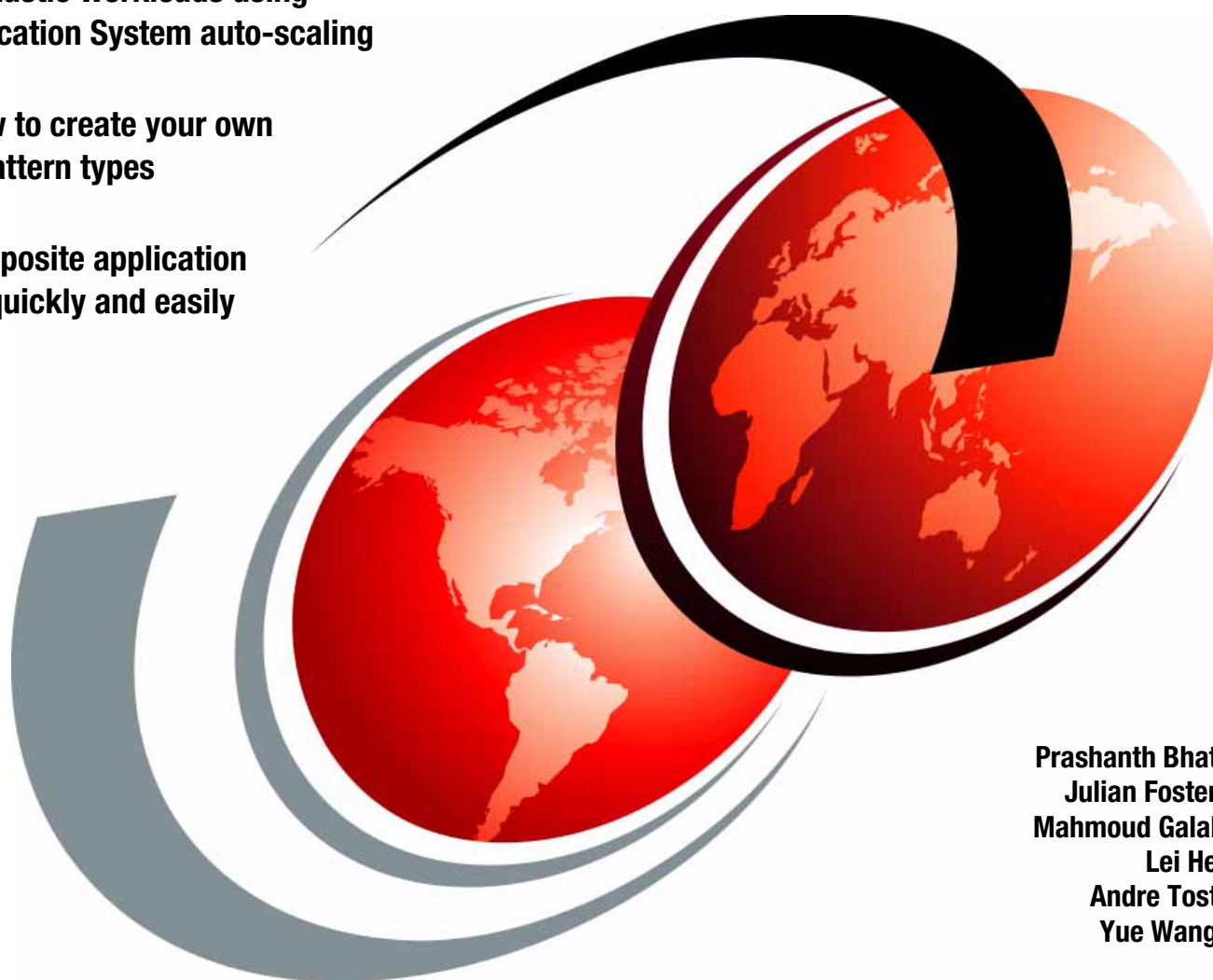


Creating Composite Application Pattern Models for IBM PureApplication System

Support elastic workloads using
PureApplication System auto-scaling

Learn how to create your own
custom pattern types

Build composite application
patterns quickly and easily



Prashanth Bhat
Julian Foster
Mahmoud Galal
Lei He
Andre Tost
Yue Wang

Redbooks



International Technical Support Organization

**Creating Composite Application Pattern Models for
IBM PureApplication System**

August 2013

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

First Edition (August 2013)

This edition applies to IBM PureApplication System Version 1.1.

© Copyright International Business Machines Corporation 2013. 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

Notices	vii
Trademarks	viii
Preface	ix
Authors	ix
Now you can become a published author, too!	xi
Comments welcome	xii
Stay connected to IBM Redbooks	xii
Part 1. Laying the foundation	1
Chapter 1. Introduction	3
1.1 System overview	5
1.2 PureApplication System as a cloud platform	6
1.2.1 Workload Deployer and PureSystems Manager	7
1.2.2 Virtualization System Manager and PureFlex System Manager	7
1.3 Patterns	7
1.3.1 A brief history	8
1.3.2 Virtual system patterns	8
1.3.3 Virtual application patterns	10
1.3.4 Shared services	11
1.3.5 Classes of pattern models	12
1.3.6 Pattern composition	14
1.3.7 Endpoint resolution	15
1.3.8 Policies	15
Chapter 2. The pattern engine	17
2.1 Pattern transformation	18
2.2 Basic plug-in overview	21
2.3 Transforms	22
2.4 Methods for use in virtual application lifecycle management	27
2.4.1 Maestro module for Python	29
2.4.2 Maestro global parameters	29
2.5 Pattern types	29
Chapter 3. Application pattern models	31
3.1 Virtual application pattern terminology	32
3.1.1 Virtual application	32
3.1.2 Virtual application pattern	32
3.1.3 Virtual application instance	32
3.1.4 Virtual Application Builder	32
3.1.5 Virtual application template	35
3.1.6 Virtual application layer	36
3.1.7 Virtual application pattern type	36
3.1.8 Virtual application pattern plug-in	38
3.2 Virtual application pattern model design	39
3.2.1 Prerequisites	40
3.2.2 Planning your virtual application	40
3.2.3 Identifying component and underlying middleware capabilities	41

