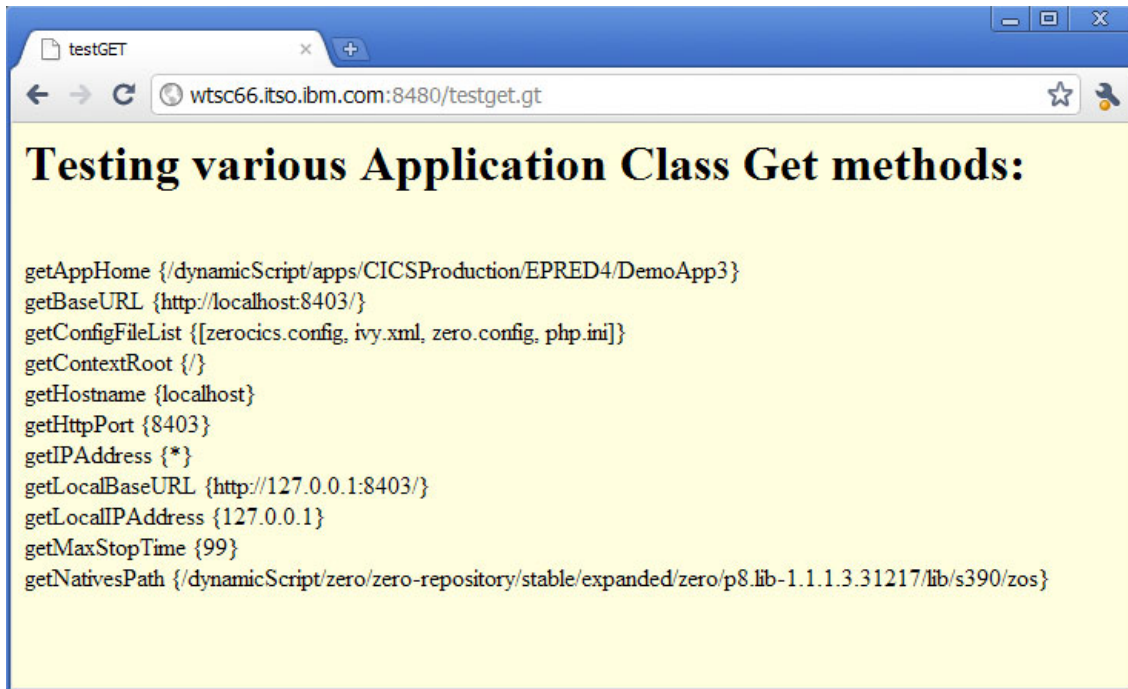


Figure 7-19 Display of various Application Class get methods

The value of maxStopTime is set to 99. If you want to change any of your application configurations settings, simply make an entry in your application's zero.config file. For example, to change the value of maxStopTime to 101 seconds, or any other value you want, you can add the following line in your application's zero.config file, as shown in Figure 7-20:

```
/config/maxStopTime = 101
```

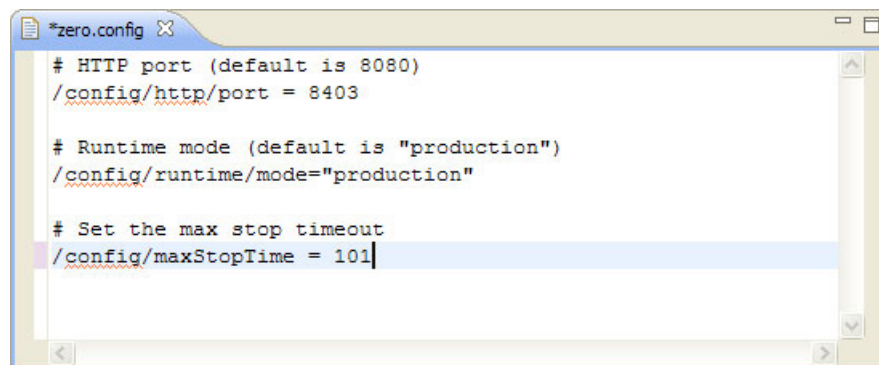


Figure 7-20 Changing configuration settings through zero.config



## 7.2 JVMServer tuning

In this section, we cover JVMServer tuning.

### 7.2.1 Use of the JVMServer with CICS Dynamic Scripting

A CICS Dynamic Scripting application runs in a multi-threaded Java Virtual Machine (JVM) provided by the JVMServer resource. The JVMServer resource was added CICS TS V4.1. The PHP and Groovy interpreters that are part of the CICS Dynamic Scripting Feature Pack, also written in Java, interpret your PHP and Groovy scripts into Java bytecode. This means that the bytecode generated by the PHP and Groovy interpreters is eligible for just-in-time (JIT) compilation so that the execution speed of your scripts can improve the more your application is used.

The level of Java required by the CICS Dynamic Scripting Feature Pack is Java 6 SR8 or later. The CICS Dynamic Scripting Feature Pack also requires APARs PM08649, PM08661, PM11157, and PM11791.

#### What is a JVMServer

JVM server is a Java environment running in CICS, capable of taking requests and running multiple CICS tasks as threads within a single JVM. The threads must retain the task-state and have the ability to call CICS services. JVMSERVER is introduced in CICS TS 4.1 and an architectural enhancement that is provided for CICS to perform system processing only. Dynamic Scripting is the first Feature Pack to use it. Currently it cannot be used for application code.

With CICS's legacy approach to JVMs where CICS maintains a pool of JVMs, although the JVM is multi-threaded, only one thread in the JVM is available to run application code. With the new JVMServer, from 1 to 256 threads can be used to run CICS-based Java application code. In this way, it solves the possible problem of short storage with JVMPOOL because of multiple threads. However, one JVMSERVER can support only one Dynamic scripting application on CICS. Therefore, in one CICS region the quantity of dynamic applications is still limited because each JVMServer takes a certain amount of storage.

A new type of task control block (TCB), a T8 TCB, is mapped to a JVM thread in a JVMServer to allow a Java application program to run and to interact with CICS. Your Java applications are strongly discouraged from creating threads in their code because the threads they create will not be mapped to T8 TCBs. Therefore, CICS is not aware of the threads and any code running in these threads cannot interact with CICS.

Figure 7-21 on page 128 shows the design of JVMServer in CICS as we described.

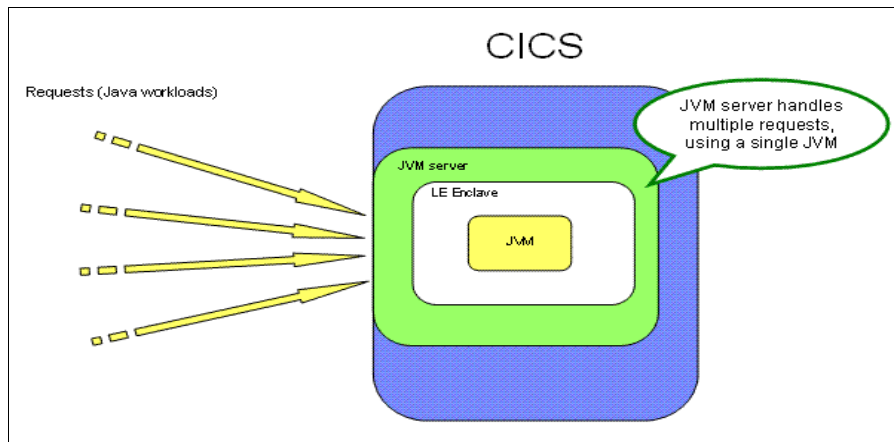


Figure 7-21 JVMServer design in CICS

## JVMServer Resource Definition

JVMServer resource must point to the location of the JVM profile, a thread limit and Language Environment run options for the Language Environment enclave.

- ▶ **LERUNOPTS:** Specify the program module that can create a Language Environment enclave and a JVM Server on a CICS region. The default module CICS that is provided is DFHAXRO.
- ▶ **THREADLIMIT:** Specify the maximum number of threads (T8 TCBs) that are allowed in the enclave for the JVMServer.

THREADLIMIT can be set from 1 to 256 threads (15 is the default). There can be a maximum of 1024 threads for a CICS region. The threads specified in the THREADLIMIT are for concurrent application usage. This means that a single JVMServer with a higher THREADLIMIT, depending on the request arrival rate, could be capable of handling multiple thousands of users. In addition to threads for application usage, the JVM allocates non-application threads for garbage collection and other functions. The number of JVMServers that can run in a CICS region are influenced by the available number of T8 TCBs per CICS region, and influenced by the size of the JVM that is implemented by the JVMServer resource and available memory in the CICS region.

The following statistics are available on JVMServer:

- ▶ Use count
- ▶ Maximum number of T8 TCB-related thread that can be attached to the JVMServer
- ▶ Current number of T8 TCBs attached to the JVMServer

- ▶ Peak number of T8 TCB-related threads that were attached to the JVMServer
- ▶ Number of tasks that waited for a free T8 TCB to attach a thread
- ▶ Amount of time tasks waited for a free T8 TCB to attach a thread
- ▶ Number of tasks that are currently waiting for a free T8 TCB
- ▶ Peak number of tasks that waited for a free T8 TCB to attach a thread

Most of the JVMServer statistics are viewable from the CICS Explorer™. The statistics can also be viewed using normal methods of displaying CICS statistics.

When using the CICS SPI (System Programmer Interface), the INQUIRE JVMPROFILE, and the CREATE, DISCARD, INQUIRE and SET JVMSERVER commands are subject to command security checking.

### JVMProfile file

JVMProfile is a zFS (UNIX System Services) file under a z/OS UNIX directory, which the CICS region can access. When CICS starts the JVM running with the characteristic specified in your JVMSERVER resource, CICS needs to know the JVM's characteristics. As with other JVMs in CICS, the characteristics of the JVMServer are specified in a JVMProfile file. The JVMProfile name is specified in the JVMServer resource definition with the file being located in the directory specified in the JVMPROFILEDIR= *System Initialization (SIT)* parameter. See Example 7-13. Be sure to turn off “caps” (capitalization) or else all will be in capital letters.

*Example 7-13 JVMServer file name*

---

```
JVMPROFILEDIR=/cics/ds/JVMProfiles
```

---

JVMProfile includes many settings. The most important settings are as follows:

- ▶ CICS\_HOME=/cics/cics660  
Specifies the path for the home directory for CICS files on z/OS UNIX. The value of this option is used to build the base library path and the base class path for the JVM. By default, this directory is /cics/cics660, which is defined by the USSDIR installation parameter when you installed CICS TS for z/OS, Version 4.1 with Dynamic Scripting Feature Pack.
- ▶ JAVA\_HOME=/java/java60\_bit31\_sr8/J6.0  
Specifies the install location for IBM SDK for z/OS, Java 2 Technology Edition in z/OS UNIX. This location contains subdirectories and JAR files required for Java support.
- ▶ LIBPATH\_PREFIX, LIBPATH\_SUFFIX=pathnames

Specifies directory paths to be searched for native C dynamic link library (DLL) files that are used by the JVM, which have the extension .so in z/OS UNIX, including those required to run the JVM and additional native libraries loaded by application code or services. The base library path for the JVM is built automatically using the directories specified by the CICS\_HOME and JAVA\_HOME options in the JVM profile. The base library path is not visible in the JVM profile. It includes all the DLL files required to run the JVM and the native libraries used by CICS.

You can extend the library path by using the LIBPATH\_SUFFIX option. This option adds directories to the end of the library path, after the base library path. Use this option to specify directories containing any additional native libraries that are used by your applications, or by any services that are not included in the standard JVM setup for CICS. For example, the additional native libraries might include the DLL files needed to use the DB2 JDBC drivers.

The LIBPATH\_PREFIX option adds directories to the beginning of the library path, before the base library path. Use this option with care, because, if DLL files in the specified directories have the same name as DLL files on the base library path, they are loaded in place of the CICS-supplied files. You might need to do this for problem determination purposes.

Any DLL files that you include on the library path for use by your applications must be compiled and linked with the XPLink option for optimum performance. The DLL files supplied on the base library path, and the DLL files used by services such as the DB2 JDBC drivers, are built with the XPLink option.

► CLASSPATH\_PREFIX, CLASSPATH\_SUFFIX=class\_pathnames

The standard class path specifies directory paths, JAR files, and ZIP files to be searched by the JVM for application classes and resources. You can specify entries on separate lines by using a backslash ( \ ) character at the end of each line that is to be continued. CLASSPATH\_PREFIX adds class path entries to the beginning of the standard class path, and CLASSPATH\_SUFFIX adds them to the end of the standard class path. With Version 6 of the SDK, all application classes are placed on the standard class path, and they are all eligible to be loaded into the shared class cache. CICS builds a base class path for the JVM using the /lib subdirectories of the directories specified by the CICS\_HOME and JAVA\_HOME options in the JVM profile. This base class path contains the JAR files supplied by CICS and by the JVM. It is not visible in the JVM profile. You do not specify these files again in the class paths in the JVM profile.

These options are the most important ones you have to know about JVMProfile. For other options, refer to CICS information center. In CICS Dynamic Scripting development, there is no need for us to create this profile by ourselves. What we

need to do is set up /zero/config/zerocics.config under the installation directory, and then the JVMProfile is generated automatically when we use **zero start** command to start our application.

Setting up zerocics.config, we only have to update two options as required in this file, which are CICS\_HOME and LIBPATH\_PREFIX, and other options can be kept as default. See Example 7-14.

*Example 7-14*

---

```
[JVMPROFILE]
CICS_HOME=/cics/cics660
LIBPATH_PREFIX=/java/java60_bit31_sr8/J6.0/bin/classic:/java/java60_bit
31_sr8/J6.0/bin
@ZEROCICS-JVMOPTIONS@
-Xmx64M
-Xms128K
-Xiss64K
-Xss256K
```

---

After successfully starting your CICS dynamic application with **zero start** command, you can check the JVMProfile directory specified in SIT option JVMPROFILEDIR. There, you find the JVM Profile named start with 'JC' characters. Open it to see the details such as JAVA\_HOME. At the same time, JVMServer resource is created automatically by CICS, you can logon the CICS region and input a command **CEMT I JVMS** to view the status of JVMServer as in Figure 7-22. It should be in enabled (Ena) status.

If the **zero start** command returns error message because JVMServer cannot be enabled, you might not have set the correct value for CICS\_HOME and LIBPATH\_PREFIX.

```
I JVMS
STATUS: RESULTS - OVERTYPE TO MODIFY
      Jvms(JC000002) Ena      Jvm(JC000002) Ler(DFHAXRO )
      Threadc(000) Threadl( 015 ) Installt(10/13/10 09:50:29)
```

*Figure 7-22 JVMServer status*

From member MSGUSR of the CICS log, you can see one JVMServer created, enabled and discarded, which is not the JVMServer you see on the terminal. This JVMServer is for ZERO commands, so when you input a command such **zero start**, **zero stop**, or any ZERO command (even simply typing zero each time), one JVMServer is created and enabled for the ZERO command to run, but is discarded as it exits.

## 7.2.2 How CICS Dynamic Scripting Feature Pack uses JVMServer

The CICS Dynamic Scripting Feature Pack is an implementation of the Project Zero technology. An overview of the Project Zero technology is described in Chapter 2, “Project Zero and WebSphere sMash features and capabilities overview” on page 11. A Project Zero application is created and administered by using a command-line interface (CLI). Both CLI commands and Project Zero applications run in a JVMServer.

### Interacting with the CICS-provided Project Zero Environment

As with all other Project Zero implementations, CICS provides a CLI for application creation and administration. The CICS-provided CLI communicates to an associated CICS region using EXCI.

The name of the CICS CONNECTION definition in the target CICS region is obtained from the installation `config/zerocics.config` file or from the Zero Application's `config/zerocics.config` file. A CONNECTION definition with the name specified in the `zerocics.config` file must be available in the target CICS region to establish communications.

Each ZERO command issued from the CLI, and each dynamic scripting application runs in its own JVMServer.

After a Zero Application starts running, HTTP requests are routed to the Zero Application running in the CICS environment using the following information:

- ▶ A TCPIPService definition to tell CICS to listen on a specified port,
- ▶ A URIMAP definition to CICS to direct HTTP requests to a specified PIPELINE,
- ▶ A PIPELINE resource whose pipeline configuration file specifies a handler that turns the HTTP request over to a JVMServer (which is running your Zero Application).

The TCPIPService, URIMAP, PIPELINE, and JVMSERVER resources, along with the pipeline configuration file and the JVMProfile file are dynamically created to run each CLI command and to run each Zero Application. When the CLI command is complete or the Zero Application is stopped, CICS dynamically removes these resources. Although CICS dynamically creates these resources, you can tell CICS the desired characteristics of these resources in the `zero.config` and the `zerocics.config` files.

The resource definitions dynamically created by CICS two letters and a six-digit number. The number is maintained in a TSQ named ZEROJID and is reset at each CICS cold start. The reason for this resource naming convention is

because multiple sets of CICS resources might exist at the same time (for various ZERO commands and Zero Applications), and each set must be unique.

The port that your application will listen on is taken from a `config/zero.config` file in your application's directory and placed in the dynamically created TCPIPService definition. The TCPIPService resource definition name starts with the letters TP or TS depending on whether or not you are using Secure Sockets Layer (SSL).

A URIMAP definition is created (which starts with either 'UP' or 'US' depending on whether you are using SSL or not), to direct all incoming HTTP activity from the port specified in the TCPIPService definition to your JVMServer through a PIPELINE definition.

The URIMAP definition does not point directly to your JVMServer, but points to a PIPELINE definition (starts with 'PI') whose corresponding pipeline configuration file specifies a handler that directs all activity to your JVMServer. The dynamically created pipeline configuration file used by the PIPELINE resource is placed in your application's `.zero/private/cics` directory. The pipeline configuration file contains the JVMSERVER ID and a handler. The name of the handler is `zero.cics.CicsAdapter`.

The characteristics of the JVM represented by the dynamically created JVMSERVER resource definition are taken from a dynamically created JVM Profile file. The JVMServer name starts with the 'JC' letters. The JVMProfile file that is dynamically created for you is placed in the directory specified by the `JVMPROFILEDIR=` specified in the SIT (system initialization) parameters.

All resource definitions and configuration files that are dynamically created are also dynamically removed when the JVMServer is shut down. The exception is that you can set an environment variable to tell CICS to leave some of the configuration files containing details about the actions that were taken. This environment variable is set before issuing the CLI command in question, and is `CICS_DEBUG=ON`. You can find more information about troubleshooting CLI commands in the CICS information center and also in Chapter 8, "Troubleshooting" on page 181.

### 7.2.3 Zeroing in on JVMServer usage

To affect the dynamically created resources used for CICS Dynamic Scripting, you can indicate values in the `zero.config` and `zerocics.config` files. For example, the port that your Zero Application will listen on (and that is placed in the generated TCPIPService definition) is specified in the `zero.config` file. To affect the JVMServer resource and its associated JVMProfile file, you must use

the `zerocics.config` file. Both the `zero.config` and `zerocics.config` files have ASCII encoding.

## The `zerocics.config` file

A `zerocics.config` file is used to tell CICS to use non-default values for dynamically created resources. Although most values can be set using the `zerocics.config` file, some values are set using the `zero.config` file. Because the JVMServer and its associated JVMProfile file are affected by the contents of the `zerocics.config` file, we will discuss this file here.

Before issuing any ZERO commands, environment variables must be set to indicate the location of the CICS Dynamic Scripting Feature Pack installation files, the location of the Java installation you are using, the PATH so that CLI commands can be located, and a STEPLIB to allow the CLI to use EXCI to communicate to a CICS region.

A `zerocics.config` file resides in the CICS Dynamic Scripting installation's config directory. The characteristics specified in this file are used for ZERO commands that are not used in an application context. An example of a CLI command that is not used in an application context is the **zero create** command. Because a `config/zerocics.config` file can contain only one `CICS_APPLID=`, and `CICS_NETNAME=` parameter, all ZERO commands that do not run in an application context go to the same CICS region.

A Zero Application can optionally contain its own `zerocics.config` file in the application's config directory. When CLI commands are run from an application context, if the application's `config/zerocics.config` file is present, it is used, otherwise the installation's `config/zerocics.config` file is used. An example of running a CLI command in an application context would be to change directory to your Zero Application's home directory, and entering `zero start`. You will want to have a `zerocics.config` file in your Zero Application's home directory if, for example, you want to run your Zero Application in a CICS region other than the region pointed to by your installation's `config/zerocics.config` file. Likewise, if you want to supply JVM tuning information that is specific to a particular Zero Application, you must place a `zerocics.config` file in your application's config directory.

## The `zerocics.config` file composition

The `zerocics.config` file contains sections where you can place information to override defaults used when CICS dynamically creates resources for running ZERO commands and to run your Zero Application. The '[ZEROCLI]' section is used to specify the APPLID, netname, user ID, and TRANID when using CLI commands. Two sections affect the TCIPSERVICE definition: one when using HTTP and the other when using HTTPS. Likewise, two sections affect the



dynamically created URIMAP definition: one for HTTP and the other when using HTTPS. There is also a section to affect the dynamically created PIPELINE definition. Of most interest in this section are those sections that affect the JVMServer: '[JVMSEVER]' and '[JVMPROFILE]' sections. In general, you will want to start with the defaults and change them based on your Zero Application's operation characteristics.

### ***Setting the '[JVMSEVER]'* characteristics**

You can customize the Language Environment enclave and maximum number of threads for your JVMSEVER in `zerocics.config` file.

By default, the '[JVMSEVER]' section of the `zerocics.config` file contains only several comments, which means that the dynamically created JVMServer resource contains all default characteristics. The default maximum number of threads that will be associated with T8 TCBs is 15. This value is the maximum number of concurrent requests your Zero Application can handle at one time. Although this is an acceptable number to start with, you will want to look at your JVMServer's thread usage to determine the best `THREADLIMIT` for your Zero Application. Zero Application usage patterns vary greatly so thread limits for Zero Applications also vary greatly vary.

The default value of `LERUNOPTS` is `DFHAXRO`, which is the program module to create Language Environment enclave and a JVMServer. You can change it with your customized module as you like.

Example 7-15 shows that changes can be made as you need.

*Example 7-15 LERUNOPTS sample*

---

```
[JVMSEVER]
LERUNOPTS(DFHAXRO)
THREADLIMIT(15)
```

---

### ***Setting the '[JVMPROFILE]'* characteristics**

The JVMProfile file is dynamically created for you and placed in the directory specified in the `JVMPROFILEDIR` parameter in the SIT (system initialization) parameters. Its name is generated and is `JCnnnnnn` (where `nnnnnn` is the six-digit ascending number discussed previously).

The contents of JVMProfile file are built from the following information:

- ▶ Command line generated by zero script's class path and Java command-line options)
- ▶ Information in `zerocics.config` '[JVMPROFILE]' section
- ▶ The zero application's resolved.properties file (`LIBPATH`)

- ▶ Dynamically created Java system properties defined for use by Java code (z/OS UNIX code page, ZERO command arguments)
- ▶ WORK\_DIR is set to your Zero Application's home directory

As indicated, you can affect the contents of the generated JVMPROFILE file by specifying values in the zerocics.config file. If the values you want to set are specific to your Zero Application, you should specify the values in your Zero Application's config/zerocics.config file.

Values that you might be interested in setting, for example, are Java dump options or JVM options. JVM options include sizing options, so as your Zero Application usage increases and you want to change parameters that pertain to garbage collection, this section will be of interest to you. Likewise, if you want to set interactive application debugging or profiling parameters, this section will be of interest to you.

### **The stdout and stderr files**

Your Zero Application's standard out and standard error files are placed in your Zero Application's home directory and have the following names (where nnnnnn is the six-digit ascending number discussed previously):

```
<cics_region_applid>.JCnnnnnn.dfhjvmout
<cics_region_applid>.JCnnnnnn.dfhjvmerr
```

## **7.2.4 Share Class Cache**

Class data sharing in IBM Java 6, on all of the platforms on which the IBM implementation of the JVM ships, allows all system and application classes to be stored in a persistent dynamic class cache in shared memory. Sharing reduces the overall virtual storage consumption when more than one JVM shares a cache. Sharing also reduces the startup time for a JVM after the cache has been created. The shared class cache is independent of any running JVM and persists until it is destroyed or an IPL of the z/OS image occurs.

Using shared class cache is the default for the JVMServer. Using JIT classes are not cached and are unique so a specific JVM. For more information about shared class cache, and how to display and control it, see the IBM Java information center and the various developerWorks articles detailing shared class cache.

## **7.2.5 Just-in-time compiler**

Your PHP and Groovy scripts are interpreted into Java byte code and are therefore eligible for just-in-time compilation.

The just-in-time compiler (JIT) is enabled by default, and is activated when a Java method is called. The JIT compiler compiles the bytecodes of a given method using various optimizations, compiling it “just in time” to run. When a method has been compiled, the JVM calls the compiled code of that method directly instead of interpreting the method's bytecode. Theoretically, if compilation did not require processor time and memory usage, compiling every method could allow the speed of the Java program to approach that of a native application.

The following levels of JIT compilation are applicable to all recent IBM JVMs:

- ▶ No Optimizations, byte code is interpreted
- ▶ Cold
- ▶ Warm
- ▶ Hot
- ▶ Veryhot
- ▶ Scorching

JIT compilation does require processor time and memory usage. When the JVM first starts, thousands of methods are invoked. Compiling all of these methods can significantly affect startup time, even if the program eventually achieves good peak performance.

In practice, methods are not compiled the first time they are called. For each method, the JVM maintains a call count, which is incremented every time the method is called. The JVM interprets a method until its call count exceeds a JIT compilation threshold. Therefore, often-used methods are compiled soon after the JVM has started, and less-used methods are compiled much later, or not at all. The JIT compilation threshold helps the JVM start quickly and still have improved performance. The threshold has been carefully selected to obtain an optimal balance between startup times and long term performance.

After a method is compiled, its call count is reset to zero and subsequent calls to the method continue to increment its count. When the call count of a method reaches a JIT recompilation threshold, the JIT compiler compiles it a second time, applying a larger selection of optimizations than on the previous compilation. This process is repeated until the maximum optimization level is reached. The busiest methods of a Java program are always optimized most aggressively, maximizing the performance benefits of using the JIT compiler. The JIT compiler can also measure operational data at run time, and use that data to improve the quality of further recompilations.

The JIT compiler can be disabled, in which case the entire Java program will be interpreted. Disabling the JIT compiler is not recommended except to diagnose or work around JIT compilation problems (which in today's Java environment, rarely occur).

There has been much research on JIT compilation and the optimizations used by the JIT compiler so using the default JIT mechanism is almost always the best choice. However, certain JVM parameters allow you to affect the behavior of JIT processing.

CICS Dynamic Scripting specifies `-Xquickstart` as a JVM option. This option causes the JIT compiler to use a lower optimization level by default to compile fewer methods, which can improve application startup.

## 7.2.6 Garbage collection

In the object-oriented world of Java, objects created by your Java program are dynamically allocated by the JVM from a storage heap (memory specifically reserved for object allocation and deallocation). When objects are no longer used by your program (the objects are no longer referred to and are therefore *dereferenced*), the objects are available for garbage collection (GC) so that the space they occupy can be used for other objects.

As your usage of Java in CICS increases, garbage collection is something you need to be aware of and something that may need tuning.

GC contains two required steps and one optional step. During GC, the live (currently used) objects need to be found (the *mark* phase); objects that are no longer referenced are returned to the free memory list (the *sweep* phase where we sweep away the garbage).

If only *mark* and *sweep* were performed, memory would become fragmented with unusable space or not enough contiguous space for larger object allocations. The third optional phase called *compact*, which defragments the heap memory.

Traditionally, all activity in the JVM must stop during GC. This stopping (*stop the world*) can cause erratic behavior and response times in your application if not controlled properly. The IBM JVM provides four GC policies to deal with your application's workload characteristics.

JVMs in CICS's pool of JVMs normally use the default GC policy called *optthruput* that is designed for maximum throughput. This way is appropriate for JVMs in CICS's JVM pool because CICS gives them one request to deal with at a time, and we want that request to get through just as quickly as possible. When a request is complete, CICS asks the JVM if the heap consumes greater than a specified threshold, and if true, CICS runs the CJGC transaction in that JVM. The CJGC transaction requests the JVM to do garbage collection. During that GC, because this is a CICS transaction and not a user transaction, it is acceptable for us to "stop the world" during GC. Using this approach, CICS decides when to do GC so we can maintain consistent response time during user transactions.

The JVMProfile file that CICS Dynamic Scripting dynamically creates for the multi-threaded JVMServer specifies a GC policy called *gencon*, which is optimized for many short-lived transactions that usually create smaller objects. The *gencon* GC policy is more involved than a simple mark, sweep, and compact. Multiple helper threads are used for parallel mark and sweep, part of which is performed while the JVM is still processing user requests. A periodic *compact* phase takes place, but even on a compact, multiple helper threads can get involved for parallel compact. Additionally, the whole heap is not normally compacted at once. For transactional systems, newer, smaller objects are most likely to need GC more often, so heap is divided into areas (for newer and older objects). If a newer object is passed over for GC multiple times, it is moved to the area for older objects where GC is applied less often. This is not the end of what is done for the *gencon* GC policy because it has object nurseries and much more. You can look in the IBM JVM manuals for more information about the *gencon* GC policy, but this short description might be enough for you to appreciate that GC should not be considered lightly, and that IBM has invested much research to minimize the affects of GC for your workload.

## 7.2.7 JVMServer Tuning

For JVMServer tuning, you will want to start with the default values. There are many values for tuning a JVM and as with any tuning effort, only change a couple items (or preferably one item) at a time.

The major area for JVM tuning is normally heap utilization. Monitoring heap utilization is normally done with tools such as the IBM Support Assistant Health Center, or tuning based on verbose GC output and using tools to analyze the output. Several parameters are available to control GC, however the two parameters normally looked at first are the `-Xms` and `-Xmx` parameters. Both parameters specify a size. Properly tuning these parameters can reduce the overhead of garbage collection.

The `-Xms` value typically should be large enough to avoid allocation failures from the time the application starts to the time it is in a *ready* state.

Several general rules for the `-Xmx` value are as follows:

- ▶ For the `-Xmx` value, during a normal load, the free heap after GC should be greater than `minf` (a setable parameter that defaults to 30%).
- ▶ The `-Xmx` value must be large enough so that you never receive an out-of-memory error.
- ▶ During the heaviest load, if the free heap after GC is greater than `maxf` (a setable parameter that defaults to 60%), the heap size (`-Xmx`) is too large.

In general, if the heap is too small, GC will be too frequent. If the heap is too large, there will be too much GC pause when GC is performed. If the heap is greater than the available physical memory, you will see paging and swapping.

Heap size expands and contracts between the -Xms and -Xmx values as the demands of your application change. If these values are set properly, garbage collection causes the least amount of impact.

For CICS Dynamic Scripting, you can set up JVM arguments such as -Xms in `zerocics.config`, and also customize it in the application's `zero.config` as in Example 7-16. The setting in `zero.config` overrides that in `zerocics.config` file.

*Example 7-16 The zero.config file*

---

```
/config/zso/jvmargs += [  
    "-Xmx64M"  
]
```

---

Many more tuning parameters are documented in the IBM JVM manuals. Additionally, many articles are available on developerWorks and other places about IBM JVM tuning.

The IBM Support Assistant Health Center is a good place to start with JVM tuning and also gaining other information about your JVM with minimal performance overhead. See Chapter 8, “Troubleshooting” on page 181 for more details.

## 7.2.8 Use of zAAP by CICS Dynamic Scripting

A zAAP processor is a specialty engine that can be added to a z/OS environment. When available, a zAAP processor can run Java workloads. Because CICS Dynamic Scripting runs in a JVMServer, one might initially think that 100% of CICS Dynamic Scripting activity is offloadable to a zAAP processor, but unfortunately this is not true.

Many of the PHP functions are implemented in C routines. CICS commands require JNI (the Java Native Interface which allows access to modules written in other languages) and the CICS commands themselves do not execute in Java. Additionally, use of the WMQ API and DB2 cause execution outside the Java environment.

Because of this, predicting how much of your Zero application will be eligible for zAAP is not possible. After your application is written, z/OS tools are available to measure the amount of CPU usage that is being run on a zAAP processor.

## 7.3 Common zero commands for an administrator

This section introduces common ZERO commands which are used frequently and are helpful. The three kinds of ZERO commands are module commands, module group commands, and repository commands.

### 7.3.1 Module commands

These commands support the life cycle of maintaining a module. You can use various options to start, stop and restart Dynamic Application modules. Several useful commands are listed in this section.

#### Compile

The **compile** command compiles all Java source files under the module's /java directory.

*Example 7-17 Compile module*

---

Usage:

`zero compile [-arg=<java-arg>] [-@=<file>] [-ecj=<path>]`

---

The command compiles all Java source files (\*.java) under the module's java directory, the resulting class files (\*.class) are generated under the classes directory. The command does not compile or recompile any source files that are part of the module's dependencies. Java compile arguments can be passed in one of two ways, directly or wrapped with the **-arg** and **-@** option.

Any argument that the command does not recognize is passed to the underlying **compile** command, for example **zero compile -Xlint:unchecked**. To determine the supported arguments use the **zero compile -help** command.

The wrapped approach uses the **-arg** or **-@** options, and is consistent with other commands. Individual compile arguments can be specified with **-arg**, for example, **zero compile -arg=-Xlint:unchecked**. The **-@** option can be used to pass a file that contains the compile options.

The **compile** command by default uses the javac found using the environment variable `java.home`. The **-ecj** option allows using the Eclipse compiler for Java instead by specifying the path to the `ecj.jar`.

The return codes for this command are as follows:

- ▶ 0 - success
- ▶ 1 - command failed

## Create

The **create** command creates a new module in the current directory. See Example 7-18.

### *Example 7-18 Create command*

---

Usage:

```
zero create <name> [-ignorepeers] [-dir=<dir>] [-module=<name>]
zero create <name> linkto <org:module:revision> [-ignorepeers]
[-dir=<dir>] [-module=<name>]
zero create <name> from <org:module:revision> [-ignorepeers]
[-dir=<dir>] [-module=<name>]
```

---

The command creates a new module under the current working directory. The **create** command has three uses: create a new module, create a module linked to an existing module, and create a copy of an existing module. The first usage creates a new module from the template `zero.application.template`. The **linkto** create also creates a module using the template `zero.application.template` but the module dependencies are modified to the `linkto` module. The **from** create creates a copy of the existing module and then modifies the module name to be the specified name.

## Start

The **start** command starts the application as a background process. The application is available immediately, but the resources required by the application is loaded only on demand. See Example 7-19.

### *Example 7-19 Start application*

---

Usage:

```
zero start [-fg] [-d] [-v] [-G={config}] [-jvmarg={jvmarg}]
[-pretaskarg={pretaskarg}] [-arg={arg}]
```

---

The options are as follows:

- |               |                                                                                                                  |
|---------------|------------------------------------------------------------------------------------------------------------------|
| <b>-fg</b>    | Allows the application to be run directly on the local console in the foreground, rather than in the background. |
| <b>-d</b>     | Enables minimal logging.                                                                                         |
| <b>-v</b>     | Enables all logging.                                                                                             |
| <b>-G</b>     | Allows the overriding values in the application configuration.                                                   |
| <b>-force</b> | Allows forcing application to attempt to start, regardless of current status.                                    |



The following options allow the passing of arguments to the jvm of the application. Any number of each argument is allowed:

- jvmarg**        These parameters are passed as a jvm argument.
- pretaskarg**   These parameters are passed before the target task (usually run).
- arg**            These parameters are passed after the target task.

The return codes for this command are as follows:

- ▶ 0 - success
- ▶ 1 - command failed

## Stop

The **stop** command stops the application, releasing the ports used for serving requests, and releasing all resources. See Example 7-20.

*Example 7-20 Stop command*

---

Usage:

zero stop [-force]

---

The **-force** option causes the stop command to kill any remaining zero processes related to the application if it not possible to stop the application gracefully.

The return codes for this command are as follows:

- ▶ 0 - success
- ▶ 1 - command failed

## Recycle

The **recycle** command equals stop and then start. It forces the application to recycle, or to release all of its resources. The application will still be available to serve requests, but all resources are dropped, and the configuration is reloaded. See Example 7-21.

*Example 7-21 Recycle command*

---

Usage:

zero recycle

---

The return codes for this command are as follows:

- ▶ 0 - success
- ▶ 1 - command failed

## Resolve

The **resolve** command determine the module's dependencies. See Example 7-22.

### *Example 7-22 Resolve command*

---

Usage:

```
zero resolve [-ivy=<file>] [-resolver=<name>] [-report] [-ignorepeers]
[-natives=<arch_os>] [-dir=<dir>] [-module=<name>]
```

---

The **resolve** command is used to determine the dependencies for a module, including any transitive dependencies, and verifying that all the dependencies are available. The module's dependencies are determined based on the module's config/ivy.xml file.

The results of the **resolve** command can be found in the following file:

`.zero/private/resolved.properties`

The information includes a fully qualified class path, a list of all dependencies, and the location of the dependencies. The list of dependencies is for the virtual directories feature of WebSphere sMash.

The **resolve** command first looks for modules in the private repository, workspace modules, local repository modules and if a match is not found then uses the configured remote repositories. The modules that are found with the workspace resolver can be configure through the -dir and -module option. The directory specified with -dir is used to list all modules in that directory. The -module option is used to specify a single module. The default for workspace modules is to find modules using the parent directory of the current module. The workspace resolver can be disabled with the -ignorepeers option.

The **resolve** command creates a `.zero/shared/resolve.history` file that is used to lock resolve to use the same module revisions for future resolves. The subsequent **resolve** command use the history file to filter out non matching revisions of modules. An application can be updated to the latest revisions in the local repository by using the zero update command. The difference between the resolve and the update process is that the update does not lock the modules to the revisions stored in the history file. If an update causes undesired changes to the application, then the resolve can be undone using the rollback command.

The options are as follows:

<b>-ivy</b>	Allows specifying the ivy file to use, the default is to use the <code>ivy.xml</code> in the <code>config</code> directory.
<b>-resolver</b>	Allows specifying the resolver to use; the default is <code>localThenRemote</code> .
<b>-report</b>	Report the results of the resolve without actually performing the resolve.
<b>-ignorepeers</b>	Perform the resolve without any peer modules.
<b>-dir</b>	Allows specifying the parent root directory for modules that can be included in the resolve.
<b>-module</b>	Allows specifying individual modules that can be included in the resolve.
<b>-natives</b>	Allows you to specify which native dependencies you want to work with. The default will be for your current platform.

The return codes for this command are as follows:

- ▶ 0 - success
- ▶ 1 - command failed

## Update

The **update** command resolves a module to its latest dependencies. See Example 7-23.

*Example 7-23 Update command*

---

Usage:

```
zero update [-ivy=<file>] [-resolver=<name>] [-report] [-ignorepeers]  
[-natives=<arch_os>] [-dir=<dir>] [-module=<name>]
```

---

The command allows resolving the module ignoring the previous resolve history. The **resolve** command is locked into a revision in local repository if it has a history file. Use the **update** command when you want to resolve against all revisions available in the local repository.

The options are as follows:

<b>-ivy</b>	Allows specifying the ivy file to use, the default is to use the <code>ivy.xml</code> in the <code>config</code> directory.
<b>-resolver</b>	Allows specifying the resolver to use, the default is <code>localThenRemote</code> .

- report** Displays what revisions would be used, but doesn't actually make the change. This allows you to see what updates are available without any modifications.
- ignorepeers** Performs the resolve without any peer modules.

For other module commands, go to the following address:

[http://www.projectzero.org/sMash/1.1.x/docs/zero.devguide.doc/zero.cli.tasks/CliModuleTasks.html#module\\_commands](http://www.projectzero.org/sMash/1.1.x/docs/zero.devguide.doc/zero.cli.tasks/CliModuleTasks.html#module_commands)

## 7.3.2 Module group commands

These commands support the life cycle of maintaining a module group. The most useful module group command is **addurl**, which is described in this section.

### **modulegroup addurl**

This command adds a remote repository to the module group. See Example 7-24.

*Example 7-24 The modulegroup addurl command*

---

Usage:

```
zero modulegroup addurl <url> [-validate]
```

---

The command adds a repository to the set of remote repositories contained in the module group resolver chain.

In the command, <url> is the URL to the remote repository. Currently, two types of repositories are support: WebSphere sMash repositories and maven repositories. The type of repository can be indicated with zero= or maven=, respectively.

- ▶ A WebSphere sMash repository is assumed if the type is not specified with the url, but can be explicitly set with zero=url. An example of a WebSphere sMash repository is as follows:

```
zero=https://www.projectzero.org/repo/zero.repo.latest
```

- ▶ The second type of repository is a maven repository, indicated by maven=, for example the following maven:

```
maven=http://repo1.maven.org/maven2/
```

The -validate option will verify that the URL is valid by trying to connect to it.

The return codes for this command are as follows:

- ▶ 0 - success
- ▶ 1 - command failed
- ▶ 2 - command failed, host already exists

For other module group commands, go to the following address:

[http://www.projectzero.org/sMash/1.1.x/docs/zero.devguide.doc/zero.cli.tasks/CliModuleTasks.html#modulegroup\\_commands](http://www.projectzero.org/sMash/1.1.x/docs/zero.devguide.doc/zero.cli.tasks/CliModuleTasks.html#modulegroup_commands)

### 7.3.3 Repository commands

These commands support the life cycle of maintaining a repository. Here are some common ones.

#### **repository info**

This command prints basic configuration data all repositories (module groups). See Example 7-25.

*Example 7-25 Repository info*

---

Usage:

```
zero repository info [-json]
```

---

The command prints basic configuration data about all of the available repositories.

If the -json option is used, the report is formatted as a JSON document.

The return codes for this command are as follows:

- ▶ 0 - success
- ▶ 1 - command failed

#### **repository list**

This command lists the modules contained in specified module group. See Example 7-26.

*Example 7-26 Repository list*

---

Usage:

```
zero repository list <modulegroup> [-json]
```

---

The command lists the modules contained in the specified module group. The module group can be one of the configured module groups or the path to a private shared repository on the file system.

If the **-json** option is used, the report is formatted as a JSON document.

The return codes for this command are as follows:

- ▶ 0 - success
- ▶ 1 - command failed

For other repository commands, go to the following address:

[http://www.projectzero.org/sMash/1.1.x/docs/zero.devguide.doc/zero.cli.tasks/CliModuleTasks.html#repository\\_commands](http://www.projectzero.org/sMash/1.1.x/docs/zero.devguide.doc/zero.cli.tasks/CliModuleTasks.html#repository_commands)

## 7.4 Security (file systems, applications,CICS), Active Content Filtering

Most sample and demo applications have security disabled because of time constraint limitations; in most cases, security topics can potentially fill an entire volume, rather than a chapter or section. In this section we provide some direction related to securing your CICS Dynamic Scripting applications and help you understand the bigger picture in regards to setting up security.

One key aspect of CICS Dynamic Scripting is that all applications built for and run in your CICS environment maintain your existing CICS security measures that are in place. Therefore, all CICS Resources are protected by your existing security process and procedures. The inverse is also true, any resources that are currently not protected are not be protected when exposed through your new CICS Dynamic Scripting applications, therefore, you should perform a security review and test before installing new applications, adding new technologies or access techniques, such as the CICS Dynamic Scripting Feature Pack.

The CICS Dynamic Scripting Feature Pack is built on technology from WebSphere sMash v1.1.1.3, but does not implement all of its features and has several restrictions. Go to the CICS Transaction Server v4.1 information center:

<http://publib.boulder.ibm.com/infocenter/cicsts/v4r1/index.jsp>

At the information center, select **CICS Dynamic Scripting Feature Pack → Restrictions**.

Two items of specific interest in regards to Security are the following restrictions:

- ▶ The zero security model and APIs in the `zero.core.security.*` package are not supported in Dynamic Scripting. For information about security in Dynamic Scripting, see “Securing dynamic scripting applications” at the information center.
- ▶ In Dynamic Scripting, SSL settings for an application that listens on HTTPS are configured in `zerocics.config` and not using the `/config/https/sslconfig` Global Context fields in `zero.config` as on WebSphere sMash. For more information, see “Securing dynamic scripting applications” at the information center.

These restrictions are because CICS Dynamic Scripting applications run inside a CICS Transaction Server region, which already provides a robust and secure environment. thus CICS based security features are used to protect your CICS resources while running in the CICS Environment.

We review the content from the “Securing dynamic scripting applications” section, but always consult the latest addition of the CICS information center.

A CICS Dynamic Scripting application, as shown in Figure 7-23 on page 150, consists of several components that may reside on separate platforms, each of which may rely on separate security mechanisms:

1. Application installation, maintenance, and control

z/OS UNIX System Services components:

- Dynamic Scripting product components
- Application components

2. Client access

HTTP through TCPIPService:

- Browser front ends
- Restful interfaces

3. Application execution

CICS environment:

- Dynamic scripting applications
- Resources such as temporary storage queues (TSQs), files, databases, and so on

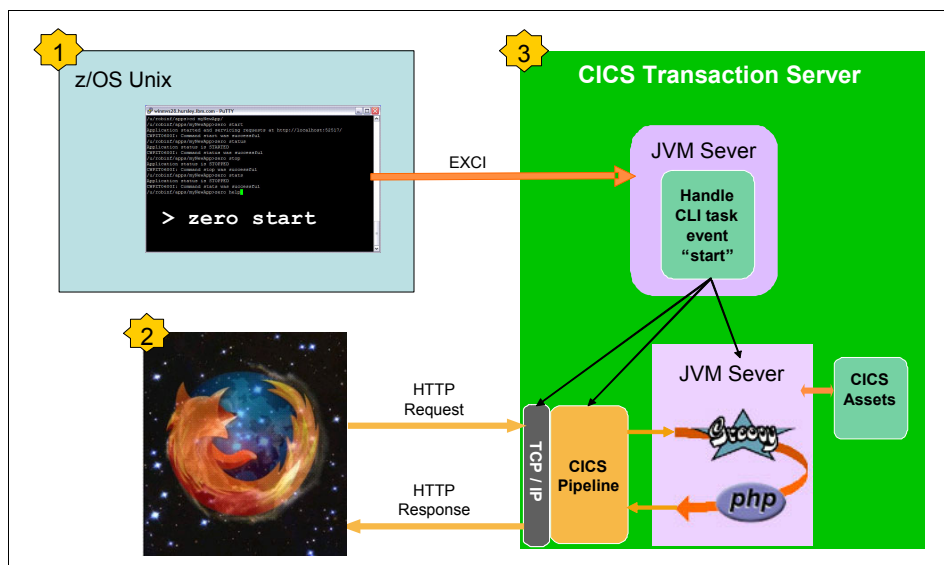


Figure 7-23 CICS Dynamic Scripting: Control, client access, and execution

In 7.4.1, “z/OS UNIX System Services security” on page 150 we describe how z/OS UNIX security mechanisms are used to secure access to, and control execution and updates to, both the product resources and your dynamic scripting application scripts.

In 7.4.2, “Client Access Security” on page 154, we describe how you use Basic Authentication or SSL/TLS security to authenticate and secure access to your applications.

In 7.4.3, “Application Execution Security” on page 157, we describe how your z/OS security product, such as RACF, is used to secure access to and control the execution and updates to both your application programs and resources.

## 7.4.1 z/OS UNIX System Services security

CICS Dynamic Scripting applications are located in z/OS UNIX directories, using a specific directory structure. Figure 7-24 on page 151 shows a **zero start** command being issued from z/OS UNIX to start a CICS Dynamic Scripting application by loading it from the z/OS UNIX application directory and then running it in a JVMServer in a CICS Transaction Server region.



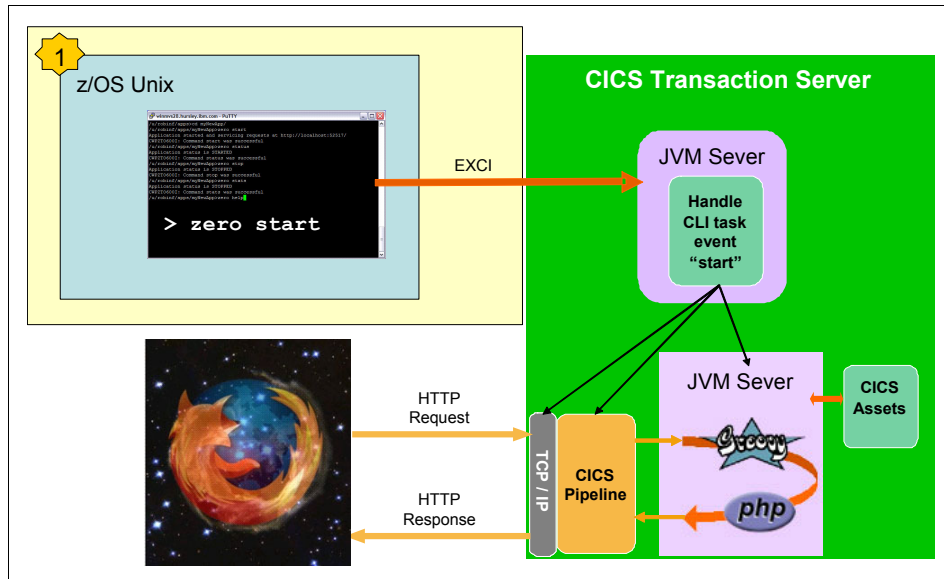


Figure 7-24 Application Installation, Maintenance and Control Security

Because our CICS Dynamic Scripting applications are located in z/OS UNIX files and directories, we are relying on its security to protect both our Groovy and PHP Scripts along with who can control them through ZERO commands.

z/OS UNIX provides several types of file systems to store our scripts, (HFS, ZFS, TFS, NFS), all of which provide two main features:

- ▶ A method of accessing, organizing, and storing files and directories
- ▶ z/OS UNIX maintains UNIX file and directory permissions for each file and directory in the file system.

When a user wants access to a file or directory, the system matches the user identifier (UID) and group identifier (GID) against security information associated with each file or directory. All access control decisions for z/OS UNIX are made by the installed external security manager, in our case RACF.

Figure 7-25 shows the relationship between z/OS UNIX and RACF in regards to controlling and authorizing access to resources

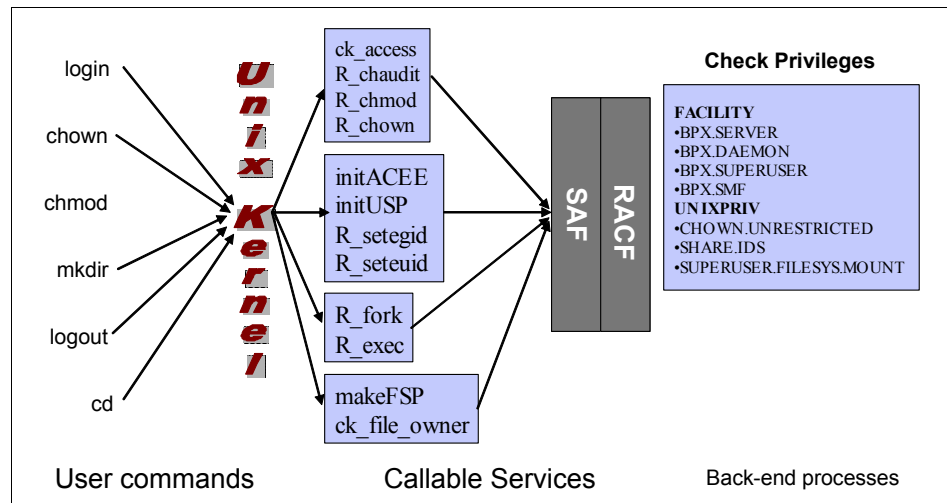


Figure 7-25 Interaction between z/OS UNIX and RACF

Each z/OS UNIX file or directory has a series of permission bits associated with it that are used to control access rights to specific users and groups of users. Example 7-27 shows a listing of a directory structure containing a CICS Dynamic Scripting apps directory; by using the `ls` command with the `-l` option, the permission bits for each file or directory are shown.

By matching the user's UID and GID against these permission bits, RACF determines who should be allowed to read, write, or execute the files.

Example 7-27 Output of a `ls -l` command showing the permission bits

```
CICSR4 @ SC66:/dynamicScript>ls -l
total 286410
-rw-r--r--  1 CICSR6  SYS1   129100442 Oct 27 22:57 CICSR1.JVM.TDUMP
drwxr-xr-x  2 CICSR2  SYS1      832 Jun 11 2010 Licenses
drwxr-xr-x 38 CICSR2  SYS1     2880 Dec 10 01:17 apps
drwxr-xr-x  2 CICSR2  SYS1      480 Jun 11 2010 cics
-rw-r----- 1 CICSR2  SYS1   17367757 Oct 14 21:33 cics_dynamic_scri
dr--r--r--  3 CICSR2  SYS1      320 Oct 27 22:18 pipelineShelf
```

The apps directory has the permission bits shown in Figure 7-26 on page 153. The owner's permissions are `rw`, which correlate to read, write, and execute. The group permissions are `r-x`, which correlate to read and execute but no Write

authority. All other users have the same authority, read and execute but no write authority.