3.2.4	Identifying links	42
3.2.5	Identifying policy	42
3.2.6	Classifying pattern type and plug-in	42
Chapter 4.	Plug-in Development Kit	45
4.1	Overview of the Plug-in Development Kit	46
4.2	The Workload Plug-in Development perspective	46
4.2.1	Structure of a Workload Pattern Type project	47
4.2.2	Structure of a Workload Plug-in project	50
4.2.3	Upload the pattern type and plug-in from the PDK	64
4.3	The Workload Plug-in Runtime perspective	65
4.4	Setting up the PDK environment	66
4.5	Create a simple plug-in project with the PDK	67
Part 2.	Creating and implementing an application pattern model	69
Chapter 5.	Case study	71
5.1	A business problem for a life science company	72
5.1.1	Background	72
5.1.2	Solution requirements	72
5.1.3	Spend record processing use case	73
5.2	The spend record processing application	74
5.2.1	Validation and compliance rule application artifact	74
5.2.2	Spend record process application artifact	76
5.2.3	Start validation and compliance rules in process	77
5.3	Composite application pattern for Promotional Spend Compliance	79
5.3.1	Rule application component and transformed topology	82
5.3.2	Process application component and transformed topology	83
5.3.3	Process and rule link transformed topology	85
Chapter 6.	Implementing the model	87
6.1	The Business Rule Application pattern type	88
6.1.1	Creating rule application components	88
6.1.2	Define a transformer	100
6.1.3	Create a usage intent policy	122
6.1.4	Create a scaling policy	130
6.2	The Business Process Application pattern type	135
6.3	The business process and business rule link	136
Chapter 7.	Debugging and testing	149
7.1	The debug and unlock plug-ins	150
7.1.1	Mock deployment	150
7.1.2	Deployment for manual debugging	152
7.2	Troubleshooting the transformer	153
7.3	Using the Workload Plug-in Runtime perspective	154
7.3.1	Upload the plug-in from your Eclipse workspace	155
7.3.2	Create and deploy a virtual application pattern	155
7.3.3	Connect to the deployed pattern	157
7.3.4	Resume from the point of failure	159
7.4	Manually debug the scripts	162
Chapter 8.	Leading practices for plug-in design and implementation	165
8.1	Before you start	166
8.1.1	Basic checklist	166

8.1.2 Advanced checklist	166
8.2 Associating a plug-in with pattern types	166
8.2.1 Primary pattern type	166
8.2.2 Secondary pattern type	167
8.2.3 Linked option	167
8.2.4 Prerequisites	168
8.3 Plug-in design hints and tips	168
8.4 How to manage binary files	169
8.5 Using persistent VMs	169
8.6 Platform consideration	169
8.6.1 Managing supported versus non-supported platforms	170
8.6.2 Organizing files and scripts for multiple platforms	171
8.6.3 Writing platform-neutral and platform-specific scripts	173
8.6.4 Supporting Microsoft Windows	173
8.7 Using versions to accelerate development	173
8.8 Naming convention	174
8.9 How to set an endpoint URL to the pattern	176
Appendix A. Additional material	177
Locating the web material	177
Using the web material	177
Downloading and extracting the web material	178
Related publications	181
IBM Redbooks	181
Other Publications	181
Online resources	181
How to get Redbooks	182
Help from IBM	182

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 grant 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 websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites 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.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

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:

AIX®
CloudBurst®
Cognos®
DB2®
developerWorks®
IBM®
IBM Flex System™

IBM SmartCloud®
POWER®
POWER7+™
PowerVM®
PureApplication™
PureFlex™
PureSystems™

RackSwitch™
Redbooks®
Redbooks (logo) ®
Storwize®
Tivoli®
WebSphere®

The following terms are trademarks of other companies:

Intel, Intel Xeon, Intel logo, Intel Inside logo, and Intel Centrino logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Linux is a trademark of Linus Torvalds 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.

Java, and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

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

Preface

This IBM® Redbooks® publication describes how IBM PureApplication™ System supports the creation of virtual systems and virtual applications. PureApplication System does so using a pattern model that enables you to take advantage of predefined, pre-configured, and proven middleware topologies and deployments.

This book also presents an abstraction level that focuses on functional capabilities and applications, completely encapsulating the underlying middleware. It describes in detail the model and the associated frameworks in PureApplication System, as well as a methodology and approach toward designing and implementing a custom pattern model. This book shows concrete implementation examples that you can use when creating your own pattern model, paired with a collection of leading practices.

This IBM Redbooks publication gives critical guidance to, and serves as a reference for, independent software vendors (ISVs) who want to create patterns for their packaged applications on PureApplication System. Clients who want to extend and enhance their existing patterns can also use this book.

Authors

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



Prashanth Bhat is a Senior Staff Software Engineer in the IBM Software Labs, India. He has been with IBM for the past nine years. He holds a Masters degree in Software Systems from the Birla Institute of Technology, Pilani, India.

He has worked in different roles, including build automation and installation development, on IBM WebSphere® technologies and cloud computing. He is part of the development team for the IBM WebSphere eXtended Transaction Runtime distributed environment.

His most recent project was to enable C and COBOL workloads as virtual applications on IBM PureApplication System using IBM Mixed Language Application Modernization Pattern.



Julian Foster has been working with IBM cloud technologies for over five years, starting with IBM Tivoli® Service Automation Manager and Tivoli Provisioning Manager, IBM WebSphere CloudBurst® Appliance and IBM Workload Deployer, and now PureApplication System and IBM SmartCloud® services.

He has extensive experience in virtualization technologies, and in managing large data centers supporting hundreds of remote users. In addition, he is heavily involved in WebSphere Application Server and virtualization consulting at enterprise-level accounts for the Haddon Hill Group, a premiere IBM Business Partner.

He is certified in Technical Sale Mastery for Publicly Available Specification (PAS), WebSphere 7 Administrator, Tivoli Monitoring v6.2.x, and IBM Systems Director v6.3.x.



Mahmoud Galal is an Executive Information Technology (IT) Specialist at IBM Software Group (SWG) services. He has more than 21 years of experience in IT. He has had several different roles, from developer to IT Specialist and team leader.

His mission is to provide hands-on consulting services to IBM clients, and other IBM organizations, for WebSphere Business Integration products and PureApplication System, mainly in the Middle East and Africa (MEA) region.



Lei He is the chief architect of China Lab Service. She led the composite application pattern design and development for PureApplication System. Throughout her 10 years in IBM, she has built considerable experience in software development, industry solution building, and client engagement.

She has deep knowledge of cloud computing, service-oriented architecture (SOA), Business Process Management (BPM), and cross-brand software integration for industry solutions.



Andre Tost works as a Senior Technical Staff Member in the IBM WebSphere organization, where he works as the Lead Patterns Architect for PureApplication System.