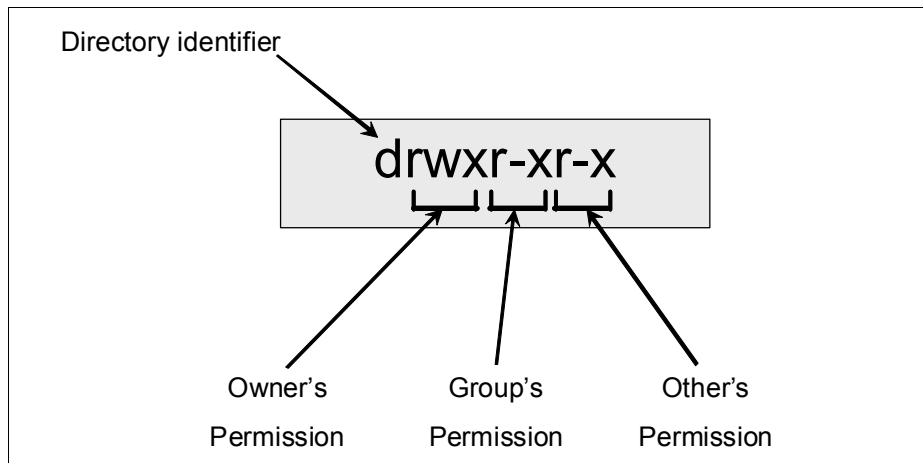


Figure 7-26 Exploring the Permission bits for the **apps** directory

The **chmod** command can be used to manage a file or directories permission bits, normally your z/OS UNIX System Administrator sets up and manages the security structure for you.

The z/OS UNIX user ID that the a CICS region uses must have write access to the applications directory and files. Example 7-28 shows a **chmod** command being used to give the group class write authority to the application directory and all subdirectories. This command assumes the z/OS UNIX user ID is in a common group with the CICS region user ID.

*Example 7-28 Granting CICS Access to the application directories*

---

```
chmod -R g+w /apps
```

---

For information about setting up CICS security, see the following resources:

- ▶ 6.3, "Configuring CICS security" on page 79
- ▶ CICS Transaction Server v4.1 information center:  
<http://publib.boulder.ibm.com/infocenter/cicsts/v4r1/index.jsp>

## 7.4.2 Client Access Security

Clients access CICS Dynamic Scripting applications through web browsers or RESTful interfaces using HTTP. Figure 7-27 shows a client interacting with a CICS Dynamic Scripting application running in a CICS region.

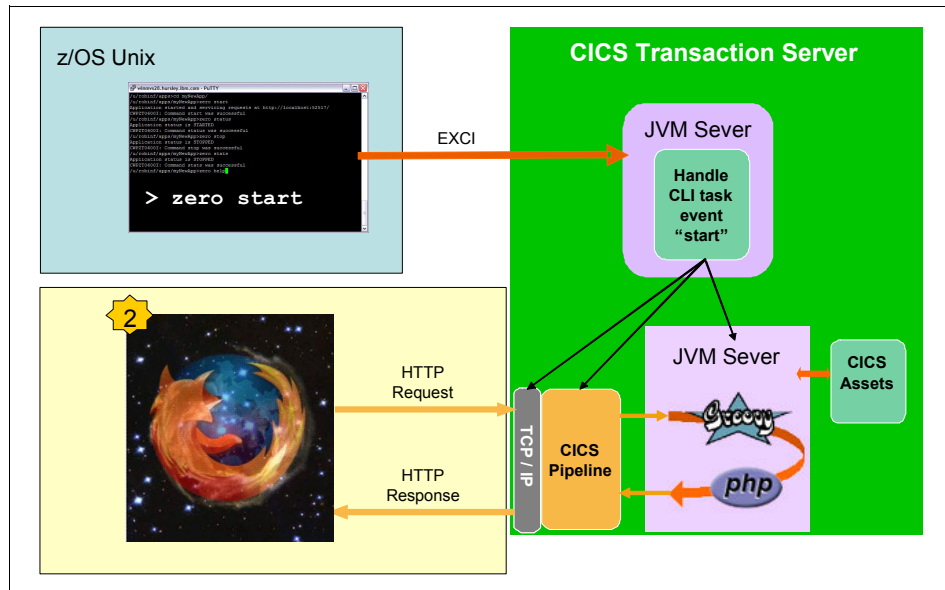


Figure 7-27 Client Access through HTTP Security

Because our Groovy or PHP scripts are actually running in a CICS environment, the client access security is controlled by the CICS region running the application. As shown in Figure 7-27, clients connect to CICS through HTTP, using a TCPIPService connection defined to the CICS runtime region.

Client authentication and message encryption are controlled by the **AUTHENTICATE** and **SSL** attributes defined in the TCPIPService port that is in use. So what we are showing you at this point is that the security used to control access to your CICS Dynamic Scripting application is the same security that is used by CICS web applications. The only difference is that you do not directly define the TCPIPService definitions, they are created for you dynamically by the CICS Dynamic Scripting application when it is started by a **ZERO** command.

To set the appropriate **AUTHENTICATE** and **SSL** attributes that will be used by your CICS Dynamic Scripting applications TCPIPService port, you modify the global configuration file, `zerocics.config`, located in your installation config directory, and optionally the `zerocics.config` file that is located in your applications config directory. Complete details of how to set up HTTP Basic

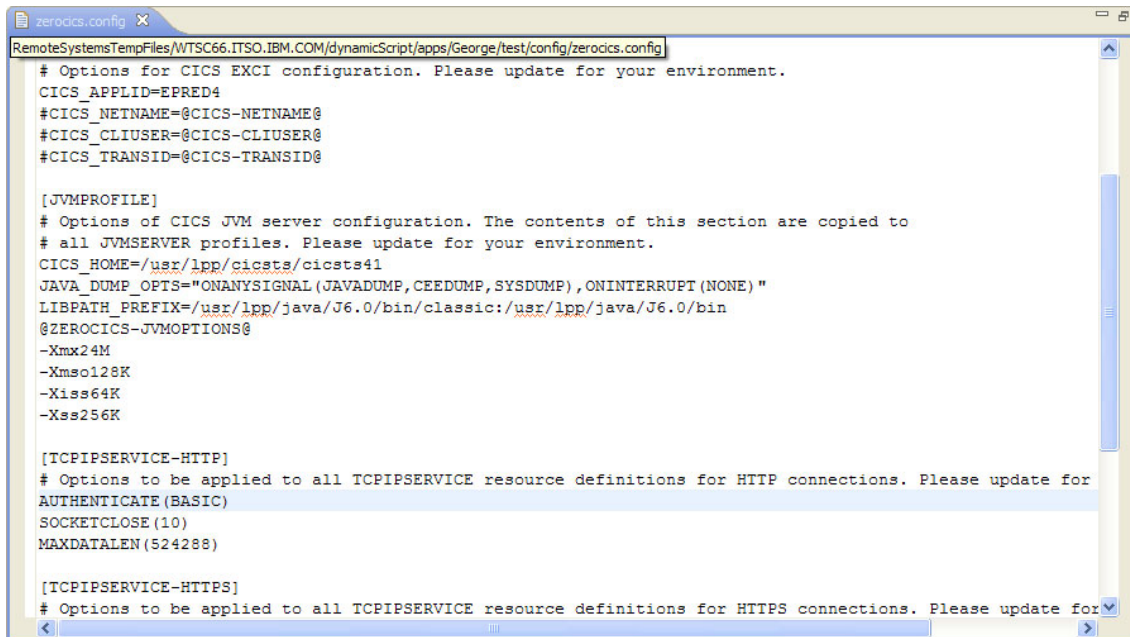
Authentication or HTTPS is in the CICS Transaction Server v4.1 information center:

<http://publib.boulder.ibm.com/infocenter/cicsts/v4r1/index.jsp>

At the information center, select **CICS Dynamic Scripting Feature Pack** → **Securing dynamic scripting applications**.

Details about the values and attributes of the AUTHENTICATE and SSL parameters, and other TCPIPService parameters are also at the CICS Transaction Server v4.1 information center in the “Security for CICS Web Support” or “Security for TCP/IP Clients” section. Only topics relating to CICS acting as a server are applicable.

For a simple test of HTTP Basic Authentication, the zerocics.config file for a test application was updated to include the AUTHENTICATE(BASIC) option as shown in Figure 7-28.



```
RemoteSystemsTempFiles\WTSC66.ITSO.IBM.COM\dynamicScript\apps\George\test\config\zerocics.config

# Options for CICS EXCI configuration. Please update for your environment.
CICS_APPLID=EPRED4
#CICS_NETNAME=@CICS-NETNAME@
#CICS_CLIUSER=@CICS-CLIUSER@
#CICS_TRANSID=@CICS-TRANSID@

[JVMPROFILE]
# Options of CICS JVM server configuration. The contents of this section are copied to
# all JVMSERVER profiles. Please update for your environment.
CICS_HOME=/usr/lpp/cicsts/cicsts41
JAVA_DUMP_OPTS="ONANYSIGNAL(JAVADUMP,CEEDUMP,SYSDUMP),ONINTERRUPT(NONE)"
LIBPATH_PREFIX=/usr/lpp/java/J6.0/bin/classic:/usr/lpp/java/J6.0/bin
@ZEROCICS-JVMOPTIONS@
-Xmx24M
-Xms128K
-Xss64K
-Xss256K

[TCPIPService-HTTP]
# Options to be applied to all TCPIPService resource definitions for HTTP connections. Please update for
AUTHENTICATE(BASIC)
SOCKETCLOSE(10)
MAXDATALEN(524288)

[TCPIPService-HTTPS]
# Options to be applied to all TCPIPService resource definitions for HTTPS connections. Please update for
```

Figure 7-28 Adding AUTHENTICATE parm to zerocics.config

After saving the changes and recycling the application through a ZERO RECYCLE CLI command, a test shows that access is blocked until a valid username (ID) and password are entered (Figure 7-29).

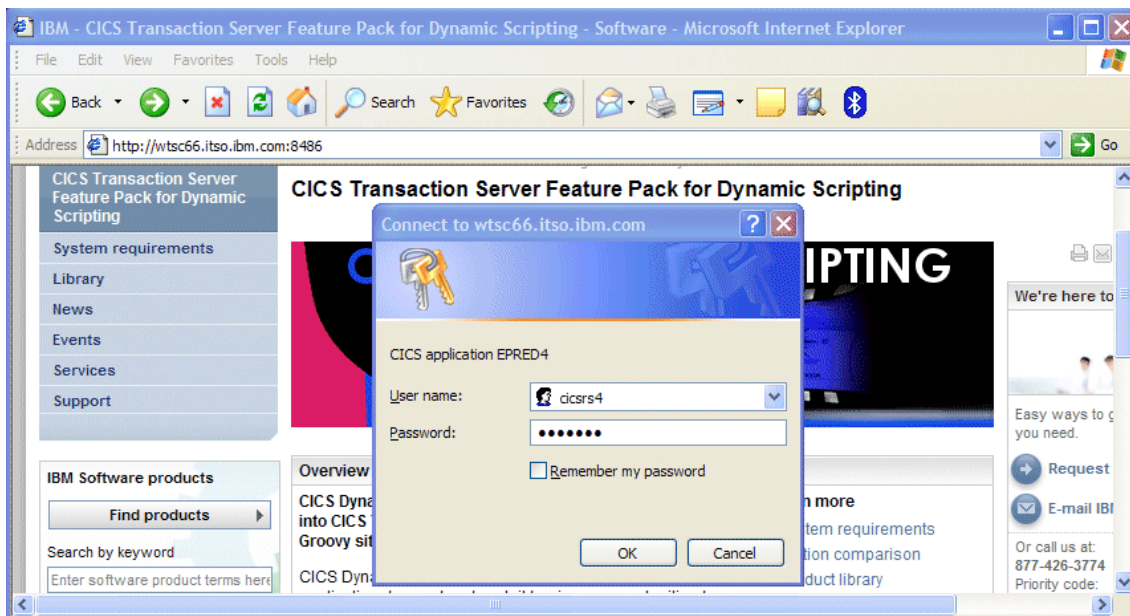


Figure 7-29 HTTP Basic Authentication forces a Signon before allowing access

After authentication is turned on, review your CLI. Specifically STOP and STATUS commands require a user ID and password to properly function, which can be fixed by adding a user ID/password combination to your application's zero.config configuration file that will be used when processing CLI commands.

Figure 7-30 shows executing a CLI command that fails after turning on HTTP Basic Authentication:

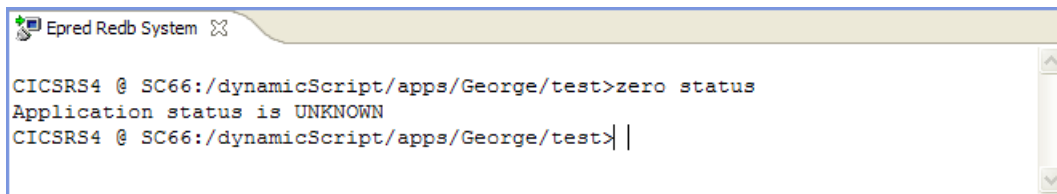
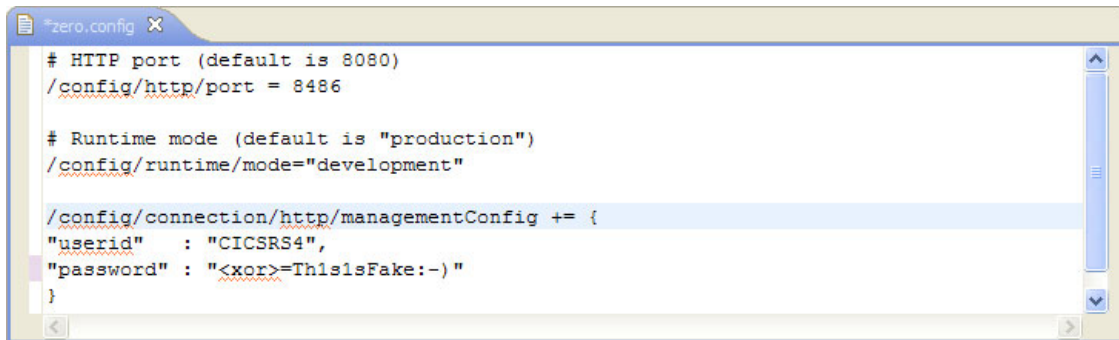


Figure 7-30 Failed command-line interface command

Figure 7-31 on page 157 shows an update to the test applications zero.config file. Note that the fictitious password is defined using XOR encoding to avoid storing the password in clear text. As explained in 7.4.1, “z/OS UNIX System

Services security” on page 150, the zero.config configuration file can be protected so that only authorized users can view and change it, to further protect the user ID and password information.



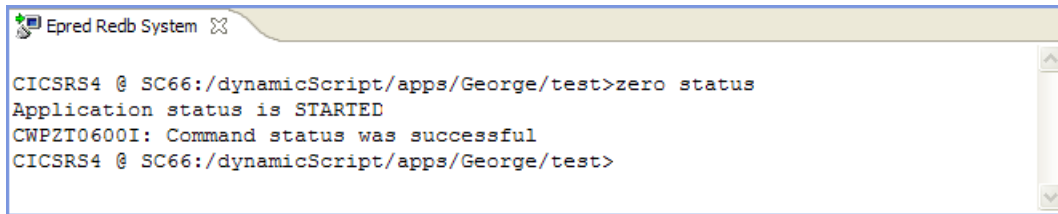
```
# HTTP port (default is 8080)
/config/http/port = 8486

# Runtime mode (default is "production")
/config/runtime/mode="development"

/config/connection/http/managementConfig += {
  "userid" : "CICSR54",
  "password" : "<xor>ThisIsFake:-)"
}
```

Figure 7-31 Updating the zero.config file with CLI user ID/password information

Figure 7-32 shows executing the same ZERO STATUS CLI command after adding user ID and password information to the test applications zero.config configuration file.



```
CICSR54 @ SC66:/dynamicScript/apps/George/test>zero status
Application status is STARTED
CWPZT0600I: Command status was successful
CICSR54 @ SC66:/dynamicScript/apps/George/test>
```

Figure 7-32 Successful command-line interface command after zero.config updates

Set up and configuration steps for both the zero.config and zerocics.config configuration files are described in 6.2, “Customizing the default configuration file” on page 75. As outlined in section 2.2.12, only one zero.config configuration file exists per application, and there may be more than one zerccics.config configuration file: one in the global install directory and one in your application directory.

### 7.4.3 Application Execution Security

CICS Dynamic Scripting applications are run in a CICS Region after being loaded from z/OS UNIX files, through the ZERO START CLI command. Figure 7-33 on page 158 shows a CICS Dynamic Scripting application running in a CICS region being accessed from a web browser client through HTTP. This

section reviews how you secure your CICS resources and data when running in a CICS region.

We previously covered the following information:

- ▶ How to secure the application under z/OS UNIX in 7.4.1, “z/OS UNIX System Services security” on page 150.
- ▶ How to authorize clients coming into the CICS region in 7.4.2, “Client Access Security” on page 154.

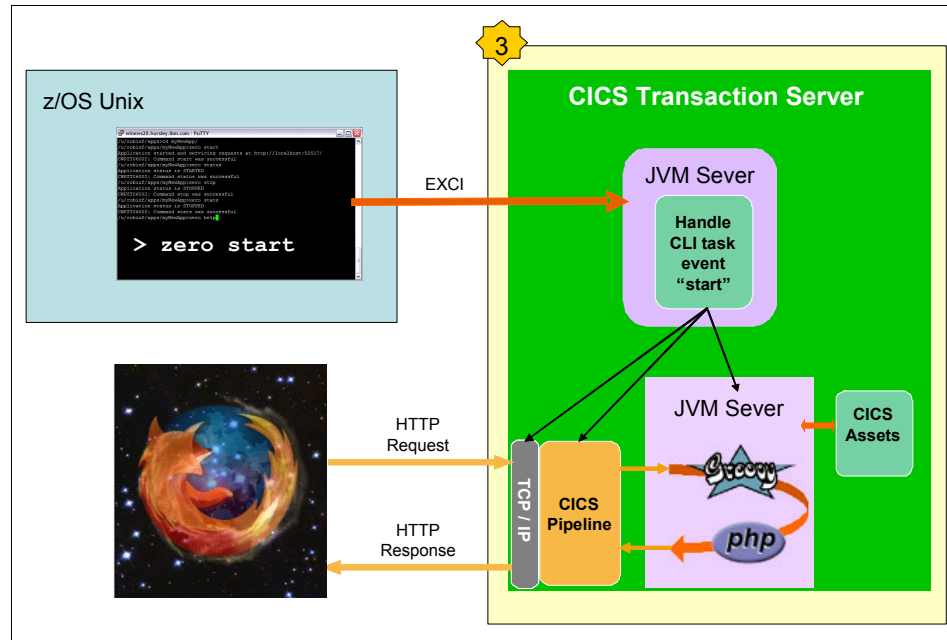


Figure 7-33 CICS runtime security

At this point in the execution of your CICS Dynamic Scripting application, CICS and your z/OS External Security Manager provide the necessary security functions. Using the client credentials, from 7.4.2, “Client Access Security” on page 154, CICS uses your existing security configuration to authorize access to your application components and resources using RACF or what ever external security manager that you have installed.

## 7.4.4 Active content filtering

With CICS Dynamic Scripting, you can filter out or cleanse active content, such as JavaScript, Applet, and ActiveX objects, from the information flowing into and out from your CICS Dynamic Scripting applications.



A person who uses your application can introduce malicious items into your application with the intent of causing potential damage or fraud. An example is shown in “Browser redirection ACF example” on page 164 where a small snippet of JavaScript is entered into a browser parameter field with the intent of redirecting the client to a different web page.

Active Content Filtering (ACF) provides several levels of filtering and control to the application designer to help detect and remove unwanted objects:

- ▶ ACF default enablement configuration
- ▶ ACF custom configuration
- ▶ ACF Programmatic API

The ACF default enablement configuration is as simple to use as setting a parameter, just add a single dependency to your `ivy.xml` configuration file and you're done. The Default Enablement also includes protection for cross-site request forgery.

For more control you can use custom configuration files by tailoring your application's `zero.config` file to specify ACF configuration options or to reference alternate filter rules files.

For more complex scenarios you can take advantage of the ACF Programmatic APIs to control the filtering directly in your application.

By configuring and using ACF, you can validate and remove untrusted active content in requests and responses. More information is available in the IBM WebSphere sMash information center by searching on ACF.

## **ACF default enablement configuration**

By default (turning on ACF through your `ivy.xml` configuration file), ACF identifies and removes unwanted active content. The default configuration that is included with the `zero.acf` package performs the following tasks:

1. Removes active content from all HTTP request parameters sent to any URIs. These parameters are considered as HTML fragments.
2. Removes active content from all string values in an inbound JSON object sent to any URIs when the content type of the request is either of the following types:
  - `application/json`
  - `text/json`

3. Removes active content from all String values in an outbound JSON object sent by any URIs when the content type of the response is either:
  - application/json
  - text/json
4. Enables protection against cross-site request forgery (CSRF).

By default, ACF also provides basic protection against CSRF scenarios by using a transparent instrumentation process to create CSRF credentials during initial authentication and provide it for verification for each request to a protected resource. Transparent instrumentation is available only to applications that access protected resources through static links in HTML pages or through XHR in Dojo. For more complex requirements see “ACF Programmatic API” on page 164.

### Turning on ACF for default enablement

To turn on Active Content Filtering for an application, simply enter a dependency for `zero.acf` to your applications `ivy.xml` file located in your `config` directory. Figure 7-34 shows an `ivy.xml` file with the `zero.acf` dependency included.

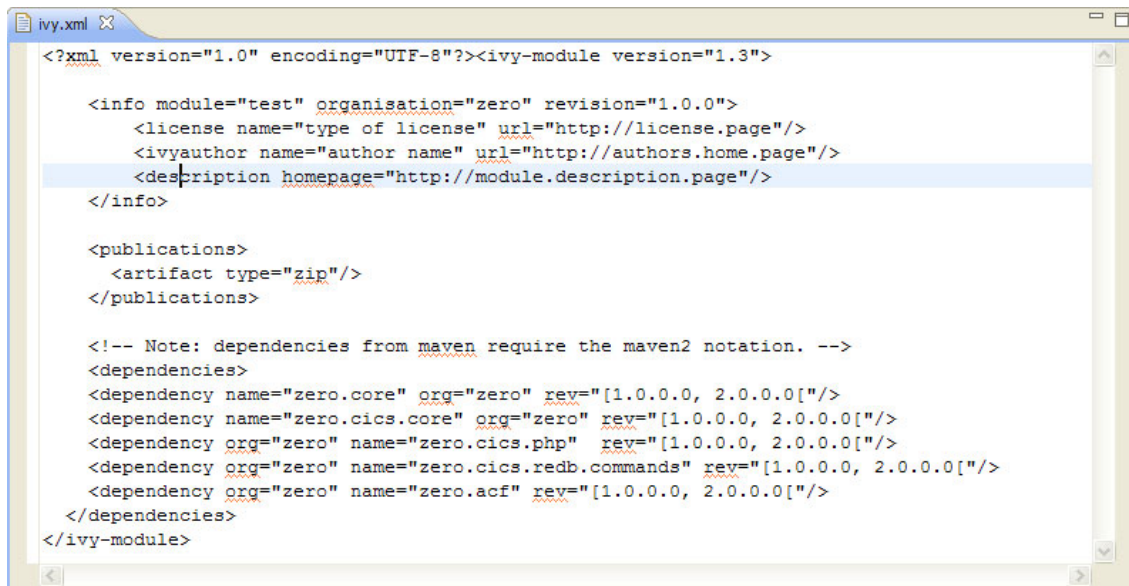


Figure 7-34 Adding `zero.acf` to your `ivy.xml`

You must issue **zero stop**, **zero resolve**, and **zero start** commands to recycle and resolve the dependency.

With no further customization, ACF will provide default enablement filtering to your application by identifying and removing active content from request parameters, such as the name field in a form, before it is passed to your application.

See “Browser redirection ACF example” on page 164, for an example that shows how to turn on Active Content Filtering using the default enablement.

## ACF custom configuration

The default enablement protects against most of the possible vulnerabilities with user generated content, but for situations where the default ACF configuration does not meet your security needs you can further customize the ACF configuration and filter files to modify or enhance the filtering configuration and rules.

To use a custom ACF configuration, you first turn off the default enablement by adding a parameter in your applications `zero.config` configuration file and then add ACF configuration options. Example 7-29 shows the parameter that you use to turn off default enablement, it is not required to turn on default enablement but after it has been added to your `zero.config` configuration file, it acts as a toggle and can be used to flip between default enablement, `true`, or a custom configuration, `false`.

With default enablement, CSRF is automatically enabled. The second line (Example 7-29) relates to CSRF protection; because you are turning off default enablement through the first line, you must tell the system what you would like to do in regards to CSRF.

*Example 7-29 zero.config setting to turn off Default Enablement*

---

```
/config/acf/enableByDefault=false  
/config/security/token/enableCsrfProtection=""
```

---

For `enableCsrfProtection`, you have two options shown in Table 7-1. For our test, CSRF was set to disabled.

*Table 7-1 The enableCsrfProtection= parameter values*

Value	Description
" "	Disabled
"REQUEST"	Automatic Token Generation, must use CSRF API

For more information about CSRF, see the IBM WebSphere sMash information center and search for CSRF.

Next, you modify the ACF configuration to tell the system to use a custom filter rule file. Using the ACF configuration options, outlined in Table 7-2, add an include parameter (shown in Example 7-30) to your `zero.config` configuration file.

*Example 7-30 zero.config ACF Configuration options*

---

```
@include "${/config/dependencies/zero.acf}/config/acf.config"{
    "conditions" : "/request/path =~ /ACFTest.gt",
    "contentType" : ["text/html"],
    "filterRuleFile" : "acf-custom.xml"
}
```

---

The parameters used in Example 7-30 tells the system to use the custom ACF filter rule file, `acf-custom.xml`, and to apply those rules when the `ACFTest.gt` application is run as identified by the conditions parameter. In this example, the conditions parameter's request path is assuming our script is located in the public directory.

*Table 7-2 ACF Configurations options*

Parameter	Description
conditions (mandatory)	The URI pattern in the <code>/request/path</code> , which is protected. These regex patterns are the same that are supported as part of the condition operators in WebSphere sMash.
contentType (optional)	A list of the content types of the input. The default value is <code>["application/json", "text/json"]</code> . When the content type in the request/response header includes one of the content types defined here, ACF is invoked. When you want to remove active content from an HTML document, set this property to <code>["text/html"]</code> .
target (optional)	A list of the targets of ACF processing. The default value is <code>"RESPONSE"</code> . When you want to remove active content from inbound request parameters (for HTML) or a inbound JSON object (for JSON), you need to set this property to <code>"REQUEST"</code> . To remove active content from both inbound data and outbound data, set this property to <code>"REQUEST_RESPONSE"</code> .
filterJsonDuringEncode (optional)	The flag that indicates whether JSON should have active content removed (if found) during JSON serialization and deserialization. The default value is true. If set to false, the application developer can filter the JSON data structure using the programmatic APIs.

Parameter	Description
filterRuleFile (optional)	The name of the file including your custom filter rules. For HTML, ACF works based on filter rules. When ACF filters inbound request parameters or an outbound HTML without this property, it uses the default filter rules bundled with ACF. But you can use your own custom filter rules when this property is set. ACF assumes that the configuration file is put in the <code>conf i g</code> directory in your application.

To complete the setup to run a custom ACF configuration, you must create a custom filter rules file. In our example, we have a custom ACF filter rule file called `acf-custom.xml`. The contents of this file describe the filter rules and are built using the rules described in Table 7-3.

*Table 7-3 Basic Filter Rules*

Rule	Description
rule	The atomic filter rule, such as removing tags with a certain name.
rule/@c14n	True if canonicalization of attribute values is required before the ACF processing.
rule/@dataschema	True if decoding of the values based on the data URL scheme (RFC2397) is required.
rule/@url	True if decoding of URL encoded values is required.
rule/@entityreference	True if entity reference in the values needs to be resolved.
rule/@whitespace	True if any whitespace character (tab, carriage return, and line feed) needs to be removed.
rule/@all	True if all canonicalization techniques we described are required.
rule/@tag	The name of the tag in the message such as 'iFrame' and 'applet'. You need to specify at least one of the 'tag' and the 'attribute' values shown in the following information. If you do not set the 'tag' value, the filter rule is applied to all tags.
rule/@attribute	The part of the attribute name such as 'href' and 'style'. You need to specify either the 'tag' or the attribute.
rule/@attribute-criterion	How to compare the attribute. You can select one of the following ways: <ul style="list-style-type: none"> <li>► equals (default): Use this if the attribute name matches with the 'attribute' completely.</li> <li>► starts-with: Use this if the attribute name starts with the 'attribute.'</li> <li>► contains: Use this if the attribute name contains the 'attribute.'</li> </ul>
rule/@value	The value of the attribute such as 'javascript'.

Rule	Description
rule/@value-criterion	How to compare the attribute value. You can select one of the following ways to do this: <ul style="list-style-type: none"> <li>▶ equals (default): Use this if the attribute value matches with the 'value' completely.</li> <li>▶ starts-with: Use this if the attribute value starts with the 'value.'</li> <li>▶ contains: Use this if the attribute value contains the 'value.'</li> </ul>
rule/@action	What the ACF does if it finds the tag or the attribute that matches the filter rule. You can select one of the following ways to do this: <ul style="list-style-type: none"> <li>▶ remove-tag: Remove the tag.</li> <li>▶ remove-attribute: Remove the attribute. You can use this value only when you set the 'attribute' attribute.</li> </ul>

You might want to copy one of the existing filter files provided with ACF as a base when creating your own filter file. See “Custom filter rules ACF example” on page 171, for an example that shows how to create and use a custom filter rule file.

## ACF Programmatic API

The Programmatic APIs provide an application with the ability to perform fine grained or specific levels of detection and filtering on your application interfaces.

One reason you might want to use the ACF API is when the default ACF enablement does not meet your performance goals. Another use is when working with cross-site request forgery scenarios. In these scenarios, applications might require maximal security, and Programmatic Instrumentation can be used.

For more information about the ACF Programmatic APIs or cross-site request forgery scenarios, see IBM WebSphere sMash information center; search for CSRF or API Documentation.

## Browser redirection ACF example

For this example, we use a simple Groovy Template application that is designed to present an input field to the browser and upon entering any text, will echo it back to the browser when the submit button is pressed.

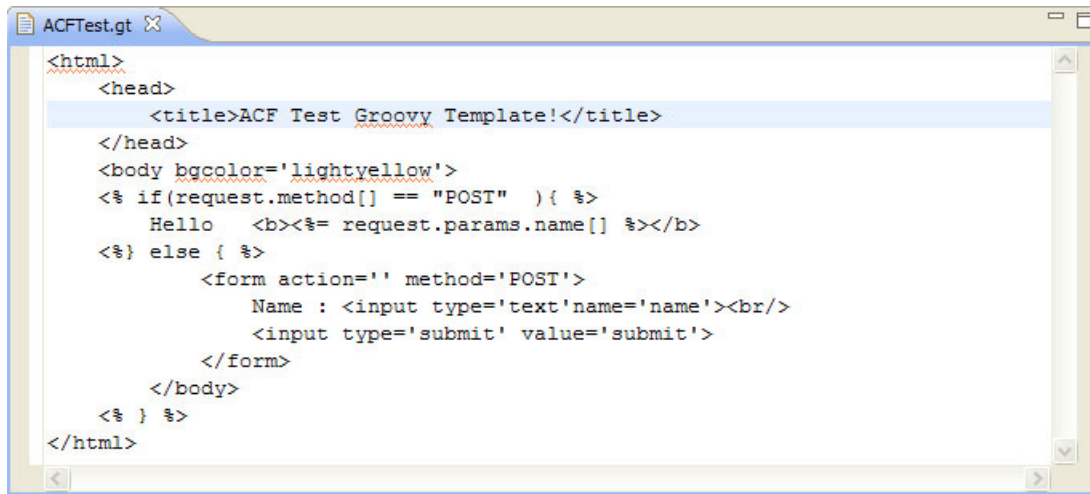
This sample gives us a chance to experiment with entering various malicious payloads in an attempt to disrupt the application. For our example, we enter the JavaScript listed in Example 7-31 on page 165.

*Example 7-31 Malicious input to our ACFTest.gt test application*

```
<script>location.href="http://ibm.com/cics"</script>
```

This small fragment of JavaScript causes an immediate redirection to a new URL from within the current browser window. By turning on Active Content Filtering and using the default enablement that is included with ACF, we can quickly protect our application from harm by filtering out most forms of active content that are entered.

Figure 7-35 contains a sample groovy template called ACFTest.gt that is designed to receive input through an input field and echo it back to the browser.

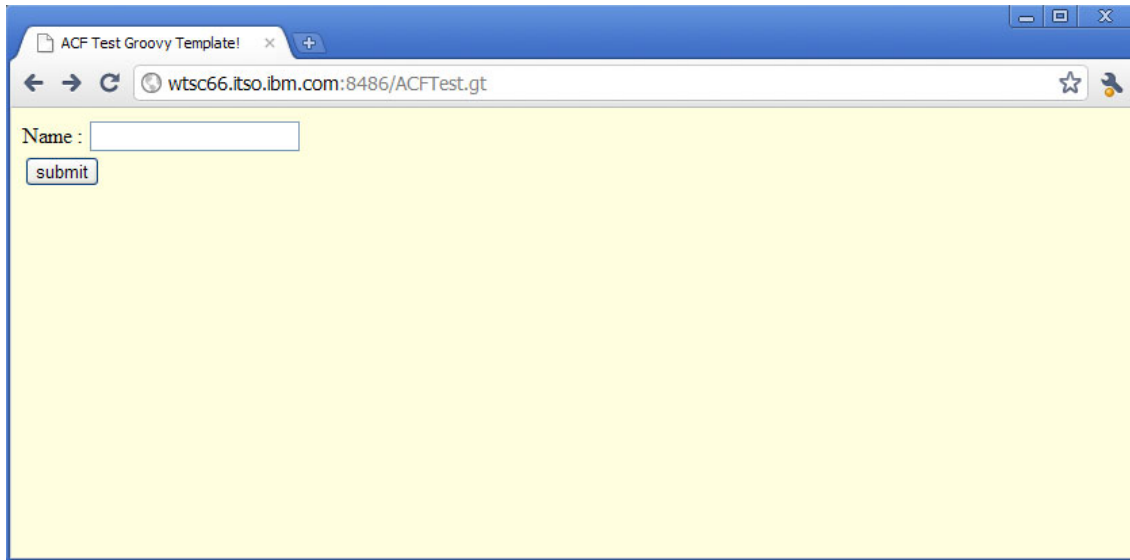


```
<html>
<head>
  <title>ACF Test Groovy Template!</title>
</head>
<body bgcolor='lightyellow'>
  <% if(request.method[] == "POST" ) { %>
    Hello  <b><%= request.params.name[] %></b>
  <% } else { %>
    <form action='' method='POST'>
      Name : <input type='text' name='name'><br/>
      <input type='submit' value='submit'>
    </form>
  <% } %>
</body>
</html>
```

*Figure 7-35 ACFTest.gt: sample groovy template to test ACF*

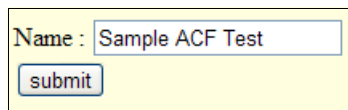
Using this application script and several ZERO commands, we can create and save our sample application as ACFTest.gt in the public directory.

After starting our application and accessing it from a browser, we get the web page listed in Figure 7-36, containing a simple input field, called Name.



*Figure 7-36 Input page generated by ACFTest.gt ACF test script*

We enter sample text into the name field (Figure 7-37).



*Figure 7-37 Inputting some text that should be echoed back*

Then we click **submit**. The result is shown in Figure 7-38 on page 167.





Figure 7-38 Expected results of ACFTTest.gt ACF test script

We test the sample application to determine how it handles active content. First, we enter malicious data (Figure 7-39) as opposed to normal text input. We click the back arrow button to return to the input page and enter the active content sample script listed in Example 7-31 on page 165.

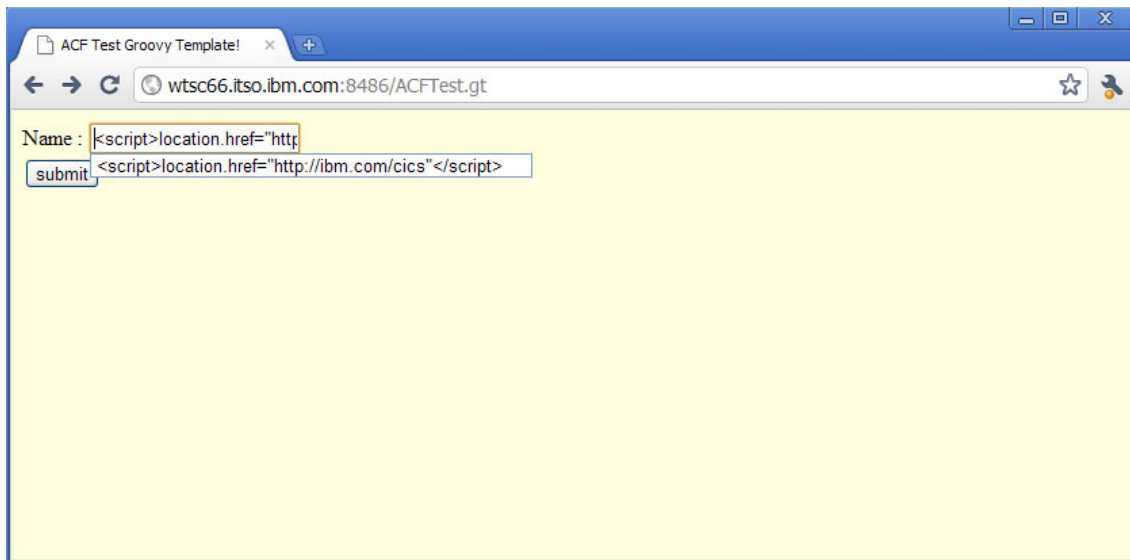


Figure 7-39 Entering malicious data into ACFTTest.gt ACF test script

The normal response would have been to echo the input back to the browser but instead our input is executed by the browser as Active Content and immediately redirected to the web page listed in the HREF property (Figure 7-40).

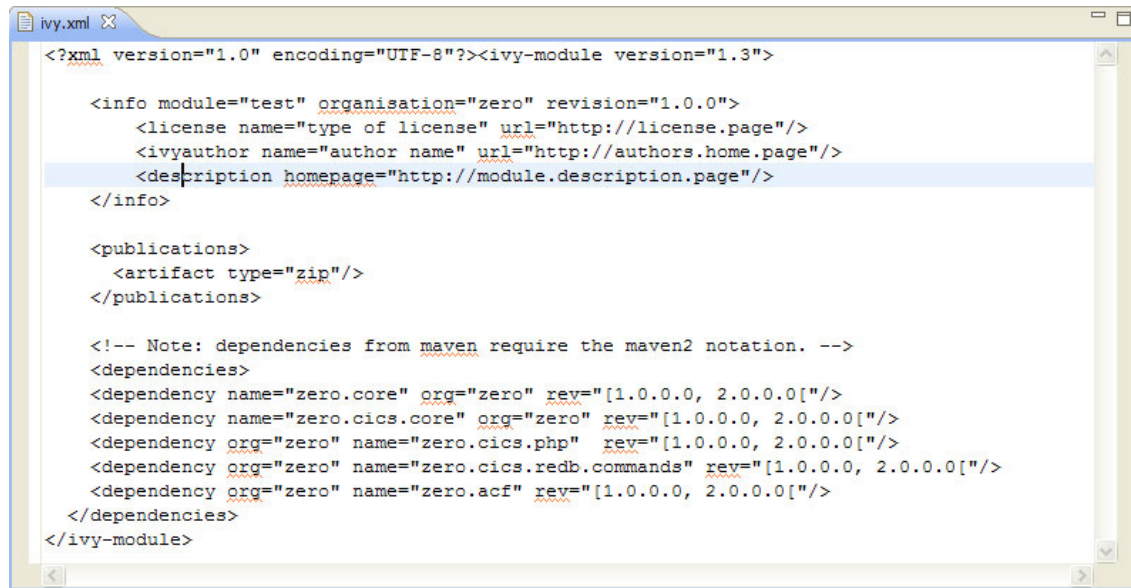


Figure 7-40 We are redirected to another site

Although this is a helpful web page, it obviously is not the response to our application that we expected.

What we can do to fix this exposure is to turn on Active Content Filtering using the default enablement provided with the ACF package and re-test our application.

Using a simple text editor, we add a dependency for `zero.acf` to our test applications `ivy.xml` configuration file, as shown in Figure 7-41 on page 169.



```
<?xml version="1.0" encoding="UTF-8"?><ivy-module version="1.3">

  <info module="test" organisation="zero" revision="1.0.0">
    <license name="type of license" url="http://license.page"/>
    <ivyauthor name="author name" url="http://authors.home.page"/>
    <description homepage="http://module.description.page"/>
  </info>

  <publications>
    <artifact type="zip"/>
  </publications>

  <!-- Note: dependencies from maven require the maven2 notation. -->
  <dependencies>
    <dependency name="zero.core" org="zero" rev="[1.0.0.0, 2.0.0.0["/>
    <dependency name="zero.cics.core" org="zero" rev="[1.0.0.0, 2.0.0.0["/>
    <dependency org="zero" name="zero.cics.php" rev="[1.0.0.0, 2.0.0.0["/>
    <dependency org="zero" name="zero.cics.redb.commands" rev="[1.0.0.0, 2.0.0.0["/>
    <dependency org="zero" name="zero.acf" rev="[1.0.0.0, 2.0.0.0["/>
  </dependencies>
</ivy-module>
```

Figure 7-41 Adding zero.acf dependency to the ivy.xml configuration file

We are using all the defaults thus no additional customization or changes are required. To finalize our changes we will use OMVS, from ISPF option 6, to shut down our application and resolve our newly added dependency. Figure 7-42 on page 170 shows the ZERO commands being entered to stop, resolve and start our application, the ZERO RESOLVE CLI command is required to retrieve the required modules from the repository.

```

Session A - [24 x 80]
File Edit View Communication Actions Window Help
CICSR4 @ SC66:/dynamicScript/apps/George/test>zero status
Application status is STARTED
CWPZT0600I: Command status was successful
CICSR4 @ SC66:/dynamicScript/apps/George/test>zero stop
Application status is STOPPED
CWPZT0600I: Command stop was successful
CICSR4 @ SC66:/dynamicScript/apps/George/test>zero resolve
CWPZT0901I: The following module(s) are not currently in the local repository:
zero:zero.acf:Y1.0.0.0, 2.0.0.0Y
CWPZT0902I: Trying to locate the module(s) using one of the configured remote re
positories
CWPZT0545I: Retrieving zero.acf-1.1.1.3.30959.zip from host file:///dynamicScri
pt/repository/base/
CWPZT0600I: Command resolve was successful
CICSR4 @ SC66:/dynamicScript/apps/George/test>zero start
Application started and servicing requests at http://localhost:8486/
CWPZT0600I: Command start was successful
CICSR4 @ SC66:/dynamicScript/apps/George/test>

==> _
INPUT
ESC=¢ 1=Help 2=SubCmd 3=HlpRetrn 4=Top 5=Bottom 6=ISO
7=BackScr 8=Scroll 9=NextSess 10=Refresh 11=FwdRetr 12=Retrieve
21/007
Connected to remote server/host wtsc66.itso.ibm.com using lu/pool TCP66011 and port 23

```

Figure 7-42 Recycling and Resolving the test application

We perform another test. At this point, our test application is running again but now with ACF configured. By entering the same malicious input as our previous test in Figure 7-39 on page 167, we see the results in Figure 7-43 on page 171, showing a successful execution of our application without any unexpected results.

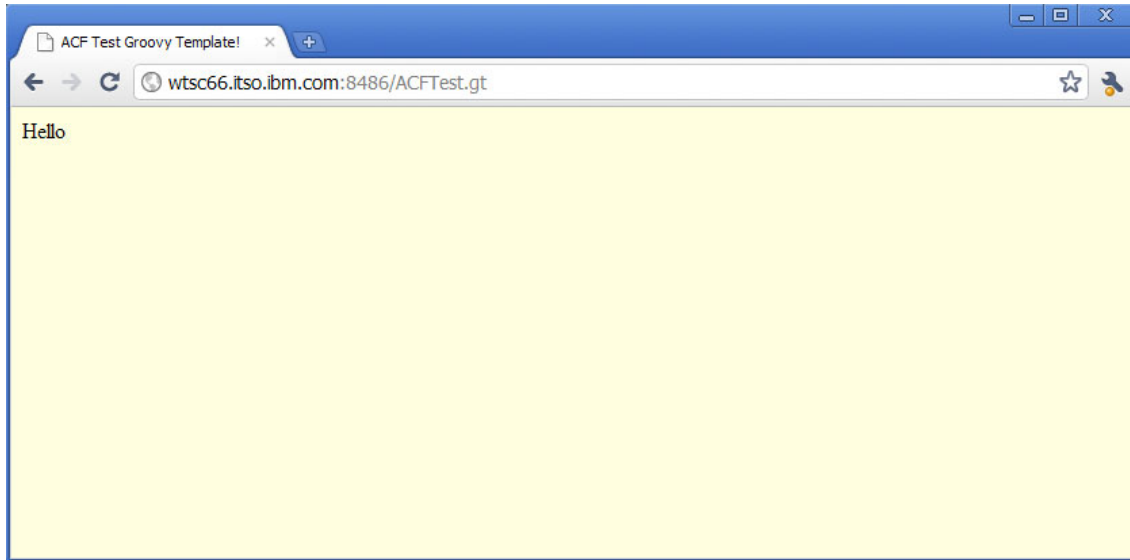


Figure 7-43 Successful malicious content filtering of ACFTest.gt ACF test script

Actually the output in Figure 7-43 does not look correct. It has no echoed output because ACF removed the `<script>` tags and all input included within the script-begin and script-end tags. Therefore, in this example there is nothing to display, so the output is correct and exactly what we would want displayed based on the input entered.

### Custom filter rules ACF example

This example shows how to implement custom filter rules to allow an application to provide a finer level of control over the active content filters that are provided by the default enablement you get by just turning on ACF.

We use the same test application from our previous example, “Browser redirection ACF example” on page 164, called `ACFTest.gt`. However, we use different input test data, which is not filtered out by the default enablement that is provided with ACF.

Therefore, we must first test our `ACFTest.gt` application with input that does not pass the default filters that are provided with ACF. The text listed in Example 7-32 uses several HTML tags to alter the text displayed on the browser.

*Example 7-32 Alternate malicious input to our ACFTest.gt test application*

---

here is some `<big><big>BIG</big></big>` text

---

This sample input uses tags that the Default Enablement does not filter out.

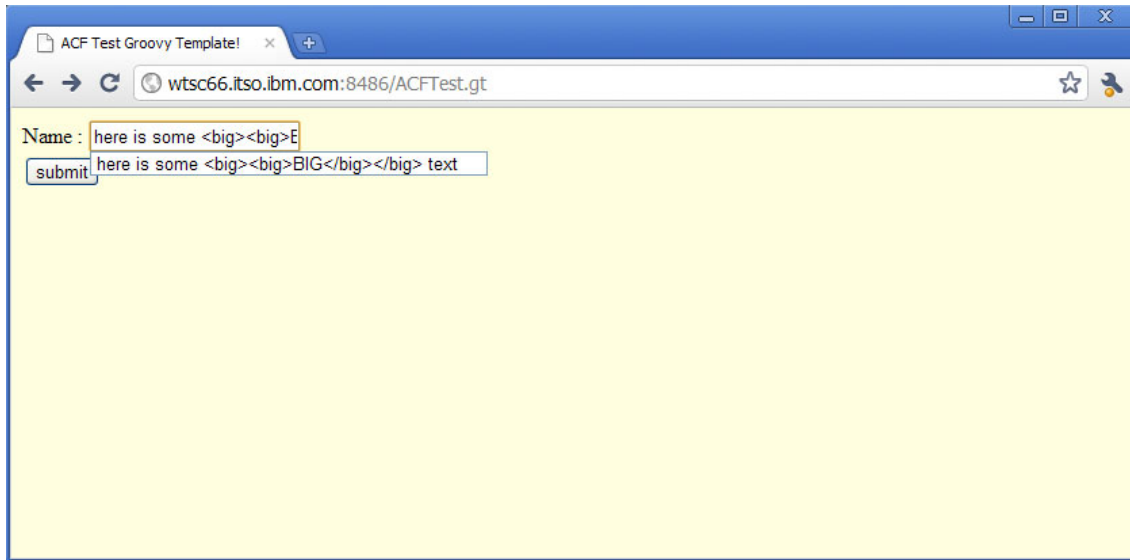


Figure 7-44 Entering malicious data into ACFTest.gt ACF test script

Although ACF is now turned on, the default enablement does not provide filter rules to handle this sample tag. Therefore, we expect to see the active content processed and also see some BIG text (Figure 7-45).



Figure 7-45 Expected results of ACFTest.gt ACF test script

What we can do to fix this exposure is add custom filtering rules to our ACF configuration to filter out this new tag.

The first step is to make a copy of the current default filter rules and then add new rules to filter out our sample tags. You can issue a ZERO REPOSITORY INFO command to find the path of your repository and also list the module groups. Using that information, you can look in the appropriate module group directory and then in the expanded/zero directory looking for the zero.acf directory that is related to the version of ACF that is in use. Located in that directory is a config/ directory that contains the provided filter rules files.

Because the filter rules file is large, Figure 7-46 shows only part of the file contents but includes the two lines that were added to filter out the <big> tags. Also note that the cursor was placed on the default filter rules file tab and thus it shows the location of where the default filter rules was copied from.

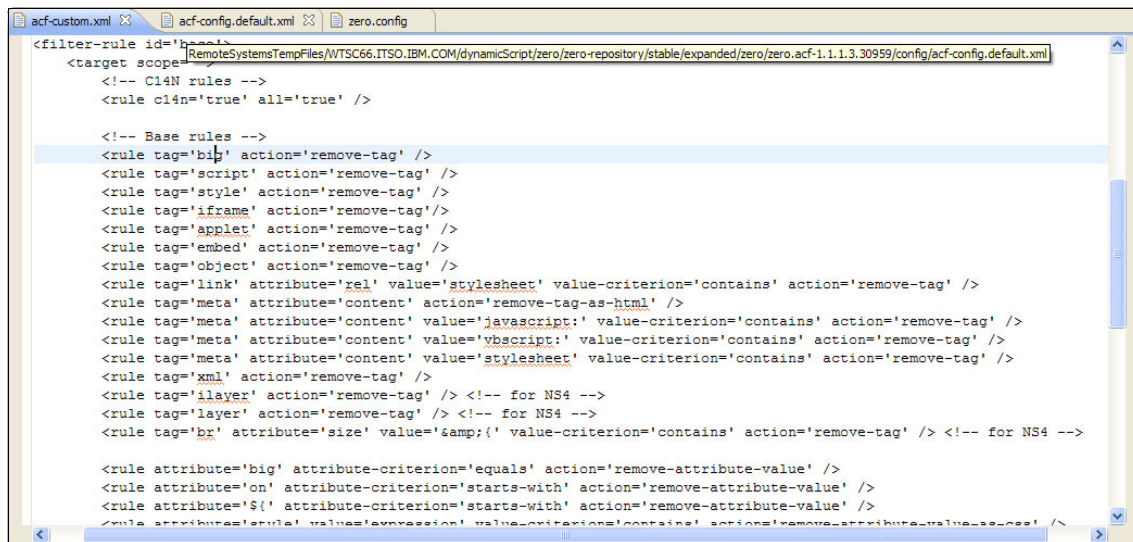
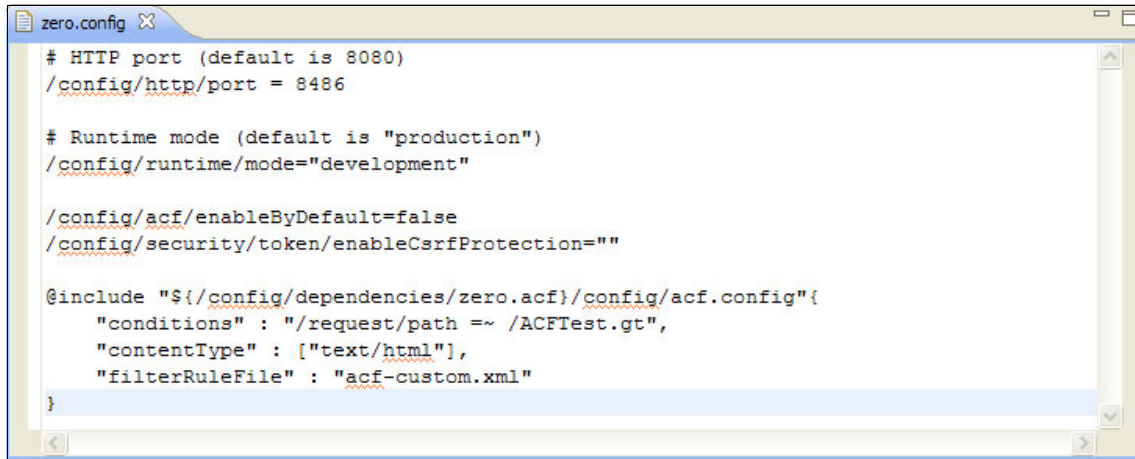


Figure 7-46 Custom filter rule file used in this example: acf-custom.xml

Saving our new acf-custom.xml filter rules file in the test applications config/ directory, we now update the test applications zero.config configuration file to tell ACF to use our new custom rules filter file. Figure 7-47 on page 174 shows the parameters required for performing the following tasks:

1. Turn off default enablement.
2. Disable CSRF.
3. Equate acf-custom.xml filter rules with the ACFTTest.gt script.

A screenshot of a text editor window titled 'zero.config'. The editor contains the following configuration code:

```
# HTTP port (default is 8080)
/config/http/port = 8486

# Runtime mode (default is "production")
/config/runtime/mode="development"

/config/acf/enableByDefault=false
/config/security/token/enableCsrfProtection=""

@include "${/config/dependencies/zero.acf}/config/acf.config"{
  "conditions" : "/request/path =~ /ACFTest.gt",
  "contentType" : ["text/html"],
  "filterRuleFile" : "acf-custom.xml"
}
```

Figure 7-47 *zero.config* modifications to enable Custom ACF filter rule

Saving our changes and a simple ZERO RECYCLE command is all that is needed to make our new changes active.