His focus at the moment is on cloud computing platforms and integrated expert systems. Before his current assignment, he spent over 10 years in various consulting, development, and architecture roles in IBM.

He has worked with large IT organizations across the globe on SOA and BPM projects, and has acted as the lead architect for many large IT projects, specifically around the enterprise service bus (ESB) pattern. He started his career at IBM as a C++ and Java developer, and still enjoys developing code.

Andre has coauthored five books on various technical topics, is an IBM developerWorks® Master Author, and speaks at conferences worldwide.



Yue Wang (Eva) is a staff software engineer in IBM Software Group. She has participated in the development of WebSphere Service Registry and Repository.

She has also delivered many solutions in the telecom, banking, healthcare, and education industries, among others. Currently, she is working in cloud computing. She has deep understanding and rich experience in SOA, cloud computing, and industry solutions.

Additional contributors

This project was led by *Margaret Ticknor*, a Redbooks Project Leader in the Raleigh Center. She primarily leads projects about WebSphere products and IBM PureApplication System. Before joining the ITSO, Margaret worked as an IT specialist in Endicott, NY. Margaret attended the Computer Science program at State University of New York at Binghamton.

Thank you to the following people for their contributions to this project:

- ▶ Jose DeJesus
- ▶ Chin Huang
- ▶ Ted Kirby
- ▶ James K. Kochuba
- ▶ Christina Lau
- ▶ Ton Ngo
- ▶ Vanessa Oktem

Thank you to the following people for supporting this project:

- ▶ Shari Deiana, IBM Redbooks IT Support
- ▶ Elise Hines, IBM Redbooks Technical Writer
- ▶ Tamikia Lee, IBM Redbooks Residency Administrator
- ▶ Linda Robinson, IBM Redbooks Graphics Editor
- ▶ Sreya Sarkar, IBM Redbooks Video Editor

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

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

- Send your comments in an email to:

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

Laying the foundation

This part explains the technical platform supporting patterns. It covers the following topics:

- ▶ Chapter 1, “Introduction” on page 3
- ▶ Chapter 2, “The pattern engine” on page 17
- ▶ Chapter 3, “Application pattern models” on page 31



Introduction

IBM PureApplication System represents a significant shift in how information technology (IT) environments are built and maintained. It combines hardware resources, such as compute nodes, network appliances, and storage, with software, including virtualization software, operating systems, middleware, and applications.

The system is part of the IBM PureSystems™ family of products, which combine hardware and software to support a more integrated, pre-configured, and pretested system that enables quicker rollout and more efficient management.

PureApplication System supports private, on-premise cloud environments by offering a built-in *platform as a service (PaaS)* layer that enables you to deploy enterprise applications and the underlying middleware within minutes. You use the PaaS layer to define your topologies and application environments in the form of reusable patterns.

Patterns are abstract models of IT deployments that encapsulate leading practices for installation, configuration, and management of middleware and applications. Patterns can be deployed into PureApplication System repeatedly, avoiding the need to provision these environments individually and manually.

This book describes the support for patterns in PureApplication System. The description is divided into two core parts:

- ▶ Part 1 explains the technical platform supporting patterns, or the *pattern engine*.
- ▶ Part 2 takes you through an example of building your own custom pattern type using development tools that come as part of the system.

After reading this book, you will have a solid understanding of how to create, deploy, and manage instances of patterns that are available for PureApplication System. In addition, you will learn how to create your own model.

Details about how to operate PureApplication System in your data center are included there, and are not described in this book. You can use both books together based on your specific interests.

This chapter provides a high-level introduction to PureApplication System, and an overview of its support for patterns. The following topics are covered in this chapter:

- ▶ System overview
- ▶ PureApplication System as a cloud platform
- ▶ Patterns

1.1 System overview

This section provides an overview of the basic components contained in PureApplication System. This section is deliberately short. You can consult *PureApplication Systems Best Practices*, SG248145, for a more detailed look:

<http://www.redbooks.ibm.com/redbooks/SG248145>

There are three different models of PureApplication System: two W1500 models and one W1700 model. The W1500 models are based on Intel compute nodes, and they come in both a small and a large rack size. The W1700 model is based on IBM POWER® compute nodes.

The small W1500 model can contain either 32 or 64 cores, and both large models come in four different configurations that include 96 - 608 cores. Table 1-1 lists the key configuration options for all models.

Table 1-1 Configuration options for PureApplication System models

	W1500-32 and W1500-64	W1500 96 - 608	W1700 96 - 608
Rack	25U 19 inch, 1.3 meter (m)	42U 19 inch, 2.0 m	42U 19 inch, 2.0 m
Node Chassis	1 IBM Flex System™ Chassis	3 Flex System Chassis	3 Flex System Chassis
Processor	Intel Xeon E5-2670 8 core	Intel Xeon E5-2670 8 core	IBM POWER7+™ 8 core
Compute nodes	2 or 4	6, 12, 24, or 38	3, 6, 12, or 19
CPU cores	32 or 64	96, 192, 384, or 608	96, 192, 384, or 608
Memory	0.5 or 1.0 terabytes (TB) random access memory (RAM)	1.5, 3.1, 6.1, or 9.7 TB RAM	1.5, 3.1, 6.1, or 9.7 TB RAM
Storage nodes	<ul style="list-style-type: none">▶ 1 IBM Storwize® V7000 controller▶ 1 Storwize V7000 expansion	<ul style="list-style-type: none">▶ 2 Storwize V7000 controllers▶ 2 Storwize V7000 expansions	<ul style="list-style-type: none">▶ 2 Storwize V7000 controllers▶ 2 Storwize V7000 expansions
Storage drives	<ul style="list-style-type: none">▶ Six 400-gigabyte (GB) solid-state drives (SSDs)▶ 40 600-GB hard disk drives (HDDs)	<ul style="list-style-type: none">▶ Sixteen 400-GB SSDs▶ Eighty 600-GB HDDs	<ul style="list-style-type: none">▶ Sixteen 400-GB SSDs▶ Eighty 600-GB HDDs
Storage capacity	<ul style="list-style-type: none">▶ 2.4 TB SSD▶ 24.0 TB HDD	<ul style="list-style-type: none">▶ 6.4 TB SSD▶ 48.0 TB HDD	<ul style="list-style-type: none">▶ 6.4 TB SSD▶ 48.0 TB HDD
Management nodes	<ul style="list-style-type: none">▶ 2 PureSystems Managers (PSMs)▶ 2 Virtualization System Managers (VSMs)	<ul style="list-style-type: none">▶ 2 PSMs▶ 2 VSMs	<ul style="list-style-type: none">▶ 2 PSMs▶ 2 IBM PureFlex™ System Managers (FSMs)
Network	2 IBM RackSwitch™ 64-port, 10 gigabit (Gb) Ethernet switches	2 IBM RackSwitch 64-port, 10 Gb Ethernet switches	2 IBM RackSwitch 64-port, 10 Gb Ethernet switches
Power	4 Power Distribution Units (PDUs)	4 PDUs	4 PDUs

All of the models are managed the same way, have the same management consoles, and support the same set of software and patterns.