By using the same input as is shown in Figure 7-44 on page 172, we test our new filter rules. The results are shown in Figure 7-48.



Figure 7-48 *Successful test results using the acf-custom.xml filter rules*



Notice that ACF removed the <big> tags and all input included within the big begin and big end tags, so the output is displayed with all active content removed.

## 7.5 Local and remote repositories

When creating CICS Dynamic Scripting applications, part of the design benefits of CICS Dynamic Scripting, you will be sharing modules and resources from other applications.

IBM WebSphere sMash uses an open source tool, available through the Apache Software Foundation, called *Ivy* to manage application dependencies (recording, tracking, resolving, and reporting). CICS Dynamic Scripting applications rely on a configuration file in the apps/config z/OS UNIX directory called *ivy.xml* that contains references to modules that your application requires along with revision or version information.

Figure 7-49 is a sample *ivy.xml* configuration file.

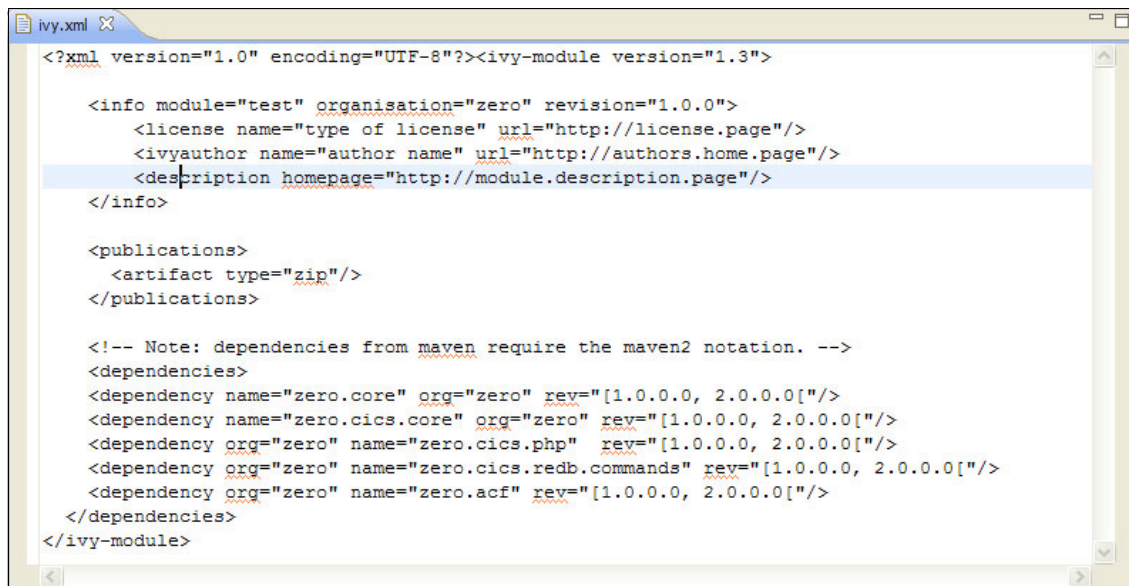


Figure 7-49 Sample *ivy.xml* configuration file

The *ivy.xml* file shows two mandatory dependencies. All CICS Dynamic Scripting applications must have a dependency on *zero.cics.core*, and if you want to use PHP then you must have a dependency on ***zero.cics.php***.

CICS Dynamic Scripting provides commands that help you manage your application dependencies:

- ▶ ZERO RESOLVE: Search and download modules matching version
- ▶ ZERO UPDATE: Resolve against all revisions ignoring version history
- ▶ ZERO ROLLBACK: Reverts affects of last Resolve or Update
- ▶ ZERO VERSION: List module dependency information

A local repository is created on your system to contain modules. All downloaded modules from remote repositories are placed into your local repository. The resolve and update commands search the local repository for modules and if not found will search remote repositories.

Module group commands can be used to update your local repository with later revisions when they become available on a remote repository.

### 7.5.1 Setting up a local shared repository

Additional local repositories, with the purpose of sharing modules, can also be created and handcrafted to include only the modules that you selectively publish to it. The repository commands are used to publish modules to a shared local repository.

Use the following command flow to package a module a publish it to a private local repository for sharing:

1. ZERO PACKAGE: Create a ZIP file of your application
2. ZERO REPOSITORY PUBLISH <path>: Publish it to a private repository

After you have a private repository setup with modules and artifacts, it can be hosted by any web server that provides directory browsing and indexing.

See the IBM WebSphere sMash information center; search on “Creating a repository for sharing modules” for more information.

## 7.6 Packaging, publishing, sharing CICS Dynamic Scripting applications

IBM WebSphere sMash and CICS Dynamic Scripting provides you with several commands that can help with building applications and sharing code, known as *modules*. An application uses an `ivy.xml` configuration file to identify dependencies required by the application.

Your system contains a local repository that was created to contain sharable modules. When you issue a ZERO RESOLVE command, the local repository is searched for matching dependencies with the version that you require. If the dependency cannot be found locally, then the configured remote repositories are searched and if found on a remote repository, the module is downloaded into the local repository. A ZERO RESOLVE command ends successfully when all required modules are found in the local repository.

Using the ZERO CREATE command, you can create new applications and optionally copy all of the dependencies, (`linkto` option), or copy all resources and dependencies, (`from` option), from an existing application.

During application build, you must issue ZERO RESOLVE commands when you add or change dependencies in your `ivy.xml` configuration file. For each dependency you define, you list revision information, which means the dependency module can become dated, you can use the ZERO UPDATE command to resolve against modules that have newer versions. Additionally you also have the ability to revert to an older version by using the ZERO ROLLBACK command.

After you have created an application, you can share it by compressing the entire application directory structure as a package by using ZERO PACKAGE command or by publishing your application to the local repository by using ZERO PUBLISH.

The ZERO PACKAGE command creates a ZIP file of your application that can then be moved to another location and expanded. The ZERO PUBLISH command publishes the ZIP file created by the ZERO PACKAGE command to the local repository.

## 7.7 Movement from development to production

As with normal CICS applications, you have multiple environments and typically move or migrate them from development, to quality test, to production. Although CICS Dynamic Scripting applications experience the same movement, because the applications are dynamic, there are additional customizations to possibly adjust the runtime configuration files. The steps are divided into two parts:

- ▶ Moving the application files and resources
- ▶ Configuring the applications target CICS run time.

### Moving application resources between z/OS UNIX systems

CICS Dynamic Scripting applications are stored in z/OS UNIX files and directories. Because they are simple files and directories you can use any suitable method of moving files, but CICS Dynamic Scripting provides you with a few options. Specifically ZERO PACKAGE will create a single ZIP file of your entire application.

Use the following steps to move your application from one z/OS UNIX to another:

1. Enter ZERO PACKAGE to create a ZIP file of your application, note the following options:
  - SHARED: The ZIP package does not include dependencies.
  - STANDALONE: The ZIP package includes dependencies.
  - INCLUDESRC: Add the java directory to your package.
2. Transfer the package, located in your applications export/ directory, to your target z/OS UNIX system by using a suitable transfer method.
3. Expand the application ZIP file in its new target directory; you can use JAR.
4. Ensure the target directories and files have the proper permissions set.
5. If you used the SHARED option, issue ZERO RESOLVE or ZERO UPDATE to resolve all application dependencies against your new target environment.
6. Continue with step 2 in “Moving applications between CICS regions” on page 178.

### Moving applications between CICS regions

CICS Dynamic Scripting applications are loaded from z/OS UNIX and run inside a CICS region, when promoting or moving an application, you perform the configuration on the z/OS UNIX side and at ZERO START time. The system resources needed to run your CICS Dynamic Scripting application are dynamically created, specifically JVMServers, URIMaps, TCPIPService and PIPELine definitions.

Use the following steps to change the target CICS region:

1. Enter ZERO STOP to stop the application. All CICS resources, (JVMServers, URIMaps, TCPIPService and PIPELine definitions), are dynamically deleted.
2. Edit the applications configuration files:
  - The `zerocics.config` file to update the CICS Applid or any other applicable settings
  - The `zero.config` file to update any applicable settings
3. Using existing CICS Systems Programmer procedures, migrate or move any required CICS resources that your application references.
4. Enter ZERO START to start the application, note if your application has moved then the ZERO START command must be run from the new location.

Step 3 is vague and can vary for each application. If your CICS Dynamic Scripting application links to any existing programs or reads a VSAM file or other CICS resource, these must be considered and migrated accordingly.

To learn what resources are associated with a dynamic scripting application, you can use any of the following methods:

- ▶ If you are in z/OS UNIX, check the application log file. The log file is in the `application_home/logs` directory. It contains the ID of the JVMSEVER resource that was created when the application started. The numeric part of the ID is the same for all the CICS resources associated with the application.
- ▶ If you are using CICS, check the shelf directory of the PIPELINE resource. The path to the shelf directory indicates which application the PIPELINE resource supports.





# Troubleshooting

In this chapter, we describe how to solve the problems in setting up the environment of CICS Dynamic Scripting.

## 8.1 Repository access problems

As noted in 5.7, “Repository access zerocics.config” on page 66, Dynamic Scripting Feature Pack requires access to a configured code repository to be able to function.

IBM provides a repository on the Internet, although it might not be accessible from your mainframe. Firewalls might block direct access to the Internet, for example, or the Domain Name Servers might not be configured to allow access to external sites. See Example 8-1.

### *Example 8-1 Example of error contacting repository*

---

```
CWPZT0901I: The following module(s) are not currently in the local
repository:
    zero:zero.cli.tasks:[1.0.0.0,2.0.0.0[
CWPZT0902I: Trying to locate the module(s) using one of the configured
remote repositories
CWPZT0911E: Retrieving module zero:zero.cli.tasks from host
http://download.boulder.ibm.com/ibmdl/pub/software/htp/cics/updates/ds/
1
.0/repo/base resulted in the following error
    "EDC8128I Connection refused."
CWPZT0915E: The module zero:zero.cli.tasks can not be located in any of
the configured repositories
CWPZT0808E: The resolve report has errors, the resolved properties will
not be created.
CWPZT0506E: Error : unresolved dependency:
zero#zero.cli.tasks:[1.0.0.0,2.0.0.0[: not found
CWPZI8503I: CICS unit of work rolled back after a CLI task returned a
non-zero return code.
initializing the cli failed
```

---

A variation of this error is as follows:

```
java.net.UnknownHostException
```

If opening an HTTP port from the mainframe to either the IBM repository or any other server is not possible, you may still proceed.

See “Downloading the repository on the mainframe” on page 183 for instructions of how to download the repository on the mainframe.



## Downloading the repository on the mainframe

If opening an HTTP port from the mainframe to either the IBM repository or any other server is not possible, you may still proceed as follows:

1. Download the IBM repository ZIP file to your own notebook.

[http://download.boulder.ibm.com/ibmdl/pub/software/htp/cics/updates/ds/1.0/cics\\_dynamic\\_scripting\\_repo\\_v1000.zip](http://download.boulder.ibm.com/ibmdl/pub/software/htp/cics/updates/ds/1.0/cics_dynamic_scripting_repo_v1000.zip)

The file is approximately 343 MB in size.

2. Using FTP, transfer the following repository ZIP file to a suitable location on the mainframe:

`cics_dynamic_scripting_repo_v1000.zip .`

The file must be transferred in binary mode.

3. On the mainframe, use the Java `jar` command to inflate the compressed file. Two directories, `base` and `samples` are created.
4. In the `zero/config` directory, edit the `bootstrap.properties` file:
  - Example 8-2 shows ISPF commands to change the location of the repository.
  - The protocol `http://` is changed to `file:///`.
  - The path must include the initial forward slash to use an absolute path.

*Example 8-2 ISPF Edit commands to change repository locations*

---

```
c
http://download.boulder.ibm.com/ibmdl/pub/software/htp/cics/updates/
ds/1.0/repo/ ##### all
c ##### file:///dynamicScript/repository all
```

---

## 8.2 Common errors (commands, code page, coding)

This section lists common problems related to installation, command-line interface, encoding, and setting up JVMSERVER.

### 8.2.1 Installation problems

The following errors might occur when you install the feature pack or create a test application.

If you type the `ZERO` command, and an error occurs (Example 8-3 on page 184), you have not set up zero library in your UNIX System Services environment.

---

*Example 8-3 zero command*

---

zero: cannot execute

---

Confirm you have set up correct JDK path, zero library path and STEPLIB path in a shell script and executed it in the correct way. A sample for the shell script is in Example 8-4.

---

*Example 8-4 Sample for shell script*

---

```
export PATH=$PATH:/usr/lpp/java/J6.0/bin/  
export ZERO_HOME=/dynamicScript/zero/  
export PATH=$ZERO_HOME:$PATH  
export STEPLIB=CICSTS41.CICS.SDFHEXCI:CICSTS41.CICS.SDFHLOAD  
export JAVA_HOME=/usr/lpp/java/J6.0/
```

---

Perhaps you enter the ZERO command and receive an EXCI error, as shown in Example 8-5.

---

*Example 8-5 Zero command*

---

```
CWPZI8704E: The CICS EXCI INIT_USER() call was unsuccessful. CICS  
NETNAME=ZEROCLI IYK3Z0HQ. Return codes are as follows:  
CWPZI8711E: Response code= 12 Reason code= 430 Sub-reason= 0.
```

---

Perform the following steps:

1. Using the MVS reason code, determine why the load failed. The most likely reason is that DFHXCPRX is not in any library included in the STEPLIB of the batch job. Ensure that the library containing the module is included in the STEPLIB concatenation.
2. If the STEPLIB concatenation is correct and the user ID has access to the STEPLIB EXCI libraries, check that the file permissions are correct in the zero directory. The user ID and the CICS region ID must belong to the same z/OS UNIX group. Listing the files in a directory displays the owner and the group. To change the group for the zero directory, enter the following command:  
  
chgrp -R group zero
3. If the previous two steps are confirmed, determine whether you have set up the CICS STEPLIB in your UNIX System Services environment properly. You must create a shell script and input the line, listed in Example 8-6, in it; then ensure you have run the script in the correct way.

---

*Example 8-6 STEPLIB*

---

```
export STEPLIB=CICSTS41.CICS.SDFHEXCI:CICSTS41.CICS.SDFHLOAD
```

---

If you type the ZERO command, and the response shown in Example 8-7 occurs, you have not upgraded your CSD with CICS Dynamic Scripting feature pack.

*Example 8-7 ZERO command*

---

```
CWPZI8854E: Unable to open Java return code file
/u/liuguan/smash/zero/IYK3Z0HQ.
JC000010.java_rc: EDC5129I No such file or directory. <129>.
CWPZI8735E: Request failed. ZEROICICS return code=0, ZEROSERV return
code=205.
initializing the cli failed
```

---

Ensure you have installed Dynamic Scripting feature pack on CICS 4.1.

## 8.2.2 Command-line interface problems

The command-line interface uses EXCI to connect to the CICS region. Each time a command runs, the feature pack creates a JVM server to handle the work in CICS. This JVM server exists until the command completes. If you have problems with the command-line interface, you might not have the correct authority to access EXCI or create resources in CICS. To determine why a ZERO command is failing, turn on debugging to produce extra messages by entering `export CICS_DEBUG=ON`.

You receive EXCI errors when you run command-line interface commands. First, we introduce the general debug process for command-line interface problems. For example, you might see the error messages in Example 8-8:

*Example 8-8 Error message*

---

```
CWPZI8707E: The CICS EXCI link was unsuccessful. EXCI return codes are
as follows:
CWPZI8711E: Response code=8 Reason code=203 Sub-reason=0.
CWPZI8735E: Request failed. ZEROICICS return code=121, ZEROSERV return
code=-1.
```

---

Every ZERO command in the feature pack requires a CICS region. Response 8 reason 203 indicates a NO\_CICS condition, for example, the CICS region could not be found. You can determine which CICS region the feature pack is using:

1. Enter `export CICS_DEBUG=ON` to turn on debugging.
2. Enter the top-level command ZERO. Example 8-9 on page 186 shows the additional diagnostics that return information about the feature pack.

*Example 8-9 ZERO command diagnostics*

---

```
CICS_DEBUG(C) [2010-05-12 10:47:47]: find_zerocics: zerocics.config
found at [/user1/workspace/trunk/test/zero/config/zerocics.config]
CICS_DEBUG(C) [2010-05-12 10:47:47]: get_CICS_info(): config file
path = "/user1/workspace/trunk/test/zero/config/zerocics.config"
CICS_DEBUG(C) [2010-05-12 10:47:47]: get_CICS_info(): CICS_APPLID =
"DYNACICS"
CICS_DEBUG(C) [2010-05-12 10:47:47]: get_CICS_info(): CICS_NETNAME =
"ZEROCLI"
CICS_DEBUG(C) [2010-05-12 10:47:47]: get_CICS_info(): CICS_CLIUSER =
"ZERouser"
CICS_DEBUG(C) [2010-05-12 10:47:47]: get_CICS_info(): CICS_TRANSID =
"ZERO"
CICS_DEBUG(C) [2010-05-12 10:47:47]: Allocate pipe:
CICS_APPLID="DYNACICS" CICS_NETNAME="ZEROCLI"
```

---

Note the following information:

- CICS\_APPLID: The CICS APPLID that is used by the feature pack.
  - CICS\_NETNAME: The CICS EXCI netname that is used by the feature pack.
  - CICS\_CLIUSER: The user ID that runs all commands from the command line.
3. If the CICS region is incorrect, edit the zerocics.config file. This file can be located in two places:
- /application\_home/zerocics.config  
If the file is present in the directory, this configuration file replaces the global configuration file in the zero directory.
  - /installation directory/zero/zerocics.config  
This configuration file is required, and contains the default settings for all dynamic scripting applications.
4. If the CICS region is correctly specified, verify that the following conditions are satisfied:
- The CICS region has an EXCI connection.
  - The EXCI connection is in service.
  - The EXCI connection has a netname.
  - The netname matches the CICS\_NETNAME.
  - IRC is started in the CICS region.

These steps are general debug steps for command-line interface. You can debug most of problems with a similar process. The following errors might occur when you run commands from the command line.

If you type command **ZERO** but it hangs without any response, you might receive a message from the member JESMSGGLG of CICS log. See Example 8-10.

---

*Example 8-10 JESMSGGLG CICS Log*

---

```
DFHAP1500 EPRED222 The CICS time-of-day is no longer synchronized with  
the system time-of-day
```

---

The message shows that the time of the CICS region is not synchronized with the system time of z/OS. The reason might be because the system time of z/OS is reset. You must then synchronize them with the CEMT **PERFORM RESET** command on the CICS region.

If you enter the **zero start** command, and the response shown in Example 8-11 occurs, you did not execute the command under the directory of dynamic scripting application. You must go to the application directory and execute it again.

---

*Example 8-11 Zero start command*

---

```
CWPZC0019E: Cannot find handler for task start  
CWPZI8503I: CICS unit of work rolled back after a CLI task returned a  
non-zero return code.
```

---

If you type **zero start** command, and the response shown in Example 8-12 occurs, the **ZERO** command was unable to communicate with your CICS region.

---

*Example 8-12 Zero start command*

---

```
CWPZI8706E: The CICS EXCI OPEN_PIPE() call was unsuccessful. Return  
codes are as follows:  
CWPZI8711E: Response code= 8 Reason code= 203 Sub-reason= 104.
```

---

The response might be for the following reasons:

- ▶ The group of **ZERO** resource has not been installed in your CICS region. Issue **CEDA INSTALL GROUP(DFH\$ZERO)** from your CICS region.
- ▶ Intersystem communication (ISC) is not started. Ensure that you have modified your SIT parameters in your CICS region that is running.
- ▶ Your CICS region is not started; check it and start it.

You receive a CWPZI8706E error message with a reason code of 203 and subreason of 92, as shown in Example 8-13.

*Example 8-13 CWPZI8706E error message*

---

CWPZI8706E: The CICS EXCI OPEN\_PIPE() call was unsuccessful. EXCI return codes are as follows:  
CWPZI8711E: Response code=8 Reason code=203 Sub-reason=92.

---

The reason is because the CICS region is not started. Check that the CICS region is running.

You receive a CWPZI8707E message and a DFHAC2008 message from CICS as shown in Example 8-14.

*Example 8-14 CWPZI8707E message*

---

CWPZI8707E: The CICS EXCI link was unsuccessful. EXCI return codes are as follows:  
CWPZI8711E: Response code=12 Reason code=414 Sub-reason=0.  
CWPZI8712E: The following message was received from the target CICS system:  
DFHAC2008 05/12/2010 11:11:07 IYCSZAW5 Transaction ZERO has been disabled and cannot be used.  
CWPZI8735E: Request failed. ZEROCICS return code=121, ZEROSERV return code=-1.

---

The ZERO transaction is disabled in the CICS region. Enable the ZERO transaction and try again.

You receive a CWPZI8707E message and a DFHAC2001 message from CICS see Example 8-15:

*Example 8-15 CWPZI8707E message*

---

CWPZI8707E: The CICS EXCI link was unsuccessful. EXCI return codes are as follows:  
CWPZI8711E: Response code=12 Reason code=414 Sub-reason=0.  
CWPZI8712E: The following message was received from the target CICS system: DFHAC2001  
05/12/2010 11:12:30 IYCSZAW5 Transaction 'ZERO' is not recognized.  
Check that the transaction name is correct.  
CWPZI8735E: Request failed. ZEROCICS return code=121, ZEROSERV return code=-1.

---

The ZERO transaction is not installed in the CICS region. Install the ZERO transaction to run command-line interface commands.

You enter a command and receive a CWPZI8707E message and a reason code of 423 see Example 8-16:

---

*Example 8-16 CWPZI8707E message*

---

CWPZI8707E: The CICS EXCI link was unsuccessful. EXCI return codes are as follows:  
CWPZI8711E: Response code=12 Reason code=423 Sub-reason=8.  
CWPZI8735E: Request failed. ZEROCICS return code=121, ZEROSERV return code=-1.

---

You also receive a ICH408I message in the system log. You do not have surrogate authority to enter CICS\_CLIUSER commands. See the “Configuring CICS security” topic in the information center.

If you type **zero start** command, and the response shown in Example 8-17 occurs, your application failed improperly last time. In another words the application stopped without the **zero stop** command last time. For example, the region shut down, but the application was running at that time. You must start it with the **-force** parameter, such as the **zero start -force** command.

---

*Example 8-17 ZERO START command*

---

CWPZC8136E: The application status is currently UNKNOWN, perhaps as the result of system crash or other error. To force application to start, use "-force" parameter.  
CWPZT0601E: Error: Command start failed  
CWPZI8503I: CICS unit of work rolled back after a CLI task returned a non-zero return code.

---

If you get the response shown in Example 8-18, the resource TCPIPSERVICE is not created successfully because port number 08680 cannot be opened. Determine whether this port is already used by a previously installed TCPIPSERVICE with the CEMT I TCPIPS command.

---

*Example 8-18 CWPZI8847E error*

---

CWPZI8847E: Open of CICS TCPIPSERVICE TP000011 on port number 08680 failed. Previously installed resources will be discarded.

---

Update the new port and try to start the application again. If it still fails, followed by the same error messages about port number, execute the **zero stop** command to clear the application in advance and then try to start it.

The command-line interface commands can access CICS, but you are experiencing errors. Use CEDX on the ZERO transaction to diagnose the problem. You can use CEDX to confirm that the ZERO transaction can run in the CICS region and determine which command is failing.

### 8.2.3 Dynamic scripting encoding consideration

Dynamic scripting is primarily a UTF-8 platform that runs in an EBCDIC environment. You must edit text files in the correct format and write application code that converts data between bytes and character data correctly.

**Note:** English characters are the same in UTF-8, ASCII, and ISO8859-1.

The feature pack encodes all log files in UTF-8 on z/OS UNIX. Similarly, all files read from z/OS UNIX, including PHP and Groovy script files, Java source files, static HTML files, JSON data files, and configuration files are expected to be encoded in UTF-8. Therefore, you must view and edit these files by using a suitable method. Several methods are as follows:

- ▶ Use the z/OS UNIX directory list utility in ISPF (option 3.17). Enter the **ea** command to edit an ASCII file.
- ▶ Use the Remote Systems Explorer in Rational Developer for System z to edit an ASCII file in z/OS UNIX. The tool defaults to EBCDIC.
- ▶ Use the Target Management plug-in in Eclipse to modify an ASCII file. This tool defaults to ASCII. The plug-in is located at the following address:

<http://eclipse.org/dsdp/tm/>

- ▶ Enter the **iconv** command in z/OS UNIX to modify an ASCII file.

For example, if the port mentioned in last section is already used, you must configure another available one in `zero.config` file for your application. Example 8-19 shows the command for converting `zero.config` from ASCII to EBCDIC, and the new `zero.config` file. EBCDIC is encoded in EBCDIC.

*Example 8-19 EBCDIC is encoded*

---

```
iconv -f ISO8859-1 -t IBM-1047 zero.config >zero.config.EBCDIC
```

---

And then we can read and modify `zero.config.EBCDIC` file using **oedit** command now. Update the port and convert it back to ASCII with the following command see Example 8-20.

*Example 8-20 Command*

---

```
iconv -f IBM-1047 -t ISO8859-1 zero.config.EBCDIC >zero.config
```

---



## 8.2.4 Setting up JVMSERVER

IBM Java 6 SR8 or later with 31 bits is required for setting up JVMSERVER. JDK editions earlier than Java 6 or Java 6 with 64 bits are not available for dynamic scripting applications.

In general, if JVMSERVER cannot be enabled, you might not have set the correct value for CICS\_HOME and LIBPATH\_PREFIX.

To debug, perform the following steps:

1. Check the CICS output member MSGUSR and find the error messages about JVMSERVER. Example 8-21 shows the JVMSERVER disabled and then discarded because the Language Environment enclave was not created.

---

### *Example 8-21 JVMSERVER disabled*

---

```
An attempt to start a JVM for the JVMSERVER resource JC000004 has
failed. Reason code: OPEN_JVM_ERROR.
JVMSERVER JC000004 is DISABLED because the Language Environment
Enclave was not created.
JVMSERVER JC000004 was successfully discarded.
```

---

2. To see more error messages, check the CICS output member CEEMSG. Example 8-22 is from CEEMSG. From the error messages, you can see that you set this up incorrectly, using JDK6.0 with 64 bits. It must be changed to the one with 31 bits.

---

### *Example 8-22 Error message*

---

```
*exc* libjvm.so not found
DLL error = CEE3588S A call was made to a function in the AMODE 64
DLL libjvm..so from an AMODE 31 caller.
IPT: *exc* Error creating JVM
```

---

3. Go to the directory of your dynamic scripting application and find the JVM error file that has a name in the following format, (where nnnnnn is the 6-digit ascending number discussed previously):

```
<cics_region_applid>.JCnnnnnn.dfjhvmerr
```

Open it to see more details in this file.

4. Update JVMProfile and JVMServer characters in zerocics.config according to the error from the previous steps. In this example, JDK must update JDK6.0 with 31 bits instead of the one with 64 bits.

If you ensure CICS\_HOME and LIBPATH\_PREFIX are both correct, consider the following errors.

You receive a CWPZI8839E message (Example 8-23) indicating that the feature pack was unable to create a JVMSERVER resource.

*Example 8-23 CWPZI8839E message*

---

CWPZI8839E: Creation of CICS JVMSERVER JC000004 failed: EIBRESP=NOTAUTH, EIBRESP2=100.  
Previously installed resources will be discarded.  
CWPZI8735E: Request failed. ZEROICICS return code=0, ZEROSERV return code=202.

---

You are not authorized to create a JVMSERVER resource. Check that your z/OS UNIX user ID has the correct authority to create resources in CICS. See the “Configuring CICS security” topic in the information center.

You receive a CWPZI8840E message (Example 8-24) indicating that the feature pack failed to enable the JVMSERVER resource.

*Example 8-24 CWPZI8840E message*

---

CWPZI8840E: Failed to enable CICS JVMSERVER JC000008.  
CWPZI8735E: Request failed. ZEROICICS return code=0, ZEROSERV return code=203.

---

Check the system log for additional messages, such as DFHAP1600, and then check the dfhjvmerr file in the current directory where you entered the command. Example 8-25 shows that the value specified for the -Xmx parameter is too large.

*Example 8-25 DFHAP1600 message*

---

12 May 2010 12:10:01 JC000007 START\_JVM: \*exc\* Create JVM failed  
12 May 2010 12:10:01 JC000007 START\_JVM: \*exc\* Unable to create JVM.  
12 May 2010 12:10:01 JC000007 Return code is -4 meaning: Not enough memory  
12 May 2010 12:10:01 JC000007 IPT: \*exc\* Error creating JVM

---

If you cannot enable JVMSERVER, the errors shown in Example 8-26 occur.

*Example 8-26 JVMSERVER error*

---

CWPZI8840E: Failed to enable CICS JVMSERVER JC000013.  
CWPZI8817E: STAT() on CICS JVMSERVER work file DFHJVMOUT failed: EDC5129I No such file or directory. <129>. File not deleted.  
CWPZI8818E: STAT() on CICS JVMSERVER work file DFHJVMERR failed: EDC5129I No such file or directory. <129>. File not deleted.  
CWPZI8735E: Request failed. ZEROICICS return code=0, ZEROSERV return code=203.

---

If you also get the message in the MSGUSR of the CICS log, shown in Example 8-27, the total THREADLIMIT of the JVMSERVER on your region is up to the maximum value of 1024. Recycle the application to re-create the JVMSERVER resource with the new value.

*Example 8-27 DFHAM4945 error*

---

DFHAM4945 W EPRED2 JVMSERVER JC000013 has been installed as disabled with a THREADLIMIT of 0.

---

You have succeeded in starting your application if you get the response shown in Example 8-28.

*Example 8-28 Application started*

---

Application started and servicing requests at http://localhost:8683/  
CWPZT0600I: Command start was successful

---

You can now enter the application in your browser (http://hostname:8683); you see the response shown in Figure 8-1.

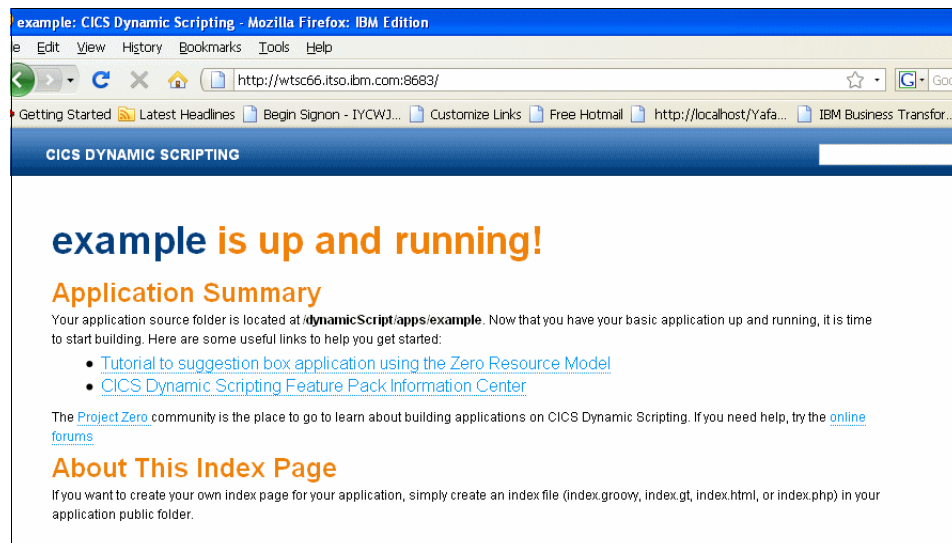


Figure 8-1 Main application window

## 8.3 Support considerations

A Dynamic Scripting problem in CICS can occur for many reasons. The most common problems are documented in 8.2, “Common errors (commands, code page, coding)” on page 183, or the CICS TS V4.1 information center. However, if you cannot solve the problems by the information in this book or from the information center, ask for help from the IBM CICS support team. You need to know what documentation the CICS support team will require to help you diagnose your problem. Be sure to gather this documentation before calling support to expedite the troubleshooting process, saving you time. This section describes what information to document.

### 8.3.1 Collecting documentation required by the support team

Gather documentation for your Dynamic Scripting problem and then call IBM support.

#### **Required documentation**

MustGather and JVM Server standard files are required.

#### ***MustGather information***

The following list describes what to do for installation and application issues:

- ▶ For installation issues, provide the following information:
  - The operating system where the issue occurred
  - The target installation directory, <install\_root>, where IBM WebSphere sMash is installed
  - Logs to collect:
    - From <install\_root> directory:  
install.log  
rte\_install.log (if available)
    - From the root tmp directory:  
For Linux: /tmp/debug.log  
For Windows: c:/tmp/debug.log

- The version of Java Development Kit (JDK) installed:  
`java -version`
- Confirmed network connectivity to the following address:  
<http://www.projectzero.org>
- For Windows, the value of the system PATH:  
`set PATH`
- For application issues, such as issues with running applications, provide the following information:
  - A description of the problem and re-creation steps
  - A simple sample where possible
  - A list of the folders under <install\_root>/zero-repository  
 The content of these folders is not required.
  - The application root directory, <app\_root>  
 This is the directory where the application is installed, for example:  
 <install\_root>/apps/<application\_name>
  - Files to collect (compress all files in these directories):  
 <app\_root>/zero  
 <app\_root>/config  
 <app\_root>/logs  
 <app\_root>/javacore\* [any Java core dumps]  
 <app\_root>/p8trace\* [any p8 trace files]
- If IBM support requests more detailed logging and tracing, two options are available:
  - Create an application specific logging.properties file:
    - i. Copy <install\_root>/config/logging.properties to the following location:  
 <app\_root>/config/logging.properties
    - ii. Change .level=INFO to .level=FINE in the following location:  
 <app\_root>/config/logging.properties
  - Specify -loglevel at application start by using the following command:  
 zero -loglevel=FINE start

If support requests more detailed logging for PHP specific issues, perform the following steps:

- a. Edit the <app\_root>/config/php.ini file.
- b. Change directive error\_reporting to a value specified by support, for example:

```
error_reporting = E_ALL | E_STRICT
```

- c. Verify the following statement:

```
log_errors = On
```

For more detailed information about MustGather, go to the following address:

<http://www.ibm.com/support/docview.wss?uid=swg21306004>

### ***JVM Server standard files***

If available, gather the JVM server standard out and standard error files. These files are created in the JVM server working directory. For CLI JVMs, this directory is the one from which you enter the command. For application JVMs, this directory is the root directory of the application.

The files are named as follows, where applid is the APPLID of the CICS region and JVMServerID is the name of the JVMSERVER resource:

- ▶ applid.JVMServerID.dfhjvmerr
- ▶ applid.JVMServerID.dfhjvmout

### **Optional documentation**

If IBM support requests more detailed logging of the interaction between Dynamic Scripting and the CICS region, perform the following steps:

1. Switch on debug model.

Enter **export CICS\_DEBUG=ON** to set the environment variable CICS\_DEBUG to ON in the z/OS UNIX shell before starting the application or running the problematic CLI command. This step result in extra logging being sent to the console and the log files.

2. Provide the contents of JVM profile directory.

This directory is defined by the JVMPROFILEDIR system initialization parameter in your CICS region.

### 8.3.2 Submitting documentation to the support team

To learn about using FTP and email for exchanging information with IBM Technical Support, using the IBM Enhanced Customer Data Repository (ECuRep), go to the following address:

<http://www.ibm.com/support/docview.wss?uid=swg21204910>

Go to ServiceLink or IBMLink to open an Electronic Technical Response (ETR). If you need instructions, go to the following address:

<http://www.ibm.com/support/docview.wss?uid=swg21220091>

To speak with an IBM technical support representative, perform either of the following steps, and always update the PMR to indicate that data has been sent.:

- ▶ If you are not in the U.S., call your country representative.
- ▶ If you are in the U.S., call 1-800-IBM-SERV.

### 8.3.3 Troubleshooting tips

The following steps might help you with diagnostics and troubleshooting:

1. Review the logs and dumps generated at the point of failure.
2. Search the CICS support site (such as information center) for known problems using symptoms like the message number and error codes.
3. If you find a fixing PTF, see the following website for the options that are available to order for CICS maintenance:  
<http://www-01.ibm.com/support/docview.wss?uid=swg21049360>
4. Gather the documentation and work with the CICS Level 2 support team to resolve your problem.

### 8.3.4 Learning more about the problem or component

For more information, see the following resources:

- ▶ Troubleshooting and support for Dynamic Scripting in CICS TS 4.1 information center:  
<http://publib.boulder.ibm.com/infocenter/cicsts/v4r1/index.jsp?topic=/com.ibm.cics.ts.smash.doc/troubleshooting.html>
- ▶ Project Zero forum to ask questions about the sMash programming model:  
<http://www.projectzero.org>

## 8.4 Health checker

The options that you specify in JVM profiles for storage heaps and garbage collection can have a significant influence on the performance of your Java applications, and on the number of JVMs that you can have in a CICS region. You can use the output from the garbage collection process in the JVM to tune these settings.

The information given in this section introduces IBM Monitoring and Diagnostic Health Center tool and provides basic guidance to help you tune JVMs in a CICS environment. The outcome of your tuning can vary depending on your Java workload, the maintenance level of CICS and of the IBM SDK for z/OS, and other factors. For more detailed information about the storage and garbage collection settings, and the tuning possibilities for JVMs, download the *IBM Developer Kit and Runtime Environment, Java Technology Edition, Diagnostics Guide* from the following address:

<http://www.ibm.com/developerworks/java/jdk/diagnosis/>

### 8.4.1 IBM Monitoring and Diagnostic Tools for Java - Health Center

The IBM Monitoring and Diagnostic Tools for Java - Health Center is a free low-overhead diagnostic tool for monitoring an application running on an IBM Java Virtual Machine.

#### Overview

The Health Center tool is provided in two parts:

- ▶ The Health Center client is a GUI-based diagnostics tool for monitoring the status of a running Java Virtual Machine (JVM), installed within the IBM Support Assistant (ISA) Workbench.
- ▶ The Health Center agent provides the mechanism by which the Health Center client obtains information about your Java application. The agent uses a small amount of processor time and memory and must be manually installed in an IBM JVM.



Figure 8-2 shows where the Health Center client and agent are located, when installed in ISA and the JVM.

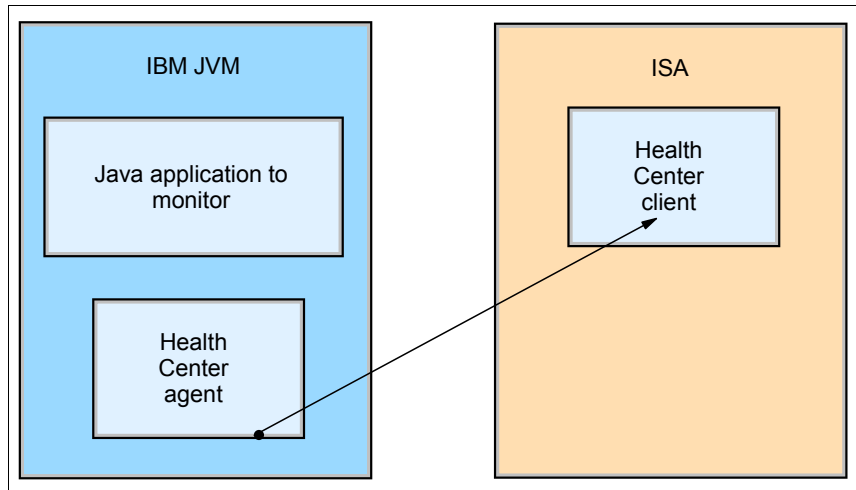


Figure 8-2 Location of Health Center client and agent

## Installing the Health Center client

To install the client, perform the following steps:

1. Download and install the IBM Support Assistant Workbench from the following address:
2. Install the Health Center in the ISA Workbench:
  - a. Start performing the steps at the following address (How to Install and Run Tools through the IBM Support Assistant):

<http://www.ibm.com/software/support/isa/>

<http://www.ibm.com/support/docview.wss?uid=swg27013279>

However, when you arrive at the “Find new tools add-ons” wizard (Figure 8-3 on page 200), type `health` in the search box, and then click the arrow next to JVM-based Tools to show matching tools.

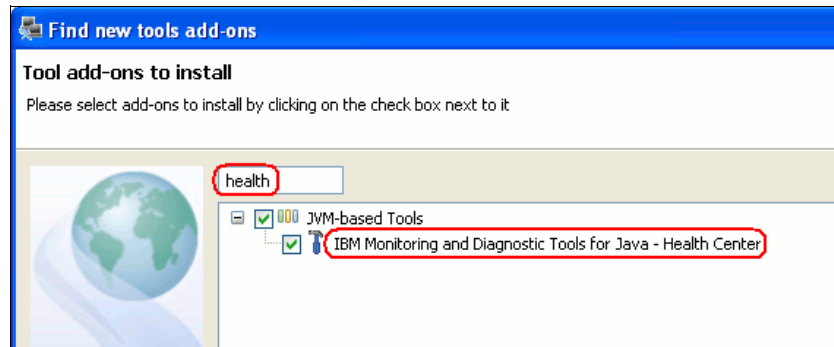


Figure 8-3 Find for the Health Center add-on

- b. Select **IBM Monitoring and Diagnostic Tools for Java - Health Center** from the list, then click **Next**.
- c. Complete the wizard by following the on-screen instructions.
- d. At the ISA prompt to restart the Workbench, select **Yes** to restart.

## Launching the Health Center client

To launch the client, perform the following steps:

1. Start the ISA Workbench and click **Analyze Problem** from the Welcome page.
2. On the Analyze Problem tab, click **Tools** (Figure 8-4).

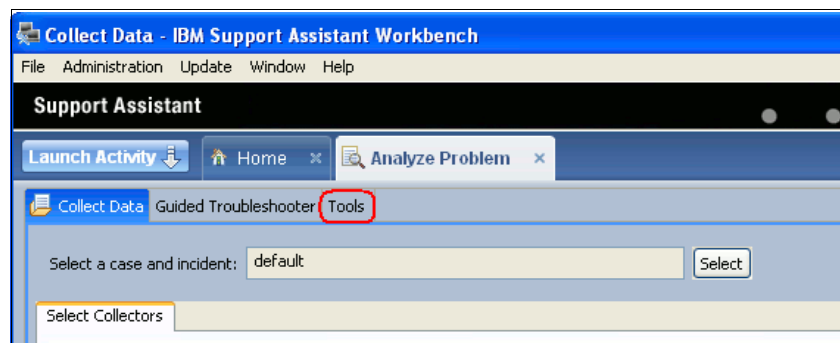


Figure 8-4 Where to find the installed tools in the ISA Workbench

3. A list of all the installed tools is displayed. Select **IBM Monitoring and Diagnostic Tools for Java - Health Center** and then click **Launch**. The Health Center application now launches and the Health Center: Connection wizard opens.

## Enabling your application for monitoring

To enable your application for monitoring, perform the following steps:

1. On the first page of the Health Center: Connection wizard, click **Enabling an application for monitoring** (Figure 8-5).

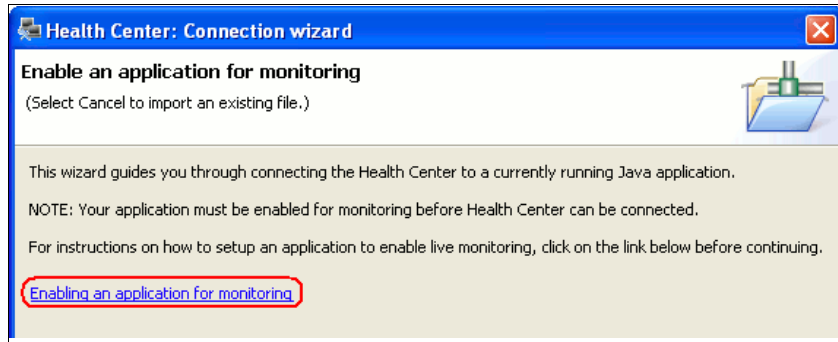


Figure 8-5 Health Center: Connection wizard application enablement link

2. Follow the help topic displayed for you application and JDK version.

To enable a CICS Dynamic Scripting application for monitoring, edit the `/applicationDir/config/zero.cics` file by adding the line (between the square brackets) shown in Example 8-29.

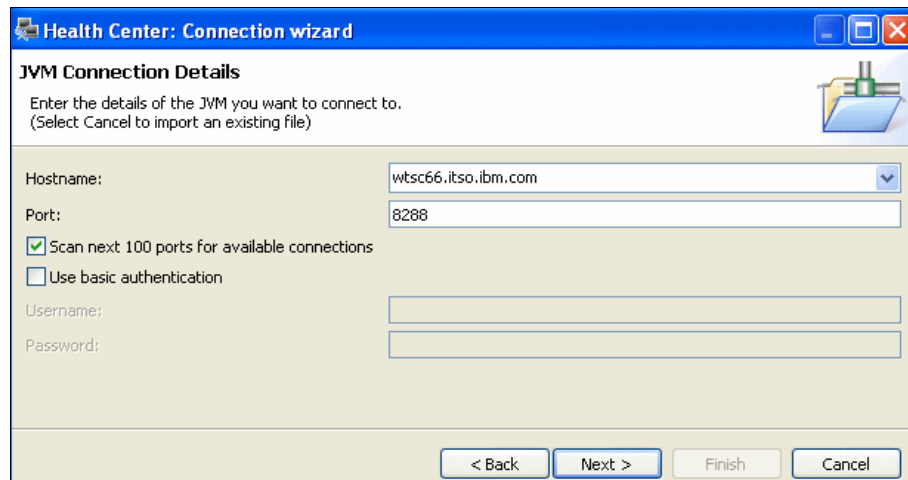
### Example 8-29 Setup monitoring

---

```
/config/zso/jvmargs += [  
  "-Xhealthcenter:port=8688"  
]
```

---

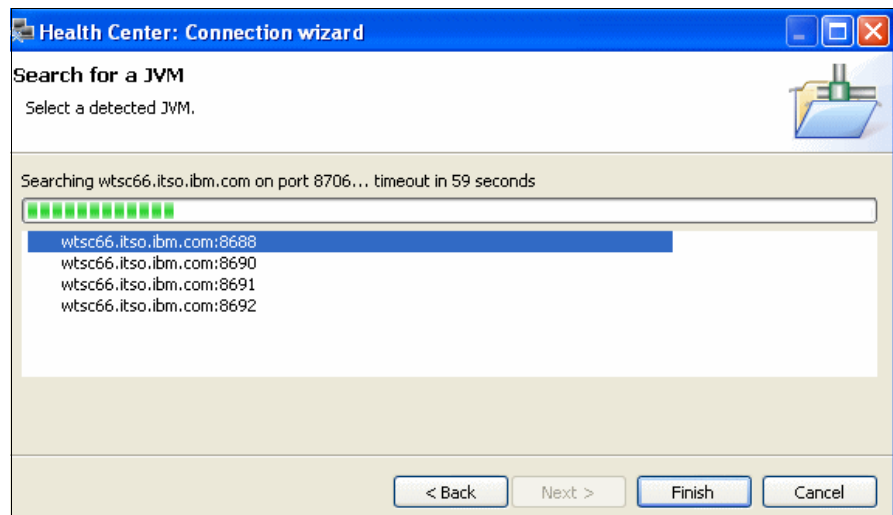
3. Input the port for Health Center agent and confirm the range of ports for searching; see Figure 8-6.



The screenshot shows the 'Health Center: Connection wizard' window, specifically the 'JVM Connection Details' step. The window has a blue title bar and a standard Windows XP-style interface. The main area is light beige. At the top, it says 'Enter the details of the JVM you want to connect to. (Select Cancel to import an existing file)'. Below this, there are several input fields: 'Hostname:' with a dropdown menu showing 'wtsc66.itso.ibm.com', 'Port:' with a text box containing '8288', a checked checkbox for 'Scan next 100 ports for available connections', an unchecked checkbox for 'Use basic authentication', 'Username:' and 'Password:' text boxes. At the bottom right, there are four buttons: '< Back', 'Next >', 'Finish', and 'Cancel'.

Figure 8-6 Specify the host and port for Health Center agent

4. Select the port that you want and click **Finish** (Figure 8-7). A connection to the Health Center agent is established.



The screenshot shows the 'Health Center: Connection wizard' window, specifically the 'Search for a JVM' step. The window has a blue title bar and a standard Windows XP-style interface. The main area is light beige. At the top, it says 'Select a detected JVM.'. Below this, there is a progress bar and a status line that reads 'Searching wtsc66.itso.ibm.com on port 8706... timeout in 59 seconds'. Below the progress bar, there is a list of detected JVMs with their hostnames and ports: 'wtsc66.itso.ibm.com:8688', 'wtsc66.itso.ibm.com:8690', 'wtsc66.itso.ibm.com:8691', and 'wtsc66.itso.ibm.com:8692'. The first entry is highlighted with a blue background. At the bottom right, there are four buttons: '< Back', 'Next >', 'Finish', and 'Cancel'.

Figure 8-7 Select a detected port for the JVM

## 8.4.2 Features and benefits

Health Center is a very low overhead monitoring tool. It runs along an IBM Java application with a small impact on the application's performance. Health Center monitors several application areas, using the information to provide recommendations and analysis that help you improve the performance and efficiency of your application. Health Center can save the data obtained from monitoring an application and load it again for analysis at a later date.

Health Center provides visibility and monitoring. See Figure 8-8.

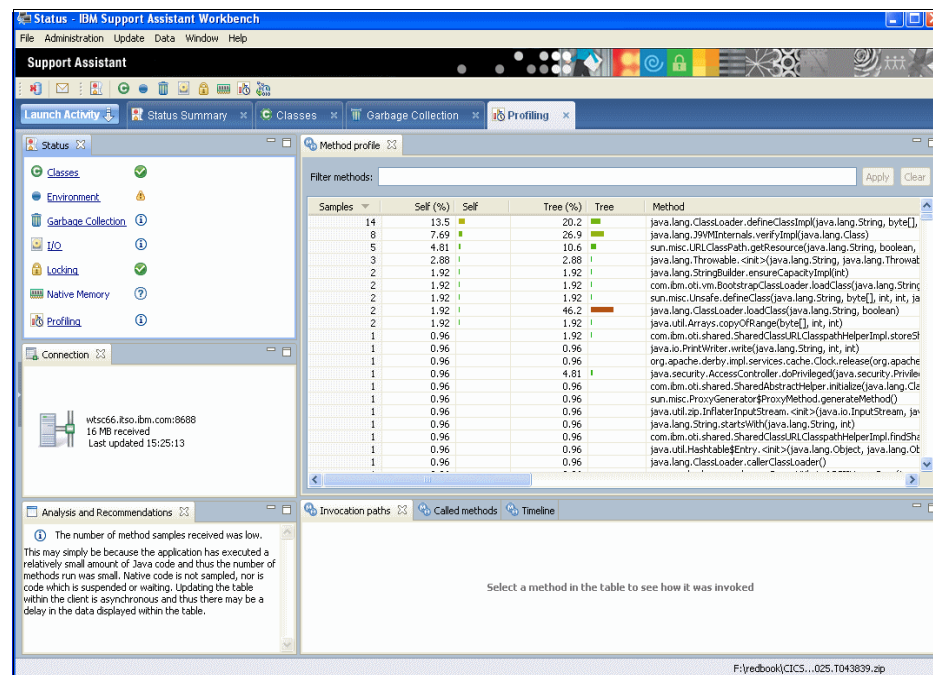


Figure 8-8 The Health Center view

Features and benefits are as follows

► Performance

- Java method profiling: The Health Center uses a sampling method profiler to diagnose applications showing high CPU usage. It is low overhead, which means there is no need to specify, in advance, which parts of the application to monitor; the Health Center simply monitors everything. It works without recompilation or byte code instrumentation and shows where the application is spending its time, by giving full call stack information for all sampled methods.
- Lock analysis: Synchronization can be a big performance bottleneck on multi-CPU systems. It is often difficult to identify a hot lock or assess the impact locking is having on your application. Health Center records all locking activity and identifies the objects with most contention. Health Center analyses this information, and uses it to provide guidance about whether synchronization is impacting performance.
- Garbage collection (GC): The performance of GC affects the entire application. Tuning GC correctly can potentially deliver significant performance gains. Health Center identifies where garbage collection is causing performance problems and suggests more appropriate command-line options.