Figure 1-1 shows an overview of the physical structure of the system, using the PureApplication System W1700-608 model as an example.

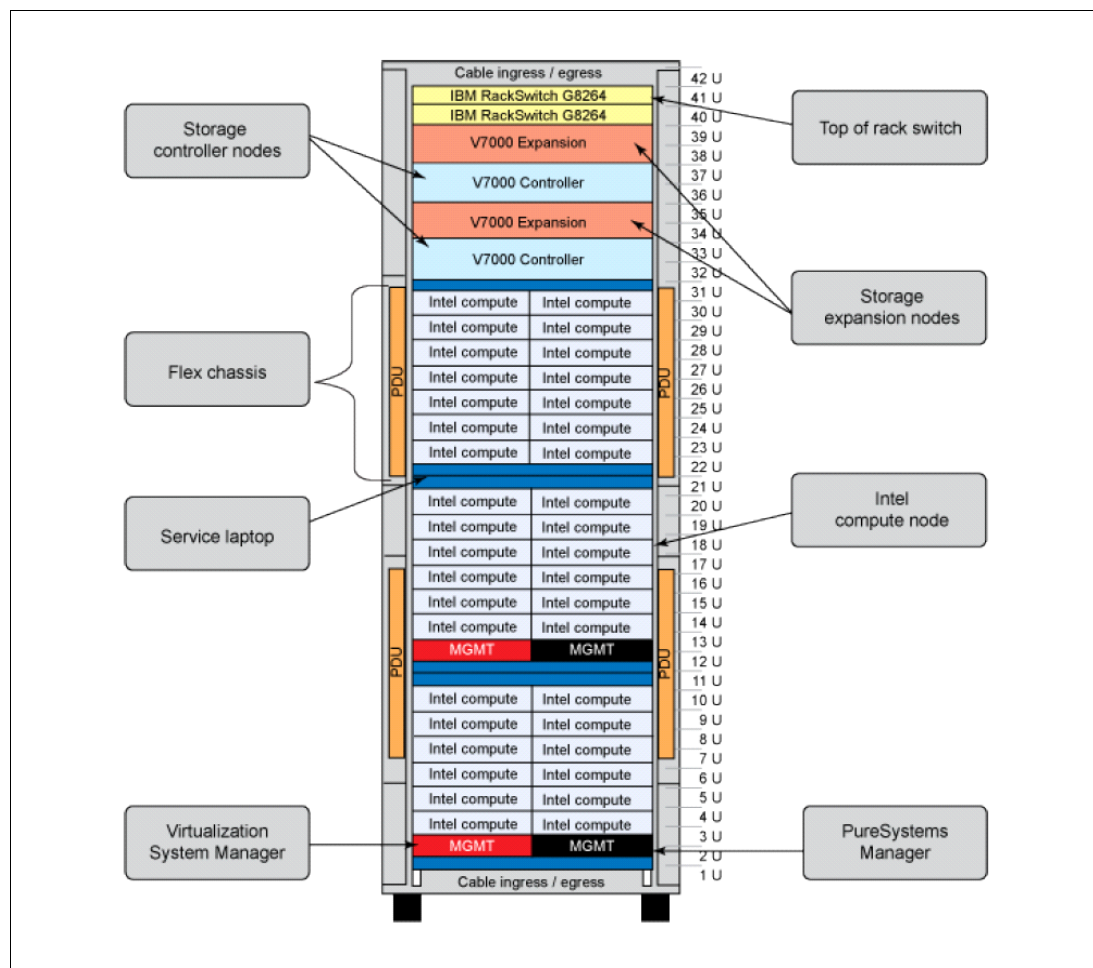


Figure 1-1 Physical structure of the W1700-608 model

For additional information, see the following DeveloperWorks article:

<http://www.ibm.com/developerworks/cloud/library/cl-ps-aim1302-hardwarepureapp/>

1.2 PureApplication System as a cloud platform

A key use case for PureApplication System describes how to build private, on-premise cloud environments. That is, it is a system that you can use to build your own cloud. An important characteristic of cloud computing is to use existing IT resources as efficiently as possible, assigning them dynamically to ever-changing workloads. This is typically achieved through virtualization.

PureApplication System is no different in this respect: you can create and remove virtual machines (VMs) dynamically and automatically, as a result of actual user demand on the solutions running in the cloud. You can also automatically configure these VMs to run middleware and applications.

Therefore, the PureApplication System functionality that enables you to dynamically create, remove, change, and manage virtual environments hosting middleware-based solutions is in direct support of one of the principles of cloud computing.

Cloud environments are often differentiated by their primary delivery models. A delivery model supporting the deployment of virtual infrastructures is called *infrastructure as a service (IaaS)*. PaaS, conversely, is a delivery model that offers a way to create and deploy entire platforms, for example, complex middleware software readily deployed on virtual topologies.

PureApplication System supports both types of cloud delivery models. However, its strength lies in its ability to be used as a PaaS environment. Patterns, which are the subject of this book, provide the key mechanism that you can use to define and deploy platforms consisting of virtual topologies. These topologies include VMs with virtualized storage and networks, appropriate middleware, and even applications.

Management components are essential elements for the support of PureApplication System when it is deployed as a private, on-premise cloud platform. The system comes with two management nodes:

- ▶ In the Intel models, PSMs and VSMs
- ▶ In the POWER model, PSMs and FSMs

1.2.1 Workload Deployer and PureSystems Manager

This component is based on the IBM Workload Deployer product, which is seamlessly embedded into the system. As its name indicates, Workload Deployer offers deployment of workloads. Moreover, it provides you with management functionality for the various runtime pieces.

Primarily, this means managing patterns and all of the deployed pattern instances. The remainder of this book provides more detailed information about Workload Deployer, but for now it is sufficient to say that all aspects of deploying a pattern instance, monitoring it, and managing its lifecycle are handled by Workload Deployer.