► Memory usage

The Health Center can identify whether your application is using more memory than seems reasonable, or where memory leaks occur. It then suggests solutions to memory issues, and also Java heap sizing guidance.

► System environment

Health Center uses an environment perspective to provide details of the Java version, Java class path, boot class path, environment variables, and system properties. This is particularly useful for identifying problems on remote systems or systems where you do not control the configuration. If the Health Center detects misconfigured applications, it provides recommendations for repairing them.

► Java Class loading

Health Center provides class loading information, showing exactly when a class has been loaded and whether it is cached or not. This information helps you determine whether your application is being affected by excessive class loading.

- ▶ File input and output

Health Center uses an I/O perspective to monitor application input or output (I/O) tasks as they run. You can use this perspective to monitor how many and which files are open, and to help solve problems such as when the application fails to close files.

- ▶ Real time monitoring

The WebSphere Real Time perspective (WRTP) helps you identify unusual or exceptional events that might occur when you run a WebSphere Real Time application. The trace information can be presented in various ways, including linear or logarithmic scales, and histograms. WRTP provides predefined trace point views that are especially helpful.

- ▶ Object allocations

Health Center allows you to view the size, time, and code location of an object allocation request that meets specific threshold criteria.







## Dynamic Scripting for unit testing of CICS programs

In this chapter, we describe the unit testing area in a CICS dynamic scripting environment. We cover how to use the JUnit framework for unit testing of our dynamic scripting applications. With JUnit framework, using dynamic scripting testing framework can also be a powerful tool for unit testing of our traditional CICS programs by writing dynamic scripting test cases.

In this chapter, we include an introduction to testing, such as test phases and benefits of using a test framework, to help you become familiar with the basic test concepts. Next, we provide an overview of the JUnit testing frameworks and also dynamic scripting support of it. The remainder of the chapter provides some common scenario example of using the features of JUnit within dynamic scripting feature.

## 9.1 Introduction to application testing

Within a typical development project, various types of testing are performed during phases of the development cycle. Project requirements based on size, complexity, risks, and costs determine the levels of testing to be performed. The focus of this section is on component testing and unit testing.

### 9.1.1 Test phases

This section lists test phases:

- ▶ Unit test
- ▶ Component test
- ▶ Build verification test (BVT)
- ▶ Function verification test (FVT)
- ▶ System verification test (SVT)
- ▶ Performance test
- ▶ Customer acceptance test

#### Unit test

Unit tests are informal tests that are generally executed by the developers of the application code. The tests are often quite low-level in nature, and test the behavior of individual software components, such as individual Java classes, scripts, servlets, or EJBs.

Because unit tests are usually written and performed by the application developer, they tend to be “white-box” in nature, that is, they are written using knowledge about the implementation details and test-specific code paths. Not all unit tests have to be written this way; one common practice is to write the unit tests for a component based on the component specification, before developing the component itself. Both approaches are valid, and you might want to make use of both when defining your own unit testing policy.

#### Component test

Component tests are used to verify particular components of the code before they are integrated into the production code base. Component tests can be performed on the development environment. A developer configures a test environment and supporting testing tools such as JUnit. Using the test environment, you can test customized code without having to deploy this code to a runtime system.

### **Build verification test (BVT)**

For these tests, members of the development team check their source code into the source control tool, and mark the components as part of a build level. The build team is responsible for building the application in a controlled environment, based on the source code available in the source control system repository. The build team extracts the source code from the source control system, executes scripts to compile the source code, packages the application, and tests the application build.

The test run on the application of the build produced is called a *build verification test (BVT)*. The BVT is a predefined and documented test procedure to ensure that basic elements of the application are working properly, before accepting the build and making it available to the test team for a function verification test (FVT) or system verification test (SVT).

### **Function verification test (FVT)**

These tests are used to verify individual functions of an application. For example, you can verify whether the taxes are being calculated properly within a banking application.

### **System verification test (SVT)**

These tests are used to test a group of functions. A dedicated test environment should be used with the same system and application software as the target production environment. To get the best results from such tests, you have to find the most similar environment and involve as many components as possible, and verify that all functions are working properly in an integrated environment.

### **Performance test**

Performance tests simulate the volume of traffic that you expect to have for the application, or applications, and ensure that the system can support this stress, and to determine whether the system performance is acceptable.

### **Customer acceptance test**

In this level of testing, all aspects of an application or system are thoroughly and systematically tested to demonstrate that it meets business and non-functional requirements. The scope of a particular acceptance test is defined in the acceptance test plan.

## **9.1.2 Benefits of unit and component testing**

It might seem obvious as to why we want to test our code. Unfortunately, many people do not understand the value of testing. Simply put, we test our code and

applications to find defects in the code, and to verify that changes we have made to existing code do not break that code. In this section, we highlight the key benefits of unit and component testing.

Perhaps looking at the question from the opposite perspective is helpful: Why do developers *not* perform unit tests? In general, the simple answer has to do with difficulty or lack of enforcement. Writing an effective set of unit tests for a component is not a trivial undertaking. Given the pressure to deliver that many developers find themselves subjected to, the temptation to postpone the creation and execution of unit tests in favor of delivering code fixes or new functionality is often overwhelming.

In practice, this way of economizing is usually false, because developers rarely deliver bug-free code, and the discovery of code defects and the costs associated with fixing them are simply pushed further out into the development cycle, which is inefficient. The best time to fix a code defect is immediately after the code has been written, while it is still fresh in the developer's mind.

Furthermore, a defect discovered during a formal testing cycle must be documented, prioritized, and tracked. All these activities incur cost, and might mean that a fix is deferred indefinitely, or at least until it becomes critical.

Based on our experience, we believe that encouraging and supporting the development and regular execution of unit test cases ultimately leads to significant improvements in productivity and overall code quality. The creation of unit test cases does not have to be a burden. If done properly, developers can find the intellectual challenge quite stimulating and ultimately satisfying. The thought process involved in creating a test can also highlight shortcomings in a design, which might not otherwise have been identified when the main focus is on implementation.

Be sure to take the time to define a unit testing strategy for your own development projects. A simple set of guidelines, and a framework that helps to ease development and execution of tests, pays for itself surprisingly quickly.

After you have decided to implement a unit testing strategy for your project, the first hurdles to overcome are the factors that dissuade developers from creating and running unit tests in the first place. A testing framework can help ease the following tasks:

- ▶ Writing tests
- ▶ Running tests
- ▶ Rerunning a test after a change

Tests are easier to write, because much of the infrastructure code that you require to support every test is already available. A testing framework also

provides a facility that makes it easier to run and re-run tests, perhaps through a GUI. The more often a developer runs tests, the sooner the problems can be located and fixed, because the difference between the code that last passed a unit test, and the code that fails the test, is smaller.

### 9.1.3 Benefits of using the test framework

Testing frameworks also provide other benefits:

- ▶ Consistency: Every developer is using the same framework. All of your unit tests work in the same way, can be managed in the same way, and report results in the same format.
- ▶ Maintenance: A framework has already been developed and is already in use in a number of projects, and you spend less time maintaining your testing code.
- ▶ Ramp-up time: If you select a popular testing framework, you might find that new developers coming into your team are already familiar with the tools and concepts involved.
- ▶ Automation: A framework can offer the ability to run tests unattended, perhaps as part of a daily or nightly build.

**Note:** A common practice in many development environments is the use of daily builds. These automatic builds are usually initiated in the early hours of the morning by a scheduling tool.

## 9.2 JUnit overview

A unit test is a collection of tests designed to verify the behavior of a single unit. A unit is always the smallest testable part of an application. In object-oriented programming, the smallest unit is always a class.

JUnit provides a framework to test your class by scenario, and you have to create test cases that use the following elements:

- ▶ Instantiate an object
- ▶ Invoke methods
- ▶ Verify assertions

**Note:** An assertion is a statement that allows you to test the validity of any assumptions made in your code.

The zero.test module uses the JUnit 4 testing framework. This chapter focuses on how to invoke JUnit tests in Groovy and Java, and several extensions that sMash provides to ease application testing. Testing server-side PHP functionality is possible, but requires writing Groovy or Java JUnit tests as clients.

Because web applications are layered architectures, they can often be complex to test. The zero.test provides conveniences and utilities to help, and provides a test execution environment that helps you verify that your code is doing what it should be doing.

## 9.2.1 Setting up zero.test

To begin using zero.test, you first add it as a dependency in your application. IBM WebSphere sMash dependency manager can find the necessary dependencies and add them to your application's class path.

After creating your application, modify the `config/ivy.xml` file in your application by adding the line, shown in Example 9-1, in the dependencies section.

*Example 9-1 Adding the dependency to your application for unit testing*

---

```
<dependency org="zero" name="zero.test" rev="[1.0.0.0, 2.0.0.0[" />
```

---

After adding the dependency, issue the resolve command using the CLI. For instance, from the command line, type the following command:

```
zero resolve
```

After the dependency is resolved, you are now ready to write test cases.

## 9.2.2 Writing tests

This section describes writing tests.

### Test cases

Your application code is tested by test case class. Each test is implemented by a Java method in the test case class. The method should be declared as public method and take no parameter and return void, which is required to be driven by the test runner.

In previous JUnit releases, all the test method names had to begin with the word test followed by three periods (test...) so that the test runner could automatically find and run them.

In JUnit 4.x, this is no longer required, because we mark the test methods with the at symbol (@Test) annotation. See Example 9-2.

Make sure to import all the annotation class and assert class in the test cases.

*Example 9-2 Generic JUnit test case written in Java*

---

```
import org.junit.Test;
import org.junit.Before;
import static org.junit.Assert.assertTrue;

public class MoneyTest {
    private Money f12USD, f14USD;
    private Money expected,result;

    @Before
    public void setUp() {
        f12USD= new Money(12, "USD");
        f14USD= new Money(14, "USD");
        expected= new Money(26, "USD");
    }

    @Test
    public void simpleAdd() {
        result= f12USD.add(f14USD);
        assertTrue(expected.equals(result));
    }
}
```

---

Do not be concerned at this point if you are not familiar with annotation and assert statements. We talk about it in detail later.

## Annotations

For the first time in its history, JUnit 4 has significant changes to previous releases. It simplifies testing by using the annotation feature, which was introduced in Java 5 (JDK 1.5). One helpful feature is that tests no longer rely on subclassing, reflection, and naming conventions. JUnit 4 allows you to mark any method in any class as an executable test case, just by adding the `@Test` annotation in front of the method. See Table 9-1.

*Table 9-1 New annotations supported in JUnit 4*

Annotation name	Descriptions
<code>@Test</code>	Marks that this method is a test method.
<code>@Test(expected=ExceptionClassName.class)</code>	Tests if the method throws the named exception.
<code>@Test(timeout=100)</code>	Fails if execution of method takes longer than 100 milliseconds.
<code>@Ignore</code>	Ignores the test method.
<code>@BeforeClass</code>	Marks the method that must be executed once before the start of all the tests; for example, to connect to the database.
<code>@AfterClass</code>	Marks the method that must be executed once after the execution of all the tests; for example, to disconnect from the database.
<code>@Before</code>	Marks the method that must be executed before each test (setUp).
<code>@After</code>	Marks the method that must be executed after each test (tearDown).



## Asserting the test result

JUnit provides a number of static methods in the `org.junit.Assert` class that can be used to assert conditions and fail a test if the condition is not met. Table 9-2 summarizes the provided static methods.

Table 9-2 JUnit provided assertion

Method name	Description
<code>assertEquals</code>	Asserts that two objects or primitives are equal. Compares objects using equals method, and compares primitives using == operator.
<code>assertFalse</code>	Asserts that a boolean condition is false.
<code>assertNotNull</code>	Asserts that an object is not null.
<code>assertNotSame</code>	Asserts that two objects do not refer the same object. Compares objects using != operator.
<code>assertNull</code>	Asserts that an object is null.
<code>assertSame</code>	Asserts that two objects refer to the same object. Compares objects using == operator.
<code>assertTrue</code>	Asserts that a boolean condition is true.
<code>fail</code>	Fails the test

All of these methods include an optional String parameter that the writer of a test can use to provide a brief explanation of why the test failed. This message is reported along with the failure when the test is executed. The full JUnit 4 API documentation is at the following address:

[http://junit.sourceforge.net/javadoc\\_40/index.html](http://junit.sourceforge.net/javadoc_40/index.html)

## Test suite

Test cases can be organized into test suites. In JUnit 4.x, the way to build test suites has been completely replaced and no longer uses sub classing, reflection, and naming conventions. The following example shows how to build a test suite class in JUnit 4.x. See Example 9-3.

*Example 9-3 Simple example for a test suite*

---

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;

@RunWith(Suite.class)
@SuiteClasses({MoneyTest.class})
public class TestsSuite {}
```

---

The AllTests class is a simple placeholder for the @RunWith and @SuiteClasses annotations, and does not require a static suite method. The @RunWith annotation tells the JUnit 4 test runner to use the org.junit.runners.Suite class for running the AllTests class. With the @SuiteClasses annotation, you can define which test classes to include in this suite and in which order. If you add more than one test class, the syntax is as follows:

```
@SuiteClasses({TestClass1.class, TestClass2.class})
```

### 9.2.3 Running tests

Tests are run by executing the zero test CLI task:

```
zero test
```

By default, this task runs every JUnit test in the current application where the class ends with Test. For instance, my.custom.UnitTest would be executed but my.custom.TestingClass would not. Tests located in resolved dependencies are not executed.

You can be specific in which tests are executed in a run. To filter a test run, filter by specifying one or more fully qualified class name as arguments to the zero test CLI task as follows:

```
zero test my.custom.TestingClass my.util.MyTest
```

Tests are not filtered by class name when specific tests are provided as arguments this way.

Alternate patterns for including classes may be specified in `zero.config` as a list of regex patterns under `/config/test/classes`, for example:

```
/config/test/classes = [ "(.*)Test" ]
```

Java classes must be compiled before invoking the zero test task:

```
zero compile
```

JUnit tests written in Groovy work the same way as traditional Java JUnit tests. Class name filtering using the `/config/test/classes` or CLI arguments. However, the bindings available in application code are not. Groovy tests must be located in the `/app/scripts` folder of the testing module.

## 9.3 Unit testing dynamic script application

We can use JUnit framework for unit testing of dynamic scripting in the following ways:

- ▶ Test base on static logic of classes (without running the application)
- ▶ Test base on HTTP response (for example test response headers/content output)
- ▶ In-container while processing an HTTP request (for example test values in the /request zone)
- ▶ Dependent on a configured Global Context, but not a running application (for example JSON APIs (custom converters are registered through GC))

### Testing static logic of classes

If you want to test the static behavior of your Java classes or Groovy class, write a simple test case to drive the class and perform assertion of the expected results. See Example 9-4.

*Example 9-4 Groovy script test case to test the add method of Money class*

---

```
package example
```

```
import org.junit.Test
import org.junit.Before
import static org.junit.Assert.assertTrue
```

```
class MoneyTest implements zero.core.groovysupport.ZeroObject {
    Money f12USD, f14USD;
    Money expected,result;
```

```

@Before
public void setUp() {
    f12USD= new Money(12, "USD");
    f14USD= new Money(14, "USD");
    expected= new Money(26, "USD");
}

@Test
public void simpleAdd() {
    result= f12USD.add(f14USD);
    assertTrue(expected.equals(result));
}
}

```

---

When we try to run this test, we get a similar output if everything is successful. See Example 9-5.

*Example 9-5 Sample output of running the test case*

---

```

CICSR5 @ SC66:/u/cicrs5/cicsds/apps/sampleApp>zero test
example.MoneyTest
.
CWPZC9504: Time: 0.043

CWPZC9502: OK (1 test

CICSR5 @ SC66:/u/cicrs5/cicsds/apps/sampleApp>

```

---

**Note:** In this scenario, starting the zero application is unnecessary because our test is not related to any runtime factor, only the static behavior of the class.

## Test base on HTTP response

Sometimes we need to perform runtime tests based on real application output. Because our dynamic scripting application follows the RESTful interface, we can build test cases to start the application, send HTTP request and perform assertion on the HTTP response, such as response code and response body.

We build the test case to test one scenario used in our previous `vsamSimpleDemo` example. Recall that the PHP resource handler handles the retrieve event and performs a lookup in a CICS VSAM file using the passed ID number as the key, then sends back the entire record content as an HTTP response. See Figure 9-1 on page 219.

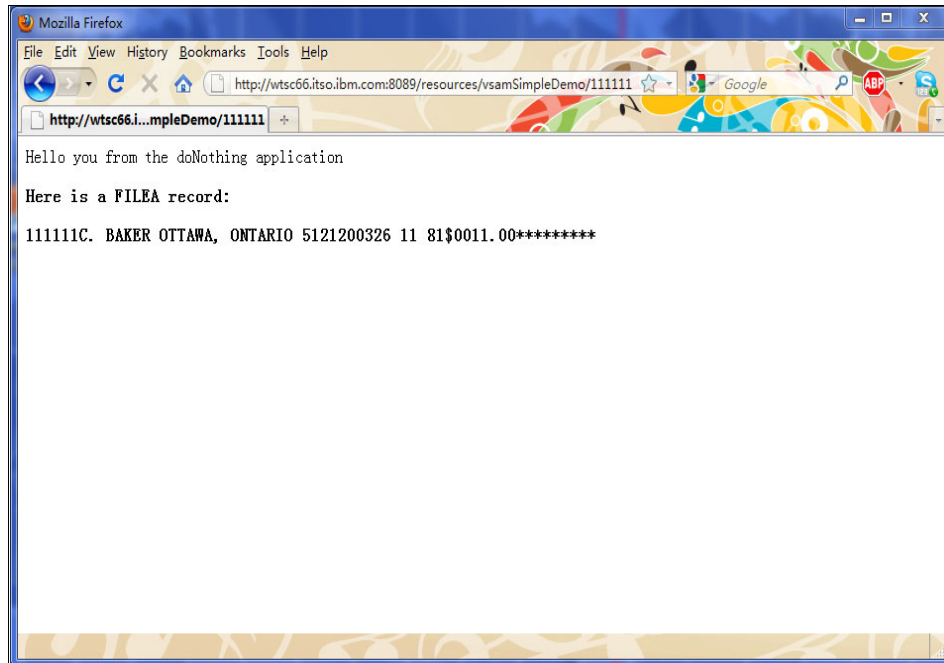


Figure 9-1 The sample VSAM browse application

The zero.test module provides utilities as a convenience when writing tests. Although Example 9-6 shows Groovy samples, use in Java also applies.

*Example 9-6 Sample test case to test base on HTTP response of the application*

---

```
// File: app/scripts/example/ESDSBrowseTest.groovy
package example

import static org.junit.Assert.assertEquals
import static org.junit.Assert.assertTrue
import java.net.HttpURLConnection
import java.net.URL
import org.junit.Test
import zero.core.context.GlobalContext
import zero.core.context.GlobalContextURIs.Config
import zero.util.http.HttpConnectionContentHandlerFactory
import zero.util.launcher.ApplicationLauncher

class ESDSBrowseTest implements zero.core.groovysupport.ZeroObject {

    @Test
    void testBrowseESDSResponse() {
```

```

        HttpURLConnectionContentHandlerFactory.register()

        ApplicationLauncher launcher = new
        ApplicationLauncher("ESDSBrowseTest", ".")

        try {
            launcher.startApplication()
            URL u = new URL("http://localhost:" + config.http.port[] +
"/resources/vsamSimpleDemo/111111")
            HttpURLConnection conn = (HttpURLConnection)
u.openConnection()

            assertEquals(HttpURLConnection.HTTP_OK,
conn.getResponseCode())
            String myContent = (String) conn.getContent()
            assertTrue("Content not as
expected", (myContent.indexOf("BAKER") != -1))
        } finally {
            launcher.stopApplication()
        }
    }
}

```

---

In the example, `HttpURLConnectionContentHandlerFactory` implements basic content handlers for `HttpURLConnection`. User code invokes the static `register` method before using `HttpURLConnection` objects. It supports returning text/plain and text/html content as `String`, application/json and text/json content as `Json.decode`, and application/xml as instance of `org.w3c.dom.Document`.

`ApplicationLauncher` is a utility for embedding WebSphere sMash applications within test cases. `ApplicationLauncher` launches the specific zero applications in a separate thread in the same `JVMServer` process.

The constructor of the class takes the following parameters:

```

public ApplicationLauncher(java.lang.String name,
                           java.lang.String applicationDirectory,
                           java.lang.String... mainArgs)

```

The parameters have the following meanings:

<b>name</b>	Name of the thread spawned to run the application.
<b>applicationDirectory</b>	Root directory of the Zero application
<b>mainArgs</b>	Optional command-line arguments for <code>zero.core.Main</code> .

The following call of the `startApplication()` method starts the application running on a new thread. This method returns after the Zero application is ready to receive requests and the global context is fully configured. Note whether the application fails to start; this method throws a `RuntimeException` with the cause. For this reason, always call `stopApplication` within a `finally` block (as shown in the previous example) to ensure the application is fully stopped.

Unit testing classes in Groovy have the standard bindings that sMash offers. For instance, `GlobalContext.zget("/config/http/port")` can be shortened to `config.http.port[]`. To get access to the bindings, set the marker interface on the JUnit test case class of `zero.core.groovysupport.ZeroObject` as illustrated in Example 9-6 on page 219.

## Tests running in-container

In certain cases, you might want to assert the values in the container environment and you want to capture these assertion errors in your test cases. You may use the in-container test support.

In-container tests also use `zero.util.launcher.ApplicationLauncher` to embed the application within the test case. To support in-container assertions, requests are sent through the `zero.test.TestUtils.sendTestRequest` method. This utility method handles the setup required to propagate in-container assertion errors (`AssertionError`) from the container thread back to the test client. See Example 9-7.

### *Example 9-7 Request*

---

```
package example

import java.net.HttpURLConnection
import java.net.URL
import org.junit.Test
import zero.core.context.GlobalContext
import zero.core.context.GlobalContextURIs.Config
import zero.util.launcher.ApplicationLauncher
import zero.test.TestUtils

class InContainerTest implements zero.core.groovysupport.ZeroObject {

    @Test
    void testInContainer() {
        ApplicationLauncher launcher = new
        ApplicationLauncher("testInContainer", ".")

        try {
```

```

        launcher.startApplication()
        URL u = new URL("http://localhost:" + config.http.port[] +
"/incontainer.groovy")
        HttpURLConnection conn = (HttpURLConnection)
u.openConnection()
        conn.setRequestProperty("foo", "bar")

        TestUtils.sendTestRequest(conn)

    } finally {
        launcher.stopApplication()
    }
}
}

```

---

Process the request through a utility method that handles the setup required to project assertion errors from the container thread back to the test client.

Example 9-8 shows the container scripts.

*Example 9-8 Sample script public/incontainer.groovy*

---

```

assert 'GET' == request.method[]
assert 'bar' == request.headers.in.foo[]

```

---

Example 9-9 shows good output.

*Example 9-9 Sample output*

---

```

CICSR5 @ SC66:/u/cicsrs5/cicsds/apps/sampleApp>zero test
example.InContainerTest
.
CWPZC9504: Time: 9.483

CWPZC9502: OK (1 test

CICSR5 @ SC66:/u/cicsrs5/cicsds/apps/sampleApp>

```

---

## Tests that are dependent on a configured global context

In certain cases you might want to test your configured global context without starting the application. For example, perhaps you want to test JSON custom converter defined in your configuration file. See Example 9-10 on page 223.



#### *Example 9-10 Custom JSON converter configuration*

---

```
/config/json/converters += {  
    "java.sql.Time" : "zero.json.converters.java.sql.TimeConverter",  
    "java.sql.Timestamp"  
    "zero.json.converters.java.sql.TimestampConverter"  
    "java.sql.Date" : "zero.json.converters.java.sql.DateConverter"  
    "java.util.Date" : "zero.json.converters.java.util.DateConverter"  
}  
  
/config/json/derivedConverters += {  
    "groovy.lang.Writable" :  
    "zero.json.converters.groovy.lang.WritableConverter"  
}
```

---

The `zero.core.config.ConfigLoader` utility class provides APIs for creating a new Global Context and loading configuration files for a specified module and all its dependencies. See Example 9-11.

#### *Example 9-11 Test case for testing custom JSON convertor*

---

```
package example  
  
import static org.junit.Assert.assertEquals  
import org.junit.Test  
import zero.core.config.ConfigLoader  
import zero.json.Json  
  
class GCDependentTest {  
  
    @Test  
    void testJavaUtilDate() {  
        ConfigLoader.initialize(".")  
  
        java.util.Date d = new java.util.Date()  
        String jsonStr = Json.encode(d)  
        assertEquals("Failed to match Date", d,  
        Json.toObject(Json.decode(jsonStr), java.util.Date.class))  
    }  
}
```

---

JSON `.encode` and `.decode` depend on the configured GC for custom converters, including converters for `java.util.Date`. In the example, the `initialize` method in `ConfigLoader` creates a new Global Context and loads configuration files for the specified module and all its dependencies.

## Testing tasks

Dynamic scripting feature for CICS inherits WebSphere sMash, providing an open framework to implement CLI tasks. In fact, the zero test is implemented using this framework. The zero.test module provides a utility to check for success and failure of your custom task.

Several conveniences it provides are setting up logging for the invoked CLI task, asserting that the task succeeded or failed with specific messages displayed on the console.

The abstract zero.test.TaskTester class must be implemented to use this utility. Simply extend it and provide your custom task's usage message when a user does not use the task as documented as a list of strings. Each string represents a line in the message.

Example 9-12 and Example 9-13 on page 225 show several uses. For a complete reference, see the zero.test.TaskTester Javadoc.

---

### *Example 9-12 Example of the usage of TaskTester*

---

```
// The workhorse for the other convenience methods in , executes a CLI task and
// returns an instance of on which other methods in this class make assertions on.
// zero create testapp from zero:zero.data
run("create", "testapp", "from", "zero:zero.data");

// Executes the CLI task for the given arguments and asserts that the return code
// for the executed process equals the parameter .
// zero model sql sync
runForCode(0, "model", "sql", "sync");

// Executes the CLI task for the given arguments and asserts that the return code
// is that of success "0". runForSuccessMsg also asserts that task output contains the
// text in the parameter.
// zero runsql sync.sql
runForSuccess("runsql", "sync.sql");
runForSuccessMsg("SQL file sync.sql executed successfully", "runsql", "sync.sql");

// Executes the CLI task for the given arguments and asserts that the return code
// is that of failure "1". runForFailureMsg also asserts that task output contains the
// text in the parameter.
// zero runsql not-a-file.sql
runForFailure("runsql", "not-a-file");
runForFailureMsg("SQL file sync.sql executed successfully", "runsql",
"not-a-file.sql");

// Executes the CLI task for the given arguments and asserts that the return code
// is that of failure "1".
// zero model loaddata
failsWithUsage("model", "loaddata");
```

---

Example 9-13 shows a test case for testing tasks.

*Example 9-13 Sample test case for testing tasks*

---

```
// File: app/scripts/example/CustomTaskTest.groovy
package example

import java.util.List
import java.util.ArrayList
import zero.test.TaskTester
import org.junit.BeforeClass
import org.junit.Test

public class CustomTaskTest extends TaskTester {

    List<String> getUsageStrings() {
        return ["USAGE:", "zero custom correct-usage [--optionalflag]"]
    }

    @BeforeClass
    static void ensureLogging() {
        setupLogging()
    }

    @Test
    void testFailureAndUsage() {
        failsWithUsage("custom incorrect-usage")
    }

    @Test
    void testCustomMessage() {
        runForFailure("custom dumpdata --optionalflag=somevalue",
            "--optionalflag cannot have a value");
    }

    @Test
    public void testSuccess() {
        runForSuccess("custom correct-usage",
            "Custom success message")
        runForSuccess("custom correct-usage --optionalflag",
            "Custom success message with optional flag")
    }
}
```

---

## 9.4 Unit testing traditional CICS programs

The JUnit testing framework provided by the dynamic scripting feature can be a useful tool for testing even your traditional CICS programs.

The test team can use their existing skill of the familiar JUnit framework and knowledge of Java or Groovy languages to quickly develop the test cases without having to know CICS or a mainframe programming language like COBOL or PL/I.

By use the JCICS API provided, the test case can easily link to the existing target CICS programs, examining the COMMAREA output from the CICS program with the expected result, compare and validate the content of FILES or TSQs or TDQs generated by the target program as output.

The test cases can be managed easily and the test cases can be rerun anytime.



# Part 3

## Scenarios

This part of the book has the following scenarios:

- ▶ Derby and DB2 example
- ▶ Exposing CICS resources in a RESTful style
- ▶ Command-line sample





## Development options

IBM WebSphere sMash provides three development modes:

- ▶ Command-line interface (CLI)
- ▶ App Builder
- ▶ Eclipse plug-ins.

In this chapter, we introduce the three development modes with Project Zero, and examine their advantages and disadvantages when applied to the Dynamic Scripting Feature Pack.

## 10.1 Overview of the development environment

Dynamic scripting is an agile web application platform for developing and running modern web applications. It offers a simple environment for creating, assembling and running applications based on popular web technologies.

In addition to this overview, this chapter covers the following topics:

- ▶ **Application**  
Description of an application in terms of Dynamic Scripting. See 10.2, “What is an application” on page 231.
- ▶ **Development environment**  
Development environments and the various options for developing applications for Dynamic Scripting deployment. See 10.3, “Development options” on page 234.
- ▶ **Dependencies**  
Applications include additional function by declaring dependencies on modules such as `zero.cics.php`. See 10.4, “Dependencies” on page 251.
- ▶ **Module groups**  
Module groups can be used to group applications together. See 10.5, “Module groups” on page 254.
- ▶ **Dependency management**  
Applications can depend on other applications, and the commands to deal with dependencies. See 10.6, “Dependency management commands” on page 259.
- ▶ **Moving applications**  
Moving applications developed on distributed platforms to run under Dynamic Scripting. See 10.7, “Moving applications to Dynamic Scripting” on page 262.
- ▶ **Issues with code pages**  
Dynamic Scripting runs inside a JVM server inside CICS on a mainframe. See 10.8, “Dynamic scripting encoding considerations” on page 266.



## 10.2 What is an application

When an application is created, it will contain a set of default subdirectories that together comprise the application. These subdirectories provide a standardized structure for applications, which makes configuration and development easier by the use of convention. Applications are also sometimes called *modules*, and are also referred to as *packages* when they are declared as dependencies.

### 10.2.1 Common directories

Within the application's home or base level directory, the following folders or directories might exist:

- ▶ `.zero`
- ▶ `app`
- ▶ `classes`
- ▶ `config`
- ▶ `java`
- ▶ `lib`
- ▶ `logs`
- ▶ `public`
- ▶ `reports`

#### **.zero**

The `.zero` directory is created by the application to hold information about itself, including the history of resolved modules, caching information and so on. No user files should be created in this folder.

#### **app**

The `app` folder contains scripts, templates, models, and tasks for the key components of the application. Within the `app` folder can be found a number of subdirectories, and additional subdirectories may be created.

#### **classes**

Compiled Java classes are part of an application. Typically the source files are located in the `java` folder in the development phase.

#### **config**

Configuration information for the application is in the `config` directory. The `config` typically contains a `php.ini` file, which contains configuration information for PHP, an `ivy.xml` file, which contains the application's name, and also any declared dependencies on other application modules. The `config` also contains

a `zero.config` file, which contains configuration information, such as port numbers, for the application.

Example 10-1 shows a `ivy.xml` for an application called `zero.demo`.

*Example 10-1 ivy.xml for zero.demo application*

---

```
<?xml version="1.0" encoding="UTF-8"?>
<ivy-module version="1.3">

  <info organisation="zero" module="demo" revision="1.0.0">
    <license name="type of license" url="http://license.page"/>
    <ivyauthor name="author name" url="http://authors.home.page"/>
    <description homepage="http://module.description.page"/>
  </info>

  <publications>
    <artifact type="zip" />
  </publications>

  <!-- Note: dependencies from maven require the maven2 notation. -->
  <dependencies>
    <dependency org="zero" name="zero.core" rev="[1.0.0.0,
2.0.0.0["/>
  </dependencies>
</ivy-module>
```

---

Example 10-2 shows the `zero.config` file that is created when a new, empty application is created. The defaults are that the new application is in development mode, and that the application will listen on HTTP port 8080.

*Example 10-2 zero.config*

---

```
# HTTP port (default is 8080)
/config/http/port = 8080

# Runtime mode (default is "production")
/config/runtime/mode="development"
```

---

## java

This folder is used to store Java source files that are part of the application. These files can be compiled with the **zero compile** command and the resulting class files are placed into the `classes` folder.

### **lib**

Additional JAR files required by the application but not part of another application module can be stored in the `lib` directory. These are references as part of the application's class path.

### **logs**

WebSphere sMash applications can produce logs and trace files while running, which are stored in this folder.

### **public**

The files within the `public` folder represent the files accessible from a web browser. These include static content such as HTML, GIF, or JPEG images, and also server-side-generated content such as PHP web pages, Groovy web pages or Groovy templates.

### **reports**

The `reports` folder contains XML files indicating the name and location of any resolved dependencies. These describe exactly what and where the dependencies are located, as opposed to the `ivy.xml` file which states what dependencies exist.

## **10.2.2 Common tasks**

In addition to creating and populating the file structures, the other activity associated with an application is the issuing of commands. Applications must be started and stopped. Dependencies must be resolved. Java code must be compiled. Databases must be declared and built.

## **10.2.3 What is necessary to develop an application**

Developing an application requires the following tasks:

- ▶ The ability to create and edit the files that make up the application
- ▶ The ability to issue the necessary commands to start the application

## 10.3 Development options

IBM WebSphere sMash offers three modes of developing applications when downloaded to your workstation.

- ▶ Command-line interface
- ▶ App Builder
- ▶ Eclipse plug-ins

### 10.3.1 The command-line interface (CLI)

The CLI is the starting point for developing applications. It can be run on the workstation or within UNIX System Services.

#### Installing the CLI on workstation

Perform the following steps:

1. Download the `zero.zip` file from the Project Zero website:  
<http://www.projectzero.org/sMash/1.1.x/download/>
2. Extract the downloaded file in your local file system.
3. Add the `zero` directory to the user's `PATH` environment variable.
4. Add the `bin` directory under the JDK installation directory to the user's `PATH` environment variable.

#### Verification of CLI on workstation

After installation, the following instructions can be followed to create a simple configuration, the following operations can be performed, which create a simple demo project currently provided in Project Zero. The same instructions work equally on the mainframe in a properly configured Dynamic Scripting environment:

1. Open a command prompt.
2. Type the **zero create demo** command.

Example 10-3 creates a new application named demo.

*Example 10-3 The zero create command*

```
zero create demo
CWPZT0849I: The new module will be created from module
zero:zero.application.template:latest.revision, located at
C:\DynamicRepo\zero\zero-repository\stable\modules\zero\zero.appl
ication.template\zips
\zero.application.template-1.1.1.3.31115.zip
CWPZT0840I: Module demo successfully created
```

3. Change the current working directory to demo, and type the **zero start** command to start the application. Example 10-4 shows the results.

*Example 10-4 The zero start command is successful*

```
Application started and servicing requests at http://localhost:8080/
CWPZT0600I: Command start was successful
```

4. Open a browser and type `http://localhost:8080/`. A default page is shown in the following screen capture, Figure 10-1.

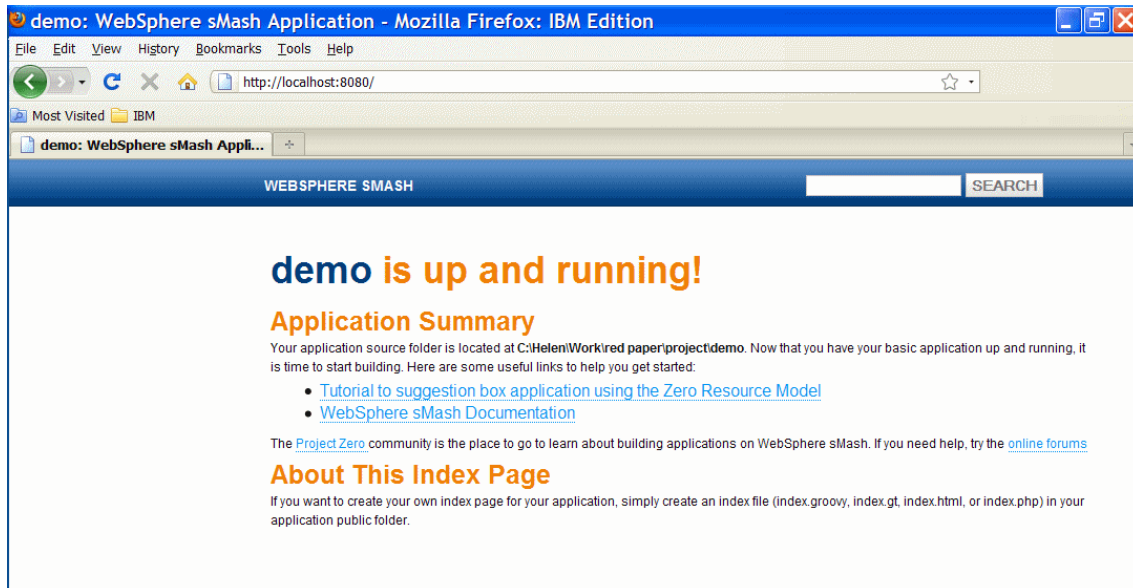


Figure 10-1 Home page for Zero demo application

For more information, visit the official website of Project Zero:

<http://www.projectzero.org>

### ***Advantages***

Advantages are as follows:

- ▶ The CLI is the smallest possible download for running WebSphere sMash on your workstation.
- ▶ For developing on the mainframe, the CLI will have already been installed as part of the installation process.

### ***Disadvantages***

Disadvantages are as follows:

- ▶ The CLI offers only commands. There are no convenient methods to edit or create files, and so you must use the editors provided by the operating system.
- ▶ There is no productivity gain to using the CLI on the workstation as opposed to using it directly on the mainframe.

## **10.3.2 The App Builder**

The WebSphere sMash App Builder is a non-released web-based tool for developing IBM WebSphere sMash applications. It can be started from the CLI and then used instead of the command line.

### **Prerequisites**

Prerequisites are as follows:

1. WebSphere sMash DE or the latest command-line interface (CLI).
2. Firefox version 3 or greater.

### **Installing the App Builder on workstation**

Perform the following steps

1. Install the CLI as described previously.
2. Issue the **appbuilder open** command. See Example 10-5 on page 237.

#### Example 10-5 Running `appbuilder open` command

---