1.2.2 Virtualization System Manager and PureFlex System Manager

PSM integrates closely with the other management components, which focus on the hardware, storage, and network resources that are included in PureApplication System and its embedded hypervisors. For example, you can partition the rack into separate groups of compute nodes or IP addresses.

To do that, PSM interfaces with both the hypervisor that runs on the system (VMWare ESX on Intel, and IBM PowerVM® on POWER), and the hypervisor management software (VMWare VCenter for Intel, and IBM Systems Director for POWER). As an administrator, you interact with the hypervisors and other management software exclusively via the PureApplication System console, or via the supported PureApplication System application programming interfaces (APIs). There is no direct access to these underlying systems.

1.3 Patterns

This book is all about patterns, and it covers quite a bit of detail about them. This section only superficially addresses what patterns are, and how the various types of patterns are supported in PureApplication System.

1.3.1 A brief history

Software and architecture patterns are not new. Patterns have been used to describe well-known concepts and ideas, in a reusable way, for quite some time.

One of the first examples of the use of patterns is the now-famous book about software design patterns, *Design Patterns: Elements of Reusable Object-Oriented Software*, by Gamma et al. It put common ways of designing software functions into an easily usable set of patterns. Even though it sounds surprising, until that time, no one had described patterns in a reusable, repeatable way. Today, the pattern language defined by that book has basically become part of software engineering.

Another influential example of patterns is the IBM Patterns for e-business. These patterns described reusable and repeatable ways of building IT solutions for the then-emerging world of e-business. They also defined a common language.

Since then, the IT industry has used patterns in a variety of ways. However, any use of patterns shares the goals of *capturing the things learned about systems* and *making those things repeatable*. These goals led to the emergence of an entirely new discipline within software engineering: *pattern-based engineering*.

The pattern support in PureApplication System is no different. It implements these ideas:

- ▶ It defines pattern languages. These languages in turn represent reusable abstractions of common IT solutions in the context of a cloud.
- ▶ You can deploy these patterns. That is, you can take the abstract description and turn it into an actual system. To be sure, it is a virtual system, not a physical one, as you would expect from a solution targeting a cloud.

Moreover, PureApplication System comes with a set of predefined patterns. These patterns focus on solutions using IBM middleware. They represent years of experience gathered in production environments, called *patterns of expertise*. The idea behind this is that IBM delivers a system that has the built-in knowledge of IBM experts about how to best configure and integrate IBM middleware products.

You can build two styles of patterns in PureApplication System:

- ▶ *Virtual system patterns* (VSPs) describe topologies.
- ▶ *Virtual application patterns* (VAPs) describe an application-centric view of a solution.

Both styles are described separately. Note that this book focuses mainly on application patterns, and the underlying framework for them.

After a pattern is deployed into the system, it is called a *pattern instance*. You can deploy many instances of a particular pattern. For example, a pattern can be deployed for use within a test environment, and simultaneously for production use, within the same physical rack. Patterns support a set of parameters that are specified at deployment time, and that influence details of the instance that is deployed.

1.3.2 Virtual system patterns

Another important use case for PureApplication System describes how to automate the provisioning, installation, and configuration of a specific topology. Topologies often consist of more than one VM. These VMs contain multiple middleware components that have to be configured to work with each other.

For example, a typical setup of WebSphere Application Server includes application server instances, a deployment manager, and one or more HTTP server instances. Often, a database server is also needed.

To dynamically provision these types of environments, simply creating new VMs is not sufficient:

- ▶ Multiple VMs have to collaborate.
- ▶ Software has to be installed.
- ▶ Configuration scripts have to be run.

Most of our clients have developed scripts that automate some aspects of this process. In most cases, however, creating new environments, or removing old ones, is a tedious and time-consuming task. In addition, you must consider the need to scale these environments to address actual usage patterns.

To accomplish this, PureApplication System uses the concept of VSPs. A VSP offers an abstract view of a topology, represented as a visually composed graph, in which each node stands for one VM. For each node, you define the software image that is installed on the VM.

A software image is stored in a standard format called Open Virtualization Archive (OVA). The image encapsulates the operating system and any other software that is running on the VM. Moreover, each node in the VSP can contain script packages that implement further configuration required for the VM to function.

Properties defined for one VM (for example, its IP address) are available to scripts running in another VM, which is useful if multiple VMs have to be integrated. One such instance could be a Java Enterprise Edition (Java EE) data source that is created on an application server, which points to a separate database server.

You can also control the sequence in which the various elements of a VSP are run when deployed. Using the same application server plus database example, the data source in the application server cannot be defined until the database server has been provisioned and installed.

You define all this using drag operations in a graphical tool (as shown in Figure 1-2).

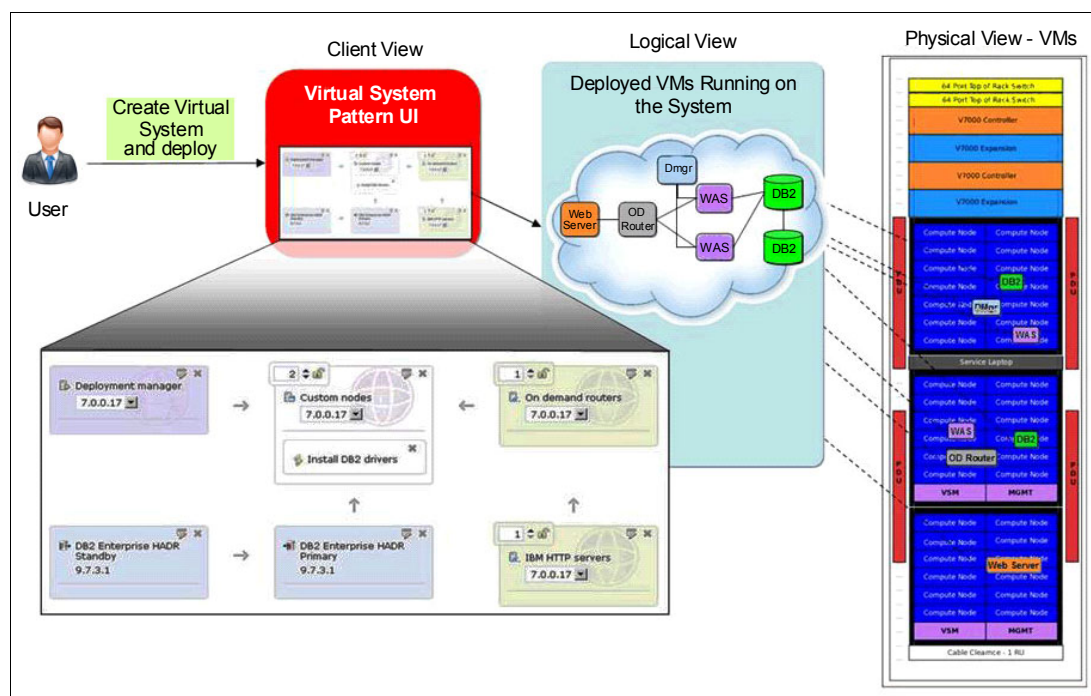


Figure 1-2 Virtual system pattern

A topology described in a VSP is deployed and managed as one unit. With one click (or via the execution of a client script), you can perform all of these functions:

- ▶ Deploy a topology.
- ▶ Provision a set of VMs.
- ▶ Install an operating system and additional middleware on these VMs.
- ▶ Run whatever configuration is required.

PureApplication System has a console that enables you to monitor and manage the resulting VMs. Built-in placement policies make sure that the components of the topology are spread across multiple physical compute nodes.

You can script the administration of the installed middleware. If the administration is not scripted, it is done in the traditional way, using administrative consoles or whatever method of access the middleware provides. In that respect, a virtualized environment deployed as a VSP is no different than a physical environment with actual servers.

1.3.3 Virtual application patterns

A VAP offers a view of the system that is disconnected from the underlying topology. Its model consists of elements that do not directly map to VMs:

- ▶ Components
- ▶ Links
- ▶ Policies

The VAP model is converted into a deployed set of VMs, including all of the middleware and the configuration that is required for it to function, via a series of transformations. Plug-ins, which hold the information about how to turn the abstract VAP model into a concrete topology model, implement these transformations.

In addition to delivering a topology model, a VAP model can also include support for managing the complete lifecycle of the contained artifacts. You can use appropriate scripts and other logic to start and stop all of the resources related to the virtual application, and to apply changes. This support for lifecycle management is an important difference between the current VSP and VAP models.

In that respect, the outcome of deploying a VAP instance appears on a superficial level to be the same as that for a VSP instance: a number of configured VMs, deployed in a cloud group within PureApplication System. The difference lies in the way that the required system is built. A VAP simply offers a more abstract, more coarsely-grained view.

Therefore, the number of components included in a VAP does not necessarily indicate the number of VMs that will be deployed. All components can represent functionality that is combined into one single VM, or one component can lead to the deployment of several different VMs.

Note, however, that a VAP model can be designed in a way that it looks like a VSP, where each component of a pattern type can indeed map to a specific product. In addition, there can be a one-to-one mapping between components and the resulting VMs. That is, the application pattern framework does not stop you from implementing a VAP model that is topology-centric. This chapter describes this concept in more detail.

Each model element can carry parameters that influence how the resulting system is configured. In the associated plug-in, you define which parameters are shown. However, the parameters offer a glimpse into some of the underlying details of the configuration that you want.

The VAP elements, components, links, and policies are grouped in pattern types. Just as with VSPs, PureApplication System comes with a number of predefined pattern types and predefined patterns. However, you can also build your own model by implementing custom plug-ins, using the Plug-in Development Kit (PDK).

The VAP model supports the idea of displaying abstract capabilities, with prescriptive topology configurations in the background, better than the VSP model can. Thus, much of the guidance contained in later sections of this book focuses on the creation of the correct VAP model elements, and how to use the PDK appropriately to implement the associated plug-ins.

This book describes the PDK in detail in Chapter 4, “Plug-in Development Kit” on page 45.

1.3.4 Shared services

PureApplication System supports a third way to deploy and use pre-integrated functionality: through *shared services*. Shared services are virtual applications that are only deployed once per cloud group. The expectation is that a shared service is used by more than one virtual application instance, which increases the level of resource reuse within the system. Examples of shared services that ship with the system are the *caching service* and the *proxy service*, which supports elastic load balancing (ELB).

Instances of shared services are accessible to a consuming VAP. Any VAP model element can contain logic that appropriately configures the resulting topology to interact with the shared service. A VSP can be configured to take advantage of a shared service via a script package.

In addition to using the shared services that come with the system, you can define your own custom shared service. A shared service is a special case of a VAP instance, so the way that it is built is similar to building a VAP (and its underlying model).

The decision about whether to expose a given function as a shared service is influenced by the required level of isolation of separate solutions that run on the system. A high degree of isolation keeps all of the used system resources independent from each other, to prevent any error in one solution from negatively affecting any other solution. The downside of using such a high degree of isolation is less-efficient use of the available resources, which cannot be used across solutions.

1.3.5 Classes of pattern models

The VAP model support in PureApplication System is generic. It does not prescribe how to map a particular product functionality into components, links, and policies, nor does it explicitly favor smaller patterns over larger patterns.

A number of different styles are applied when you build a VAP model. These styles introduce different levels of abstraction, and different degrees of decoupling the model from the underlying topologies. The following sections provide definitions of common styles, and describe what their characteristics are.

Function-centric VAP models

A *function-centric VAP model* contains components (and related links) that are focused on the functional aspects of a solution. They often define the explicit notion of an application (web application, process application, rule application, and so on), and thus encapsulate the business function or purpose of the solution.

At the same time, they do not contain any components that directly represent servers. Instead, the servers required to support the function are implicitly contained in the application components, and will be created and deployed based on the need to support this function.

The policies defined by such a model show details about how a particular function is expected to behave after it is deployed. This often exposes some lower-level technical details, and it can always be at an abstraction level that is within the scope of the business function. An example is the definition of a scaling policy that determines when to add or reduce capacity at run time. Ideally, some of the required policies can be derived from non-functional requirements that have been defined for the business solution.

Integration-centric VAP models

An *integration-centric VAP model* focuses on integration that is required to connect existing functions. Its components encapsulate traditional enterprise service bus (ESB) patterns, such as transformation or routing. For instance, a VAP model based on WebSphere Message Broker can have a Map component, a Route component, and so on.

Note that this does not mean that there are no components that represent business logic. Figure 1-3 on page 13 shows the elements of an integration layer that contains support for exposure of existing functions (or services), and also provides support for new provider composition.