```
-----  
Creating the AppBuilder application.  
This step may take a while.  
-----  
CWPZT0849I: The new module will be created from module  
zero:zero.application.template:latest.revision, located at  
C:\DynamicRepo\zero\zero-repository\stable\modules\zero\zero.applica  
tion.template\zips  
\zero.application.template-1.1.1.3.31115.zip  
CWPZT0840I: Module appbuilder.bootstrap successfully created  
CWPZT0600I: Command update was successful  
CWPZT0600I: Command modulegroup create was successful  
CWPZT0901I: The following module(s) are not currently in the local  
repository:  
           zero:zero.cli.tasks:[1.0.0.0,)  
CWPZT0902I: Trying to locate the module(s) using one of the  
configured remote repositories  
...
```

---

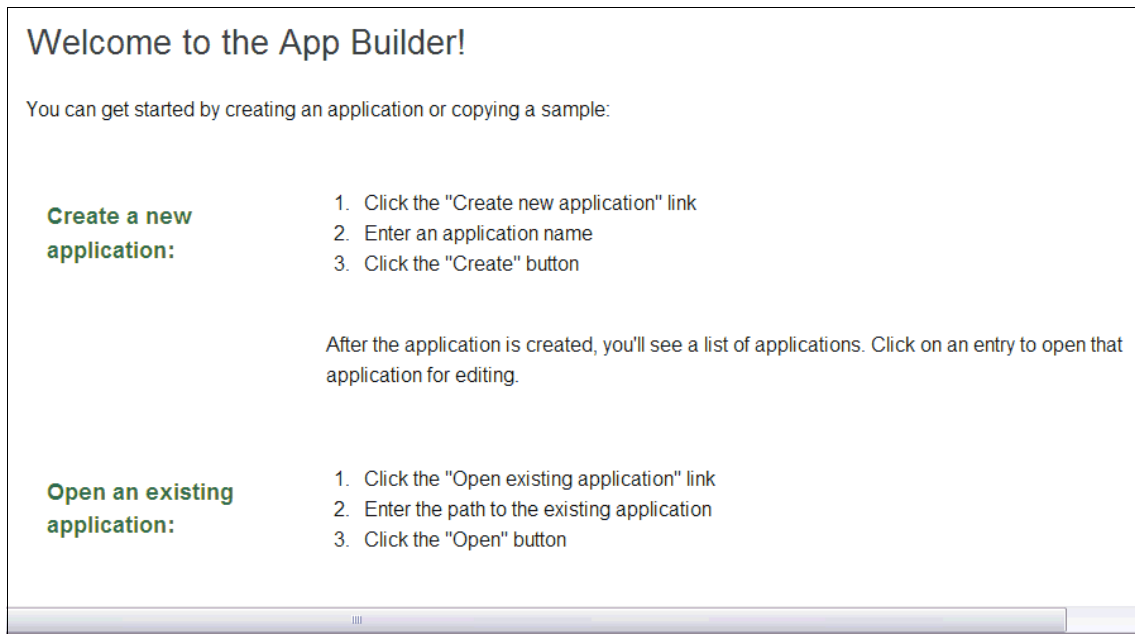
**Note:** The first invocation might take additional time to complete (depending on your network connectivity) as the App Builder dependencies are downloaded and your copy of App Builder is created. Subsequent invocations complete within seconds because they operate on the local copy.

After the command has completed, the browser will be started and will navigate to `http://localhost:8070/`. This way is the easiest for getting started.

The App Builder continues to run in the background even if the web browser is closed. To stop the App Builder, issue the **appbuilder stop** command.

If you want to access the App Builder remotely, it can be started without opening a web browser by using the **appbuilder start** command.

Figure 10-2 shows part of the App Builder.



*Figure 10-2 Part of the App Builder page*



## Verification of App Builder on workstation

Perform the following steps:

1. In the Actions section of the App Builder window, click **Create new application** as shown in Figure 10-3.

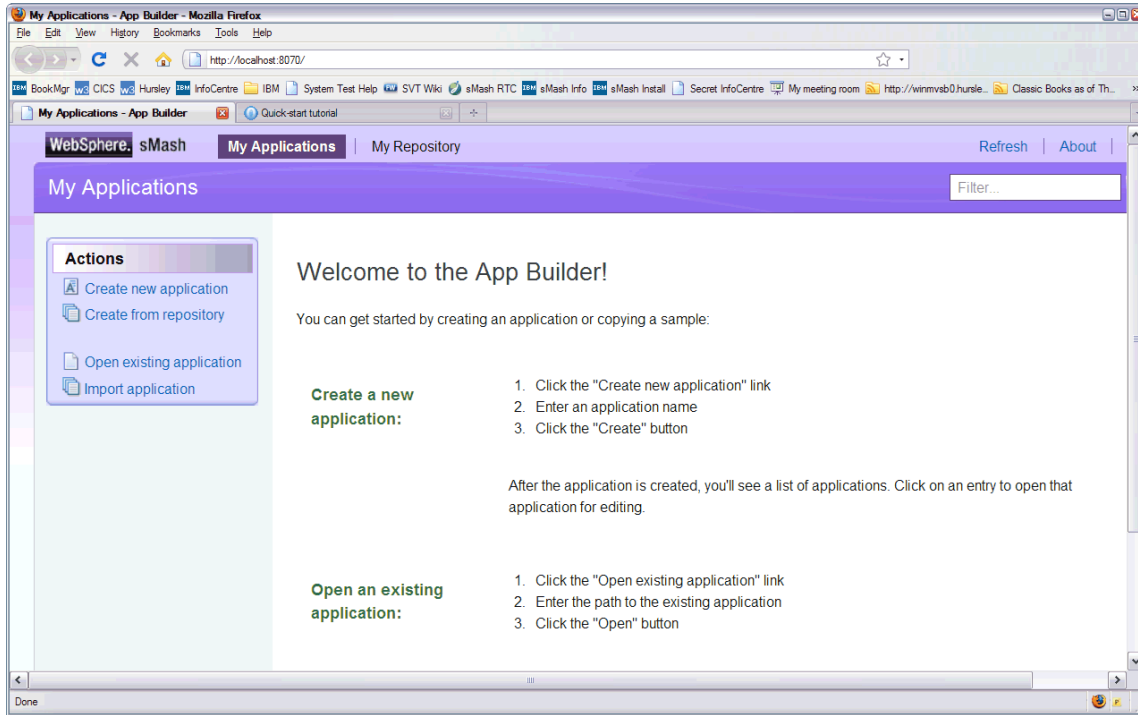


Figure 10-3 Select “Create new application” in App Builder

2. In the Create new application window that opens, type `zero.demo` as the name of the new application, as shown in Figure 10-4 on page 240.

Figure 10-4 Entering the new application name

The new application is created and added to the list of known applications. The default application address and a start button are shown in Figure 10-5.

Figure 10-5 New application zero.demo displayed in App Builder

The running application is shown in Figure 10-6.

Figure 10-6 The zero.demo is running

For more information about the App Builder, see the tutorial:

<http://www.projectzero.org/zero/indy.dev/latest/docs/zero.gettingstarted.doc/zero.gettingstarted.doc/Tutorial.html>

### ***Advantages***

Advantages are as follows:

- ▶ The App Builder provides editors for PHP, Groovy, JSON and Dojo based web pages.
- ▶ The App Builder provides command short-cuts and also a built-in console.
- ▶ Samples can be easily created, tested, started and stopped from within the App Builder.

### ***Disadvantages***

Disadvantages are as follows:

- ▶ The App Builder is not supported by the Dynamic Scripting Feature Pack. Therefore development using CICS resources is not possible.
- ▶ The initial download of the App Builder might take a considerable amount of time before it can be used.
- ▶ Applications must be packaged and transported before they can be deployed into CICS.
- ▶ The App Builder offers both experimental and stable modules. Only a subset of stable modules are supported by Dynamic Scripting Feature Pack for CICS, so it is possible that you can build an application that cannot run under CICS. Details of supported modules are in the information center.

## **10.3.3 Eclipse**

The Eclipse platform offers an open and extensible development platform for developing, debugging and deploying software.

Eclipse itself provides basic functionality and a framework for extension points to plug into it. These extension points are called Eclipse plug-ins.

**Note:** The Eclipse foundation's website offers many versions of Eclipse to download. Several versions include additional functionality. We describe the options in this section. For more information, see the Eclipse download page:

<http://www.eclipse.org/downloads/>

## Using Eclipse for PHP Developers

Eclipse for PHP Developers provides PHP editors that offer context-sensitive help and auto-completion for PHP code. PHP debuggers can also be downloaded and used to help diagnose application problems.

### ***Advantages***

Writing PHP using an editor that supports PHP is easier. Color coding of comments and keywords and content-assist improve productivity.

### ***Disadvantages***

Disadvantages are as follows:

- ▶ No access to remote file systems exists. Therefore, you cannot develop directly inside CICS; you must develop and then deploy applications later. This approach can be mitigated by using the Target Management plug-in.
- ▶ No command line is provided. Commands must be entered by using the native command-line prompt. This probably can be mitigated by using the Target Management plug-in or the WebSphere sMash plug-in.

## Using Target Management plug-in

Eclipse Target Management offers a remote files explorer that allows the creation, editing, and navigation of files on a remote file system within Eclipse.

<http://www.eclipse.org/dsdp/tm/>

### ***Adding Target Management to your Eclipse environment***

Perform the following steps:

1. Go to the download site for Target Management, and download a ZIP file to your workstation. The size is approximately 5 MB. The exact size varies with the version:

<http://download.eclipse.org/dsdp/tm/downloads/index.php>

2. Extract the ZIP file.
3. In the Eclipse menu, click **Help** → **Install New Software**, which opens a window where you can select available software, as seen in Figure 10-7 on page 243.

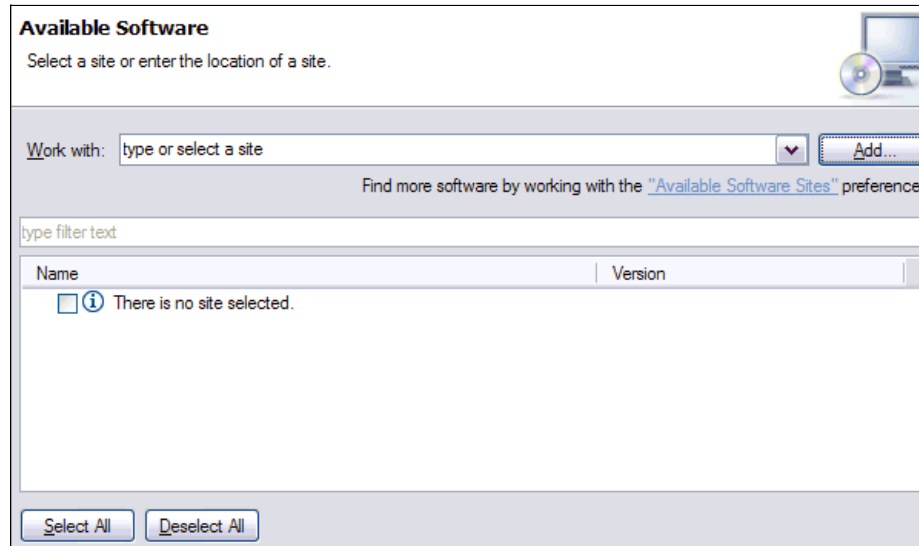


Figure 10-7 Available Software

4. Select **Local Site**, click **Add**, and then navigate to the location of the compressed directory.
5. Clear the **Group Items by Category** check box. Items in Target Management have no category and will not be shown by default.
6. Select All and then install. See Figure 10-8. Restart Eclipse when you are prompted.

Name	Version	Id
Remote System Explorer End-User Runtime	3.2.1.v20100819...	org.eclipse.rse.feature.group
RSE Core	3.2.1.v20100819...	org.eclipse.rse.core.feature.group
RSE DStore Services	3.2.1.v20100819...	org.eclipse.rse.dstore.feature.group
RSE FTP Service	3.0.200.v201005...	org.eclipse.rse.ftp.feature.group
RSE Local Services	2.1.201.v201009...	org.eclipse.rse.local.feature.group
RSE SSH Services	3.0.100.v201005...	org.eclipse.rse.ssh.feature.group
RSE Telnet Service	2.2.100.v201005...	org.eclipse.rse.telnet.feature.group
RSE Terminals UI	1.0.101.v201008...	org.eclipse.rse.terminals.feature.group
Target Management Terminal Widget	3.0.201.v201008...	org.eclipse.tm.terminal.feature.group

Figure 10-8 Items installed as part of Target Management

Eclipse with Target Management may be used to edit files on the local file system and also on remote file systems. It also provides a command-line section. See Figure 10-9 on page 244.

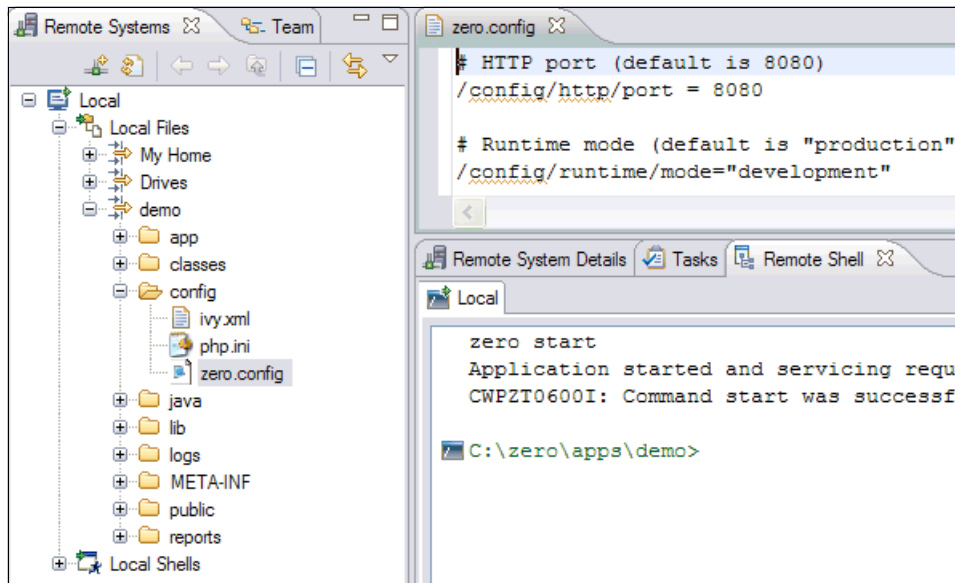


Figure 10-9 Eclipse with Target Management, command-line at bottom

### Advantages

Advantages are as follows:

- ▶ Access is available to both remote and local file systems. Applications can be developed both locally or on the mainframe.
- ▶ A command line is provided for both remote and local file systems. Applications can be controlled both locally or on the mainframe.

### Disadvantages

By itself, any “intelligent editor” capability is not provided. This lack can be mitigated by a combination with a PHP plug-in or the WebSphere sMash plug-in.

### Using the Eclipse WebSphere sMash plug-in

IBM WebSphere sMash plug-ins for Eclipse are non-released components and generally for platform developers which are available for Java/Groovy and PHP.

The plug-ins provide support for creating, resolving, and running WebSphere sMash applications, and also basic repository management.

The WebSphere sMash CLI is a prerequisite for the WebSphere sMash plug-ins. The plug-ins must be configured for the CLI for each workspace.

### ***Adding the PHP and Groovy plug-ins to your Eclipse environment***

IBM WebSphere sMash as Eclipse plug-ins are currently provided in IBM WebSphere PHP and Groovy plug-ins. To integrate with Eclipse, perform the following the steps:

1. In the Eclipse menu, click **Help** → **Install New Software**. This opens the Available Software window.
2. Click **Add Site**, type the following site URL, and click **OK**:  
<http://www.projectzero.org/zero/indy.dev/latest/update/zero.eclipse>
3. Expand the added URL. The window shown in Figure 10-10 opens. Select **WebSphere sMash** to install.

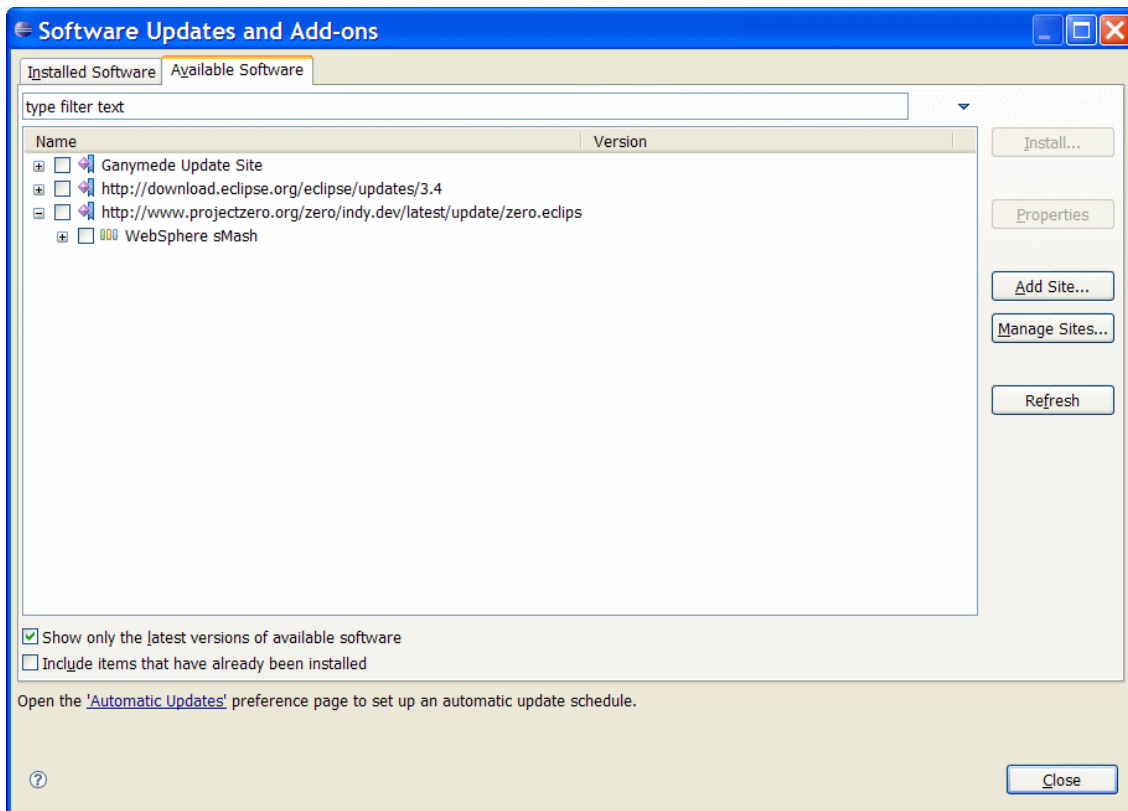


Figure 10-10 Installing IBM WebSphere sMash Eclipse plug-in

4. Click **Install** to begin the installation, and then follow the steps to restart Eclipse.

5. Configure CLI in the Preferences page:
  - a. Open the Preferences page by clicking **Windows** → **Preferences**.
  - b. Select **WebSphere sMash**, specify the zero home of the CLI, then click **OK**. See Figure 10-11.
6. Until you have added PHP support, repeat steps 2 - 4 by using the following URL and click **OK**. This step adds PHP support to the WebSphere sMash plug-in.

<http://www.projectzero.org/zero/indy.dev/latest/update/zero.eclipse.php>

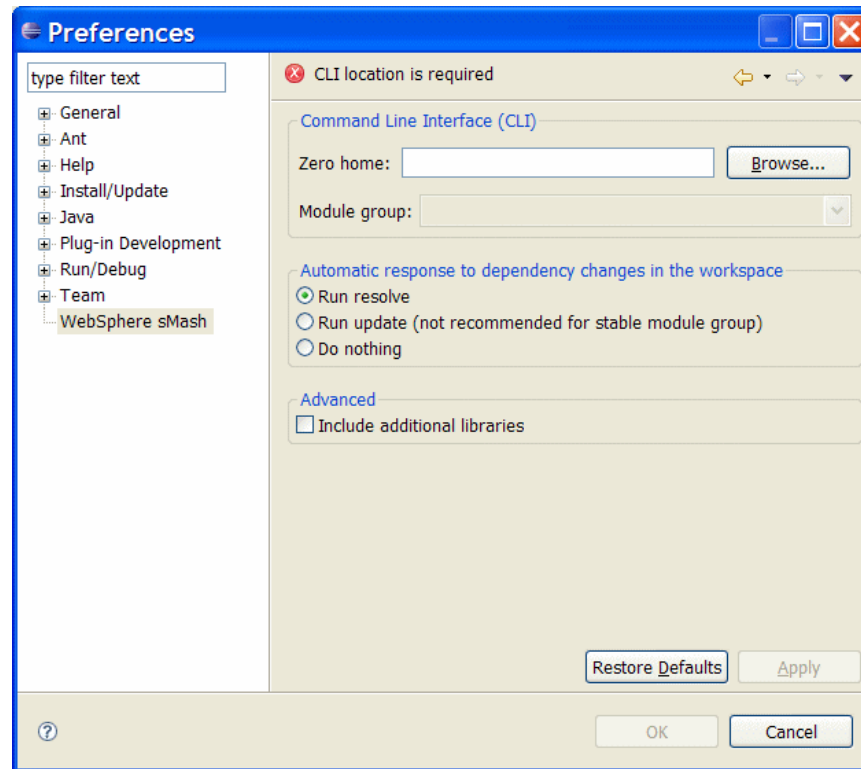


Figure 10-11 Preference page to configure CLI for Eclipse plug-in



## Advantages

Advantages are as follows:

- ▶ New WebSphere sMash applications can be created without using the command line.
- ▶ WebSphere sMash samples can be created easily by selecting **New** → **Other** → **Example**. A menu lists all available samples in the installation; see Figure 10-12.

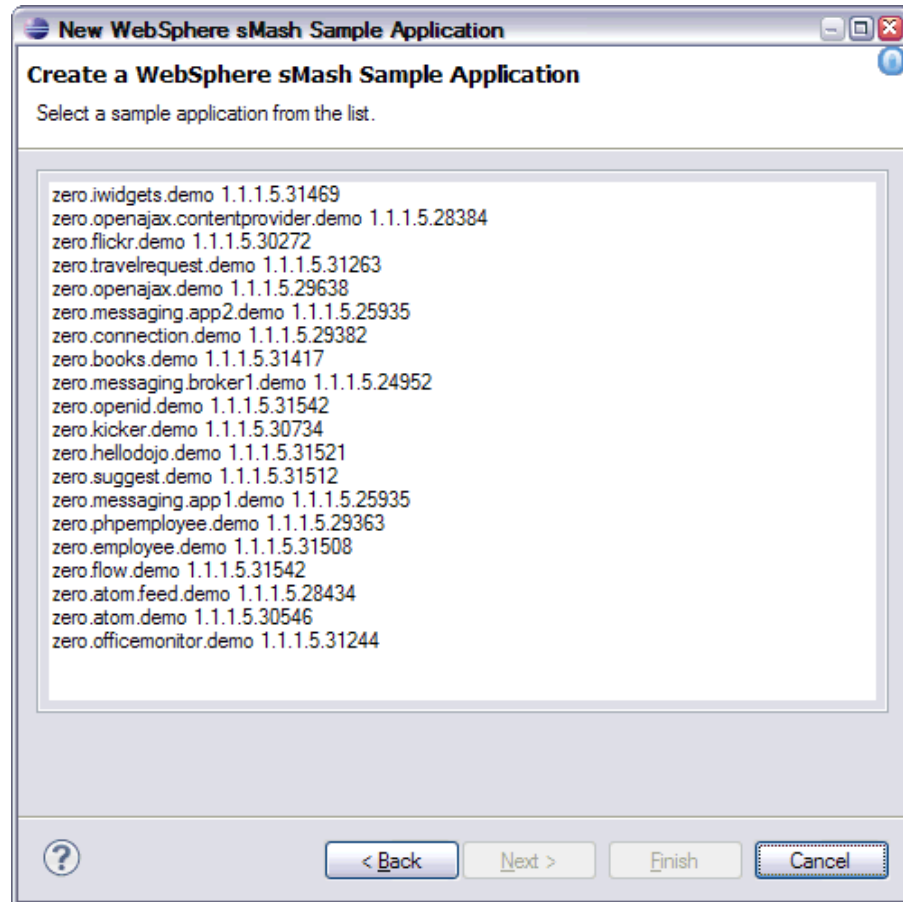


Figure 10-12 Creating a new Sample Application with the plug-in

- ▶ Dependency management is simplified greatly with a custom dependency editor.

## ***Disadvantages***

Disadvantages are as follows:

- ▶ Applications on remote systems cannot be managed in this way.
- ▶ The application will not have access to the CICS APIs and will not be running in CICS.
- ▶ You must add a PHP editor, Groovy editor, or debugger if you require them.
- ▶ The repository used by the plug-in is that of the CLI on your workstation, not that of Dynamic Scripting. Therefore, the same modules may not be available in both environments.

## **Using IBM Rational Developer for System z with Java**

The IBM z/OS Eclipse-based software development integrated development environment (IDE) is in widespread use, and can be used to develop dynamic scripting applications.

IBM Rational Developer for System z environment also empowers veteran time sharing option (TSO) developers. It does this through faithful interactive System Productivity Facility (ISPF)-emulation and exceptional tooling, for tough everyday z/OS software maintenance and support tasks (data flow analysis, control flow analysis, and so on). Through its GUI tools, the IBM Rational Developer for System z utility has helped refurbished the skill sets of veteran TSO developers, helping them become effective, contributing project staff members for today's application requirements.

IBM Rational Developer for System z already allows access to remote file systems (z/OS and UNIX System Services) and also provides two separate CLIs, a TSO shell, and a UNIX System Services shell.

## Configuring access to z/OS for Dynamic Scripting

When accessing the z/OS system, the very first step should be to create connections to your MVS system, as follows:

1. Change IBM Rational Developer for System z to z/OS perspective, and double click **New Connection** in Remote Systems view.

The new Connection window (Figure 10-13) opens.

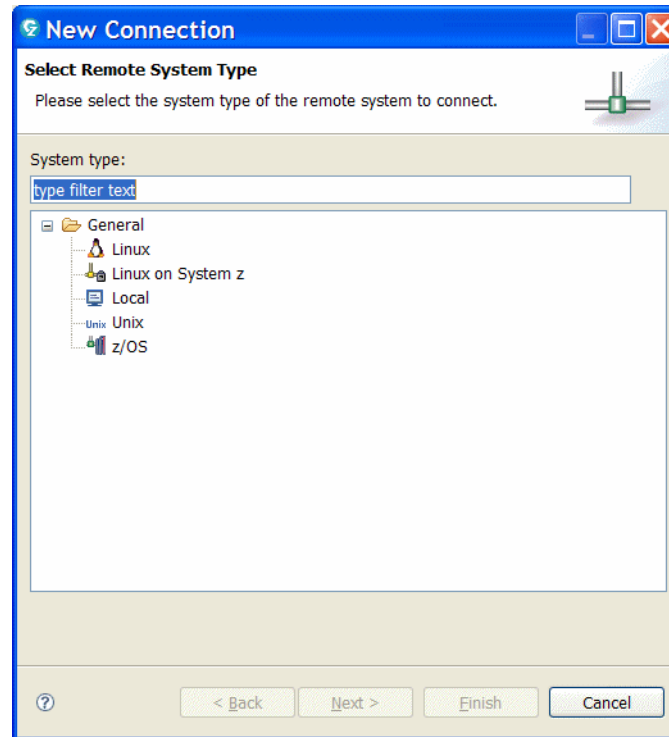
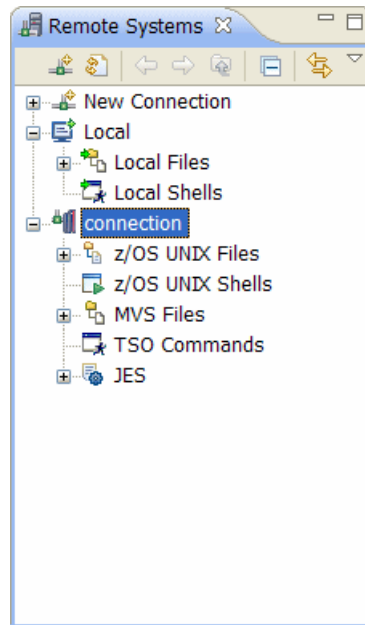


Figure 10-13 Creating new connections to z/OS through IBM Rational Developer for System z

2. Select z/OS, and click **Next**. Fill the Host name with your remote system name, and click **Next**.

**Note:** If you select the Verify host name check box, the IBM Rational Developer for System z will verify the host to the remote z/OS system. if valid the wizard has access to the z/FS file system and all JES pages.

3. After all this configuration, click **Finish** to close the Connection Creation wizard, then a connection with the name you specified is shown in the connection view. See Figure 10-14.



*Figure 10-14 New connection to z/OS finish page*

Right-click the new connection to open the connection properties window, as shown in Figure 10-15 on page 251. Set the default encoding to UTF-8. This step applies only to files in UNIX System Services and can be changed v if you are not connected. This way is convenient because most files in Dynamic Scripting are in ASCII/UTF-8 and not in EBCDIC.

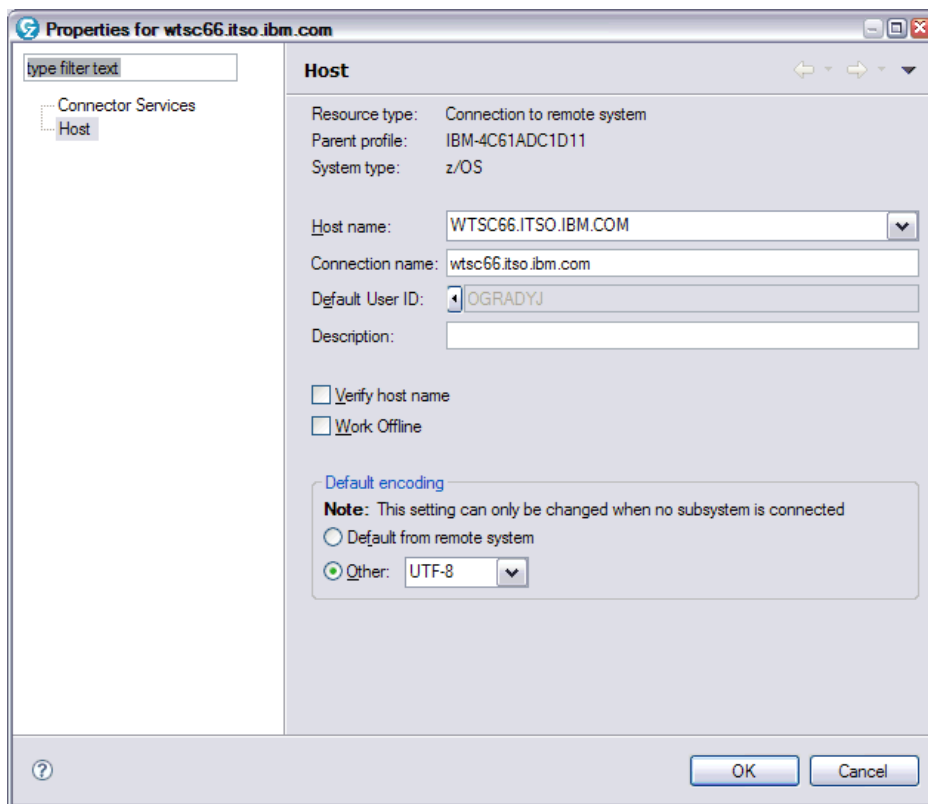


Figure 10-15 Changing the default encoding to UTF-8

More details and the newest release for IBM Rational Developer for System z are provided at the following address:

<http://www.ibm.com/developerworks/rational/products/rdz/release.html>

## 10.4 Dependencies

All the constituent parts of WebSphere sMash and Dynamic Scripting are built as modules. These modules have dependencies on each other.

For example, any application has a dependency on the following two modules:

- ▶ zero.core
- ▶ zero.cics.core

Any application using PHP has a dependency on zero.cics.php.  
zero.cics.php, which has a dependency on zero.php.

This dependency can be considered similar to the dependencies an application has on CICS, in terms of system initialization parameters and resources. CICS in turn has dependencies on z/OS, DB2, and Language Environment.

Dependencies may contain files and resources, commands, help files, and configuration information.

## 10.4.1 Configuration of dependencies

To add or administer the dependencies of your application, modify and manage the ivy.xml file by editing the <dependencies> section. If you use the WebSphere sMash Eclipse plug-in, an editor is provided, which can help you with these tasks.

The default ivy.xml file in Project Zero demo project is shown in Example 10-6.

*Example 10-6 The ivy.xml file from a dynamic scripting application*

---

```
<?xml version="1.0" encoding="UTF-8"?>
<ivy-module version="1.3">

  <info organisation="zero" module="demo" revision="1.0.0">
    <license name="type of license" url="http://license.page"/>
    <ivyauthor name="author name" url="http://authors.home.page"/>
    <description homepage="http://module.description.page"/>
  </info>

  <publications>
    <artifact type="zip" />
  </publications>

  <!-- Note: dependencies from maven require the maven2 notation. -->
  <dependencies>
    <dependency org="zero" name="zero.core" rev="[1.0.0.0,
2.0.0.0["/>
    <dependency org="zero" name="zero.cics.core" rev="[1.0.0.0, 2.0.0.0["/>
  </dependencies>
</ivy-module>
```

---

After editing ivy.xml, issue a **zero resolve** command. This command is explained in 10.6.1, “The zero resolve command” on page 259.

## 10.4.2 Frequently used dependencies

This section lists various dependencies.

### The `zero.cics.core` dependency

To run in CICS, all dynamic scripting applications must have a dependency on the `zero.cics.core` module. Applications that you create with the **zero create** command have this dependency by default.

Applications ported to a Dynamic Scripting environment from WebSphere sMash must have this dependency added.

The `zero.cics.core` module contains the default web page shown when no `index.html` file is provided by the application. This web page contains links to the IBM information center.

Example 10-7 shows the dependency in an `ivy.xml` file.

*Example 10-7 The `zero.cics.core` dependency*

---

```
<dependency org="zero" name="zero.cics.core" rev="[1.0.0.0, 2.0.0.0["/>
```

---

### The `zero.cics.php` dependency

If your application uses PHP, the `ivy.xml` file must include a dependency on the `zero.cics.php` module. Applications ported to a Dynamic Scripting environment from WebSphere sMash will need to have this dependency added in place of the existing `zero.php` module. Example 10-8 shows the dependency in an `ivy.xml` file.

*Example 10-8 The `zero.cics.php` dependency*

---

```
<dependency org="zero" name="zero.cics.php" rev="[1.0.0.0, 2.0.0.0["/>
```

---

### The `zero.core.webtools` dependency

The `zero.core.webtools` module provides utilities to assist in the development of dynamic scripting applications:

- ▶ A default web page that links to the utilities
- ▶ A Virtualized Directory viewer that shows the resources available to the resolved application, including resources provided by the dependencies.
- ▶ A Request Data viewer. This shows HTTP headers, cookies, and so on.
- ▶ Developer-centric error page. With `zero.core.webtools`, error pages contain more information.

Example 10-9 shows the dependency in an `ivy.xml` file.

*Example 10-9 Dependency*

---

```
<dependency org="zero" name="zero.core.webtools" rev="[1.0.0.0, 2.0.0.0["/>
```

---

### The zero.cics.db2 dependency

This dependency is mandatory when connecting to DB2 using Zero Resource Model (ZRM) or using the `zero.data` APIs and you are not explicitly creating multiple connections in the same thread. See Example 10-10.

*Example 10-10 The zero.cics.db2 dependency*

---

```
<dependency org="zero" name="zero.cics.db2" rev="[1.0.0.0, 2.0.0.0["/>
```

---

**Note:** For a full list of modules that are supported in Dynamic Scripting Feature Pack for CICS, go to the following information center address:

[http://publib.boulder.ibm.com/infocenter/cicsts/v4r1/index.jsp?topic=/com.ibm.cics.ts.smash.doc/supported\\_packages.html](http://publib.boulder.ibm.com/infocenter/cicsts/v4r1/index.jsp?topic=/com.ibm.cics.ts.smash.doc/supported_packages.html)

## 10.5 Module groups

This section describes module groups and how to manage them.

### 10.5.1 Overview of module groups

A *module group* is a local repository of modules, typically retrieved from remote repositories. An application must be resolved against one module group.

When installed, Dynamic Scripting contains only one module group, `stable`. Accordingly, all applications that are created are part of the `stable` module group.

IBM WebSphere sMash defines two module groups to isolate released code (stable module group) from the latest code (experimental). The experimental module group may also include trial modules, which provide interesting functionality for early community feedback but not yet planned for a release. When new applications are created in the App Builder, you can choose whether they are in the stable or the experimental module group.



New module groups can be created. Module groups can have their modules updated as a group from remote repositories. Each module can then be updated in turn.

## 10.5.2 The management of module groups

Managing module groups consists of creating them, switching between them, getting information about an application's module group, adding a repository, listing contents, updating, removing unused modules, and destroying the module group.

### Creating the module group

We create a new module group named *production*. This module group will represent applications that are running in our production environment. Module groups are created with the **zero modulegroup create** command. This command takes two arguments:

- ▶ Name of the module group
- ▶ Address of a repository that the module group will use.

Example 10-11 shows the creation of a new production module group with the `file:///` URI pointing at a location in the ZFS file structure.

*Example 10-11 Create production group*

---

```
zero modulegroup create production  
file:///dynamicScript/repositoryProduction/base
```

---

### Switching between module groups

It is possible to switch an application from one module group to another, but switching is not intended to be a common operation. Most developers use the free WebSphere sMash DE download, which provides tooling support and a stable run time for testing and running applications. The resultant applications may be deployed without change to the WebSphere sMash supported run time.

The one scenario where you might need to switch module groups is to convert an experimental application that started out as a trial exercise and developed into a significant component that you now want to deploy. You can use the **zero switch** command to switch the module group, but additional testing is required to ensure that the application runs correctly with the new module group.

You might also want to switch the module group of an application to ensure that it uses a separate local repository.

The **zero switch** command switches the module group of the application to the named module group, as shown in Example 10-12.

*Example 10-12 Switch between module group*

---

```
zero switch production
```

---

## Getting information about an application's module group

The **zero modulegroup info** command shows the following information about the application and its module group:

- ▶ The module name
- ▶ The location of the module group configuration
- ▶ Configured repository locations for the module group

Example 10-13 shows this information.

*Example 10-13 The zero modulegroup info command*

---

The module group name is production.

The location for the modulegroup production configuration is  
/dynamicScript/zero/zero-repository/production/site.xml.

The following are the list of resolvers for resolving maven artifacts:  
    resolver name is maven.3 and location is  
http://repo1.maven.org/maven2/

The following are the list of resolvers for resolving zero modules:  
    resolver name is zero.2 and location is  
file:/dynamicScript/repositoryProduction/base/

The chain resolver remote includes the following resolvers:  
    zero.2  
    maven.3

CWPZT0600I: Command modulegroup info was successful

---

## Adding another repository to the module group

Additional repository locations can be added to the module group. The command to do this is **zero modulegroup addurl** command. The repository can be an additional repository available on the Internet, or a locally configured repository.

WebSphere sMash provides a sample repository, which contains additional samples that can run inside the Dynamic Scripting Feature Pack. This repository is not available by default; it must be added before samples can be created.

Example 10-14 shows the command that is used to add the WebSphere sMash samples repository to the list for the current module group.

*Example 10-14 The zero modulegroup addurl to add samples repository*

---

```
zero modulegroup addurl  
http://www.projectzero.org/sMash/1.1.x/repo/samples/
```

---

## Listing the contents of the local repository

Each module group, including the default stable module group, keeps a copy of used module's in its local repository. The contents of this repository can be listed with the **zero modulegroup list** command.

Example 10-15 shows the response from this command for the stable module group at the end of the Redb repository.

*Example 10-15 The zero modulegroup list command*

---

```
itso:redb.commands:1.0.0  
org.apache.derby:derby:10.3.3.1  
dojo:dojo:1.1.1.26717  
dojo:dojo:1.2.3.30640  
dojo:dojo:1.4.1.31117  
zero:zero.network.support:1.1.1.3.30955  
zero:p8:1.1.1.3.31217  
zero:zero.management.spi:1.1.1.3.29370  
zero:doNothingDemo:1.0.0  
zero:zero.kernel:1.1.1.3.31215  
zero:zero.dojo:1.1.1.3.31196  
zero:zero.resource:1.1.1.3.31144  
zero:zero.cics.tsq.demo:1.0.0.0.4093  
zero:zero.cics.smpe.demo:1.0.0.0.1  
zero:zero.cics.redb.commands:1.0.0  
... continues ...
```

---

## Updating the local repository

The **zero modulegroup update** command can be used to refresh local copies of modules. For each module within the local repository, Dynamic Scripting checks the configured repositories and searches for a more recent version. These versions are then stored in the local repository. You might need to do this if IBM or another vendor provides updates to a module in a remote repository.

Example 10-16 shows the results of the **zero modulegroup update** command.

*Example 10-16 The zero modulegroup update command updates local repository*

---

```
CWPZT0896I: This operation can take some time to complete, please be
patient.
CWPZT0545I: Retrieving dojo-1.2.3.30640.zip from host
file:/dynamicScript/repository/base/
CWPZT0531I: Module List :
dojo:dojo:1.2.3.30640
CWPZT0600I: Command modulegroup update was successful
```

---

### Removing unused modules from the local repository

Because various versions of modules are used, possibly during the development cycle, old versions can build up within the local repository. For example, you might create an application and publish it as version 1. Later, you might add a Derby database to it and publish it as version 1.0.1. Later still, you add a VSAM resource handler and publish it as version 2.0.

As result, within the local repository, three versions would exist, as Example 10-17 shows. There are three separate copies of versionApp in the local repository for the module group.

*Example 10-17 The zero modulegroup list command shows multiple copies of module*

---

```
zero:versionApp:1.0.1
zero:versionApp:1.2.0
zero:versionApp:1.0.0
```

---

The **zero modulegroup prune** command can be used to remove unused versions of modules from the repository. Example 10-18 shows the results of a successful command.

*Example 10-18 The zero modulegroup prune command removes unused versions*

---

```
CWPZT0600I: Command modulegroup prune was successful
```

---

Example 10-19 shows the results of a **zero modulegroup list** command after a successful **zero modulegroup prune** command.

*Example 10-19 The zero modulegroup list shows only one version of versionApp*

---

```
zero:versionApp:1.2.0
CWPZT0600I: Command modulegroup list was successful
```

---

## Destroying module groups

The module group can be destroyed by **zero modulegroup destroy** command, as Example 10-20 shows.

*Example 10-20 Destroy experimental module group*

---

```
zero modulegroup destroy experimental
```

---

If the module group does not exist, the following error message is issued:

CWPZT0828E: Modulegroup, experimental, does not exist.

## 10.6 Dependency management commands

The commands are as follows:

- ▶ zero resolve
- ▶ zero version
- ▶ zero update
- ▶ zero switch target\_module\_group

### 10.6.1 The zero resolve command

The **zero resolve** command reads the declared dependencies in the `ivy.xml` file, and also any Java archive (JAR) files in the `lib` directory. The command first looks in the local repository for the latest version of a module that matches the version information within the `ivy.xml`. If one is found, that copy is used. Example 10-21 shows the messages that are issued when a local copy is not found.

*Example 10-21 The zero resolve command searching for modules*

---

```
CWPZT0901I: The following module(s) are not currently in the local
repository:
    zero:zero.cics.tsq.demo:[1.0.0.0, 1.2.0.0[
CWPZT0902I: Trying to locate the module(s) using one of the configured
remote repositories
CWPZT0545I: Retrieving zero.cics.tsq.demo-1.0.0.0.4093.zip from host
file:/dynamicScript/repository/samples/
CWPZT0600I: Command resolve was successful
```

---

**Note:** a repository reached by the `file:///` protocol, even though it is by definition present on the local file system, is still referred to as a remote repository. Remote repositories are *any* repositories that are not stored with installation directory, in the `zero-repository` folder.

After an application's dependency has been resolved, newer versions of the module in any remote repository will not be used by any subsequent **resolve** command. That is, you can issue the same **resolve** command and be sure that you will get the same version of the module unless you change the `ivy.xml`.

## 10.6.2 The zero version command

The **zero version** command can be used to list the exact versions of modules used by the command line. If issued from within an application, it will also list the exact versions of modules used by the application.

The `ivy.xml` file lists required dependencies and modules. The **zero version** command shows the versions that are currently in use. Example 10-22 shows the results of the **zero version** command issued from within an application. In addition the command shows the Java version in use.

*Example 10-22 The zero version command showing modules currently in use*

---

Command-line Version: 1.1.1.4.31129 20100601 0917

Command-line Information:

Name: zero.cics.cli

Version: 1.0.0.0.4172

Location: /dynamicScript/zero

Modulegroup: stable

Dependencies:

zero:zero.cics.cli.tasks:1.0.0.0.4171 (userhome)

zero:zero.cli.tasks:1.1.1.3.31130 (userhome)

zero:zero.kernel:1.1.1.3.31215 (userhome)

Module Information:

Name: versionApp

Version: 1.2.0

Location: /dynamicScript/apps/versionApp

Modulegroup: production

Dependencies:

zero:p8.lib:1.1.1.3.31217 (userhome)

zero:p8:1.1.1.3.31217 (userhome)

```

zero:zero.cics.cli.tasks:1.0.0.0.4171 (userhome)
zero:zero.cics.core:1.0.0.0.4171 (userhome)
zero:zero.cics.php:1.0.0.0.4093 (userhome)
zero:zero.cics.tsq.demo:1.0.0.0.4093 (userhome)
zero:zero.cli.tasks:1.1.1.3.31130 (userhome)
zero:zero.core:1.1.1.3.31203 (userhome)
zero:zero.data.php:1.1.1.3.30971 (userhome)
zero:zero.data:1.1.1.3.30742 (userhome)
zero:zero.kernel:1.1.1.3.31215 (userhome)
zero:zero.management.monitor:1.1.1.3.28430 (userhome)
zero:zero.management.native.process:1.1.1.3.30955
(userhome)
zero:zero.management.spi:1.1.1.3.29370 (userhome)
zero:zero.management.zso:1.1.1.3.30778 (userhome)
zero:zero.network.support:1.1.1.3.30955 (userhome)
zero:zero.network:1.1.1.3.31204 (userhome)
zero:zero.php:1.1.1.3.31175 (userhome)
zero:zero.test:1.1.1.3.31150 (userhome)

```

The java command path is /usr/lpp/java/J6.0/bin/java

```

java version "1.6.0"
Java(TM) SE Runtime Environment (build pmz3160sr8fp1-20100624_01(SR8
FP1))
IBM J9 VM (build 2.4, JRE 1.6.0 IBM J9 2.4 z/OS s390-31
jvmmz3160sr8ifx-20100609_59383 (JIT enabled, AOT enabled)
J9VM - 20100609_059383
JIT - r9_20100401_15339ifx2
GC - 20100308_AA)
JCL - 20100624_01

```

---

### 10.6.3 The zero update command

The **zero update** command searches the local repository for later versions of dependencies. If the module was created with a dependency on zero.cics.module version 1.0.0, and a new version becomes available in the local repository, for example zero.cics.module version 1.1.0, the **zero update** command must be used to replace the version 1.0.0 copy with the more recent version.

The **zero update** command does not, however, search the configured remote repositories. To get version 1.2.0 from the IBM repository into the application, **zero modulegroup update** and then **zero update** command must be used.

Example 10-23 shows the results of a successful **zero update** command. Note that no information about updated modules is provided.

*Example 10-23 Command successful*

---

```
CWPZT0600I: Command update was successful
```

---

## 10.6.4 The zero switch command

The **zero switch *target\_module\_group*** command is used to change the module group from the current module group to the specified module group. Each module group has a separate local repository and may have differently configured remote repositories. See Example 10-24.

*Example 10-24 The zero switch command*

---

```
zero switch production  
CWPZT0600I: Command switch was successful
```

---

**Note:** The **zero switch** command performs the equivalent of an **update** command after changing the module group. This succeeds as long as the dependencies can be satisfied from the new module group.

For more zero commands, go to the following address:

[http://publib.boulder.ibm.com/infocenter/cicsts/v4r1/topic/com.ibm.cics.ts.smash.doc/smash\\_clicommands.html](http://publib.boulder.ibm.com/infocenter/cicsts/v4r1/topic/com.ibm.cics.ts.smash.doc/smash_clicommands.html)

## 10.7 Moving applications to Dynamic Scripting

Applications may be developed entirely within Dynamic Scripting, using the command-line interface, IBM Rational Developer for System z, or Eclipse with Target Management.

Alternatively, applications can be developed off-platform and later ported to run under CICS. These applications might be developed using Eclipse or the App Builder.

The **zero package** command can be used to help move your applications. The **zero package** command produces a .zip file of the application within the export directory, which can then be transferred and extracted onto another server, which includes a ZFS structure on the mainframe.



The **zero package** command takes the following options:

- ▶ `shared | standalone`

For deploying to Dynamic Scripting, always select `shared`, which is the default. The `standalone` option will package native libraries with the application, which will then not be resolvable on the mainframe.

- ▶ `-includeSrc`

By default, Java source code within the `java` directory is *not* included within the package. The reason is so you can protect your source code if you want to distribute a dynamic scripting application. Specifying `-includeSrc` option instructs WebSphere sMash to include these files.

Perform the following steps:

1. Issue the **zero package** command in the application's home directory. See Example 10-25.

For example, to package application `demo` located at `C:\sMash\project\demo`, change directory to `C:\sMash\project\demo` then issue **zero package**.

*Example 10-25 invoke package command*

---

```
C:\>cd C:\sMash\project\demo
C:\sMash\project\demo>zero package
CWPZT0628I: Packaging C:\sMash\project\demo
CWPZT0504I: Build shared package demo
CWPZT0502I: Created shared package C:\sMash\project\demo\export\demo-1.0.0.zip
CWPZT0600I: Command package was successful
```

---

2. Transfer the generated `.zip` file to the mainframe. Console output from **zero package** includes a pointer to the generated file, which will be located in the application's export directory. Note the following information:
  - The file name of the `.zip` file includes the version number.
  - Ensure that the file transfer takes place in binary mode, otherwise corruption of the data might occur.
  - Ensure that the `.zip` file is stored in the correct location for your installation.
3. Extract the `.zip` file on the mainframe. The **jar** command extracts the `.zip` file into the current directory. Example 10-26 on page 264 shows example output. See Example 10-26 on page 264.

*Example 10-26 The jar -xvf command extracts the ZIP file in verbose mode*

---

```
jar -xvf demo-1.0.0.zip
created: demo/
  created: demo/.zero/
  created: demo/app/
  created: demo/classes/
  created: demo/config/
inflated: demo/config/ivy.xml
inflated: demo/config/php.ini
inflated: demo/config/zero.config
created: demo/lib/
created: demo/META-INF/
created: demo/public/
created: demo/.zero/shared/
inflated: demo/.zero/shared/MANIFEST.MF
inflated: demo/.zero/shared/resolve.history
created: demo/app/errors/
created: demo/app/resources/
created: demo/app/scripts/
created: demo/app/views/
```

---

4. The application must be resolved using the **zero resolve** command and possibly a **zero update** command before it can be started. See Example 10-27. An application cannot be started if it is not resolved.

*Example 10-27 The zero start command fails; resolve required*

---

```
CWPZC0019E: Cannot find handler for task start
CWPZI8503I: CICS unit of work rolled back after a CLI task returned
a non-zero return code.
```

---

The **zero resolve** might fail if the application already has satisfied dependencies at a higher level than are available in the local repositories. WebSphere sMash and Dynamic Scripting do not share the same repositories, and so distributed versions may have more up to date versions of modules.

Example 10-28 shows this happening. The `zero.core` was already in use at version 1.1.1.5.31479, but the latest available version was 1.1.1.3.31203. This is a different version and so does not satisfy the dependency.

*Example 10-28 The zero resolve fails because the current version is not available*

---

CWPZT0901I: The following module(s) are not currently in the local repository:

zero:zero.core:[1.0.0.0, 2.0.0.0[

CWPZT0902I: Trying to locate the module(s) using one of the configured remote repositories

CWPZT0671W: The module zero:zero.core:1.1.1.3.31203 was found, but is not considered for the resolve because version 1.1.1.5.31479 is specified in history file

CWPZT0671W: The module zero:zero.core:1.1.0.0.27578 was found, but is not considered for the resolve because version 1.1.1.5.31479 is specified in history file

CWPZT0671W: The module zero:zero.core:1.1.0.1.29295 was found, but is not considered for the resolve because version 1.1.1.5.31479 is specified in history file

CWPZT0671W: The module zero:zero.core:1.1.1.0.30349 was found, but is not considered for the resolve because version 1.1.1.5.31479 is specified in history file

CWPZT0671W: The module zero:zero.core:1.1.1.1.30754 was found, but is not considered for the resolve because version 1.1.1.5.31479 is specified in history file

CWPZT0915E: The module zero:zero.core can not be located in any of the configured repositories

CWPZT0916E: The resolve failed, but some modules were filtered because of the resolve history. You might want to consider using zero update instead.

CWPZT0808E: The resolve report has errors, the resolved properties will not be created.

CWPZT0537W: Resolve could not find the following dependencies

zero:zero.core:[1.0.0.0, 2.0.0.0[

CWPZT0601E: Error: Command resolve failed

CWPZI8503I: CICS unit of work rolled back after a CLI task returned a non-zero return code.

---

5. Use the **zero update** command to update the dependencies to the latest available dependencies.
6. The `ivy.xml` file must be updated to add or alter dependencies. Applications in WebSphere sMash all have a dependency on `zero.core`. Applications in Dynamic Scripting all have an additional dependency on `zero.cics.core`.

Failure to add `zero.cics.core` can result in a message appearing on a **zero start** command. Example 10-29 shows the results.

---

*Example 10-29 You cannot start an application without zero.cics.core*

---

```
CWPZC8029E: ZSO could not be located for the current platform of
OS=zos, arch=s390.
CWPZT0601E: Error: Command start failed
CWPZI8503I: CICS unit of work rolled back after a CLI task returned
a non-zero return code.
```

---

7. WebSphere sMash applications that use PHP have a dependency on `zero.php`. This must be replaced with `zero.cics.php` to run in Dynamic Scripting. Failure to do so will result in the application being unable to start, as can be seen in Example 10-30.

---

*Example 10-30 CWPZI8605E message*

---

```
CWPZI8605E: CICS sMash PHP application must have a resolved
dependency on zero.cics.php in its config/ivy.xml.
CWPZT0601E: Error: Command start failed
CWPZI8503I: CICS unit of work rolled back after a CLI task returned
a non-zero return code.
```

---

## 10.8 Dynamic scripting encoding considerations

Dynamic scripting is primarily a UTF-8 platform that runs in an EBCDIC environment. To run the application normally, the encoding of the application must be in correct format. As a result, additional application code that converts data between bytes and character data is essential.

On z/OS UNIX, the feature pack encodes all log files in UTF-8. Similarly, all files read from z/OS UNIX, including PHP and Groovy script files, Java source files, static HTML files, JSON data files, and configuration files are expected to be encoded in UTF-8. Therefore, you must view and edit these files by using a suitable method. For more information, see 6.6, “Editing an ASCII file on an EBCDIC system” on page 85.

When you use FTP to transfer dynamic scripting files to and from the z/OS platform, use binary mode so that the files are not transcoded.

If you use Eclipse with Target Management to edit the files directly on the mainframe, you can edit the files directly in ASCII.

## 10.8.1 Converting between Java Strings and byte arrays

Several standard Java library methods convert between Java strings and byte arrays by using the default encoding of the JVM. Pay particular attention to encoding issues if you use any of these methods. When possible, use overloaded versions that take a character set explicitly. See Example 10-31.

*Example 10-31 Java methods that might lead to encoding issues*

---

```
java.lang.String.getBytes()
java.io.FileReader(String filename)
java.io.FileWriter(String filename)
java.lang.String(byte[] bytes)
java.io.ByteArrayOutputStream.toString()
java.io.FileReader(String filename)
java.io.FileReader(File file)
java.io.FileReader(FileDescriptor fileDescriptor)
java.io.FileWriter(String filename)
java.io.FileWriter(File file)
java.io.FileWriter(FileDescriptor fileDescriptor)
java.io.InputStreamReader(InputStream input)
java.io.OutputStreamWriter(OutputStream output)
java.io.PrintStream(File file)
java.io.PrintStream(OutputStream output)
java.io.PrintStream(String string)
java.io.PrintWriter(File file)
java.io.PrintWriter(OutputStream output)
java.io.PrintWriter(String string)
java.util.Scanner(InputStream input)
java.util.Formatter(String filename)
java.util.Formatter(File file)
java.util.Formatter(OutputStream output)
```

---

In Groovy, you may use the code in Example 10-32 to obtain IBM1047 EBCDIC bytes from a Java String.

*Example 10-32 Explicitly setting encoding when creating IBM1047 EBCDIC bytes*

---

```
def myBytes = myString.getBytes("IBM1047")
```

---

**Note:** Various third-party Java libraries are not written in a character set independent manner, and can fail if the default platform encoding is not ASCII-compatible. These libraries require that the default character set of the JVM is changed to ASCII using the `file.encoding` system property, for example by running with the JVM option:

```
-Dfile.encoding=UTF-8
```

Setting the `file.encoding` system property to an ASCII-compatible character set affects the behavior of the JCI CS library, and is not supported in CICS.

## 10.8.2 Working with character data in PHP

A characteristic of the PHP language is that PHP strings are raw byte sequences. The PHP strings can contain any sequence of bytes and are not tied to a specific character set.

By convention, PHP string literals in Dynamic Scripting are UTF-8 byte sequences. The script in Example 10-33 demonstrates this fact by using the PHP function `bin2hex`, which returns the hexadecimal representation of a PHP string.

*Example 10-33 The `bin2hex` method in PHP returns hexadecimal values*

---

```
<?php
$ascii = 'ABC';
echo bin2hex($ascii); // prints 414243, the hex values of ABC in ASCII
```

---

If you want to convert a string to EBCDIC bytes, you can use the PHP function `mb_convert_encoding` in Example 10-34.

*Example 10-34 Example to convert string to EBCDIC bytes*

---

```
<?php
$ascii = 'ABC';
$ebcdic = mb_convert_encoding($ascii, 'IBM1047', 'UTF-8');
echo bin2hex($ebcdic); // prints c1c2c3, the hex values of ABC in
EBCDIC
```

---

## 10.8.3 PHP strings and the PHP-Java bridge

In dynamic scripting applications, you can use the PHP-Java bridge to invoke Java methods from PHP scripts.

## Passing a PHP string to a Java method that takes a byte array

When passing a PHP string to a Java method that takes a Java byte array, the method receives the bytes of the PHP string unmodified.

If the PHP string is from a string literal, the Java method receives an array of UTF-8 bytes.

## Passing a PHP String to a Java method that takes a Java String

When passing a PHP string to a Java method that takes a Java String, the PHP-Java bridge converts the bytes to a Java string, interpreting them as UTF-8 bytes.

If a PHP string contains EBCDIC bytes, you must use `mb_convert_encoding()` to transcode it to UTF-8 before passing it to a Java method that expects a Java String.

## Passing a PHP String to overloaded Java methods

Be particularly attentive when calling overloaded Java methods that can take either a byte array or a Java string.

See Example 10-35 regarding the `Container.put()` in JCICS.

*Example 10-35 Problematic sample of method `Container.put()` in JCICS*

---

```
<?php
java_import('com.ibm.cics.server.Task');
$myBinContainer = Task::getTask()
    ->createChannel("MYCHANNEL")
    ->createContainer("MYCONT");

$myBytes = 'My data';

// Call Container.put(String value)
$myBinContainer->put($myBytes);

// Prints garbage - the container contains corrupt bytes
echo $myBinContainer->get();
```

---

The PHP-Java Bridge defaults to calling the string version. In this example, the default is problematic, because the PHP-Java bridge correctly interprets the bytes as UTF-8 when converting them to a Java string. However, JCICS then converts that string back to a byte array using the EBCDIC encoding of the JVM, which corrupts the data.

To fix this problem, the invoke the `JavaSignature` class (Example 10-36), so that it selects the version of `Container.put()` that takes a byte array to ensure that JICIS does not transcode the data, and therefore prevents corruption.

*Example 10-36 Fix overload of `Container.put()` by `JavaSignature` class*

---

```
<?php
java_import('com.ibm.cics.server.Task');
$myBinContainer = Task::getTask()
    ->createChannel("MYCHANNEL")
    ->createContainer("MYCONT");

$myBytes = 'My data';

// Use the JavaSignature() construct to select an overloaded method:
$byteArraySig = new JavaSignature(JAVA_BYTE | JAVA_ARRAY);
// Call Container.put(byte[] value)
$myBinContainer->put($byteArraySig, $myBytes);

// Prints 'My data' - the container contains the bytes we expect
echo $myBinContainer->get();
```

---

### Calling a Java method that returns a String

When calling a Java method that returns a Java String through the PHP-Java bridge, the PHP call site receives a PHP string containing a sequence of UTF-8 bytes.

## 10.8.4 Character sets of HTTP responses

HTTP text responses is the encoding of your choice, but you must set the `charset` parameter of the `Content-Type` HTTP response header.

Few HTTP clients expect or support EBCDIC. To ensure that your application can be accessed by a wide range of clients, respond with UTF-8 or ISO8859-1 data. If your script writes character data to the response stream without specifying an encoding, UTF-8 is used.

By placing this Groovy script in `/application_directory/public` and calling it with an HTTP request, a UTF-8 response is sent, as shown in Example 10-37.

*Example 10-37 Receive UTF-8 characters*

---

```
println "Hello World"
```

---



Using this Groovy script, a UTF-16 response is sent (Example 10-38).

*Example 10-38 Receive UTF-16 response*

---

```
request.headers.out.'Content-Type' = "text/plain; charset=UTF-16"  
def bytes = "Hello World".getBytes("UTF-16")  
request.outputStream[].write(bytes)
```

---

**Note:** For further reading and detailed code examples of encoding issues in dynamic scripting applications, read the topic in the information center:

[http://publib.boulder.ibm.com/infocenter/cicsts/v4r1/index.jsp?topic=/com.ibm.cics.ts.smash.doc/smash\\_dynamicscripting\\_encodingconsiderations.html](http://publib.boulder.ibm.com/infocenter/cicsts/v4r1/index.jsp?topic=/com.ibm.cics.ts.smash.doc/smash_dynamicscripting_encodingconsiderations.html)





## Derby and DB2, create easy example

Various techniques are available for accessing DB2 with sMash including ZRM and Zero.data and also through the PHP APIs and ibm\_db2 interface.

## 11.1 Database access

The CICS Dynamic Scripting Feature Pack can connect to databases from a variety of vendors. In this book, we focus on various methods of interacting with the data stored in the embedded Derby database and also DB2 databases. This chapter covers setting up a connection to the database and also both of the following methods of interacting with data in a database:

- ▶ Zero Resource Model (ZRM)

The ZRM provides a way of creating a RESTful interface to your data with a minimal amount of coding, allowing the developer to focus on the application and not the details of data storage.

- ▶ The zero.data API

The zero.data API is a layer on top of Java's standard JDBC interfaces that simplifies data access. Additionally, JDBC may also be used to access the data. However, because using JDBC in the dynamic scripting environment is no different from other Java environments, we do not cover examples in this book.

## 11.2 Configuring a connection

A dynamic scripting application uses the embedded Derby database by default, unless extra configuration is done. If you intend to use the Derby database, the only extra configuration needed depends on the method that you choose to use for accessing the data. These methods are described in more detail later in this chapter.

### 11.2.1 Configuring a DB2 connection

If you want to use DB2 with a dynamic scripting application, the following configuration is required:

1. The application configuration file, `zero.config`, requires the lines in Example 11-1 to be able to access DB2.

*Example 11-1 DB2 Access*

---

```
/config/db/mydb = {  
    "class" : "com.ibm.db2.jcc.DB2DataSource"  
}  
  
/config/resource/dbKey = "mydb"
```

---

Depending how user IDs are configured on your system, you may have to specify the schema under which SQL statements are to be executed from the dynamic scripting application. This step can be achieved by adding the following line to the connection specification in `zero.config` file:

```
"currentSchema" : "CICSR9"
```

The `driverType` property under `/config/db/mydb` is set to 2 by default. This setting causes the DB2 JDBC driver to behave as a type 2 driver, which is required under CICS. Do not attempt to configure a type 4 connection to DB2 because CICS accepts only a type 2 connection. Most properties of the connection are configured in CICS.

2. Dynamic scripting applications configured to use DB2 will require access to the DB2 client driver. Perform the following steps:

- a. Copy the Java libraries from your DB2 installation into the `lib` folder of the application, or use symbolic links. The following command sets up symbolic links for you:

```
ln -s db2_installation/classes/* application_home/lib
```

In the command, `db2_installation` is the location where DB2 is installed on your system, and `application_home` is the root folder of your dynamic scripting application.

- b. Add the DB2 JNI library to your application, because it is a requirement for the type 2 JDBC driver. This step involves creating a folder with the following command:

```
mkdir -p application_home/lib/s390/zos
```

- c. Create a symbolic link to the binary by using the following command:

```
ln -s db2_installation/lib/libdb2jcct2zos.so  
application_home/lib/s390/zos
```

3. Add the required module dependencies to your application. Add the following line to the `config/ivy.xml` file:

```
<dependency name="zero.cics.db2" org="zero" rev="[1.0.0.0,  
2.0.0.0["/>
```

4. Issue the **zero resolve** command for the module to be downloaded from the repository.

The zero.cics.db2 module has the configuration, shown in Example 11-2, in the zero.config file. Any application that uses DB2 must adhere to this configuration.

*Example 11-2 The zero.config file*

---