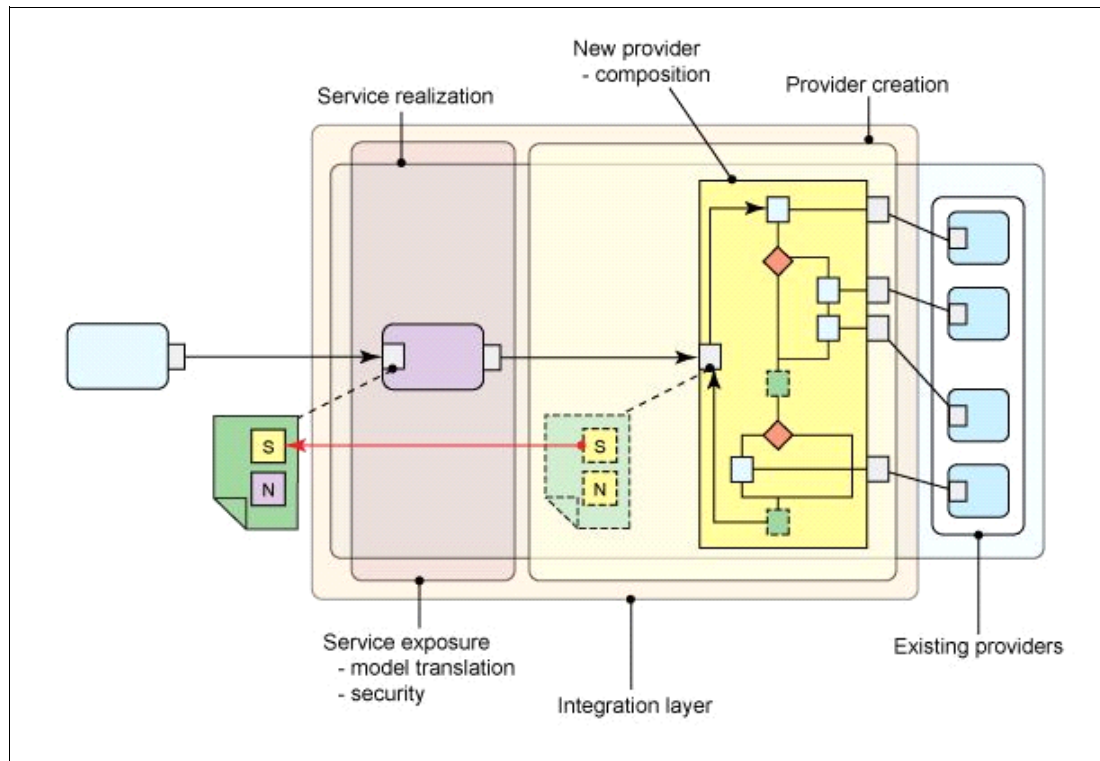


Figure 1-3 The structure of the Integration layer

This composition typically requires business logic. The related capability, if supported by an underlying product, can be shown separately in the VAP model. For example, this can lead to a component type named *service composition*.

Non-functional characteristics of an integration-centric model are often shown by defining components rather than policies, albeit not exclusively. For example, support for a gateway function for certain message encryption requirements can be included as an encryption component (although defining an encryption policy is also possible).

Topology-centric VAP models

Another style of VAP model, called a *topology-centric VAP model*, provides a direct abstraction of an underlying middleware topology. A topology-centric model does not express components related to solution artifacts (for example, *<nnn> application*), but contains components that represent servers and their supported functions.

Such a model is commonly defined when the targeted middleware does not have an explicit notion of an application, or if it supports functionality that cannot be tied to one specific business use case.

Examples for those cases are *Master Data Management* or *Federated Identity Management*. Given that the supported functionality is applied across many distinct business solutions, these models are often candidates for being deployed as shared services.

This style is similar to a VSP, or you can think of a VSP as an example of a topology-centric pattern model.

Tool-centric VAP models

This style of pattern, called a *tool-centric VAP model*, supports the deployment of tools. It applies when one or more tools, possibly pre-configured for use for a specific purpose, are supplied to users in a centrally managed way, that is, via the pattern instance.

The components in this kind of pattern model represent specific tools, with links that implement integration with existing supply chain management (SCM) systems. There is not much usage of policies.

The resulting topology in most cases is rather simple: it results in deployed VMs that either host the tool itself, or act as a server that downloads install images of the tools that can be used locally.

In the first case, this requires that the tool supports being shared across multiple users. For example, the IBM Integration Designer tool can be hosted on a VM. It can be used by multiple users simultaneously logged in, each using their own workspace. Other tools can offer a web-enabled front end.

1.3.6 Pattern composition

To build patterns supporting complex solutions, you often must compose elements from distinct pattern models. Components from different models can be used within the same pattern (assuming that the required links are included in the models), but it can also become necessary to compose completely separate patterns.

One reason for this is if you want to compose elements from different classes of pattern models. For example, you can express the integration between two process applications by explicitly introducing an integration layer.

You should not mix different types of VAP models in one pattern. It leads to patterns that display an inconsistent level of abstraction and detail. Moreover, it does not support a good separation of concerns, where different aspects of the solution cannot be developed and maintained separately.

In other cases, multiple patterns have to be composed even though they stem from the same class, simply because they were developed by different development teams.

Integrating VAP and VSP is another form of pattern composition, namely in cases where a VAP model for a given function or product is not available, but exists as a VSP.

Finally, another example of a case where pattern composition is required is the use of shared services. Even though a shared service is based on a VAP model, it is deployed and managed separately.

PureApplication System offers various approaches toward the composition of patterns. One approach is to take advantage of a VAP model policy element. A policy can be applied to a component in a concrete VAP, and the associated plug-in implementation injects the appropriate configuration into the resulting topology model.

This approach works for composition with other concrete pattern instances, and for integration with a shared service. The information about how to configure the source pattern to interact with the target pattern is encapsulated in the policy details.

The scaling policy, which is included with the system today, is an example where this approach is used. It integrates a VAP with the caching and proxy shared services.

An alternative approach is to define an explicit component within your pattern that represents the link. This approach is similar to the previously described mechanism. The associated plug-in contains *transforms* that inject configuration scripts that manage the integration into the resulting topology.

In some cases, it is necessary to define such reference components on both sides, in the source and the target. Specific configuration steps might be required to prepare the target for serving requests by the consumer.

1.3.7 Endpoint resolution

A major challenge of supporting pattern composition is the requirement that pattern instances can find each other. PureApplication System does not currently offer APIs that enable the automatic retrieval of VMs deployed as part of a pattern deployment for subsequent use by another pattern instance to integrate with that VM.