```
# Add this module's DataSource provider to the list of providers.
/config/data/providers/datasource += {
    "cdsp" : "zero.cics.db2.CICSDataSourceProvider"
}

# Configure the default DataSource provider to the one provided in
this module.
# Similar configuration might be needed elsewhere if the default
provider isn't used.
/config/defaults/datasource/provider = "cdsp"
```

---

## 11.2.2 Validating the connection

To test that the steps (from 11.2.1, “Configuring a DB2 connection” on page 274) to determine whether configuring the connection has worked, issue the **zero validatedb** command. This command verifies a successful database connection. If successful the output is similar to the following output:

```
/dynamicScript/apps/db2Chapter>zero validatedb mydb
CWPZC9069I: Successfully connected to configured database -> mydb
CWPZT0600I: Command validatedb was successful
```

## 11.2.3 Connection limitation

A Java application running in CICS may have at most one JDBC connection or SQLJ connection context open at any one time. This limitation applies to each JVM and therefore also to the JVM server in which a dynamic scripting application runs. Only connections to DB2 are affected by this limitation; Derby remains unaffected. If your application attempts to create multiple connections to DB2 the following exception is thrown:

```
com.ibm.db2.jcc.a.SqlException: [jcc] [50053] [12310] [3.52.99] T2z0S
exception:
[jcc] [T2zos] com.ibm.db2.jcc.t2zos.T2zosConnection.initialize: Multiple
Connections are disallowed in current pre-existing attachment
environment ERRORCODE=-4228, SQLSTATE=null
```

To avoid this exception, you can configure your application to use one of the following solutions:

- The first solution is to configure the JVM server, in which the dynamic scripting application is running, to have a thread limit of 1. This limit means that if only one thread is executing in the application, multiple connections are avoided. To achieve this result, add the following line to the `zerocics.config` file in the JVMSERVER stanza:

```
THREADLIMIT(1)
```

The obvious drawback to this solution is that it restricts the application to running single-threaded all the time, even when no DB2 access is required.

- The second solution is to use an enqueue around the code that accesses DB2 ensuring that code making use of a connection is single-threaded, thus having only one connection. This solution can be achieved with the code in Example 11-3.

*Example 11-3 Enqueue around the code*

---

```
import com.ibm.cics.server.NameResource;

String application = GlobalContext.zget("/config/name");
NameResource myNameResource = new NameResource();
myNameResource.setName(application);
myNameResource.enqueue();

// Code to access DB2
ZRM.delegate()
// End of DB2 code

myNameResource.dequeue();
```

---

This code obtains the name of the application and, using the JCICS class `NameResource`, creates an enqueue on the name of the application. After the code to access the database has executed, the enqueue can be released.

## 11.3 Using ZRM for data access

With ZRM, you can quickly create a RESTful interface to your data. With little programming, you can add list, create, receive, update, and delete (LCRUD) operations to your data and store it in a database. This RESTful service can be implemented with only two files:

- ▶ A model definition to define the resource model
- ▶ A resource handler that defines the operations to be performed

We examine how you implement both of these files. In addition to the RESTful service, ZRM also provides database table creation functionality.

### 11.3.1 Obtaining ZRM functionality

To use the functionality offered by ZRM, you must add the following dependency to the application's `ivy.xml` file:

```
<dependency name="zero.resource" org="zero" rev="[1.0.0.0, 2.0.0.0["/>
```

After adding this dependency, issue the **zero resolve** command to ensure that this module is retrieved from the repository.

Notice that when you issue the **ZERO** command from within your application, the list of valid commands now includes the **model** command. A more detailed explanation of this command is covered in 11.3.4, “CLI commands” on page 282.

### 11.3.2 Creating a model definition

For ZRM to function, it needs an explanation of the data that it is receiving. This step is achieved by creating a JavaScript Object Notation (JSON) file in the `app/models` folder of your application. This JSON file is referred to as a model definition because it defines the fields in this resource model.

The model definitions are used as the basis for creating the tables in the database, and also for allowing the application to understand the data it is receiving.



Example 11-4 is a basic JSON model definition.

*Example 11-4 JSON model*

---

```
{
  "fields": {
    "first_name": {"type": "string"},
    "second_name": {"type": "string"}
  }
}
```

---

The JSON model definition must contain a "fields" list. Within this list are the fields that this model declares. The basic example shows only two fields, both of "string" type. A model definition might be more complex and specify certain constraints on the field. Example 11-5 is a more complex example to illustrate this point.

*Example 11-5 String fields*

---

```
{
  "fields" : {
    "name": {"type": "string", "max_length":50},
    "date_of_birth": {"type": "date"},
    "shift_start": {"type": "time"},
    "shift_end": {"type": "time"},
    "hourly_pay": {"type": "decimal", "max_digits": 4,
"decimal_places": 2},
    "house_number": {"type": "integer"},
    "street_name":{"type":"string", "max_length": 60},
    "city": {"type": "string", "max_length": 45},
    "state": {"type": "string", "format":"region"},
    "phone": {"type": "string", "format":"phone"},
    "email": {"type":"string", "format":"email", "required":false},
    "additional_info": {"type": "string", "format": "large"}
  }
}
```

---

The more complex example has introduced several new types and constraints that can be applied to the fields.

## Valid field types

Within a model definition, the types listed in Table 11-1 are valid.

Table 11-1 Valid types

Type	Description
string	For character-based values. May take the max_length and format parameters.
boolean	A true or false field. Valid values are as follows: <ul style="list-style-type: none"><li>▶ true, 't', 'true', 'True', '1', or 1</li><li>▶ false, 'f', 'false', 'False', '0', or 0</li></ul>
date	A field representing the date in the form yyyy-MM-dd. May take the parameters auto_create and auto_update.
date-time	A time stamp field. Valid values are of the following forms: <ul style="list-style-type: none"><li>▶ yyyy-MM-dd</li><li>▶ yyyy-MM-dd HH:mm</li><li>▶ yyyy-MM-dd HH:mm:ss.</li></ul>
time	A field representing the time, independent of a specific date. Valid values may be in the form of HH:mm and HH:mm:ss.
decimal	Represents a decimal number. Requires the parameters max_digits and decimal_places.
integer	A type for integer values.
float	A type for floating-point numbers.

## Field parameters: string

The string type may take the following optional parameters:

**max\_length**      Maximum number of characters allowed in this field.

**format**            Applies specific format validation to the string.

The format parameter may take the following values to alter the validation of the field:

**large**             Is for very large amounts of character-based data.

**email**             Uses a regular expression to validate the data.

**Phone**            Uses a regular expression to validate the data.

**Region**           Is a two-character abbreviation for a U.S. state.

**Any**               Stores large byte arrays. When transferred through HTTP this format is BASE64-encoded.

The formats described that use regular expressions during validation may have the default expressions overridden by modifying the application's `zero.config` file. For example, the following line means that string types of the email format will be validated with the regular expression specified:

```
/config/resource/field/types/string/email/regex =  
"\\w+([-+.]\\w+)*@\\w+([-+.]\\w+)*\\.\\w+([-+.]\\w+)*"
```

### Field parameters: decimal

The decimal type requires the following parameters:

**max\_digits**      The maximum number of digits allowed in the number.

**decimal\_places**    The number of decimal places to store.

### Field parameters: date

The date field may take the following optional parameters:

**auto\_create**      When a member is created this field will be set to the current date.

**auto\_update**      When a member is created and then subsequently updated, this field will be set to the current date.

**Note:** These parameters are mutually exclusive.

### Additional options

The options in Table 11-2 apply to fields of all types.

Table 11-2 Options

Option	Description
required	If false, this field does not require a value. If true then a value must be specified.
label	A user-friendly name for the field. If none is provided, ZRM uses the field name as a label.
description	Allows a programmer to provide a more detailed description of the field. Useful for documentation and also providing assistance for the user.
default_value	Used when a member is created and no value has been specified for the field.

### 11.3.3 Creating a resource handler

As discussed previously in this chapter, ZRM requires a file to define the operations to be performed on the resource. This step is the responsibility of the resource handler. Resource handlers are stored in the `app/resources` folder of the application and define the list, create, retrieve, update, and delete (LCRUD) operations.

The most basic resource handler includes the following single line of code:

```
ZRM.delegate();
```

This code enables HTTP access to the resource through a RESTful interface, allowing you to perform the LCRUD operations on the resource collection by varying the URL and HTTP method. Table 11-3 shows how the various HTTP methods and URIs map to the methods within the resource handler.

*Table 11-3 Mapping HTTP and URI to methods*

HTTP	URI	Groovy method	Description
GET	/resources/messages	onList()	Lists all members of the collection
POST	/resources/messages	onCreate()	Creates a new member
GET	/resources/messages/1	onRetrieve()	Retrieves one member
PUT	/resources/messages/1	onUpdate()	Updates a member
DELETE	/resources/messages/1	onDelete()	Deletes a member

In Example 11-5 on page 279, the responsibility for the resource handler was delegated to ZRM and the individual methods were not implemented. However, it is possible to write custom resource handlers, implementing only those methods which are applicable. For example, you might not want items being modified after they have been added, therefore the implementation for the `onUpdate()` method would be omitted.

### 11.3.4 CLI commands

ZRM provides the functionality for creating a RESTful interface to your resources, and also the functionality for creating and managing the database artifacts to backup these resources.

## The zero model sync command

This command creates the necessary database tables based on the model definitions it finds in the `app/models` folder of the application. Additionally, it populates those tables with data found in the `initial_data.json` file.

The `initial_data.json` file must be placed in the `app/models/fixtures` folder of your application. When the **zero model sync** command is issued, the data in this file is used to populate the database. One file can be used to populate the tables of multiple resource types. Example 11-6 shows the contents of an `initial_data.json` file.

*Example 11-6 JSON sample*

---

```
[
  {
    "type": "lecturer",
    "fields": {
      "username": "JerryCuomo",
      "firstname": "Jerry",
      "lastname": "Cuomo",
      "office": "MB205",
      "phonenummer": "1-XXX-XXX-XXXX"
    }
  },
  {
    "type": "student",
    "fields": {
      "username": "JasonMcGee",
      "firstname": "Jason",
      "lastname": "McGee",
      "student_id": "T296614"
    }
  }
]
```

---

Each instance has a type specified, which has a corresponding model definition in the application. Each of the fields has a value in accordance with the model definition defining that type. When the **zero model sync** command is run, the tables for the lecturer and student resources contain one entry each.

## The zero model reset command

This command restores the database artifacts to the state that they were in immediately after the **zero model sync** command was issued. This is the equivalent of dropping the tables and issuing a **zero model sync** command.

## The zero model loaddata command

This command loads data from the specified file or files into the database. For example:

```
zero model loaddata test_data.json
```

If the specified file cannot be found in the working directory, the command searches the `app/models/fixtures` directory for it. Further files can be added to the end of the command as extra arguments.

## The zero model dumpdata command

This command writes the contents of the database to a file, in JSON format. To print out all of the data to a single file, use the following variation of the command:

```
zero model dumpdata output.json
```

To target a specific model specify a value for the `--model` argument on the command as follows:

```
zero model dumpdata --model=staff staff_output.json
```

Alternatively, using the `--split` argument of the command you are able to dump all of the data into separate files:

```
zero model dumpdata --split --prefix=output app/models/fixtures
```

This command produces a JSON file for each model type that is defined in the application. For example, if the application has a `staff` model defined, all member instances of type `staff` would be written to `output_staff.json`. The `--prefix` argument is optional and means that each file will have the prefix specified in the command. If `--prefix` is not specified, the default value is `dump`.

## The zero model sql command

This command outputs the SQL statements to create and drop tables when the other zero model commands are issued. The command is used as follows:

```
zero model sql {sync|drop|reset} filename
```

For example, the following command produces the SQL that is used when the **zero model sync** command is run:

```
zero model sql sync sync.sql
```

This command writes the SQL statements to a file called `sync.sql` in the working directory so you can view the statements before they are executed and edit them if required. If you choose to edit them, you may then run them manually against the database. Details of are covered in 11.5, “Running SQL directly against the database” on page 296.

## Editing SQL statements

One instance where editing the SQL can be useful is when you are using DB2. If the user ID under which you are running the ZERO commands does not have authority to create tables in the default storage group, you will not be able to run the **zero model sync** command. Instead if you obtain the SQL statements that this command will use you can modify the storage group being used and then run the SQL manually. For instance, you may add something similar to the following example before the CREATE TABLE SQL statements:

```
SET CURRENT SQLID = 'USER123';  
CREATE STOGROUP STAFF VOLUMES('*', '*', '*') VCAT DSN910PB;  
CREATE TABLESPACE STAFF USING STOGROUP STAFF;
```

The first statement changes the SQL authorization ID of the process, which might or might not be needed depending on the user IDs that you have set up. The second statement is creating a storage group, which is then used by the third statement to create a table space.

Details of how to run SQL statements manually are in 11.5, “Running SQL directly against the database” on page 296.

### 11.3.5 ZRM and existing data

When ZRM is used, it automatically adds two extra fields to the model definitions. For example, when the **zero model sync** command is used to build the database tables, the table for any given resource will contain the fields that are defined in the model definition, and also an ID and an updated field.

The ID field is an integer that begins at 100 and increments by one for each instance added. The updated field is a date-time field that stores a time stamp for when the instance was last updated (or created).

This might be a problem when you try to use ZRM to access existing data, because existing tables are unlikely to have these two fields. One option is to alter the existing tables and add those fields that are required by ZRM. An alternative option is to use the zero.data API to implement the individual methods within the resource handler. This way provides much more control over the data that is being written to the database. An example of how to implement these methods, using the zero.data API, is provided in 11.4.3, “Using zero.data in Groovy” on page 288 and 11.4.4, “Using zero.data in PHP” on page 292.

### 11.3.6 Testing ZRM

If you want to test the RESTful interface, you may use an HTTP client to post data to the collection. Figure 11-1 uses Poster, an add-on for Firefox, which posts the data to the collection to create a new entry.

This example uses the model definition described earlier in this section and creates a new employee record in the database. Note the use of quotation marks around all non-numeric data. If the input data is badly formed, you receive an HTTP response of 400 Bad request, indicating that some change to the data is required. Additionally, you receive the same response if the input data fails to conform to the constraints placed on the field in the model definition.

The image shows the Poster application window, which is used for testing HTTP requests. The window has a title bar with the name "Poster" and a close button. The main area is divided into several sections:

- Request:** This section contains a text field for the URL, which is set to "http://example.com:19000/resources/employee". Below the URL field are fields for "User Auth:" with the value "AUSER" and a password field filled with dots. There is also a "Google Login" button. A "Timeout:" slider is set to 30. Below these are "Settings:" buttons for "Save", "Import", and "Store".
- Actions:** This section contains buttons for "GET", "POST", "PUT", and "DELETE", along with a "Submit" button. The "POST" button is currently selected.
- Content to Send:** This section has tabs for "Content to Send", "Headers", and "Parameters". Under "Content to Send", there is a "File:" field with a "Browse..." button, a "Content Type:" field set to "text/xml", and "Content Options:" buttons for "Base64" and "Parameter Body". Below these is a text area containing a JSON object: 

```
{ "name": "Adam Rice", "date_of_birth": "1900-01-01", "shift_start": "08:00", "shift_end": "17:00", "hourly_pay": 5.67, "house_number": 20, "street_name": "New Road", "city": "Southampton", "state": "NY", "phone": "423-4567", "email": "adamrice@uk.ibm.com", "additional_info": "No additional info" }
```

Figure 11-1 Poster add



## 11.4 Using the zero.data API for data access

The CICS Dynamic Scripting Feature Pack also provides a data access API, referred to as zero.data, which is a thin wrapper around SQL that simplifies data access. Using the zero.data.groovy.Manager class you are able to manipulate your data with very little additional supporting code that is typically needed when using JDBC.

### 11.4.1 Obtaining the zero.data functionality

To obtain zero.data API functionality is contributed through the zero.data module. Add the following dependency to the `ivy.xml` file in your application to obtain this API:

```
<dependency org="zero" name="zero.data" rev="[1.1.1.4, 2.0.0.0[" />
```

After adding this line, issue the **zero resolve** command to ensure that this module is retrieved from the repository.

### 11.4.2 Configuring the connection

Unlike ZRM, the zero.data API does not provide functionality for creating database artifacts, so we must connect to an existing database. If you are using DB2, your connection is already configured by following the relevant information in 11.2.1, “Configuring a DB2 connection” on page 274.

Example 11-7 is what must be added to the `zero.config` file if you are using the embedded Derby database with zero.data.

*Example 11-7 Derby DB use*

---

```
/config/resource/dbKey = "employee_db"

/config/db/employee_db = {
    "class" : "org.apache.derby.jdbc.EmbeddedDataSource",
    "databaseName" : "db/resource"
}
```

---

This addition provides a key, by which you can access the database and also specifies the database connection. In 11.3.5, “ZRM and existing data” on page 285, we created a database using ZRM’s **zero model sync** command, which is the database we reference with “db/resource” in Example 11-7.

### 11.4.3 Using zero.data in Groovy

To illustrate the use of the zero.data API, we implement each of the methods for the LCRUD operations in a resource handler using the Groovy API. The following method implementations would be written in the resource handler, replacing the `ZRM.delegate()` line.

#### Implementing onList method

The `onList()` method returns all elements within a collection. If this method is defined within a resource handler, it is invoked when an HTTP GET request is made against the collection. Example 11-8 shows implementation of this method.

*Example 11-8 HTTP get example*

---

```
def onList() {
    try {
        // Get configured DataManager for data access
        def data = zero.data.groovy.Manager.create('employee_db')

        // Retrieve employee records via Data Zero
        def result = data.queryArray('SELECT * FROM employee')

        request.view = 'JSON'
        request.json.output = result
        render()
    } catch (Exception e){
        if (e.getCause() instanceof java.sql.SQLException) {
            request.status = HttpURLConnection.HTTP_INTERNAL_ERROR
            request.error.message = "The db may not have been
initialized."
            request.view = "error"
            render()
        }
    }
}
```

---

The first step is to obtain an instance of a Manager object. This is achieved by using the `create` method in the Manager class and passing it the database key, in this case `employee_db`, as specified in the `zero.config` file.

After we have an instance of the Manager we can use it to execute SQL against the database. In this case, as the method must return all of the members of the collection we use a `SELECT` without a `WHERE` clause to return everything. The method `queryArray` will return the result set in an array. This can then be easily be returned in the JSON format using a view renderer.

This use of a view renderer to return the object in the JSON format can be seen in the last three lines of the try block. The first of these lines sets the view renderer to the JSON type. The next line sets the object to be serialized to the output stream, and the final line invokes the renderer.

## Implementing onRetrieve method

The `onRetrieve()` method is implemented to return a single element of the collection. Similar to the `onList()` method, it is invoked when the application receives an HTTP GET request, but the difference here is that the request is for an individual element. For example the request might use a URI similar to the following line:

```
/resources/employee/100
```

In the line, `employee` is the name of the collection and `100` is the ID of one of the elements within that collection. Example 11-9 shows an implementation of this method.

*Example 11-9 The onRetrieve method*

---

```
def onRetrieve() {
    String id = request.params.employeeId[]

    // Get configured DataManager for data access
    def data = zero.data.groovy.Manager.create('employee_db')

    // Retrieve employee record via Data Zero
    def empRecord = data.queryFirst("SELECT * FROM employee WHERE
id=$id")

    if(empRecord != null) {
        // Use ViewEngine JSON rendering
        request.view = 'JSON'
        request.json.output = empRecord
        render()
    } else {
        // Error handling
        request.status = HttpURLConnection.HTTP_NOT_FOUND
        request.error.message = "Employee $id not found."
        request.view = "error"
        render()
    }
}
```

---

This method shares many similarities with the `onList()` method, however some differences exist. First, because this method will be retrieving a particular element from the collection we must determine which element has been requested. The first line of this method obtains the ID of the employee from the HTTP request. Then, after obtaining an instance of a Manager object we can use the ID variable in the SQL statement that is run against the database.

Notice that in this example we use the `queryFirst()` method to run the SQL query. This method returns the first processed row, which is the ideal method to use because we want to retrieve only one row from the database. Finally, as with the `onList()` method, the object is returned in the JSON format, using a view renderer.

## Implementing `onCreate` method

The `onCreate()` method is used for adding new elements to the resource collection. It is invoked when an HTTP POST request is received on the URI of the collection. Example 11-10 shows implementation of this method.

*Example 11-10 The `onCreate` method*

---

```
def onCreate() {
    // Convert entity to JSON object
    def emp = zero.json.Json.decode(request.input[])

    // Get configured DataManager for data access
    def data = zero.data.groovy.Manager.create('employee_db')

    // Insert employee record via Data Zero APIs
    data.update(""" INSERT INTO employee (name, date_of_birth, id,
updated)
                VALUES ($emp.name, $emp.date_of_birth, DEFAULT, CURRENT
TIMESTAMP)""")

    // Set a Location header with URI to the new record
    locationUri = getRequestedUri(false) + '/' + emp['name']
    request.headers.out.Location = locationUri
    request.status = 201;
    request.view = 'JSON'
    request.json.output = emp
    render()
}
```

---

This method is quite different to the previous methods as it must obtain all of the data for the new element from the request. The first line of the method parses the JSON data received in the request so that we may use it later in the method. As

with the previous methods, we obtain an instance of the Manager and use it to run a SQL statement to INSERT the new element. The example illustrates how to reference the individual fields within the emp JSON object in the SQL query. Notice that the method used for running the SQL query is now the update() method, which is different to the previous examples.

## Implementing onUpdate method

When you want to modify an existing element in a collection, the onUpdate() method is used and is invoked with an HTTP PUT request for an individual element within a collection. In this case, the URI used is the same as the URI for the onRetrieve() method. The difference is the HTTP method used for the request. Example 11-11 shows how this method may be implemented.

*Example 11-11 The onUpdate method*

---

```
def onUpdate() {
    def emp = zero.json.Json.decode(request.input[])

    def data = zero.data.groovy.Manager.create('employee_db')

    data.update("UPDATE employee "
        + "SET firstname=$emp.firstname, lastname=$emp.lastname "
        + "WHERE id=$emp.id")

    request.status = HttpURLConnection.HTTP_NO_CONTENT
}
```

---

In this method, similar to the onCreate() method, the JSON data is parsed and then that object is used to update an existing element with the new fields. Again notice that the onUpdate() method in the Manager object being is being used to run the SQL statement. The ID field provided in the request is used in the WHERE clause to identify which record in the table is to be updated.

## Implementing onDelete method

The onDelete() method possibly has the simplest implementation of all the methods in the resource handler. It is invoked by the HTTP DELETE method using the URI for a specific element in the collection, for example:

/resources/messages/100

In this line, 100 is the ID for a particular employee. Example 11-12 shows how this method might be implemented using the zero.data API.

*Example 11-12 The zero data API*

---

```
def onDelete() {
    def id = request.params.employeeId[]
    def data = zero.data.groovy.Manager.create('employee_db')

    data.update("DELETE FROM employee WHERE id=$id");

    request.status = HttpURLConnection.HTTP_NO_CONTENT
}
```

---

This method obtains the ID of the element that is to be deleted and then, after obtaining an instance of the Manager object it uses that ID in the DELETE SQL statement.

## 11.4.4 Using zero.data in PHP

The zero.data API is also available in PHP, which can be used instead of Groovy if that is your preferred language. The following examples illustrate how to run various SQL statements using the PHP API and can be used to implement the LCRUD methods in the resource handler, as shown in 11.3, “Using ZRM for data access” on page 278.

### Obtaining the PHP functionality

To use PHP in your application, the following dependencies must be in your application’s ivy.xml file:

```
<dependency org="zero" name="zero.php" rev="[1.1.1.3, 2.0.0.0[" />
<dependency org="zero" name="zero.data.php" rev="[1.1.1.3, 2.0.0.0[" />
```

After adding these dependencies, issue the **zero resolve** command to retrieve them from the repository.

### Obtaining a Manager Instance

Similar to the Groovy API, you must obtain an instance of the Manager class that will be used to run SQL statements. Use the following line of code to obtain the instance:

```
$dataManager = dataManager("employee_db");
```

## PHP SELECT statement

Example 11-13 shows how to run a SELECT statement and then print the results on to the web page using the PHP API.

### *Example 11-13 PHP API*

---

```
<?php
$dataManager = dataManager("employee_db");
$sql = "SELECT * FROM employee";
$params = array(1);
$items = dataExec($dataManager,$sql,$params);
// Result manipulation is the same for both cases.
if (is_array($items)) {
    foreach ($items as $row) {
        foreach ($row as $columnName => $value) {
            echo "$columnName => $value\n";
        }
    }
}
else if (is_null($items)) {
    // Error
}
?>
```

---

This example can be placed in a file in the public folder of your application and invoked directly from a web browser or an HTTP client. Another possibility is to write a resource handler using the zero.data API in PHP. See Example 11-14.

### *Example 11-14 The zero data API in PHP*

---

```
<?php
// Get DataManager for data access
$dataManager = data_manager('employee_db');
class Employee {
    function onList() {
        global $dataManager;
        $sql = "SELECT * FROM employee";
        $params = array(1);
        $employees = dataExec($dataManager,$sql,$params);
        // Result manipulation is the same for both cases.

        // Use the zero global context to render the data as JSON
        zput('/request/view', 'JSON');
        zput('/request/json/output', $employees);
        render_view();
    }
}
?>
```

---

This code example would be placed in the following location with the `onList()` method being invoked by an HTTP GET request through the collection URI:

`application_home/app/resources/employee.php`

This resource handler can easily be extended by defining functions for the other methods in the `Employee` class.

## PHP INSERT statement

Example 11-15 shows how to run an INSERT statement using PHP to add a new element to the collection resource. This function can be added to the `Employee` class in Example 11-14 on page 293 to build up the PHP resource handler.

*Example 11-15 PHP INSERT statement*

---

```
function onCreate() {
    global $dataManager;
    // Convert the raw JSON stream in to a PHP array
    $emp = json_decode($HTTP_RAW_POST_DATA);
    $result = dataExec($dataManager, "INSERT INTO employee (name,
date_of_birth, id, updated) ".
        "VALUES (?, ?, DEFAULT, CURRENT_TIMESTAMP)", array($emp['name'],
$emp['date_of_birth']));
    // verify that an entry was inserted
    // Set a Location header with URI to the new record
    zput('/request/status', 201);
    $locationUri = zget('/request/path') . "/" . $emp['name'];
    zput('/request/headers/out/Location', $locationUri);
    zput('/request/headers/out/Content-Type', 'text/json');
    echo json_encode($emp);
}
```

---



## PHP UPDATE statement

Example 11-16 shows how the onUpdate() method may be implemented in the PHP resource handler, using the SQL UPDATE statement.

*Example 11-16 PHP update*

---

```
function onUpdate() {
    // Get configured DataManager for data access
    global $dataManager;
    $empId = zget('/request/params/employeeId');
    $emp = json_decode($HTTP_RAW_POST_DATA);
    $result = dataExec($dataManager, "UPDATE employee SET name=?,
updated=CURRENT_TIMESTAMP, id=?".
        "WHERE id=?", array($emp['name'], $empId, $empId));
    zput("/request/status", 204);
}
```

---

## PHP DELETE statement

Example 11-17 shows an implementation of the onDelete() method for the PHP resource handler. Again, this can be added to the Employee class in the example we are using to build up the PHP resource handler.

*Example 11-17 PHP delete*

---

```
function onDelete() {
    // Get configured DataManager for data access
    global $dataManager;
    $empId = zget("/request/params/employeeId");
    // Delete employee record via Query Zero
    $result = dataExec($dataManager, "DELETE FROM employee WHERE id=?",
array      ($empId));
    zput("/request/status", 204);
}
```

---

## 11.5 Running SQL directly against the database

The `zero.data` module provides some command-line tasks, one of which allows you to manually run SQL statements against your database. The command takes the following form:

```
zero runsql [dbkey] filename
```

In the command, `dbkey` is an optional parameter that specifies the key of the database as defined in the `zero.config` file; `filename` is a file that contains valid SQL statements. If the `dbkey` parameter is omitted, the command uses the configuration for the database that is configured, if there is only one. If more than one exists, the command looks for the default database, which is defined in the `zero.config` file by the following line:

```
/config/defaults/dbKey = "mydb"
```

### Valid statements

Valid statements can be any statements that are valid for the particular database and those that do not return anything. This means that `SELECT` statements are not allowed in the file. Therefore, valid statements are `INSERT`, `UPDATE`, `DELETE`. Additionally, any SQL statement that returns nothing, such as a DDL statement, is also valid. For example `CREATE`, `DROP` or `ALTER` are valid statements that can be used.

Statements in the file are executed as a single database transaction. If any statement fails, the changes made up to that point are not committed.



# Dynamic scripting scenarios

This chapter describes simple scenarios as an illustration of the CICS TS V4.1 dynamic scripting capabilities.

## 12.1 Exposing CICS resources in a RESTful style

In this section, we address the following issues:

- ▶ The ability to conform to RESTful style

We use the power of the dynamic scripting feature to help you more easily understand and perform the action (HTTP method) against a resource (URI convention).

- ▶ The ability to access CICS resources from a dynamic script

We use the power of the JCICS API to more easily translate the RESTful style request into a CICS action.

- ▶ The ability to have a direct access to the various elements of the messages that are exchanged

### 12.1.1 A simple do-nothing RESTful dynamic script

We first address the RESTful style capability. This scenario builds a dynamic script skeleton that reacts to REST style interaction patterns. This is the goal of our doNothingDemo zero application.

The steps are as follows:

1. Create a doNothingDemo application from the **zero create doNothingDemo** command.
2. Change the default 8080 port number according to our local environment, that is 8102, in the `config/zero.config` file of the doNothingDemo application.
3. Copy or create a customized `zerocics.config` file that is to target the correct CICS dynamic scripting engine.

4. We use a PHP resource handler (could have been groovy), so the PHP dependency must be added to the `ivy.xml` file in the config directory, followed by the **zero resolve** command:

```
<dependency name="zero.cics.php" org="zero" rev="[1.0.0.0,
2.0.0.0[/>
```

5. A good practice is to have a personalized home page, so we create an `index.html` page in the public directory, as shown in Figure 12-1.

```
<html>
<title>Welcome to the doNothing Application</title>
<body bgcolor=#8221;#FF99CC;#8221;>
<h1>ITSO doNothing Application</h1>
</body>
</html>
```

*Figure 12-1 The doNothing welcome HTML page*

6. Create the **PHP resource handler** in the following directory:  
doNothingDemo/app/resources directory

```
<?php
echo "Hello you from the doNothing application";
$event = zget('/event/_name');
switch($event)
{
case 'list':
echo "<h1>This was an HTTP GET of the Collection URI</h1>";
break;
case 'retrieve':
echo "<h1>This was an HTTP GET of the Member URI</h1>";
break;
case 'create':
echo "<h1>This was an HTTP POST to the Collection URI</h1>";
break;
case 'postMember':
echo "<h1>This was an HTTP POST of the Member URI</h1>";
break;
case 'putCollection':
echo "<h1>This was an HTTP PUT to the Collection URI</h1>";
break;
case 'update':
echo "<h1>This was an HTTP PUT of the Member URI</h1>";
break;
case 'deleteCollection':
echo "<h1>This was an HTTP DELETE to the Collection
URI</h1>";
break;
case 'delete':
echo "<h1>This was an HTTP DELETE of the Member URI</h1>";
break;

default:
echo "<h1>This was something else ??</h1>";
}
?>
```

*Figure 12-2 The doNothing php resource handler*

As you see, Project Zero allows the PHP script to react to REST method and URI patterns. That is, the HTTP GET method that is used against a /resource/Collection/Member URI type is scripted into a retrieve event name.

7. Change the display\_errors from Off to On in the config/php.ini file so that we can see the echo statements on our browser.

8. Start the application (using **zero start**) from its directory.
9. Test it from a browser or from the Firefox Poster add-on:  
`http://hostname:port/resources/doNothing/1`

The browser issues an HTTP GET request to the hostname:port TCIPSERVICE against the one member of the doNothing collection. In REST terms, it means that we want to retrieve this member reference. See Figure 12-3.

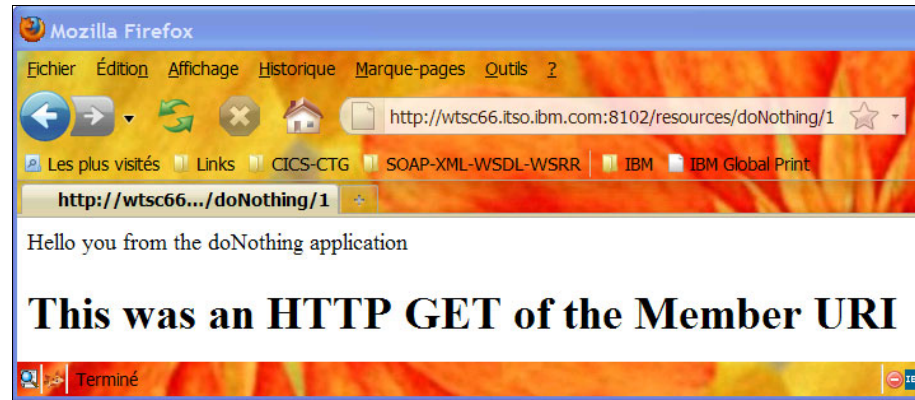


Figure 12-3 A doNothing application retrieve event invocation

Now, we can try to access a CICS resource.

### 12.1.2 A simple VSAM file resource access

We enhance the doNothingDemo application to be able to execute a CICS File Control request as determined by the RESTful pattern:

- ▶ Browse the file (list handler event)  
HTTP GET on URI /resources/vsamSimpleDemo
- ▶ Read a record (retrieve handler event)  
HTTP GET on URI /resources/vsamSimpleDemo/record\_id\_information
- ▶ Add a record (postMember handler event)  
HTTP PUT on URI /resources/vsamSimpleDemo/record\_id\_information
- ▶ Delete a record (delete handler event)  
HTTP DELETE on /resources/vsamSimpleDemo/record\_id\_information

This scenario illustrates a file control read on the FILEA vsam file.

The URL is as follows:

`http://hostname:port/resources/vsamSimpleDemo/111111`

HTTP GET method is transformed into an EXEC CICS READ FILE. Although the file name is fixed here, it could be dynamic. For example, the collection name in the URI is as follows:

`/resources/vsamSimpleDemo/111111/FILEA`

The /FILEA information is retrieved from the following simple statement:

`zget('request/event/pathInfo')`

The RIDFLD is the URI member, which in this case is 111111.

Perform the following steps:

1. Use the following command to create the new application from the previous one  
`zero create vsamSimpleDemo from doNothingDemo`
2. Make a copy of the `doNothing.php` script and rename the copy to `vsamSimpleDemo.php` file.

The retrieve routine (a GET operation on a collection member) is now shown in Figure 12-4.

```
case 'retrieve':
$aFile = new Java('com.ibm.cics.server.KSDS');
$aFile->setName('FILEA');
$aRID = zget('/request/params/vsamSimpleDemoId');
$aRID = mb_convert_encoding($aRID,"1047","iso-8859-1");
$aRecordHolder = new Java('com.ibm.cics.server.RecordHolder');
$aFile->read($aRID, $aRecordHolder);
$aRecord = $aRecordHolder->value;
$aRecord = mb_convert_encoding($aRecord,"iso-8859-1","1047");
echo "<p><b>Here is a FILEA record: <p>$aRecord";
break;
```

Figure 12-4 The `vsamSimpleDemo.php` retrieve event handler scripting logic

The PHP to Java bridge provides access to Java classes and functionality in PHP. CICS supplies the JCICS API to access CICS resources from Java applications. It is then a convenient way to issue the following line from the `vsamSimpleDemo` PHP script:

`EXEC CICS READ FILE('FILEA') RIDFLD(member information from the URI)  
INTO(a Java RecordHolder object)`



We supply more information about JCICS in 12.1.5, “More about JCICS within dynamic scripts” on page 309.

The logic is as follows:

- The Name property of a KSDS object is filled with the FILEA file name.
- The RIDFLD information is obtained from the member information of the REST URI. Notice the zget call convention (the collection name has the Id suffix):

```
/request/params/collection_nameId
```

- Because the member information is ASCII, it is converted to the suitable EBCDIC.
- A RecordHolder object is then returned from the read() method of the KSDS object.
- The record is converted to ASCII for display in the browser.

3. We start the vsamSimpleDemo application from its directory. We use the **zero start** command.

Figure 12-5 shows the result.



Figure 12-5 RESTful style FILEA VSAM file record read invocation

### 12.1.3 Returning the VSAM record as an Atom feed

Section 12.1.2, “A simple VSAM file resource access” on page 301 echoes the VSAM record to the browser. Dynamic scripting offers an easy way to present the VSAM record as an Atom feed (collection) or entry (member), as an advantage of the zero.atom component.

Details about the zero.atom component are at the following address:

<http://www.projectzero.org/sMash/1.1.x/docs/zero.devguide.doc/zero.atom/Atom.html>

To keep this scenario as simple as possible, we use only a few elements of the (rich) Atom Syndication Format data structure. The steps are as follows:

1. We copy the `vsamSimpleDemo.php` as `vsamSimpleDemoAtom.php` in the `app/resources` subdirectory of the `vsamSimpleDemo` application.
2. The following line of code can be replaced with the code in Figure 12-6.

```
echo "<p><b>Here is a FILEA record: <p>$aRecord" 1
```

```
$entry = array (
    "array (
        "id" => 1,
        "title" => "Welcome to the FILEA ATOM feed entry",
        "authorname" => "ITSO Redb",
        "updated" => time() * 1000, // milliseconds since the epoch
        "contenttype" => "TEXT",
        "content" => "This is our welcome message :o)"
    ),
    array(
        "id" => 2,
        "title" => "FILEA ATOM feed entry",
        "authorname" => "ITSO Redb",
        "updated" => time() * 1000, // milliseconds since the epoch
        "contenttype" => "TEXT",
        "content" => "Here is a FILEA record: $aRecord."
    )
);
zput("/request/view","atom");
zput("/request/docType","entry");
zput("/request/atom/output",$entry);
render_view();
```

Figure 12-6 Replacement code

The ATOM representation is an array of two entries: a welcome message and the FILEA record. The `zput` commands indicate that we want to *render* an "atom" view of Atom "entry" whose content is in the `$entry` variable.

The Atom Syndication Format information we fill is as follows:

- id
- title
- authorname
- updated
- contenttype
- content

The `render_view()` call returns the required Atom Syndication Format XML document, filled with our \$entry information using the Atom Publishing Protocol.

The Firefox Poster extension can then be used to display the content of the ATOM feed from a GET action. See Figure 12-7.

The URL to use is now as follows:

`http://hostname:port/resources/vsamSimpleDemoAtom/your_filea_recordId`

3. Note that there is no need to recycle the application when we add or modify a PHP script.

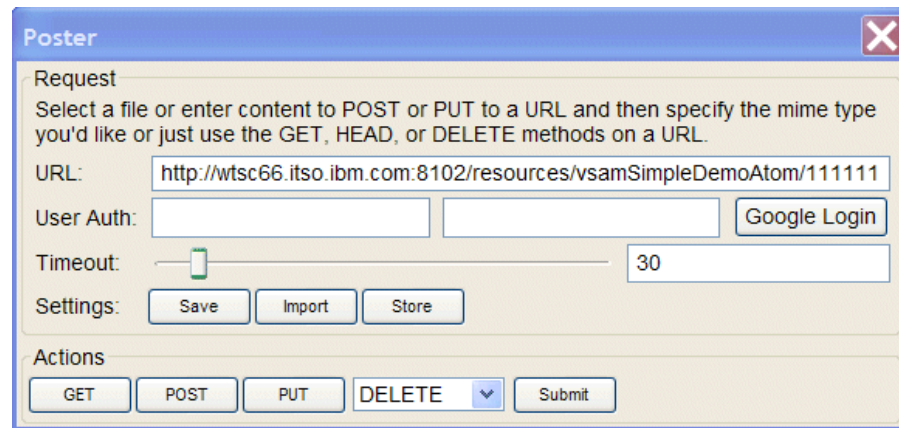


Figure 12-7 The `vsamSimpledemoAtom` script invocation

If you are familiar with the Atom Syndication Format you might recognize the ATOM XML vocabulary (Figure 12-8 on page 306).

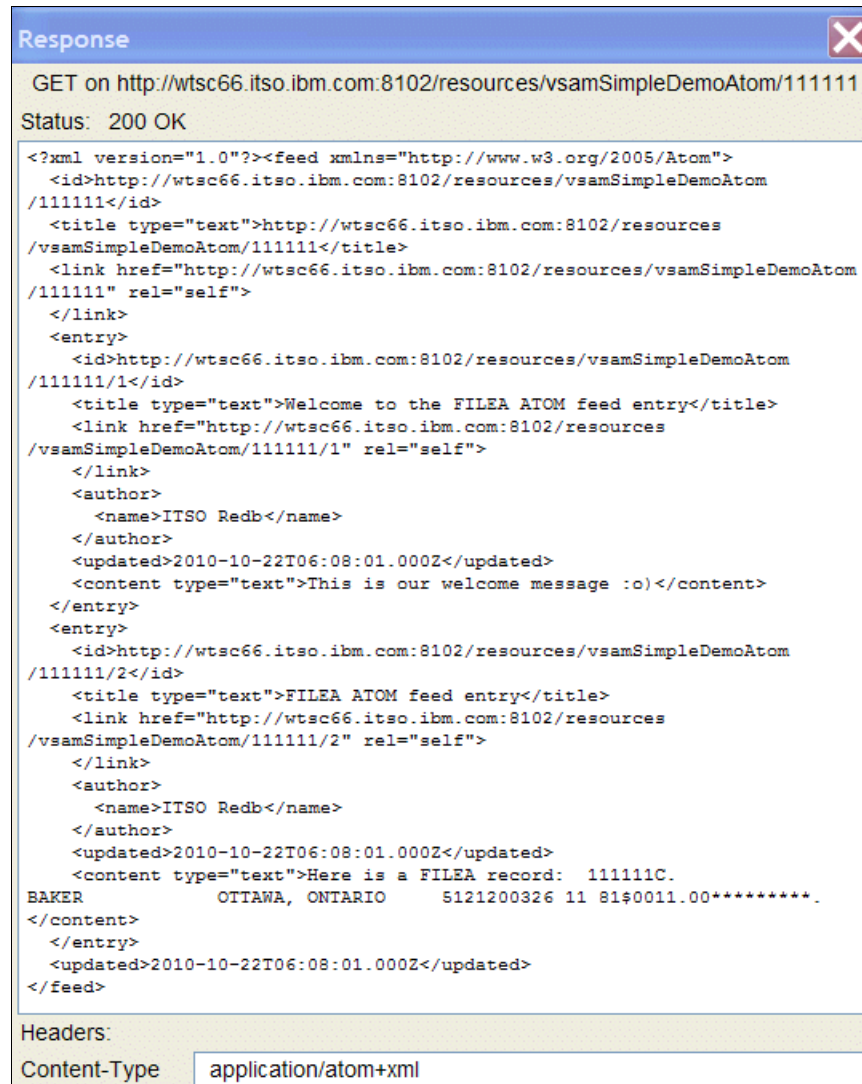


Figure 12-8 The vsamSimpledemoAtom response

ATOM is a standard for document publication. We can take advantage of any existing feed readers. RSS Bandit, an RSS feed reader, can be used to subscribe to our feed, or can use your Lotus® Notes® embedded feed reader. See Figure 12-9 on page 307.

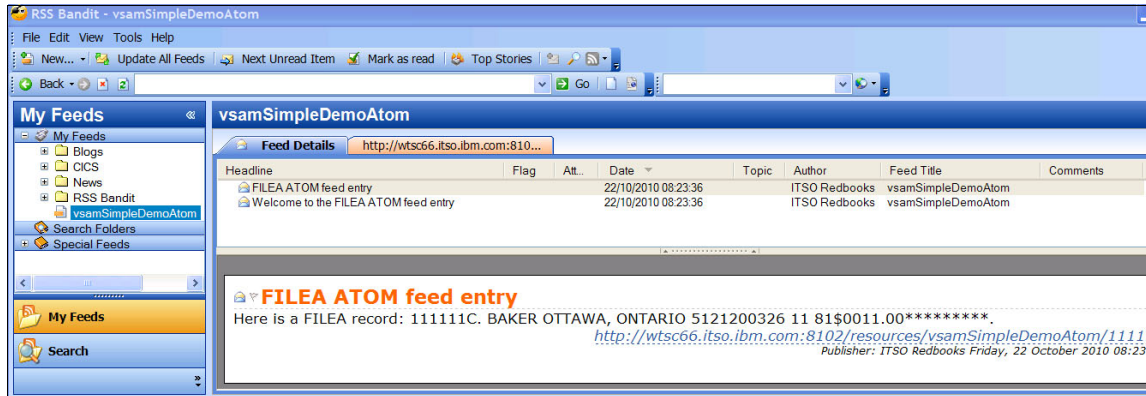


Figure 12-9 RSS Bandit vsamSimpleDemoAtom subscription result

It successfully formats our feed.

**Note:** Each time the FILEA record is updated, the feed reader issues an information pop-up for this CICS feed.

## 12.1.4 The zero.cics.tsq.demo sample

The CICS dynamic scripting feature supplies a tsqDemo sample. This scenario explores several benefits. In addition to a tsq.groovy script, the feature supplies a PHP command-line interface (CLI) to the resource handler, which means that the tsq.groovy script is executed from an HTTP connection request, and the PHP script is run from the UNIX System Services ZERO command-line interface.

This scenario describes the use of the ZERO command-line interface, which creates the new application, as follows:

```
zero create tsqSimpleDemo from zero:zero.cics.tsq.demo
```

This sample contains a tsq.groovy script in its resources subdirectory, which is where the TSQ JCICS API is used to get access to the CICS resource from a REST URI.

The sample contains tsqRead.PHP and a tsqWrite.PHP scripts in the tasks subdirectory to enable the command-line interface for the same purpose.

These samples might be useful when testing JCICS APIs locally.

## The command-line scripts

Configuring the HTTP port in the `ivy.xml` file is unnecessary.

The code is similar to the `vsamSimpleDemo` scripts with several differences:

- The JCICS classes are imported at the beginning of the script. This way can be useful when they are shared by multiple nested scripts so that they have a wide scope: this script and any included script. See Figure 12-10.

```
java_import('com.ibm.cics.server.TSQ');

>>>The arguments passed along with the command are received<<<

$args = zget('/event/args');
if (count($args) != 2) {

//>>>And the JCICS code is executed<<<

try {
    $tsq = new TSQ();
    $tsq->setName($args[0]);
    $tsq->writeItem(mb_convert_encoding($args[1], 'IBM-1047', 'UTF-8'));
} catch (Exception $e) {
    echo "Failed to write item [$args[1]] to queue [$args[0]]: " .
    $e->getMessage() . PHP_EOL;
    zput('/config/exitCode', -1);
}
```

*Figure 12-10 The `tsqWrite.php` command-line interface script*

- The **zero `tsqWrite aNiceTSQ aNiceTSQ_Item_Content`** command can be executed. Notice that the TSQ class handles a QNAME natively. See Figure 12-11.

```
SC66> /u/cicsrsl/cicsds: cd tsqSimpleDemo
SC66> /u/cicsrsl/cicsds/tsqSimpleDemo: ceTSQ aNiceTSQ Item
Data
Writing "aNiceTSQ_Item_Data" to TSQ "aNiceTSQ"
```

*Figure 12-11 The `tsqWrite.php` command-line command execution*

- CEBR can be used to check the results. See Figure 12-12.

```
CEBR TSQ aNiceTSQ          SYSID
ENTER COMMAND ==>
*****
00001 aNiceTSQ_Item_Data
***** B
```

*Figure 12-12 The aNiceTSQ TS queue read using CEBR*

This command-line interface is a convenient way to practice and test the JCICS API.

### 12.1.5 More about JCICS within dynamic scripts

The CICS dynamic scripting facility uses the JCICS API, although you do not have to be a Java expert to use it. This section introduces key information.

Our goal is to issue EXEC CICS commands from the scripts. Likely common needs are as follows:

- Program control LINK to a CICS program
- File control VSAM access to a CICS VSAM file or data table
- Temporary Storage access to a CICS TS QUEUE

In the dynamic scripting context, the JCICS API is functionally equivalent to the regular CICS command level API. This section addresses the information required to successfully interact with those CICS APIs.

#### Program control LINK

The goal is to map an EXEC CICS LINK COMMAREA or CHANNEL API call.

The JCICS API Program class supplies the link() method. The link can use a COMMAREA or a CHANNEL interface.

For a mature CHANNEL interface, the JCICS API Channel class supplies the createContainer() and getContainer() methods. The Channel object is created from the createChannel() method on the Task class (with JCICS the channel create is explicit). The Container class supplies the put() and get() methods. More advanced techniques could browse the containers which form a CHANNEL.

For an established COMMAREA interface, the commarea object is a Java bytes array.

As a simple illustration of program link access, we access two simple CICS COBOL programs:

- ▶ PONGBL, which is a regular COMMAREA-based linked subprogram.  
Its COMMAREA is composed of a first request section followed by a second response section. PONGBL traces the request section in a CICS TS queue and returns its CICS execution context EIBTRNID, APPLID, and USERID information in the response section.
- ▶ PONGIF, which enhances the PONGBL commarea interface by using a CHANNEL interface.

The request and response sections are now two separate messages. To keep the scenario as simple as possible, the same PINGPONG CONTAINER is used for both the request and the response messages. PONGIF links to PONGBL using its commarea interface.

The first program LINK scenario links to the CICS PONGIF COBOL program by using a channel named *pongIF* and the required *PINGPONG* container.

This scenario is functionally equivalent to the following COBOL code:

```
EXEC CICS PUT CONTAINER('PINGPONG') CHANNEL('PongIF')  
....FROM('o) hello PONG from a PING pgm (o:') FLENGTH(34)  
END-EXEC.  
EXEC CICS LINK PROGRAM('PONGIF') CHANNEL('PongIF') END-EXEC.  
EXEC CICS GET CONTAINER('PINGPONG') CHANNEL('PongIF')  
.....INTO(Pong-Response) FLENGTH(Pong-Response-Length)  
END-EXEC.
```

Although you might think that we are missing the required COBOL MOVEs before and after this set of commands, we address this information in 12.1.6, “More complex message interface support using JZOS” on page 321.



Figure 12-13 shows simple PHP script that links to PONGIF.

```
<?php
java_import("com.ibm.cics.server.Task");
try {
    $pongProgram = new Java('com.ibm.cics.server.Program');
    $myTask = Task::getTask();
    $pongChannel = $myTask->createChannel("pongIF          ");
    $pongContainer = $pongChannel->createContainer("PINGPONG
");

    $pongData = new Java('java.lang.String','o) hello PONG from a
PING php (o:');
    $pongContainerBytes = $pongData->getBytes();
    $pongContainerBytes =
    mb_convert_encoding($pongContainerBytes,"iso8859-1","037");
    $pongContainer->put($pongContainerBytes);

    $pongProgram->setName('PONGIF  ');
    $pongProgram->link($pongChannel);

    $pongContainerBytes = $pongContainer->get();

    echo
    mb_convert_encoding($pongContainerBytes,"UTF-8","IBM-1047")    .
    PHP_EOL;

} catch (JavaException $e) {
    echo "Problem on LINK CHANNEL to PONGIF : " . $e->getMessage();
    zput('/config/exitCode', -1);
    return;
?>
```

*Figure 12-13 The pcLINKChannel.php script to link to the PINGIF COBOL program*

The logic is as follows:

1. A Program object is created.
2. The pongIF CHANNEL is created from the createChannel() method of the Task object.
3. The PINGPONG CONTAINER is created from the createContainer() method of the Channel object. It is filled with the required Java byte array data, in the correct encoding, from the Container object put() method.
4. The Name property of the Program object is set to the CICS program name to be linked: PONGIF.

5. The EXEC CICS LINK is then executed from the link() method of the Program object.
6. This simple interface uses the same PINGPONG container for the response, so the get() method receives it into the same Java bytes array object.
7. The echo of the response must be in the correct encoding.
8. The last command exposes any exception returned (that is PGMIDERR or any other CICS condition we would like to handle).

A convenient way to test this script is to run it as a task as discussed in the tsqDemo sample application scenario (12.1.4, “The zero.cics.tsq.demo sample” on page 307):

1. Use the **zero create** command to create JCICSSandbox from tsqSimpleDemo.
2. Copy the pcLINKChannel.PHP script to the app/tasks directory.
3. Run the script from the **zero pcLINKChannel** command. See Figure 12-14.

```
SC66> /u/cicsrs1/cicsds/JCICSSandbox: zero pcLINKChannel  
ZERO EPRED2 CICSRS1  
SC66> /u/cicsrs1/cicsds/JCICSSandbox:
```

*Figure 12-14 The pcLINKChannel.php execution results*

The PINGPONG CONTAINER that is returned on the link contains the CICS PONGIF execution context information:

- ▶ The ZERO transaction code as defined in our local configuration
- ▶ The EPRED2 CICS APPLID

This information is unlikely to be the CICS APPLID that is defined in our configuration because we are likely to use a Dynamic Scripting Owning Region (DSOR), which dynamically links to a regular CICS AOR.

- ▶ The CICSRS1 RACF USERID according to the CICS ESM security rules in place.

We can also link to the COMMAREA interface used by the PONGBL program. This time the code looks like the code in Figure 12-15.

```
<?php
try {
    $pongProgram = new Java('com.ibm.cics.server.Program');
    $pongCommarea = new Java('java.lang.String','o) hello PONG from a
    PING php (o:.....');
    $pongCommareaBytes = $pongCommarea->getBytes();

    $pongProgram->setName('PONGBL ');
    $pongProgram->link($pongCommareaBytes);

    echo mb_convert_encoding($pongContainerBytes,"UTF-8","IBM-1047") .
    PHP_EOL;

} catch (JavaException $e) {
    echo "Problem on LINK COMMAREA to PONGBL : " . $e->getMessage() .
    PHP_EOL;
    zput('/config/exitCode', -1);
    return;
}
?>
```

*Figure 12-15 The pcLINKCommarea.php script*

The logic is as follows:

1. A program object is created.
2. The commarea byte array is created from the `getBytes()` method on a String object.
3. The Name property of the Program object is set to the CICS program name to be linked: PONGBL
4. The EXEC CICS LINK is then executed from the `link()` method of the Program object.
5. The echo of the response must be in the correct encoding.
6. The last command exposes any exception returned (that is PGMIDERR or any other CICS condition we would like to handle).

Perform the following steps:

1. Copy the `pcLINKCommarea.php` script to the `app/tasks` directory.
2. Run it from the **zero pcLINKCommarea** command. See Figure 12-16

```
SC66> /u/cicsrs1/cicsds/JCICSSandbox: zero pcLINKCommarea
:o) hello PONG from a PING php (o: ZERO EPRED2 CICSRS1
SC66> /u/cicsrs1/cicsds/JCICSSandbox:
```

Figure 12-16 The *pcLINKCommarea.php* execution results

This time the complete request and response commarea message is returned and displayed.

## File Control VSAM API

In this section, we use JCICS to map a VSAM File Control EXEC CICS API call.

The JCICS API supports KSDS, ESDS and RRDS files. We use the KSDS implementation.

The JCICS API supplies the `read()`, `write()`, `startbrowse()`, `next()`, and other methods on the KSDS class.

The data to be written is in a Java byte array. The data is read from a file into a `RecordHolder` Java byte array object. When browsing a file, the key of the record is represented by a `KeyHolder` Java byte array object.

The `simpleVSAMDemo` application script will be modified to read the first five records of the `FILEA` VSAM file (see Figure 12-17 on page 315):

- ▶ In REST event terms, we *list* a collection.
- ▶ In CICS terms, we use `READNEXT` of the first five records of the `FILEA` CICS file control resource.

```

case 'list':
    $aFile = new Java('com.ibm.cics.server.KSDS');
    $aFile->setName('FILEA');
    $aRID = ("\0\0\0\0\0\0");
    $aKSDSBrowse = $aFile->startBrowse($aRID);
    $aRecordHolder = new Java('com.ibm.cics.server.RecordHolder');
    $aKeyHolder = new Java('com.ibm.cics.server.KeyHolder');
    try
    {
        for($i =0 ; $i < 5 ;$i++)
        {
            $aKSDSBrowse->next($aRID, $aRecordHolder,$aKeyHolder);
            $aRID = $aKeyHolder->value;
            $aRecord = $aRecordHolder->value;
            $aRecord = mb_convert_encoding($aRecord,"iso8859-1","037");
            $oRID = mb_convert_encoding($aRID,"iso8859-1","037");
            echo"<p>Here is FILEA recordID <b> $oRID : <BR></b> $aRecord<p>";
        }
    }
    catch(Exception $e)
    {
        zput("/request/status",500);
        zput("/request/view","error");
        zput("/request/error/message","Exception: $e");
        render_view();
    }
    break;

```

Figure 12-17 The *vsamSimpleDemo.php* enhanced script

The logic is as follows:

1. The Name property of a KSDS object is filled with the FILEA file name.
2. The RIDFLD information is initialized to nulls.
3. The RecordHolder and KeyHolder objects are created.
4. The code loops on five READNEXT commands issued from the next() method.
5. Because the member information is ASCII it is converted to the suitable EBCDIC, the record is converted to ASCII for display to the browser.

The following URI returns the results shown in Figure 12-18 on page 316:

<http://hostname:port/resources/vsamSimpleDemo>

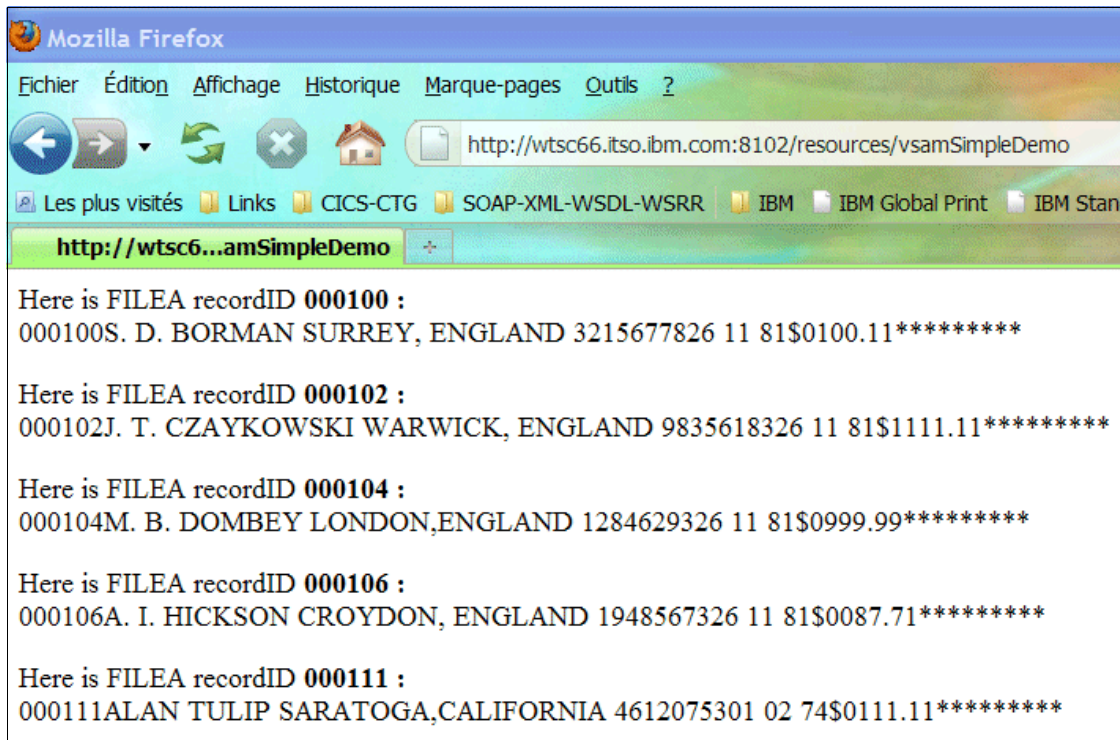


Figure 12-18 The vsamSimpleDemo browser response

## Temporary Storage API

We now use EXEC CICS to temporary storage queues (TSQs or TS queues) from JCICS.

The JCICS API applies the readItem(), readnextItem(), writeItem(), rewriteItem(), and other methods on the TSQ class.

The data to be written is in a Java byte array. The data is read from a TS queue into a ItemHolder Java byte array object.

The zero.cics.tsq.demo in 12.1.4, "The zero.cics.tsq.demo sample" on page 307 described TSQ write and delete scenarios. Browsing a TS queue is similar to the VSAM implementation.

We have the tsq.groovy (not PHP this time) dynamic script supplied as a sample in the zero.cics.tsq.demo application. Although we copied it in our simpleTSQDemo application, you know where to look for it.

As already mentioned, this script is more Java like than PHP. The first portion of the code adds several convenience methods to the TSQ base class, shown in Figure 12-19.

```
import zero.core.groovysupport.ZeroObject
import com.ibm.cics.server.*
import zero.util.http.*

class tsq implements ZeroObject {

    /** The encoding in which to store strings in the TSQs. */
    final String TSQ_CHARSET = 'IBM1047'

    /**
     * The constructor is executed on every event.
     * We use it to dynamically enhance the TSQ API with some
     * convenience methods.
     */
    def tsq() {
        // Add a static factory method
        TSQ.metaClass.static.create = { String name ->
            def tsq = new TSQ()
            tsq.setName(name)
            return tsq
        }

        // Add an iteration method
        TSQ.metaClass.each = { Closure closure ->
            def position = 0
            while (true) {
                def itemHolder = new ItemHolder()
                try {
                    delegate.readItem(++position, itemHolder)
                } catch (ItemErrorException e) {
                    break; // Reached end of queue
                }
                closure(itemHolder.value)
            }
        }
    }
}
```

Figure 12-19 The *tsq.groovy* script: part 1

Next, the REST event pattern handling routines are implemented:

1. We look first at the onRetrieve handling of an HTTP GET for /ressources/tsq/queueName. See Figure 12-20.

```
/**
 * Read the full content of a TSQ. Each item in the queue is assumed
 * to be a string encoded in TSQ_CHARSET.
 */
def onRetrieve() {
    def tsq = TSQ.create(request.params.tsqId[])
    def items = []
    try {
        tsq.each { item -> items += new String(item, TSQ_CHARSET) }
    } catch (InvalidQueueIdException e) {
        reportNotFound()
        return
    }
    println items
}

/**
 * Convenience method to report error HTTP 404: resource not found.
 */
def reportNotFound() {
    request.status = HttpURLConnection.HTTP_NOT_FOUND
    println "No such queue: ${request.params.tsqId[]}."
}
```

*Figure 12-20 The tsq.groovy script: part 2*

2. Then we look at the onUpdate() handling of an HTTP PUT for /ressources/tsq/queueName with the following header:  
text/\* Content-Type HTTP header.



The TS data resides in the HTTP body. See Figure 12-21.

```
/**
 * Write to a TSQ. The TSQ is created if it does not exist.
 */
def onUpdate() {
    // Verify the request's Content-Type
    def contentType = request.headers.in.'Content-Type'[]
    if (!contentType || !contentType.startsWith('text')) {
        request.status = HttpURLConnection.HTTP_NOT_ACCEPTABLE
        println "Request Content-Type must be text/*"
        return
    }

    // Get text sent in the request
    def inputText = getRequestText()
    if (!inputText) {
        request.status = HttpURLConnection.HTTP_BAD_REQUEST
        println "Request must contain data"
        return
    }

    // Write data to the TSQ, transcoded to TSQ_CHARSET.
    def tsq = TSQ.create(request.params.tsqId[])
    tsq.writeItem(inputText.getBytes(TSQ_CHARSET))

    println "Wrote [{inputText}] to tsq
    ${request.params.tsqId[]}."
}
```

Figure 12-21 The `tsq.groovy` script: part 3

3. We can use the Poster Firefox extension to test the script. See Figure 12-22.

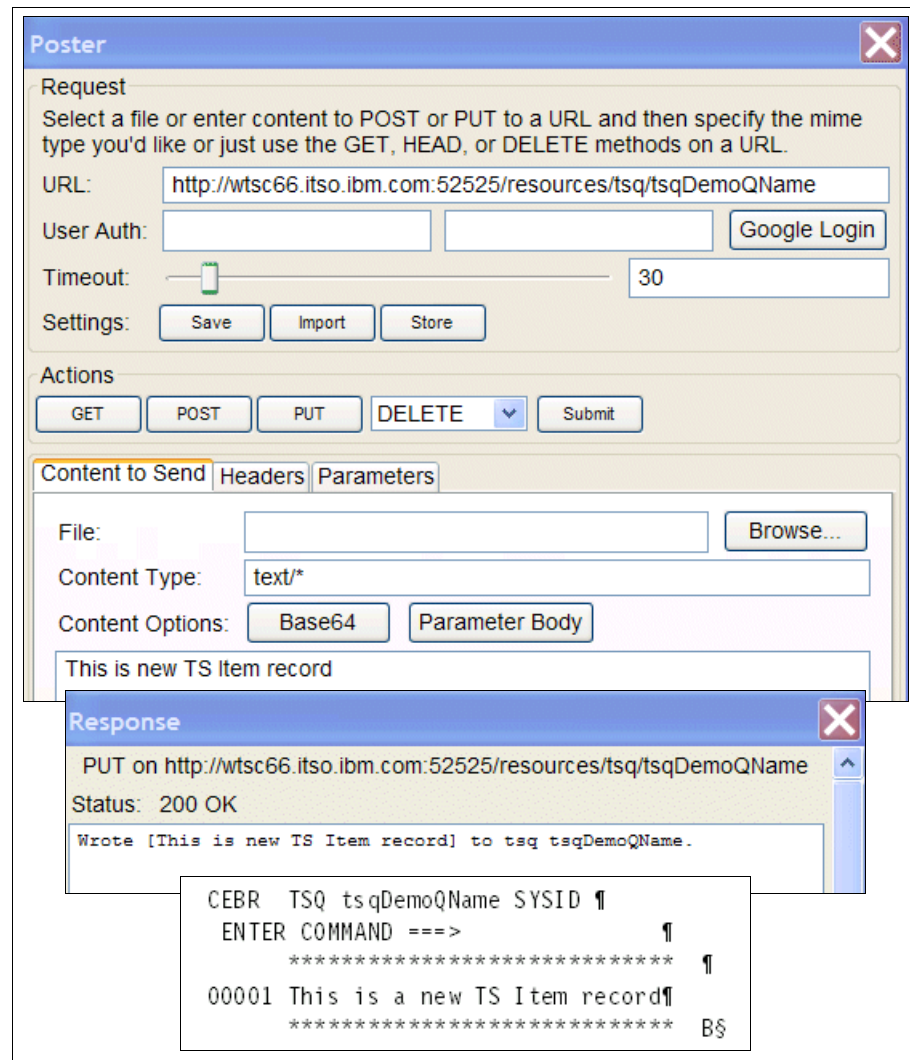


Figure 12-22 The tsq.groovy execution result

4. The `onDelete()` event is then implemented as an EXEC CICS DELETEQ TSQNAME(queueName) extracted from the `/ressources/tsq/queueName` URI. You can test it by selecting the DELETE HTTP method in the Poster Firefox extension. See Figure 12-23.

```
/**
 * Delete a TSQ.
 */
def onDelete() {
    def tsq = TSQ.create(request.params.tsqId[])
    try {
        tsq.delete()
    } catch (InvalidQueueIdException e) {
        reportNotFound()
        return
    }
    println "TSQ ${request.params.tsqId[]} deleted."
}
```

Figure 12-23 The `onDelete()` event handling

### 12.1.6 More complex message interface support using JZOS

As you may have noticed, all interfaces or record structures we used in previous sections were raw character data.

In real situations, you more likely will require access to individual fields within this interface from getter or setter methods.

Older interfaces might also use complex types that might require specific adaptors. Although it is less likely to occur, it could happen.

In this situation, the CICS PHP and Groovy dynamic scripting support of the JZOS tool is effective. The JZOS Record Generator tool creates Java classes that provide descriptive APIs for accessing data in COBOL data structures. This record generator tool supplies an easy access to the individual fields of a program control interface or CICS resource record description from the dynamic scripts.

The link between the COBOL, PL/I or Assembler data structure and Java representation is an *Associated Data* file. This file is created from the compilers. JZOS supplies the tools that create this data structure Java code access.

We illustrate its use with the pcLINKCommarea script. The PONGBL COBOL interface is shown in Figure 12-24.

```

01 PONGBLMS.
02 Message-Request.
    05 Message-Request-Xion      PIC X(4).
    05 Message-Request-Message  PIC X(30).

02 Message-Reponse.
    05 Message-Reponse-Pong     PIC X(4).
    05 FI1                      PIC X(1).
    05 Message-Reponse-APPLID   PIC X(8).
    05 FI2                      PIC X(1).
    05 Message-Reponse-USERID   PIC X(8).

```

Figure 12-24 The PONGBL COBOL interface

The ADATA generation JCL is a simple COBOL compile step (Figure 12-25).

```

//DAVCOB  JOB 'CICS COMPILE',
//          MSGCLASS=A,
//          CLASS=A,NOTIFY=&SYSUID,MSGLEVEL=(1,1)
/*JOBPARM SYSAFF=SC66
//COB     EXEC PGM=IGYCRCTL,REGION=0M,
//          PARM='NODYNAM,LIB,OBJECT,RENT,APOST,MAP,ADATA,TRUNC(BIN)'
//STEPLIB DD DSN=IGY.SIGYCOMP,DISP=SHR
//SYSLIB  DD DSN=CICSTS41.CICS.SDFHCOB,DISP=SHR
//          DD DSN=CICSR1.CICS.COPYLIB,DISP=SHR
//          DD DSN=CICSR1.CICS.APPLIS,DISP=SHR
//SYSADATA DD DSN=CICSR1.ADATA(PONGBLAD),
//            DCB=(RECFM=VB,LRECL=1020,BLKSIZE=1024),
//            DISP=(NEW,CATLG),SPACE=(CYL,(3,3,10))
//SYSPRINT DD SYSOUT=*
//SYSIN   DD DSN=CICSR1.CICS.APPLIS(PONGBLAD),DISP=SHR
//SYSLIN  DD DUMMY
//SYSUT1  DD UNIT=SYSDA,SPACE=(CYL,(1,1))
.....

```

Figure 12-25 A ADATA sample JCL extract

It can then be processed by the JZOS tools, which use a lengthy command (Figure 12-26) that can be stored in a command file. In this scenario, we name *adata*. The CICS Dynamic Infrastructure® supplies full details in its information center.

```
java -Dfile.encoding=UTF-8 -cp /dynamicScript/cics/jzos_recgen.jar
com.ibm.jzos.recordgen.cobol.RecordClassGenerator
adataFile="//'CICSR1.ADATA(PONGBLAD)'"
symbol=PONGBLMS outputDir=./java package=sample class=PONGBLMessages
genCache=false
```

*Figure 12-26 The lengthy JZOS command which creates the ADATA magic*

This command must be issued from the pcLINKCommarea application home directory (JCICSSandbox here). See Figure 12-27. The java directory is created from the **mkdir java** command. The symbol parameter references the COBOL level 01 PONGBLMS and the java class name is to be PONGBLMessages.

```
SC66> /u/cicsrsl/cicsds/JCICSSandbox: ../atata
SC66> Generating //'CICSR1.ADATA(PONGBLAD)' to
/u/cicsrsl/cicsds/JCICSSandbox/java/sample/PONGBLMessages.java
```

*Figure 12-27 Output from the magic adata command*

The PONGBLMessages.java code can be viewed at the PONGBLMS commarea access bean. See Figure 12-28.

```
package sample;
import com.ibm.jzos.fields.*

// Generated by com.ibm.jzos recordgen.cobol.RecordClassGenerat

public class PONGBLMessages {
protected static CobolDatatypeFactory factory = new CobolDataty
static { factory.setStringTrimDefault(false); }

/** <pre>
01 PONGBLMS </pre> */
public static final int PONGBLMS_len = 58;
```

*Figure 12-28 PONGBLMessages.java*

Figure 12-29 shows the access code to the following COBOL field (extracts from PONGBLMessages.java):

05 Message-Reponse-Applid PIC X(8).

```
/** <pre>
    05 Message-Reponse-APPLID      PIC X(8). </pre> */
protected static StringField MESSAGE_REPONSE_APPLID =
factory.getString(8);
....
protected byte[] _byteBuffer;
....
public String getMessageReponseApplid() {
    return MESSAGE_REPONSE_APPLID.getString(_byteBuffer);
}

public void setMessageReponseApplid(String messageReponseApplid) {
    MESSAGE_REPONSE_APPLID.putString(messageReponseApplid,
    _byteBuffer);
}
```

*Figure 12-29 COBOL Message-Response-Applid java code handling extract*

The `getMessageReponseApplid()` method returns the APPLID field as a 8-byte Java byte array.

The `setMessageReponseApplid()` method receives an APPLID Java String object as input.

Back to the application home directory, the **zero compile** can be executed to compile the code into a java class. See Figure 12-30.

```
SC66> /u/cicsrsl/cicsds/JCICSSandbox: zero compile
CWPZT0645I: javac command: /usr/lpp/java/J6.0/bin/javac
/sc66/tmp/2128799662511_0/JCICSSandbox.options
/sc66/tmp/21287996625611_0/JCICSSandbox.srcNames

CWPZT0800I: Command compile was successful/
SC66> /u/cicsrsl/cicsds/JCICSSandbox: ls -al ./classes/sample
total 10
drwxrwx--x 2 CICSRS1 SYS1 328 Oct 25 04:50 ./
drwxrwx--x 3 CICSRS1 SYS1 328 Oct 25 04:50 ../
-rw-rw---- 1 CICSRS1 SYS1 2562 Oct 25 04:50 PONGBLMessages.class
```

*Figure 12-30 The zero compile output*

The `pcLINKCommarea.PHP` script may now be modified so that it benefits from the PONGBL interface helpers. This way is closer to real situations.

Name the script *`pcLINKCommareaAdata.php`*.

Figure 12-31 shows that the code is cleaner than before.

```
<?php
    java_import("com.ibm.cics.server.Task");
    java_import("sample.PONGBLMessages");

    try {
        $pongProgram = new Java('com.ibm.cics.server.Program');
        $myTask = Task::getTask();

        $pongCommarea = new PONGBLMessages();
        $pongCommarea->messageRequestXion = ':o) ';
        $pongCommarea->messageRequestMessage = 'hello PONG from a PING
php (o: ';

        $pongProgram->setName('PONGBL ');
        $pongProgram->link($pongCommarea->byteBuffer);

        $RTransid = $pongCommarea->getMessageReponsePong();
        $RAplid = $pongCommarea->getMessageReponseAplid();
        $RUserid = $pongCommarea->getMessageReponseUserid();
        echo "Transaction Code : $RTransid" . PHP_EOL;
        echo "CICS APPLID      : $RAplid" . PHP_EOL;
        echo "RACF USERID      : $RUserid" . PHP_EOL;
```

*Figure 12-31 Cleaner code*

The logic is as follows:

1. The `PONGBLMessages` class is imported.
2. A `PONGBLMessages` object is created and its `messageRequestXion` and `messageRequestMessage` properties are filled in. Notice the convenient java names created from the COBOL structure (that is `messageRequestXion` from `Message-Request-Xion`).
3. The `PONGBLMessages` object is passed on the `LINK` as a Java byte array `commarea`.
4. The getters are used to extract the three response fields that are echoed.

The script can be executed from **zero** `pcLINKCommareaAdata`. See Figure 12-32.

```

sc66> /u/cicsrs1/cicsds/JCICSSandbox: /zero pcLINKCommareaAdata
Transaction Code : ZERO
CICS APPLID .....: EPRED2
RACF USERID .....: CICSRS1
sc66> /u/cicsrs1/cicsds/JCICSSandbox:

```

Figure 12-32 *pcLINKCommareaAdata formatted output*

## 12.1.7 More complex message interface support using Rational Developer for System z

Rational Application Developer and Rational Developer for System z supply J2E J2C Java binding tooling to create java helpers similar to JZOS ones. Figure 12-33 shows the wizard to use.

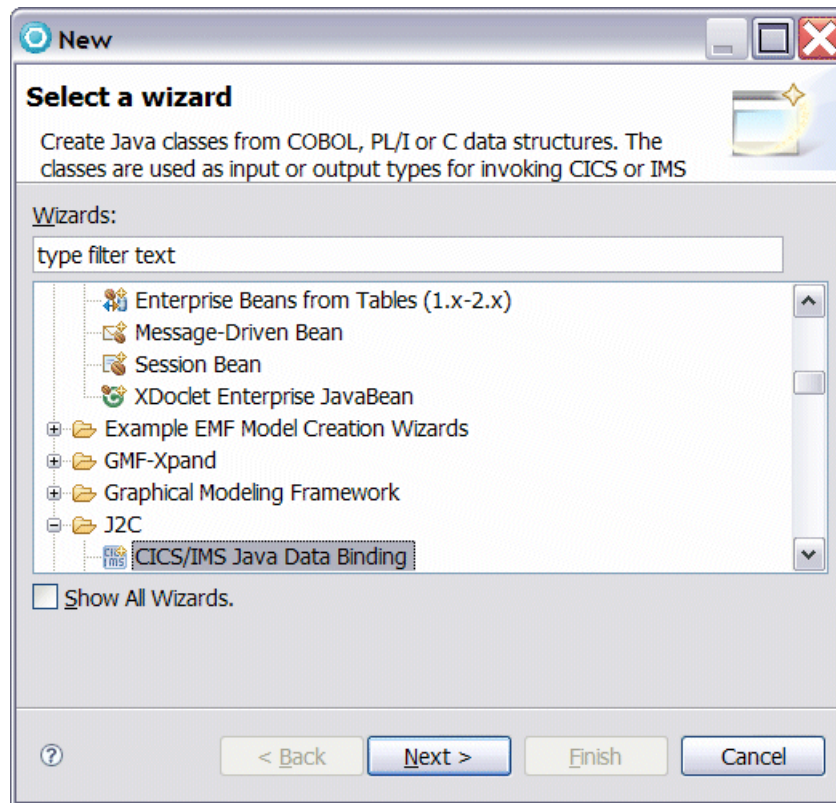


Figure 12-33 *CICS/IMS Java Data Binding wizard*

Use the wizard to import the PONGBLMS COBOL copy. See Figure 12-34.



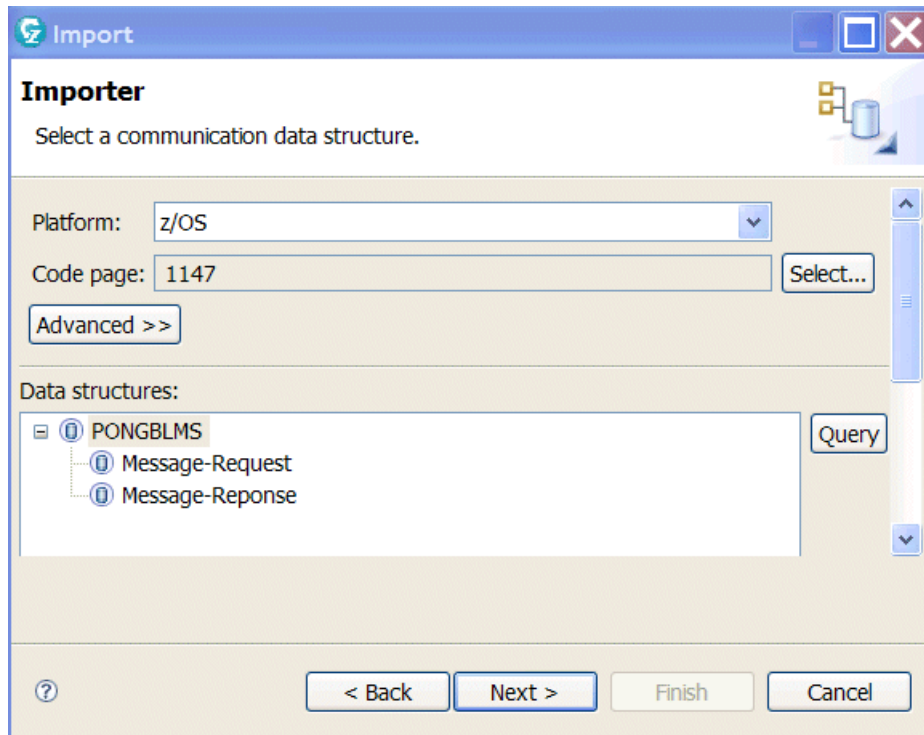
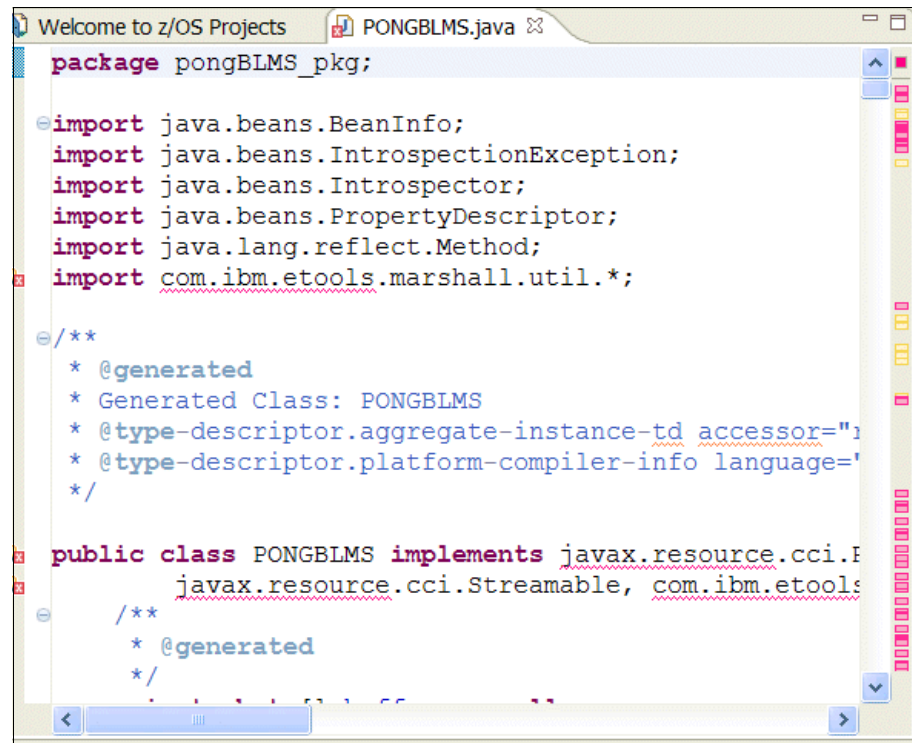


Figure 12-34 The COBOL copy importer wizard

Click **Next** and then specify the Java package and Class names.

The Java commarea bean is generated. See Figure 12-35 and Figure 12-36 on page 329.



```
Welcome to z/OS Projects PongBLMS.java
package pongBLMS_pkg;

import java.beans.BeanInfo;
import java.beans.IntrospectionException;
import java.beans.Introspector;
import java.beans.PropertyDescriptor;
import java.lang.reflect.Method;
import com.ibm.etools.marshall.util.*;

/**
 * @generated
 * Generated Class: PongBLMS
 * @type-descriptor.aggregate-instance-td accessor="1
 * @type-descriptor.platform-compiler-info language="
 */

public class PongBLMS implements javax.resource.cci.F
    javax.resource.cci.Streamable, com.ibm.etools

/**
 * @generated
 */
```

Figure 12-35 The PongBLMS bean generated code

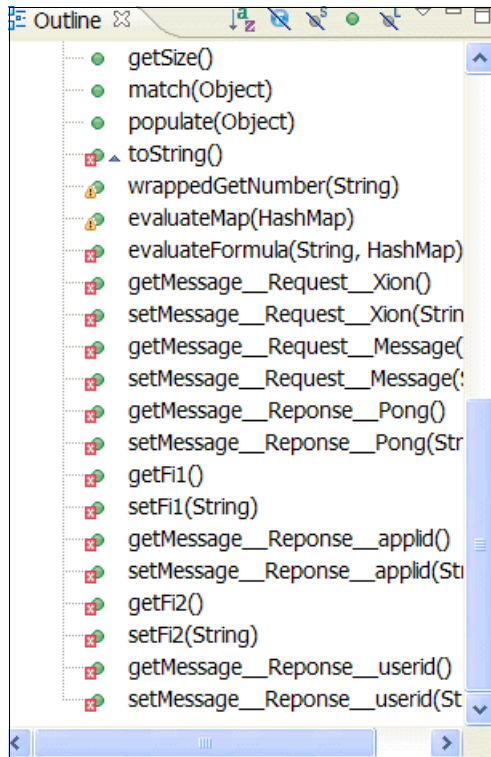


Figure 12-36 The PONGBLMS bean methods

The native Rational Developer for System z productivity environment features can be used to test, modify, and validate the PONGBLMS bean.

The bean can then be used within the dynamic scripts in the same manner as the JZOS ones, meaning that you simply import the class and use its methods.

### 12.1.8 The CICS catalog application tutorial

Find CICS samples at the following address:

[http://www.projectzero.org/wiki/Development/CicsSamples?S\\_CMP=rnav](http://www.projectzero.org/wiki/Development/CicsSamples?S_CMP=rnav)

The address also includes a tutorial named “Catalog example.” This tutorial describes how to create a dynamic scripting front-end to the *catalog application* CICS sample application. The front-end provides search and sort capabilities for the catalog and allow the user to buy more than one item at a time, where the existing 3270 interface is more restricted.

We encourage you to look at this tutorial. The major steps required to create this front-end are as follows:

1. Create a catalog.php PHP resource handler from the magic of dynamic scripting.
2. Create the RESTful interface to the catalog file.  
Add a browse file capability from the magic of JCICS for the CICS resource access and the benefit of JZOS for the VSAM record COBOL structure access. It is named Catalog in this tutorial.
3. The VSAM record is presented as JSON format data, which looks similar to rendering ATOM data. Again, the benefit of Project Zero simplicity. See Figure 12-37.

```
$myCatalogItem = new  
Java('com.ibm.cics.catalog.Catalog',$myRecordHolder->value);  
...  
zput('/request/view',"JSON");  
zput('/request/json/output',$myCatalogItem);  
render_view();
```

*Figure 12-37 VSAM record*

Dojo is an open source toolkit for JavaScript and is designed to make web page construction easier. It includes tables with clickable column headings that can support sorting. The tutorial makes use of it to enable column sorting on the catalog. It also stores the catalog items into a Dojo data store which could then be shared with other Dojo widgets.

## 12.2 Summary

The scenarios have described various capabilities of the CICS Dynamic Scripting feature:

- ▶ Handling of RESTful style interaction patterns
- ▶ Access to CICS resource using the JCICS API
- ▶ Handling of Language Structures such as COBOL or PL/I
- ▶ Rendering of the returned data such as ATOM or JSON

These capabilities are the enablers to CICS integration to mashups or situational applications.



## Command-line sample

In this chapter, we describe extending the command-line to provide additional functionality. This addition includes commands to give status information, commands to manage configuration files, and new composite commands.

## 13.1 Command-line commands

Every ZERO command is implemented within a module:

- ▶ The **zero create** command-line command is in the `zero.cli.tasks` module. You may create new command-line commands that are available to your application.
- ▶ The **zero start** and **zero stop** commands are in the `zero.core` module. All applications have a dependency on `zero.core`.

Command-line commands must be created in the `app/tasks` folder.

## 13.2 Command-line basics

Creating a PHP or a Groovy script in the `app/tasks` directory with a function declaration creates a new command with the name of that file.

Additional commands can be written in Groovy, PHP, or Java.

Creating commands in Groovy or PHP consists of creating a file within the `app/tasks` folder. The tasks directory is not created as a part of a new application, and must be created, for example with the `mkdir -p app/tasks` command.

Creating commands in Java requires additional work to register the command with Dynamic Scripting. This is referred to in the documentation as *registering an event handler*.

## 13.3 JVM servers, applications and commands

Each **zero** command runs under its own JVM server. If the application is currently running, an additional JVM Server will be created to process the command. The application's JVM Server is not used. Figure 13-1 shows the flow of control between CICS and UNIX Systems Services.

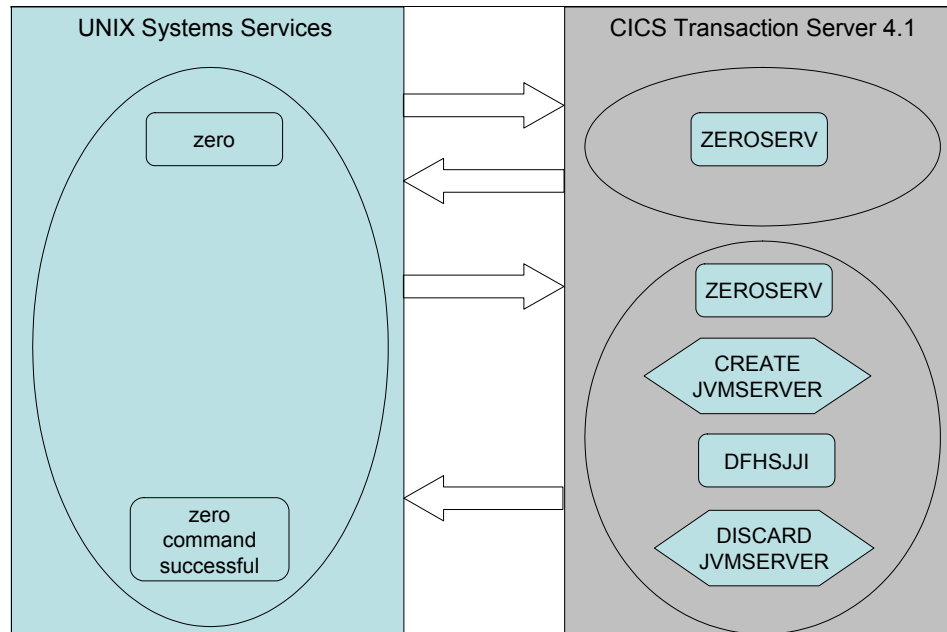


Figure 13-1 Flow of data between UNIX Systems Services and CICS

New commands created can use the jCICS API, or any of the Dynamic Scripting APIs.

The application does not have to be running for command-line commands to be issued. This is because the command runs in a separate JVM server to the application itself.

## 13.4 Creating a simple command in Groovy

To create a new command written in Groovy, we create a file within the `app/tasks` directory. The name of the file is the name of the command.

Any command written in Groovy must include the `onCliTask()` function, which is called when the command is issued.

We create a simple command that displays the name of the dynamic scripting application and the name of the CICS region. See Example 13-1.

*Example 13-1 app/tasks/cics.groovy*

---

```
import com.ibm.cics.server.Region;
def onCliTask()
{
    def task = zget("/event/task")
    zget("/event/args").each{System.out.println("arg is " + it)}
    // Obtain the name of the CICS region in which we are running in
    Region myRegion = new Region();
    String myApplid = myRegion.getAPPLID();
    // Determine the name of the application
    String application = zget("/config/name");
    System.out.println("CICS region $myApplid running application
$application");
}
```

---

This command can be run by issuing the **zero cics** command. Example 13-2 shows the output.

*Example 13-2 Sample output for zero cics command*

---

```
CICS region EPRED2 running application command.test
```

---



## 13.5 Creating a simple command in PHP

To create a new command written in PHP, we create a file within the `app/tasks` directory. The name of the file is the name of the command.

To use PHP, the application must be declared as having a dependency on `zero.cics.php`.

Any command written in PHP must define a class. The name of the class must match the name of the file, and the name of the command.

This class must include the `onCliTask()` function. This function is called when the command is issued.

We create a simple command that displays the name of the dynamic scripting application and the name of the CICS region. See Example 13-25 on page 346.

*Example 13-3* `app/tasks/cicsphp.php`

---

```
<?php
java_import("com.ibm.cics.server.Region");
class cicsphp
{
    function onCliTask () {
// Obtain the name of the command
        $task = zget("/event/task");
// Display the name of the command, followed by a new line
        echo "Task name is $task.\n";
// Get the command line options
        $args = zget("/event/args");
// Display each command line option, followed by a new line
        foreach ($args as $arg)
        {
            echo " Arg is $arg.\n";
        }
// Create a new Java object which represents the CICS region
        $myRegion = new Java('com.ibm.cics.server.Region');
// Get the CICS applid from the CICS region object
        $myApplid = $myRegion->getAPPLID();
// Get the name of the application using the Global Context API
        $myApplication = zget('/config/name');
// Display the name of the region and application
        echo "CICS Region " . $myApplid . " running application " .
$myApplication . "\n";
    }
}
?>
```

---

Example 13-4 shows the output from this command.

*Example 13-4 Sample output for zero cicsphp command*

---

```
Task name is cicsphp.  
CICS Region EPRED2 running application redb.command
```

---

## 13.6 Creating a simple command in Java

To create a new command written in Java, we create a file within the `java` directory. The name of the class is the name of the command.

The class must declare a public `cliTask()` method.

We create a simple command that displays the name of the dynamic scripting application and the name of the CICS region. See Example 13-5.

*Example 13-5 /java/cicsjava.java*

---

```
import com.ibm.cics.server.Region;  
import zero.core.connection.*;  
import zero.core.context.GlobalContext;  
  
public class cicsjava  
{  
    public void onCliTask()  
    {  
        String task = GlobalContext.zget("/event/task");  
        java.util.ArrayList args = GlobalContext.zget("/event/args");  
        for(int i = 0; i < args.size(); i++)  
        {  
            System.out.println("arg is " + args.get(i));  
        }  
        // Obtain the name of the CICS region in which we are running in  
        Region myRegion = new Region();  
        String myApplid = myRegion.getAPPLID();  
        // Determine the name of the application  
        String application = GlobalContext.zget("/config/name");  
        System.out.println("CICS region " + myApplid + " running  
application " + application);  
    }  
}
```

---

To use the new Java command, we must register the event in `zero.config`. Groovy and PHP commands in `app/tasks` are registered automatically. Example 13-6 shows the necessary lines to do this task:

- ▶ A new event is added to `/config/handlers` with the `+=` operand.
- ▶ The only event that this new event can be called for is `cliTask`. Other events include GET, POST, PUT and DELETE.
- ▶ The handler is the fully qualified name of the Java class.
- ▶ The condition is a regular expression indicating anything matching `cicsjava`.

*Example 13-6 The zero.config configuration for cicsjava command*

---

```
# cicsjava
/config/handlers += [{
  "events": "cliTask",
  "handler" : "cicsjava.class",
  "conditions" : "/event/task =~ cicsjava"
}]
```

---

The Java class must be compiled before use, and can be performed with the **zero compile** command. Example 13-7 shows the output for this command.

*Example 13-7 Sample output for the zero cicsjava command*

---

```
CICS region EPRED2 running application redb.commands
```

---

## 13.7 Providing help for commands

Issuing the **zero command** displays all currently available commands. The new commands **cics**, **cicsphp** and **cicsjava** are not shown because they do not have help provided.

To add help documentation for the commands, you must create a help directory within the tasks directory.

Within the help directory, create a file named `command.properties`. The help file may be provided in multiple languages as described at the following address:

<http://www.projectzero.org/sMash/1.1.x/docs/zero.devguide.doc/zero.cli.tasks/ExtendingCli.html>

Example 13-8 shows the contents of the `cics.properties` file. The short line is the text displayed when all commands are listed. The full text is displayed when **zero help cics** is issued.

*Example 13-8 The cics.properties file for the cics command*

---

```
short=Displays information about the application and CICS region.
full.1=Usage:
full.2=
full.3=zero cics
full.4=
full.5=The cics command will display the APPLID and application name.
It is written in Groovy.
```

---

With all three command help files in place, issuing the **zero** command in the application gives a list of commands as shown in Example 13-9.

*Example 13-9 List of commands including new commands*

---

```
Usage: zero [main-opts] <task> [task-args]
main-opts:
  -d          enable minimal logging
  -v          enable all logging
  -l=<file>   use the given file to output log to

cics          Displays information about the application and CICS
region.
cicsjava      Displays information about the application and CICS
region.
cicsphp       Displays information about the application and CICS
region.
compile       Compiles all Java source files under the module's /java
              directory.
create        Creates a new module in the current directory.
encode        Encode string, e.g., passwords, for usage in
configuration files.
help          Prints the documentation for the given command.
info          Returns basic information about the application, like
its base     URL.
modproxy      Manage Apache mod_proxy configuration.
modulegroup   Manage modulegroups.
package       Packages the module as a zip file.
php           Executes a PHP script file as a command line task.
publish       Publish the module to your local repository.
```

recycle	Recycle the application.
repository	Manage repositories.
resolve	Determine the module's dependencies.
rollback	Reverts the effects of the last resolve or update.
runsql	Executes an SQL file against a database
search repository.	Finds and prints all matching modules in the
secretkey	Generate secret key used for encrypting information
shared between	application and the client.
start	Starts the application.
stats	Returns the current statistics collected for the
application.	
status	Return the status of the application.
stop	Stops the application.
switch	Switch the module group.
update	Resolves a module to its latest dependencies.
user	Create and edit users.
validatedb	Validates if the provided configuration can
successfully connect	
	to a database.
version	Displays version information

---

## 13.8 Creating subtasks

Rather than creating many separate commands, creating one command with subcommands is more convenient.

PHP and Groovy subtasks can be created by creating a directory in `app/tasks` with the name of the main command, and then scripts within that directory for each of the subtasks.

We create a directory named `redb` in the `app/tasks` directory.

### 13.8.1 Updating `zero.config` for the main command

Entering the **zero redb** command displays the help for the command, and lists any subcommands. However, because no handler exists for the command, the command fails.

Example 13-10 on page 340 shows the `zero.config` stanzas required to implement a handler for the **redb** command. The **redb** command without a subcommand will just display the help for it and any subcommands.

*Example 13-10 The zero.config lines for the redb command*

---

```
/config/handlers += [{  
    "events": "cliTask",  
    "handler" : "zero.cli.tasks.commands.SubCommand.class",  
    "conditions" : "/event/task =~ redb"  
}]
```

---

## 13.8.2 Creating a subcommand in Groovy

To create a new subcommand written in Groovy, we create a file within the `app/tasks/redb` directory. The name of the file is the name of the subcommand.

Any subcommand written in Groovy must include the `onCliSubTask()` function, which is called when the command is issued.

We create a subcommand that displays the name of the dynamic scripting application and the name of the CICS region. See Example 13-1 on page 334.

*Example 13-11 app/tasks/redb/cics.groovy*

---

```
import com.ibm.cics.server.Region;  
def onCliSubTask()  
{  
    def task = zget("/event/task")  
    zget("/event/args").each{System.out.println("arg is " + it)}  
    // Obtain the name of the CICS region in which we are running in  
    Region myRegion = new Region();  
    String myApplid = myRegion.getAPPLID();  
    // Determine the name of the application  
    String application = zget("/config/name");  
    System.out.println("CICS region $myApplid running application  
$application");  
}
```

---

This command can be run by issuing the command **zero redb cics** command. Example 13-12 shows sample output.

*Example 13-12 Sample output for zero redb cics command*

---

```
CICS region EPRED2 running application command.test
```

---

## Extending the subcommand to add more information

We now extend the subcommand to gather more information from the Global Context API. The extended command displays the following information:

- ▶ The ZERO\_HOME directory
- ▶ The application's home directory
- ▶ The HTTP port, if one exists
- ▶ The HTTPS port, if one exists

The ZERO\_HOME directory is where in the zFS file system the Dynamic Scripting Feature Pack is installed. It can be obtained by using the `zget()` method of the Global Context API, with the `"/config/zerohome"` key, as shown in Example 13-13.

*Example 13-13 Obtaining the location of the installation of Dynamic Scripting*

---

```
// Determine where Dynamic Scripting is installed
String zerohome = zget("/config/zerohome");
System.out.println("ZERO_HOME is $zerohome");
```

---

The application's home directory can be obtained in a similar way by using the `zget()` method. In this instance, `"/config/root"` is the key, as shown in Example 13-14.

*Example 13-14 Obtaining the application's home directory*

---

```
// Determine where this application is installed
String zeroapphome = zget("/config/root");
System.out.println("Application home directory is $zeroapphome");
```

---

The HTTP and HTTPS ports can be obtained using `zget()` also. If the value is set, it is returned. If the value is not set, `null` is returned. See Example 13-15.

*Example 13-15 Determining the HTTP and HTTPS ports of an application*

---

```
Long httpPort = zget("/config/http/port");
Long httpsPort = zget("/config/https/port");
if(httpPort != null)
{
    System.out.println("Application $application accepts HTTP requests on
port $httpPort");
}
if(httpsPort != null)
{
    System.out.println("Application $application accepts HTTPS requests on
port $httpsPort");
}
```

---

Example 13-16 shows the output.

*Example 13-16 Useful information about an application*

---

```
CICS region EPRED9 running application zero.cics.redb.commands
ZERO_HOME is /dynamicScript/zero
Application home directory is
/dynamicScript/apps/zero.cics.redb.commands
Application zero.cics.redb.commands accepts HTTP requests on port 8280
CWPZT0600I: Command redb cics was successful
```

---

## **Adding additional information**

We want to add the following information to the command, to make it more useful:

- ▶ The application's JVM Server name
- ▶ The application's CICS resources (PIPELINEs, TCPIP SERVICES, URIMAPs)
- ▶ Optional: The contents of the application's stdout and stderr files

### ***The application's JVM Server name***

The currently running JVM Server name can be obtained from the JVM's system properties, as shown in Example 13-17.

*Example 13-17 Retrieve current JVM Server ID from System.getProperty()*

---

```
String jvmserver = System.getProperty("zero.cics.jvmserver.id");
```

---

However, as noted in 13.3, “JVM servers, applications and commands” on page 333, each ZERO command runs in its own JVM server, and not in that of the application. Therefore, the JVM Server name that is obtained from this command is that of the current JVM, and not that of the application.

**Note:** Obtaining the JVM Server name of the application by using an undocumented internal file within the application's file structure is possible. This task is undocumented and is therefore unsupported, and might change at any time in the future.

The JVM Server name of the application can be found by looking at the `.zero/private/cics/jvmserver_id` file. This file contains text in the local EBCDIC code page, as Example 13-18 on page 343 shows. The text contains the JVMSERVER resource ID, and also the numeric suffix used for all other CICS resources.



*Example 13-18 Contents of jvmserver\_id file*

---

JC000007,000007

---

The contents of this file can be retrieved using the Connection API and the file:/// protocol, as Example 13-19 shows.

*Example 13-19 Obtaining the JVM Server ID from the jvmserver\_id file*

---

```
String encoding = System.getProperty("zero.uss.encoding");
String jvmid =
Connection.doGET("file:///~/zero/private/cics/jvmserver_id").getRespon
seBodyAsString(encoding);
String jvmserver = jvmid.substring(0,jvmid.indexOf(','));
String suffix =
jvmid.substring(++(jvmid.indexOf(',')),--(jvmid.length()));
System.out.println("Last application JVM Server was $jvmserver");
```

---

Example 13-19 shows the following information

- ▶ The first line retrieves the system encoding from the JVM's properties. We must do this because the file is in EBCDIC and not in UTF-8.
- ▶ The Connection.doGET() method returns a ConnectionResponse object. We invoke the getResponseBodyAsString() method on this returned object using the encoding from the JVM.
- ▶ The JVM Server name can be obtained by using a subString() on the contents of the file, stopping at the first comma.
- ▶ The resource suffix can be obtained in a similar manner, starting at the character after the first comma, and omitting the last character. The reason is because the last character in the file is a new-line character.

**Note:** Because the jvmserver\_id file is an undocumented interface, we use a try-catch construction. The file might not exist if the application has never been started, for example.

### ***The application's CICS resources***

As noted in “CICS resources for applications and the command-line interface” in the CICS Transaction Server Feature Pack for Dynamic Scripting documentation, the CICS resources that are required by a running application consist of a two-character prefix, and a six-digit numeric suffix.

The JVM Server name can be simply displayed (Example 13-20).

*Example 13-20 JVM Server name displayed*

---

```
System.out.println("Last application JVM Server was $jvmserver");
```

---

We have obtained the suffix from the `jvmserver_id` file. We can now determine the names of the other CICS resources.

The PIPELINE resource begins with the characters PI. Example 13-21 shows how we can simply display the PIPELINE name.

*Example 13-21 Displaying the application PIPELINE resource name*

---

```
System.out.println("Application PIPELINE          : PI$suffix");
```

---

The names of the TCPIP SERVICES begin with TP for HTTP, and TS for HTTPS. As we previously obtained the HTTP and HTTPS ports from the Global Context, we can use these values to determine which TCPIP SERVICE is present. The same is true for the URIMAPS. The URIMAP is directly linked to the TCPIP SERVICE, and the names begin with UP for HTTP, and US for HTTPS, as Example 13-22 shows.

*Example 13-22 Displaying TCPIP SERVICE and URIMAP values*

---

```
if(httpPort != null)
{
    System.out.println("Application HTTP TCPIP SERVICE : TP$suffix");
    System.out.println("Application HTTP URIMAP       : UP$suffix");
}
if(httpsPort != null)
{
    System.out.println("Application HTTPS TCPIP SERVICE: TS$suffix");
    System.out.println("Application HTTPS URIMAP       : US$suffix");
}
```

---

Example 13-23 shows output from the command.

*Example 13-23 Output from expanded zero redb cics command*

---

```
CICS region EPRED9 running application zero.cics.redb.commands
ZERO_HOME is /dynamicScript/zero
Application home directory is
/dynamicScript/apps/zero.cics.redb.commands
Application zero.cics.redb.commands accepts HTTP requests on port 8280
Last application JVM Server was JC000029
Application PIPELINE           : PI000029
Application HTTP TCPIPService  : TP000029
Application HTTP URIMAP       : UP000029
CWPZT0600I: Command redb cics was successful
```

---

***The application's stdout and stderr files***

The final piece of this command is to add options to display the dfhjvmout and dfhjvmerr files of the application's JVM.

**Note:** the jvmserver\_id file is an undocumented interface and might change or be withdrawn in the future. In addition, in certain circumstances, the following commands will not work. For example, if the application is stopped, and the zerocics.config file is changed to reference a different CICS region, the dfhjvmout and dfhjvmerr files will not be found.

As can be seen from the directory structure of the dynamic scripting application, the JVM Server's stdout, stderr, and stdin files have a standard naming convention:

```
[CICS applid].[JVM server name].[dfhjvmin | dfhvmout | dfhjvmerr]
```

The contents of these files are in the local encoding, that is, in a variant of EBCDIC. Using the Connection API, we can obtain the contents of these files and display them as a convenience.

However, we only do this task if the correct option is specified. Options can be specified on the command line, and then tested.

Example 13-24 on page 346 shows how this approach can be done in Groovy. After declaring two variables, we obtain the command-line arguments. This is a list. Lists in Groovy contain a useful *each* operator that allows processing each argument simple.

Within the closure of the each operator, the value of each argument is held in the variable `it`. We simply compare the `it` variable with the known arguments `-stdout` and `-stderr`, and set the flags appropriately.

---

*Example 13-24 Testing*

---

```
boolean stdout_flag = false;
boolean stderr_flag = false;
zget("/event/args").each
{
    if(it == "-stdout")
    {
        stdout_flag = true;
    }
    if(it == "-stderr")
    {
        stderr_flag = true;
    }
}
```

---

Having set the flags, we can then construct the file names and retrieve the contents as we did for `jvmserver_id`. The encoding was obtained earlier from the Global Context API. Example 13-25 shows the final piece of code.

---

*Example 13-25 Displaying stdout and stderr files of the application JVM*

---

```
if(stdout_flag == true)
{
    String stdout_file = myApplid + "." + jvmserver + ".dfhjvmout";
    String stdout =
Connection.doGET("file:///~/$stdout_file").getResponseBodyAsString(encoding);
    System.out.println("===STDOUT===");
    System.out.println(stdout);
}
if(stderr_flag == true)
{
    String stderr_file = myApplid + "." + jvmserver + ".dfhjvmerr";
    String stderr =
Connection.doGET("file:///~/$stderr_file").getResponseBodyAsString(encoding);
    System.out.println("===STDERR===");
    System.out.println(stderr);
}
```

---

Example 13-26 shows the output from the following command:

```
zero redb cics -stdout -stderr
```

Note that System.out for command-line commands is routed to the shell session that issued it. Normally, System.out for CICS JVMs is routed to the dfhjvmout or dfhjvmerr file for that JVM server.

*Example 13-26 Output for zero redb cics -stdout -stderr*

---

```
CICS region EPRED9 running application zero.cics.redb.commands
ZERO_HOME is /dynamicScript/zero
Application home directory is
/dynamicScript/apps/zero.cics.redb.commands
Application zero.cics.redb.commands accepts HTTP requests on port 8280
Last application JVM Server was JC000029
Application PIPELINE      : PI000029
Application HTTP TCIPSERVICE : TP000029
Application HTTP URIMAP    : UP000029
====STDOUT====

====STDERR====
2010-10-26 05:54:50 zero.cics.entrypoints.AppInit::main Thread-0
      INFO [ CWPZI8504I: Application started in CICS with
JVMServer=JC000029,
APP_HOME=/dynamicScript/apps/zero.cics.redb.commands  ]

CWPZT0600I: Command redb cics was successful
```

---

### 13.8.3 Creating a subcommand in PHP

To create a new subcommand written in PHP, we create a file within the app/tasks/redb directory. The name of the file is the name of the subcommand.

To use PHP, the application must be declared a dependency on zero.cics.php.

Any command written in PHP must define a class. The name of the class must match the name of the file, and the name of the command.

This class must include the onCliSubTask() function, which is called when the subcommand is issued.

We create a simple subcommand that displays the name of the dynamic scripting application and the name of the CICS region, as Example 13-25 on page 346 shows.

*Example 13-27 The app/tasks/redb/cicsphp.php file*

---

```
<?php
java_import("com.ibm.cics.server.Region");
class cicsphp
{
    function onCliSubTask () {
// Obtain the name of the command
        $task = zget("/event/task");
// Display the name of the command, followed by a new line
        echo ("Task name is $task.\n");
        $subtask = zget("/event/subTask");
// Display the name of the command, followed by a new line
        echo ("Subtask name is $subtask.\n");
// Get the command line options
        $args = zget("/event/args");
// Display each command line option, followed by a new line
        foreach ($args as $arg)
        {
            echo " Arg is $arg.\n";
        }
// Create a new Java object which represents the CICS region
        $myRegion = new Java('com.ibm.cics.server.Region');
// Get the CICS applid from the CICS region object
        $myApplid = $myRegion->getAPPLID();
// Get the name of the application using the Global Context API
        $myApplication = zget('/config/name');
// Display the name of the region and application
        echo "CICS Region " . $myApplid . " running application " .
$myApplication . "\n";
    }
}
?>
```

---

Example 13-28 shows output from this command.

*Example 13-28 Output for zero cicsphp command*

---

```
Task name is redb.
Subtask name is cicsphp.
CICS Region EPRED2 running application itso:redb.commands
```

## Extending the subcommand to add more information

We now extend the subcommand to gather more information from the Global Context API. The extended command displays the following information:

- ▶ The ZERO\_HOME directory
- ▶ The application's home directory
- ▶ The HTTP port, if there is one
- ▶ The HTTPS port, if there is one

The ZERO\_HOME directory is where in the zFS file system that the Dynamic Scripting Feature Pack is installed. It can be obtained by using the `zget()` method of the Global Context API, with the key `'/config/zerohome'` as shown in Example 13-29.

### *Example 13-29 Obtaining the location of the installation of Dynamic Scripting*

---

```
// Determine where Dynamic Scripting is installed
$zeroHome = zget('/config/zerohome');
echo "ZERO_HOME is $zeroHome \n";
```

---

The application's home directory can be obtained in a similar way by using the `zget()` method. The key in this instance is `'/config/root'` as shown in Example 13-30.

### *Example 13-30 Obtaining the application's home directory*

---

```
// Determine where this application is installed
$appHome = zget('/config/root');
echo "Application home directory is $appHome \n";
```

---

The HTTP and HTTPS ports can be obtained by using `zget()` also. If the value is set, it is returned. If the value is not set, `null` is returned. See Example 13-15 on page 341.

### *Example 13-31 Determining the HTTP and HTTPS ports of an application*

---

```
$httpPort = zget("/config/http/port");
$httpsPort = zget("/config/https/port");
if($httpPort)
{
    echo "Application $myApplication accepts HTTP requests on port
$httpPort\n";
}
if($httpsPort)
{
```

```
    echo "Application $myApplication accepts HTTPS requests on port  
$httpsPort\n";  
}
```

---

Example 13-32 shows the output.

*Example 13-32 Useful information about an application*

---

```
CICS Region EPRED9 running application zero.cics.redb.commands  
ZERO_HOME is /dynamicScript/zero  
Application home directory is  
/dynamicScript/apps/zero.cics.redb.commands  
Application zero.cics.redb.commands accepts HTTP requests on port 8280  
CWPZT0600I: Command redb cicsph was successful
```

---

## Adding additional information

We want to add more information to the command, to make it more useful. The information we would like to add is as follows:

- ▶ The application's JVM Server name
- ▶ The application's CICS resources (PIPELINEs, TCPIP SERVICES, URIMAPs)
- ▶ Optional: The contents of the application's stdout and stderr files

### *The application's JVM Server name*

The currently running JVM Server name can be obtained from the JVM's system properties, as Example 13-33 shows.

*Example 13-33 Retrieve current JVM Server ID from System.getProperty()*

---

```
$system = new JavaClass('java.lang.System');  
$jvmserver = $system->getProperty("zero.cics.jvmserver.id");
```

---

However, as noted in 13.3, “JVM servers, applications and commands” on page 333, each ZERO command runs in its own JVM server, and not in that of the application. Therefore, the JVM Server name obtained from this command will be that of the current JVM, and not that of the application.

**Note:** Obtaining the JVM Server name of the application by using an undocumented internal file within the application's file structure is possible. This is undocumented and is therefore unsupported, and may change at any time in the future.



The JVM Server name of the application can be found by looking at the `.zero/private/cics/jvmserver_id` file. This file contains text in the local EBCDIC code page, as Example 13-34 on page 351 shows. The text contains the JVMSERVER resource ID, and also the numeric suffix that is used for all other CICS resources.

*Example 13-34 Contents of jvmserver\_id file*

---

JC000007,000007

---

The contents of this file can be retrieved by using the Connection API and the `file:///` protocol, as Example 13-35 shows.

*Example 13-35 Obtaining the JVM Server ID from the jvmserver\_id file*

---

```
$system = new JavaClass('java.lang.System');
$encoding = $system->getProperty("zero.uss.encoding");
$jvmidResponse =
Connection::doGET("file:///~/.zero/private/cics/jvmserver_id");
$jvmid = $jvmidResponse->getResponseBodyAsString($encoding);
$jvmserver = explode(",", $jvmid);
```

---

Example 13-35 has the following information:

- ▶ The first line constructs a new `java.lang.System` class reference. We then obtain the `System` property, on the second line, because the file is in EBCDIC and not in UTF-8.
- ▶ The `Connection::doGET()` method returns a `ConnectionResponse` object. We use double colon (`::`) characters because we are invoking a static method.
- ▶ We invoke the `getResponseBodyAsString()` method on this returned object using the encoding from the JVM.
- ▶ We use the PHP `explode` function to split the contents of the file in two, with the comma as the delimiter.
- ▶ We are left with a two-element array, `$jvmserver`.

**Note:** Because the `jvmserver_id` file is an undocumented interface, we use a try-catch construction. The file might not exist if the application has never been started, for example.

### ***The application's CICS resources***

As noted in “CICS resources for applications and the command-line interface” in the CICS Transaction Server Feature Pack for Dynamic Scripting documentation,

the CICS resources that are required by a running application consist of a two-character prefix, and a six-digit numeric suffix.

Example 13-36 on page 352 shows how we can display the name of the JVM server used by the application. We need to add a new-line character at the end for formatting.

---

*Example 13-36 Displaying the name*

---

```
echo "Last application JVM server was $jvmserver[0]\n";
```

---

We have obtained the suffix from the `jvmserver_id` file. We can now determine the names of the other CICS resources.

The PIPELINE resource begins with the characters PI. Example 13-37 shows how we can simply display the PIPELINE name.

---

*Example 13-37 Displaying the application PIPELINE resource name*

---

```
echo "Application PIPELINE          : PI$jvmserver[1]";
```

---

The names of the TCPIP SERVICES begin with TP for HTTP, and TS for HTTPS. Because we previously obtained the HTTP and HTTPS ports from the Global Context, we can use these values to determine which TCPIP SERVICE is present. The same is true for the URIMAPs. The URIMAP is directly linked to the TCPIP SERVICE, and the names begin with UP for HTTP, and US for HTTPS, which Example 13-38 shows.

---

*Example 13-38 Displaying TCPIP SERVICE and URIMAP values*

---

```
if($httpPort)
{
    echo "Application HTTP TCPIP SERVICE : TP$jvmserver[1]";
    echo "Application HTTP URIMAP       : UP$jvmserver[1]";
}
if($httpsPort)
{
    echo "Application HTTPS TCPIP SERVICE: TS$jvmserver[1]";
    echo "Application HTTPS URIMAP      : US$jvmserver[1]";
}
```

---

Example 13-39 shows output from the command.

---

*Example 13-39 Output from expanded zero redb cicsphp command*

---

```
CICS Region EPRED9 running application zero.cics.redb.commands
ZERO_HOME is /dynamicScript/zero
Application home directory is /dynamicScript/apps/zero.cics.redb.commands
```

```
Application zero.cics.redb.commands accepts HTTP requests on port 8280
Last application JVM server was JC000029
Application PIPELINE           : PI000029
Application HTTP TCPIPService : TP000029
Application HTTP URIMAP       : UP000029
CWPZT0600I: Command redb cicsphp was successful
```

---

### ***The application's stdout and stderr files***

The final piece of this command is to add options to display the dfhjvmout and dfhjvmerr files of the application's JVM.

**Note:** the jvmserver\_id file is an undocumented interface and might change or be withdrawn in the future. In addition, in certain circumstances, the following commands will not work. For example, if the application is stopped, and then the zerocics.config file is changed to reference a different CICS region, then the dfhjvmout and dfhjvmerr files will not be found.

As can be seen from the directory structure of the dynamic scripting application, the JVM Server's stdout, stderr, and stdin files have a standard naming convention:

```
[CICS applid].[JVM server name].[dfhjvmin | dfhvmout | dfhjvmerr]
```

The contents of these files are in the local encoding, that is. in a variant of EBCDIC. Using the Connection API, we can obtain the contents of these files and display them as a convenience.

However, we do this task only if the correct option is specified. Options can be specified on the command line, and then tested for.

Example 13-40 on page 354 shows how this way can be done in PHP. After declaring two variables, we obtain the command-line arguments. PHP contains a useful *foreach* construct to iterate through a list or array. Within the closure of the foreach operator, the value of each element of argument one is held in the variable named as argument two.

We simply compare the *it* variable with the known arguments -stdout and -stderr, and set the flags appropriately.

#### *Example 13-40 Using PHP*

---

```
// Get the command line options
$args = zget("/event/args");
$stdout_flag = false;
$stderr_flag = false;
// Display each command line option, followed by a new line
foreach ($args as $arg)
{
    if($arg == "-stdout")
    {
        $stdout_flag = true;
    }
    if($arg == "-stderr")
    {
        $stderr_flag = true;
    }
}
```

---

Having set the flags, we can then construct the file names and retrieve the contents as we did for `jvmserver_id`. The encoding was obtained earlier from the Global Context API. Example 13-41 shows the final piece of code.

#### *Example 13-41 Displaying stdout and stderr files of the application JVM*

---

```
if($stdout_flag)
{
    $stdout_file = $myApplid . "." . $jvmserver[0] . ".dfhjvmout";
    $stdout_response = Connection::doGET("file:///~/$stdout_file");
    $stdout = $stdout_response->getResponseBodyAsString($encoding);
    echo "\n===STDOUT===\n";
    echo $stdout;
}
if($stderr_flag)
{
    $stderr_file = $myApplid . "." . $jvmserver[0] . ".dfhjvmerr";
    $stderr_response = Connection::doGET("file:///~/$stderr_file");
    $stderr = $stderr_response->getResponseBodyAsString($encoding);
    echo "\n===STDERR===\n";
    echo $stderr;
}
```

---

Example 13-42 shows the output from the **zero redb cicsphp -stdout -stderr** command.

*Example 13-42 Output for zero redb cics -stdout -stderr*

---

```
CICS region EPRED9 running application zero.cics.redb.commands
ZERO_HOME is /dynamicScript/zero
Application home directory is
/dynamicScript/apps/zero.cics.redb.commands
Application zero.cics.redb.commands accepts HTTP requests on port 8280
Last application JVM Server was JC000029
Application PIPELINE           : PI000029
Application HTTP TCIPSERVICE  : TP000029
Application HTTP URIMAP        : UP000029
====STDOUT====

====STDERR====
2010-10-26 05:54:50 zero.cics.entrypoints.AppInit::main Thread-0
      INFO [ CWPZI8504I: Application started in CICS with
JVMServer=JC000029,
APP_HOME=/dynamicScript/apps/zero.cics.redb.commands  ]

CWPZT0600I: Command redb cicsphp was successful
```

---

### 13.8.4 Creating a subcommand in Java

To create a new subcommand written in Java we create a file within the java directory. The name of the class is the name of the subcommand.

The class must declare a public `onCliSubTask()` method.

We create a simple command that displays the name of the dynamic scripting application and the name of the CICS region. See Example 13-43.

*Example 13-43 /java/redb/cicsjava.java*

---

```
package redb;

import com.ibm.cics.server.Region;
import zero.core.connection.*;
import zero.core.context.GlobalContext;

public class cicsjava
{
    public void onCliSubTask()
```

```

    {
        // Obtain the name of the CICS region in which we are running in
        Region myRegion = new Region();
        String myApplid = myRegion.getAPPLID();
        // Determine the name of the application
        String application = GlobalContext.zget("/config/name");
        System.out.println("CICS region " + myApplid + " running
application " + application);
    }
}

```

---

To use the new Java command, we must register the event in `zero.config` file. Groovy and PHP commands in `app/tasks` are registered automatically. Example 13-44 shows the necessary lines, which are as follows:

- ▶ A new event is added to `/config/handlers` with the `+=` operand.
- ▶ The only event that this new event can be called for is `cliSubTask`. Other events include `GET`, `POST`, `PUT` and `DELETE`.
- ▶ The handler is the fully qualified name of the `redb.cicsjava` Java class.
- ▶ The condition is a regular expression indicating anything matching `redb` for the task, and `cicsjava` for the subtask.

*Example 13-44 zero.config configuration for redb cicsjava subcommand*

---

```

# redb cicsjava
/config/handlers += [{
    "events": "cliSubTask",
    "handler" : "redb.cicsjava.class",
    "conditions" : "(/event/task =~ redb) && (/event/subTask =~
cicsjava)"
}]

```

---

The Java class must be compiled, with the **zero compile** command before use.

Example 13-45 shows the output for this command.

*Example 13-45 Sample output for the zero redb cicsjava command*

---

```

Task is redb subtask cicsjava
CICS region EPRED2 running application itso:redb.commands
CWPZT0600I: Command redb cicsjava was successful

```

---

## Extending the subcommand to add more information

We now extend the subcommand to gather more information from the Global Context API. The extended command will display the following information:

- ▶ The ZERO\_HOME directory
- ▶ The application's home directory
- ▶ The HTTP port, if one exists
- ▶ The HTTPS port, if one exists

**Note:** if you are using Java to write command-line extensions, remember that you must compile Java by using the **zero compile** command after any change. However, because the JVM server for a command line is created for each and every command, you do not need to reset any JVMs.

The ZERO\_HOME directory is where in the zFS file system the Dynamic Scripting Feature Pack is installed. This can be obtained by using the `zget()` method of the Global Context API, with the `"/config/zerohome"` key, as Example 13-46 shows.

*Example 13-46 Obtaining the location of the installation of Dynamic Scripting*

---

```
// Determine where Dynamic Scripting is installed
String zerohome = GlobalContext.zget("/config/zerohome");
System.out.println("ZERO_HOME is " + zerohome);
```

---

The application's home directory can be obtained in a similar way using the `zget()` method. In this instance, `"/config/root"` is the key as Example 13-47 shows.

*Example 13-47 Obtaining the application's home directory*

---

```
// Determine where this application is installed
String zeroapphome = GlobalContext.zget("/config/root");
System.out.println("Application home directory is " + zeroapphome);
```

---

The HTTP and HTTPS ports can be obtained using `zget()` also. If the value is set, it is returned. If the value is not set, `null` is returned. See Example 13-48.

*Example 13-48 Determining the http and https ports of an application*

---

```
Long httpPort = GlobalContext.zget("/config/http/port");
Long httpsPort = GlobalContext.zget("/config/https/port");
if(httpPort != null)
{
    System.out.println("Application " + application + " accepts HTTP
requests on port " + httpPort);
}
if(httpsPort != null)
{
    System.out.println("Application " + application + " accepts HTTPS
requests on port " + httpsPort);
}
```

---

Output is shown in Example 13-49.

*Example 13-49 Useful information about an application*

---

```
CICS region EPRED9 running application zero.cics.redb.commands
ZERO_HOME is /dynamicScript/zero
Application home directory is
/dynamicScript/apps/zero.cics.redb.commands
Application zero.cics.redb.commands accepts HTTP requests on port 8280
CWPZT0600I: Command redb cicsjava was successful
```

---

## Adding more information

We want to add more information to the command to make it more useful. The information we want to add is as follows:

- ▶ The application's JVM Server name
- ▶ The application's CICS resources (PIPELINES, TCPIP SERVICES, URIMAPS)
- ▶ Optional: The contents of the application's stdout and stderr files



### ***The application's JVM Server name***

The currently running JVM Server name can be obtained from the JVM's System Properties, as Example 13-50 shows.

*Example 13-50 Retrieve current JVM Server ID from System.getProperty()*

---

```
String jvmserver =  
java.lang.System.getProperty("zero.cics.jvmserver.id");
```

---

However, as noted in 13.3, “JVM servers, applications and commands” on page 333, each ZERO command runs in its own JVM server, and not in that of the application. Therefore, the JVM Server name obtained from this command will be that of the current JVM, and not that of the application.

**Note:** Obtaining the JVM Server name of the application by using an undocumented internal file within the application's file structure is possible. This task is undocumented and is therefore unsupported, and might change at any time in the future.

The JVM Server name of the application can be found by looking at the `.zero/private/cics/jvmserver_id` file. This file contains text in the local EBCDIC code page, as Example 13-51 shows. The text contains the JVMSERVER resource ID, and also the numeric suffix used for all other CICS resources.

*Example 13-51 Contents of jvmserver\_id file*

---

```
JC000007,000007
```

---

The contents of this file can be retrieved using the Connection API and the `file:///` protocol, as Example 13-52 shows.

*Example 13-52 Obtaining the JVM Server ID from the jvmserver\_id file*

---

```
String encoding = System.getProperty("zero.uss.encoding");  
String jvmid =  
Connection.doGET("file:///~/zero/private/cics/jvmserver_id").getRespon  
seBodyAsString(encoding);  
int comma = jvmid.indexOf(',');  
int jvmidLength = jvmid.length();  
jvmidLength--;  
String jvmserver = jvmid.substring(0,comma);  
String suffix = jvmid.substring(++(comma),jvmidLength);
```

---

The example shows the following information:

- ▶ The first line retrieves the JVM property so we know the correct form of EBCDIC to use.
- ▶ The `Connection.doGET()` static method returns a `ConnectionResponse` object.
- ▶ We invoke the `getResponseBodyAsString()` method on this returned object using the encoding from the JVM.
- ▶ We use the Java substring function to split the contents of the file in two, with the comma as the delimiter. We discard the last character, which is a new-line.
- ▶ We are left with a two variables, `jvmserver` and `suffix`.

**Note:** Because the `jvmserver_id` file is an undocumented interface, we use a try-catch construction. The file might not exist if the application has never been started, for example.

### ***The application's CICS resources***

As noted in “CICS resources for applications and the command-line interface” in the CICS Transaction Server Feature Pack for Dynamic Scripting documentation, the CICS resources required by a running application consist of a two-character prefix, and a six-digit numeric suffix.

We have obtained the suffix from the `jvmserver_id` file. We can now determine the names of the other CICS resources.

The `JVMSERVER` resource is that specified in the `jvmserver[0]` variable. It always begins with `JC`. Example 13-53 shows how to display the name of the JVM server used by the application.

#### *Example 13-53 JVM server name last used by application*

---

```
System.out.println("Last application JVM Server was " + jvmserver);
```

---

The `PIPELINE` resource begins with the characters `PI`. Example 13-54 shows how we can display the `PIPELINE` name.

#### *Example 13-54 Displaying the application PIPELINE resource name*

---

```
System.out.println("Application PIPELINE          : PI" + suffix);
```

---

The names of the `TCPIP SERVICES` begin with `TP` for `HTTP`, and `TS` for `HTTPS`. Because we previously obtained the `HTTP` and `HTTPS` ports from the `Global Context`, we can use these values to determine which `TCPIP SERVICE` is

present. The same is true for the URIMAPs. The URIMAP is directly linked to the TCIPSERVICE, and the names begin with UP for HTTP, and US for HTTPS. Example 13-55 shows the code.

*Example 13-55 Displaying TCIPSERVICE and URIMAP values*

---

```
if(httpPort != null)
{
    System.out.println("Application HTTP TCIPSERVICE : TP" + suffix);
    System.out.println("Application HTTP URIMAP      : UP" + suffix);
}
if(httpsPort != null)
{
    System.out.println("Application HTTPS TCIPSERVICE: TS" + suffix);
    System.out.println("Application HTTPS URIMAP      : US" + suffix);
}
```

---

Example 13-56 shows output from the command.

*Example 13-56 Output from expanded zero redb cics command*

---

```
CICS Region EPRED9 running application zero.cics.redb.commands
ZERO_HOME is /dynamicScript/zero
Application home directory is
/dynamicScript/apps/zero.cics.redb.commands
Application zero.cics.redb.commands accepts HTTP requests on port 8280
Application PIPELINE      : PI000029
Application HTTP TCIPSERVICE : TP000029
Application HTTP URIMAP    : UP000029
CWPZT0600I: Command redb cicsphp was successful
```

---

### ***The application's stdout and stderr files***

The final piece of this command is to add options to display the dfhjvmout and dfhjvmerr files of the application's JVM.

**Note:** The jvmserver\_id file is an undocumented interface and might change or be withdrawn in the future. Also, in certain circumstances, the following commands will not work. For example, if the application is stopped, and then the zerocics.config file is changed to reference a different CICS region, the dfhjvmout and dfhjvmerr files will not be found.

As can be seen from the directory structure of the dynamic scripting application, the JVM Server's stdout, stderr, and stdin files have a standard naming convention:

```
[CICS applid].[JVM server name].[dfhjvmin | dfhvmout | dfhjvmerr]
```

The contents of these files are in the local encoding, that is, in a variant of EBCDIC. Using the Connection API, we can obtain the contents of these files and display them as a convenience. However, we do this only if the correct option is specified. Options can be specified on the command line, and then tested for.

Example 13-57 shows how this approach can be done in Java. After declaring two variables, we obtain the command-line arguments. We simply compare each argument with the known arguments -stdout and -stderr, and set the flags appropriately.

---

*Example 13-57 In Java*

---

```
java.util.ArrayList args = GlobalContext.zget("/event/args");
for(int i = 0; i < args.size(); i++)
{
    String tempString = (String) args.get(i);
    if(tempString.compareTo("-stdout") == 0)
    {
        stdout_flag = true;
    }
    if(tempString.compareTo("-stderr") == 0)
    {
        stderr_flag = true;
    }
}
```

---

Having set the flags, we can then construct the file names and retrieve the contents as we did for `jvmserver_id`. The encoding was obtained earlier from the Global Context API. Example 13-58 shows the final piece of code.

---

*Example 13-58 Displaying stdout and stderr files of the application JVM*

---

```
if(stdout_flag == true)
{
    String stdout_file = myApplid + "." + jvmserver + ".dfhjvmout";
    String stdout =
Connection.doGET("file:///~/$stdout_file").getResponseBodyAsString(encoding);
    System.out.println("====STDOUT====");
    System.out.println(stdout);
}
```

```

if(stderr_flag == true)
{
    String stderr_file = myApplid + "." + jvmserver + ".dfhjvmerr";
    String stderr =
Connection.doGET("file:///~/$stderr_file").getResponseBodyAsString(encoding);
    System.out.println("====STDERR====");
    System.out.println(stderr);
}

```

---

Example 13-59 shows the output from the **zero redb cicsjava -stdout -stderr** command.

*Example 13-59 Output for zero redb cicsjava -stdout -stderr*

---

```

CICS region EPRED9 running application zero.cics.redb.commands
ZERO_HOME is /dynamicScript/zero
Application home directory is
/dynamicScript/apps/zero.cics.redb.commands
Application zero.cics.redb.commands accepts HTTP requests on port 8280
Last application JVM Server was JC000029
Application PIPELINE      : PI000029
Application HTTP TCIPSERVICE : TP000029
Application HTTP URIMAP    : UP000029
====STDOUT====
====STDERR====
2010-10-26 05:54:50 zero.cics.entryptoints.AppInit::main Thread-0
      INFO [ CWPZI8504I: Application started in CICS with
JVMServer=JC000029,
APP_HOME=/dynamicScript/apps/zero.cics.redb.commands ]
CWPZT0600I: Command redb cicsjava was successful

```

---

## 13.9 Copying zerocics.config to the config directory

When Dynamic Scripting is installed, a default `zerocics.config` file is created in the installation's config directory.

The `zerocics.config` contains the name of the CICS region in which the application is to run.

If you want to run applications within a separate CICS region to the default, you must copy the `zerocics.config` file to the application's config directory.

We create a **zero redb config** command as an example of a more complex line command.

Note the following steps:

1. We must find where the default `zerocics.config` file is located. A copy of this file is always present in the `config` directory of the installation directory. The installation directory name can be obtained from the Global Context.
2. Changing `zerocics.config` changes the CICS region in which the application runs. Doing this step while an application is running can lead to orphaned JVMSERVER, TCPIPService, PIPELINE and URIMAP resources. Opening the same port, or starting the application in two CICS regions at once, might not be possible. Before copying the file, **zero redb config** attempts to determine if the application is running.
3. Copy the file from one directory to another.

### 13.9.1 Where is the default `zerocics.config` file

Configuration information for Dynamic Scripting and its applications is held within the Global Context. The Global Context can be queried by using the correct Global Context API. For Groovy and PHP, this task can be done with the **zget** command, specifying the correct key.

The key for the installation directory of Dynamic Scripting is `/config/zerohome`, as Example 13-60 shows.

*Example 13-60 Example of determining installation directory from Global Context*

---

```
// Where is Dynamic Scripting installed?  
String zerohome = zget("/config/zerohome");
```

---

### 13.9.2 Is the application running

Dynamic Scripting provides an Application API to assist in managing applications. The API includes commands to start and stop the application. It also has commands to determine the status of the application and the names of files in the config directory.

To use the Application API, we must build an Application object. The Application object can only be built if we know the name or the location in the file system of the dynamic scripting application. Both pieces of information are available from the Global Context API, as Example 13-61 on page 365 shows.

#### *Example 13-61 Obtaining the current Application using Application API*

---

```
import zero.management.appconfig.*;
...
String apphome = zget("/config/root");
File appHome = new File(apphome);
Application myApp = Application.getApplication(appHome);
```

---

After an Application object has been successfully created, we can use the `getStatus()` method to determine its status. An application is either started, stopped, or in an unknown state. Applications can be placed in an unknown state if, for example, the CICS region is shut down and then cold started whilst the application is running.

Application status is provided in an enumeration, and can thus be compared quite simply. Example 13-62 shows the code.

#### *Example 13-62 Determining application status*

---

```
if(myApp.getStatus() != Application.Status.STOPPED)
{
    System.out.println("Application $application is " +
myApp.getStatus() + ". Please stop application before using this
command.");
}
```

---

### **13.9.3 Does a zerocics.config file already exist**

If a `zerocics.config` file already exists, copying the default file replaces the `zerocics.config` file, which might change the CICS region and CICS resource information used by the application.

We use the Application API to query the existing configuration files for the current application. The Application API provides the `getConfigFileList()` method, which returns a String array containing the names of all files found in the application's config directory.

Having obtained this array, we iterate through it, searching for `zerocics.config`. If the file name `zerocics.config` exists within the array, we set a flag to true. Example 13-63 on page 366 shows the construction.

*Example 13-63 Searching for a zerocics.config file in the application*

---

```
String[] configFiles = myApp.getConfigFileList();
boolean zerocics_config = false;
for(int i = 0; i < configFiles.length; i++)
{
    String temp = configFiles[i];
    if(configFiles[i].compareTo("zerocics.config") == 0)
    {
        zerocics_config = true;
    }
}
```

---

### 13.9.4 Reading zerocics.config file

If the application is not started, we must read the zerocics.config file. We can use the Connection API with the file:/// protocol. The file protocol is intended to read and write files on the local file system.

Determining the installation directory is described in 13.9.1, “Where is the default zerocics.config file” on page 364. The zerocics.config file can be found within the config directory.

Example 13-63 shows the constructing of a URL for an HTTP or HTTPS connection.

Example 13-64 shows how to construct a file:/// URL. The syntax of the command is identical; the only part that changes is the URL itself.

The code is inserted within the catch part of the code.

*Example 13-64 Construct file:/// URL for the zerocics.config file*

---

```
// Build the URL for the default zerocics.config file
String myURL = "file://" + zerohome + "/config/zerocics.config";
Connection zerocics = new Connection(myURL);
```

---

In order to read the contents of the file, we need to create a ConnectionResponse object. This is automatically created by the send() method of the Connection object.

To read the file itself, we invoke the getResponseBodyAsString() method of the ConnectionResponse object, as Example 13-65 on page 367 shows. The zerocics.config file is already in UTF-8 and so does not need to be translated.



*Example 13-65 Reading the zerocics.config file into a local variable*

---

```
Connection.Response zerocicsResponse = zerocics.send();  
String zerocicsFile = zerocicsResponse.getResponseBodyAsString();
```

---

### 13.9.5 Writing zerocics.config

The Connection API allows the use of static methods, which can be used as a convenience. See Example 13-66. The **doPUT** command takes two parameters, a URL and the data to send. Dynamic Scripting offers a convenient way of referring to the application's own home directory. The tilde ( ~ ) character can be used in place of the home directory.

*Example 13-66 Creating the zerocics.config file in the application config directory*

---

```
Connection.doPUT("file:///~/config/zerocics.config", zerocicsFile);
```

---

### 13.9.6 Extending the config command to specify the CICS region

The most common reason to copy a `zerocics.config` file to an application's `config` directory is to allow the application to run in a different CICS region to the default. As a result, a logical step is to extend the **zero redb config** command to accept a parameter, which allows the CICS region to be changed by the command.

**Note:** Ensure that the CICS region you will use is configured for Dynamic Scripting and is started. Otherwise, all subsequent commands for this application fail unless you delete the application's `zerocics.config` file.

The steps are as follows:

1. Amend the arguments loop to detect a new `-cics` parameter.
2. Turn the contents of `zerocics.config` into an array.
3. Detect the line saying `CICS_APPLID` and change it.
4. Write out the file.

#### **Amend the arguments loop to detect a new -cics parameter**

Example 13-67 on page 368 shows the code necessary to detect a `-cics=APPLID` parameter, which is placed within a *for* loop. The **zget** command returns a list of arguments and then iterates through them.

If an argument is at least six characters long, we test it to see if it begins with `-cics=`. If it does, we test the size of the entire argument. If the argument is exactly six characters long, then no CICS APPLID has been specified. If the argument is greater than six characters long, we extract the extra characters to form the applid. However, if the new applid is now longer than eight characters, it is too long and is therefore invalid.

If characters are specified on the `-cics` option, we make them uppercase. The *z/OS Communications Server SNA Resource Definition Reference* notes that the following rules apply to VTAM® names (and therefore to CICS APPLIDs):

- ▶ The first character must be alphabetic (A-Z) characters, or one of the national (#, @, or \$) characters.
- ▶ The second through eighth characters must also be alphabetic (A-Z), one of the national characters (#, @ or \$), or alternatively numeric (0-9).

We assume that case sensitivity for the CICS applid within the command is not desirable, and so make all applids uppercase. However, we have not performed additional validation on the parameter.

A Boolean `newCICS` argument is used to determine whether or not a new CICS applid has been specified.

*Example 13-67 Detecting the -cics=NEWCICS argument in Groovy*

---

```
for(it in zget("/event/args"))
{
    if(it == "-overwrite")
    {
        overwrite = true;
    }
    if(it.length() >= 6)
    {
        if(it.substring(0,6) == "-cics=")
        {
            newCICS = true;
            if(it.size() == 6)
            {
                newCICS = false;
                System.out.println("-cics parameter must specify CICS
applid");
            }
            else
            {
                newCICS_applid = it.substring(6,it.size());
                if(newCICS_applid.size() > 8)
```

```

        {
            newCICS = false;
            System.out.println("-cics parameter applid must be 1 to
8 characters");
        }
        else
        {
            newCICS_applid = newCICS_applid.toUpperCase();
        }
    }
}
}
}

```

---

### Convert the contents of zerocics.config to an array

Each line of the `zerocics.config` file represents information that can be used to configure Dynamic Scripting. We need to find the `CICS_APPLID=` line.

To make this easier, we can turn the contents of the file from a `String` into an array of `Strings`. The `split()` function can be used to do this. We split the file into an array using carriage return and line feed characters as the separating characters.

Example 13-68 shows the code. We declare and instantiate a byte array containing the two characters carriage return and line feed. We then create a `String` using an explicit UTF-8 encoding. We call the `split` function to return a `String` array, and then set the `String` `zerocicsFile` to an empty `String`.

*Example 13-68 Turning zerocics.config into a String array*

---

```

byte[] crlf_array = new byte[2];

crlf_array[0] = (byte) '\r';
crlf_array[1] = (byte) '\n';

// we then declare a string using UTF-8
String crlf = new String(crlf_array,"UTF-8");

String[] exploded = zerocicsFile.split(crlf);
zerocicsFile = "";

```

---

## Detect the line saying CICS\_APPLID and change it

The String array exploded can be iterated through using the each function. For each entry, which represents a line in the zerocics.config file, we determine whether it is long enough to be CICS\_APPLID=. The line must be longer than 12 characters.

If the line matches the minimum requirement, we use the substring() and compareTo() methods to determine if we have a match with CICS\_APPLID=. If we do, we construct a new line from CICS\_APPLID=, the new CICS applid and carriage return and line feed, and then concatenate it to the zerocicsFile String.

If the line is too short, or it is not the correct line, we concatenate it back onto the zerocicsFile String with carriage return and line feed, as Example 13-69 shows.

*Example 13-69 Searching for CICS\_APPLID within zerocics.config*

---

```
exploded.each
{
    if(it.size() > 12)
    {
        // 12 is the size of CICS_APPLID=
        String CICS_APPLID = "CICS_APPLID=";
        if(it.substring(0,12).compareTo(CICS_APPLID) == 0)
        {
            // If we have found the right line, write out the new line
            String newConfigFileLine = CICS_APPLID + newCICS_applid +
            crlf;
            zerocicsFile = zerocicsFile.concat(newConfigFileLine);
            System.out.println("New CICS applid added to zerocics.config.
            CICS_APPLID=$newCICS_applid");
        }
        else
        {
            // Write out an existing line
            zerocicsFile = zerocicsFile.concat(it + crlf);
        }
    }
    else
    {
        // Write out an existing line
        zerocicsFile = zerocicsFile.concat(it + crlf);
    }
}
```

---

## Write out the file

The String `zerocicsFile` now contains the amended `zerocics.config` file. The Connection API, as before, can be used to write out the new file as Example 13-70 shows.

*Example 13-70 Writing out the new file*

---

```
Connection.doPUT("file:///~/config/zerocics.config", zerocicsFile);
```

---

## 13.10 Making commands available across applications

Commands are available within the application that contains them. However, it is not necessary to copy scripts from `app/tasks` to new applications.

Any command present in the module `redb.commands` is also present in any application that declares a dependency on the module, including HTML files in `public`, configuration data in `zero.config`, help files, and so on.

### 13.10.1 A useful file in `public/redb/`

As a convenience, we produce a Groovy file that we place in the following location:

```
zero.cics.redb.commands
```

To prevent clashes with other applications, we create the file in a new folder named `redb`. This file can now be referenced in any application that has a dependency on `zero.cics.redb.commands`.

Example 13-71 shows the contents of `redb/cics.groovy`. The web page gathers similar information to the **redb** commands.

*Example 13-71 Contents of `redb/cics.groovy`*

---

```
import com.ibm.cics.server.Region;
import zero.core.connection.*;
def onGET()
{

    // Obtain the name of the CICS region in which we are running in
    Region myRegion = new Region();
    String myApplid = myRegion.getAPPLID();
    // Determine the name of the application
    String application = zget("/config/name");
```

```

// Determine where Dynamic Scripting is installed
String zerohome = zget("/config/zerohome");

String jvmserver = System.getProperty("zero.cics.jvmserver.id");

// Determine where this application is installed
String zeroapphome = System.getProperty("zero.app.home");

// Determine the http and https ports
Long httpPort = zget("/config/http/port");
Long httpsPort = zget("/config/https/port");

// Display the information in a basic HTML table
println("<table border=2> <tr><th>CICS
region</th><th>Application</th><th>JVMServer</th><th>Application
Home</th><th>HTTP Port</th><th>HTTPS Port</th></tr>");
println("<tr><td>$myApplid</td>");
println("<td>$application</td>");
println("<td>$jvmserver</td>");
println("<td>$zeroapphome</td>");
println("<td>$httpPort</td>");
println("<td>$httpsPort</td></tr></table>");

// Next we determine the system encoding
String encoding = System.getProperty("zero.uss.encoding");

// Display the contents of the application's current dfhjvmout file
println("<h1>stdout</h1>");

// Construct a file:/// URL for the dfhjvmout file
String stdoutUrl = new String("file:///");
// dfhjvmout file is found in the application's home directory
stdoutUrl = stdoutUrl.concat(zeroapphome);
stdoutUrl = stdoutUrl.concat("/");
// the first part of the file name is the CICS applid
stdoutUrl = stdoutUrl.concat(myApplid);
stdoutUrl = stdoutUrl.concat(".");
// the second part is the JVM server name
stdoutUrl = stdoutUrl.concat(jvmserver);
stdoutUrl = stdoutUrl.concat(".");
// the final part is dfhjvmout
stdoutUrl = stdoutUrl.concat("dfhjvmout");
println("<h2>$stdoutUrl</h2>");
// Use the Connection API to get the file, and convert it to UTF-8

```

```

String stdout =
Connection.doGET(stdoutUrl).getResponseBodyAsString(encoding);
// Replace all new line characters with html <br> characters
stderr = stdout.replace("\n", "<br>");
println(stdout);
// Do the same for dfhvmerr.
println("<h1>stderr</h1>");
String stderrUrl = new String("file:///");
stderrUrl = stderrUrl.concat(zeroapphome);
stderrUrl = stderrUrl.concat("/");
stderrUrl = stderrUrl.concat(myApplid);
stderrUrl = stderrUrl.concat(".");
stderrUrl = stderrUrl.concat(jvmserver);
stderrUrl = stderrUrl.concat(".");
stderrUrl = stderrUrl.concat("dfhjvmerr");
println("<h2>${stderrUrl}</h2>");
String stderr =
Connection.doGET(stderrUrl).getResponseBodyAsString(encoding);
stderr = stderr.replace("\n", "<br>");
println(stderr);
}

```

Figure 13-2 shows the results when the page is referenced.

CICS region	Application	JVMServer	Application Home	HTTP Port	HTTPS Port
EPRED9	zero.cics.redb.commands	JC000568	/dynamicScript/apps/zero.cics.redb.commands	8280	null

**stdout**

**file:///dynamicScript/apps/zero.cics.redb.commands/EPRED9.JC000568.dfhjvmout**

**stderr**

**file:///dynamicScript/apps/zero.cics.redb.commands/EPRED9.JC000568.dfhjvmerr**

2010-11-11 03:02:08 zero.cics.entrypoints.AppInit:main Thread-0  
INFO [ CWPZI8504I: Application started in CICS with JVMServer=JC000568, APP\_HOME=/dynamicScript/apps/zero.cics.redb.commands]

Figure 13-2 The redb/cics.groovy page shown in a web browser

## 13.10.2 Packaging and publishing

We use the **zero package** and **zero publish** commands to put a copy of the application in the local repository.

Example 13-72 shows the **package** and **publish** commands. The **package** command specifies the `-includeSrc` option to include Java source code. The default is not to package the Java code, only the classes.

*Example 13-72 Commands to package and publish redb.commands*

---

```
zero package -includeSrc
zero publish
```

---

We can now create a new application, using the **zero create** command.

Entering the **zero redb** command fails, as Example 13-73 shows.

*Example 13-73 The redb command is unknown*

---

```
CWPZC0019E: Cannot find handler for task redb
CWPZI8503I: CICS unit of work rolled back after a CLI task returned a
non-zero return code.
```

---

Example 13-74 shows the addition of the dependency on `redb.commands` in the `ivy.xml` file.

*Example 13-74 Additional line in ivy.xml file of command.test*

---

```
<dependency name="redb.commands" org="itso" rev="[1.0.0.0, 2.0.0.0["/>
```

---

After editing the `ivy.xml` file, you must issue the **zero resolve** command to update the application's dependencies. You may then issue the **zero redb config** command.

Example 13-75 shows the sequence of command.

*Example 13-75 Sequence of command*

---

```
zero resolve
CWPZT0600I: Command resolve was successful

zero redb config

zerocics.config file copied from ZERO_HOME to command.test config directory
CWPZT0600I: Command redb config was successful
```

---



## 13.11 Combining existing commands

Normally, a change to a dynamic scripting application does not require that the application be restarted. New files may be added to the `public` directory, new application handlers may be added to the `app/resources` directory, and new command-line tasks may be added to the `app/tasks` directory without needing a restart of the application.

However, changes to the `zero.config`, `ivy.xml`, `php.ini`, or `zerocics.config` file always require a restart of the application. In addition, if Java code is recompiled, the JVM Server must be stopped and restarted so that CICS can pick up the latest version of the classes.

Commands already exist in Dynamic Scripting to stop an application (**zero stop**), start an application (**zero start**), refresh the dependencies list (**zero resolve** and **zero update**), and compile Java code (**zero compile**).

In this section, we create a composite command that performs the following actions:

- ▶ ZERO RESOLVE
- ▶ ZERO COMPILE
- ▶ ZERO STOP
- ▶ ZERO START

The name for the new composite command is as follows:

```
zero redb newcopy
```

### The zero resolve command

All command-line commands provided by Dynamic Scripting are in the `zero.cli.tasks.commands` package. They can be created and executed from within a command-line task.

**Note:** For information about the **zero resolve** command go to:

<http://www.projectzero.org/sMash/1.0.x/docs/apidocs/CORE/ALL/zero/cli/tasks/commands/Resolve.html>

This *internal code* might change in the future. If it does, these functions might no longer work.

The `runTask()` method of the command-line command takes as an argument a `java.util.List` object, which represents the arguments passed to the command.

Example 13-76 shows an example of how to set up and then run the **zero resolve** command within the **zero redb newcopy** command.

*Example 13-76 Running the zero resolve command programmatically*

---

```
import zero.cli.tasks.commands.*;
import zero.cli.tasks.CliException;
...
    Resolve myResolve = new Resolve();
    java.util.List args = [];
    // zero resolve
    System.out.println("About to resolve application");
    myResolve.runTask(args);
```

---

The command-line task sets a return-code indicating success or failure, but does not issue a Java Exception. In order to verify that the command worked, we must check the return-code. This can be obtained using the Global Context API. Example 13-77 shows an example of how to obtain the return-code, and some simple error handling.

*Example 13-77 Sample error handling on non-zero exit code from command*

---

```
int exitCode = zget("/config/exitCode");
if(exitCode != 0)
{
    throw new CliException("Non-zero return code on zero resolve");
}
```

---

## The zero compile command

The **zero compile** command compiles all the source code within the application's `java` directory. The code to perform this is identical with that of **zero resolve**, with the exception that the class name is `Compile`. Example 13-78 shows the code required.

*Example 13-78 The zero compile performed programmatically*

---

```
System.out.println("About to compile application");
Compile myCompile = new Compile();
myCompile.runTask(args);
exitCode = zget("/config/exitCode");
if(exitCode != 0)
{
    throw new CliException("Non-zero return code on zero compile");
}
```

---

**Note:** Documentation for the **zero compile** command is located at the following address:

<http://www.projectzero.org/sMash/1.0.x/docs/apidocs/CORE/ALL/zero/cli/tasks/commands/Compile.html>

This *internal code* might change in the future. If it does, these pieces of function might no longer work.

## The zero stop command

The **zero stop** command is provided both as a command-line command and as part of the Application API. The Application API is more useful and feature-rich, therefore we use the Application API to stop the application.

To create an Application object, you must retrieve the name of the application using the Global Context API, or provide the location in the file system of the application. Example 13-79 shows the latter.

*Example 13-79 Creating an application from the application location on the file system*

---

```
String apphome = zget("/config/root");
File appHome = new File(apphome);
Application myApp = Application.getApplication(appHome);
```

---

We can use the `start()` and `stop()` methods of the Application object to start and stop the application. However, these methods are asynchronous and they do not wait for completion. We must therefore use the `waitForStatusToEqual()` method of the Application. This way allows the task to wait until the stop is successful.

Example 13-80 shows this approach, with an arbitrary 10-second delay.

*Example 13-80 Application stop() and waitForStatusToEqual methods*

---

```
myApp.stop();
boolean stopped =
myApp.waitForStatusToEqual(Application.Status.STOPPED, 10000);
exitCode = zget("/config/exitCode");
if(exitCode != 0)
{
    throw new CliException("Non-zero return code on zero stop");
}
if(!stopped)
{
    throw new CliException("Application failed to stop within 10
seconds");
}
```

```

    }
    else
    {
        System.out.println("Application " + myApp.getName() + " is now "
+ myApp.getStatus());
    }

```

---

## The zero start command

The **zero start** is like **zero stop**. The `stop()` and `start()` methods of `Application` are used, and also `waitForStatusToEqual()`. Example 13-81 shows the final piece of code.

*Example 13-81 The zero start command used programmatically*

---

```

// zero start
System.out.println("About to issue start command");
myApp.start();

boolean started =
myApp.waitForStatusToEqual(Application.Status.STARTED,10000);
exitCode = zget("/config/exitCode");
if(exitCode != 0)
{
    throw new CliException("Non-zero return code on zero start");
}

if(!started)
{
    throw new CliException("Application failed to start within 10
seconds");
}
System.out.println("Application " + myApp.getName() + " is now "
+ myApp.getStatus());

```

---

# Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this book.

## IBM Redbooks

The following IBM Redbooks publications provide additional information about the topic in this document. Note that some publications referenced in this list might be available in softcopy only.

- ▶ *Extend the CICS Explorer: A Better Way to Manage Your CICS*, SG24-7819
- ▶ *IBM CICS Explorer*, SG24-7778
- ▶ *Implementing Event Processing with CICS*, SG24-7792
- ▶ *Smarter Banking with CICS Transaction Server*, SG24-7815
- ▶ *Threadsafe Considerations for CICS*, SG24-6351

You can search for, view, or download Redbooks, Redpapers, Technotes, draft publications and Additional materials, and order hardcopy Redbooks publications, at this website:

[ibm.com/redbooks](http://ibm.com/redbooks)

## Online resources

These websites are also relevant as further information sources:

- ▶ The Groovy website  
<http://groovy.codehaus.org>
- ▶ The Dojo website  
<http://www.dojotoolkit.org/>
- ▶ Project Zero  
<http://www.projectzero.org/>
- ▶ PHP  
<http://www.php.net/docs.php>

- ▶ PECL  
<http://pecl.php.net/>
- ▶ PHP Extension and Application Repository (PEAR)  
<http://pear.php.net/>
- ▶ Open source software  
<http://sourceforge.net/>
- ▶ JavaScript Object Notation (JSON)  
<http://json.org/>

## Help from IBM

IBM Support and downloads

[ibm.com/support](http://ibm.com/support)

IBM Global Services

[ibm.com/services](http://ibm.com/services)



# Introduction to ICS Dynamic Scripting

(0.5" spine)  
0.475" <-> 0.875"  
250 <-> 459 pages









# Introduction to CICS Dynamic Scripting

**Dynamic Scripting  
concepts, examples,  
and scenarios**

**Application  
management and  
troubleshooting**

**Feature pack  
installation and  
customization**

IBM CICS Transaction Server Feature Pack for Dynamic Scripting embeds and integrates technology from WebSphere sMash into the CICS TS V4.1 run time, helping to reduce the time and cost of CICS application development. The Feature Pack provides a robust, managed environment for a wide range of situational applications allowing PHP and Groovy developers to create reports, dashboards, and widgets, and integrate CICS assets into mash-ups, and much more.

The CICS Dynamic Scripting Feature Pack combines the benefits of scripted, Web 2.0 applications with easy and secure access to CICS application and data resources. The Feature Pack includes a PHP 5.2 run time implemented in Java and with Groovy language support, support for native Java code and access to many additional libraries and connectors to enhance the development and user experience of rich Internet applications. Access to CICS resources is achieved by using the JCICS APIs.

In this IBM Redbooks publication, we introduce the Dynamic Scripting Feature Pack, show how to install and customize it, and provide examples for using it.

## INTERNATIONAL TECHNICAL SUPPORT ORGANIZATION

### BUILDING TECHNICAL INFORMATION BASED ON PRACTICAL EXPERIENCE

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

**For more information:**  
[ibm.com/redbooks](http://ibm.com/redbooks)