A way out of this dilemma is to prescribe a specific sequence in which patterns have to be deployed, and to require specification of concrete IP addresses in component parameters.

A more elegant approach is to enter information in the consuming pattern that lets it resolve to a target VM (or a resource running on that target VM) dynamically at run time. PureApplication System offers a number of components that act as placeholders for existing resources (for example, an existing database). Support for dynamic resolution of arbitrary VMs is planned for a future release of the system.

If the target for the composition is a shared service, it becomes a lot easier, because an API to find a shared service is already supported. However, the details of what to do with the service always depend on the specific case, and therefore have to be implemented in the relevant consumer VAP model plug-in.

1.3.8 Policies

The term *policy* is used broadly, and is often applied to different domains, including business and IT. Generally, policies represent decisions, rules, or guidelines that help govern their respective domains. For these domains, there are different ways of expressing policies, and they are often related to, or even derived from, each other. Policies can be structured into domains and layers, to reflect their scope and functional context.

To support policies, you can establish a set of runtime components:

- ▶ The *Policy Administration Point (PAP)* is for authoring and managing policies.
- ▶ The *Policy Enforcement Point (PEP)* is for enforcing policies.
- ▶ The *Policy Monitoring Point (PMP)* is for monitoring policies.

How these components are implemented depends on the architecture of the system in which they operate. For example, an ESB often plays the role of a PEP in a service-oriented architecture (SOA).

PureApplication System has support for policies in its VAP model. The model does not prescribe how policies are to be used, so any aspect that drives the deployment and management of the virtual application can be expressed as a policy.

Policies, which are clearly part of the operational policy layer, can be *configuration policies* or *runtime policies*. Subsequently, they can be applied during deployment time, or during any phase of the lifecycle of the solution.

There is currently no support for an explicit central point for authoring, versioning, or managing a policy in a VAP model. Also, a policy value used in a specific pattern instance cannot be changed through an API. A policy and its value exist within the pattern, and thus get managed with it.

Note also that the only supported scope for a policy is a *component*. A policy cannot be within the scope of a pattern or even a set of patterns.

These functions are planned for a future release of PureApplication System.



The pattern engine

This chapter describes the various components of the *pattern engine*. It provides examples of plug-ins, and of the Maestro Framework elements used in *pattern transformation*. This chapter also provides an in-depth overview of *transformers* and their purposes, and a high-level overview of the steps of a *transform*. The following topics are covered in this chapter:

- ▶ Pattern transformation
- ▶ Basic plug-in overview
- ▶ Transforms
- ▶ Methods for use in virtual application lifecycle management
- ▶ Pattern types

2.1 Pattern transformation

The various components of the Maestro Framework choreograph and metamorphose the pattern from the application model into a topology that the deployment engine can understand. This is used to implement the actual virtual machines (VMs) and middleware components of the virtual application into the cloud. This topology also controls the various options and features (scaling, load balancing, and so on) while running in the cloud.

When using the Virtual Application Builder (VAB) WebUI, the GUI cannot always actually reflect the complexity of what it takes to instantiate and implement the end product. Figure 2-1 shows the JavaScript Object Notation file (appmodel.json) of an IBM Cognos® Business Intelligence Pattern.

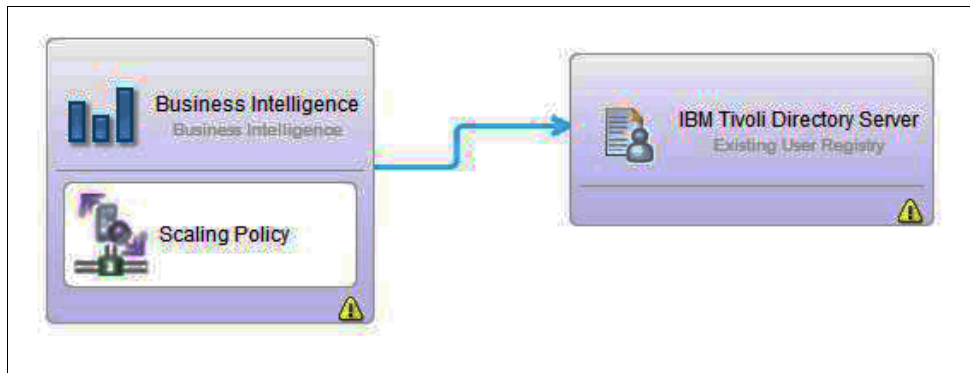


Figure 2-1 Cognos Virtual Application topology as viewed in the VAB WebUI

Example 2-1 is the source view of the appmodel.json file as seen from the VAB. In the VAB, click the Source tab to get this view of your pattern. This code is what the pattern engine transforms into topology data to be employed for the deployment and use of the virtual application lifecycle.

Example 2-1 Cognos Business Intelligence sample appmodel.json

```
{
  "model": {
    "name": "Cognos BI Sample",
    "nodes": [
      {
        "attributes": {
          "DEFAULT_THROUGHPUT_TYPE": "LOW",
          "ADMIN_CREDENTIALS_NAMESPACE": "NONE",
          "ADMIN_CREDENTIALS_USERNAME": "virtuser",
          "ADMIN_CREDENTIALS_PASSWORD": "<xor>Lz4sLCgwLTs=",
          "PROXY_HOST_ALIAS": "SAMPLE",
          "VALID_DOMAINS_AND_OR_HOSTS": "*"
        },
        "id": "Business Intelligence",
        "type": "cognosbi",
        "groups": {
          "DATASOURCE_CONNECTORS": true,
          "DEFAULT": true,
          "ADVANCED": false,
          "DB2": true,
          "Notification": true,

```



```

        "ADMIN_CREDENTIALS": true,
        "Oracle": false,
        "Microsoft": false
    }
},
{
    "attributes": {
        "LDAP_USER_LOOKUP": "(uid=${userID})"
    },
    "id": "IBM Tivoli Directory Server",
    "type": "xTDS",
    "groups": {
    }
},
{
    "attributes": {
        "initialInstanceNumber": 2
    },
    "id": "Scaling Policy",
    "type": "ScalingPolicyofCognosBI",
    "groups": {
        "None": true,
        "Basic": false
    }
}
],
"links": [
    {
        "source": "Business Intelligence",
        "target": "IBM Tivoli Directory Server",
        "attributes": {
        },
        "type": "COGNOSBIxUR",
        "id": "COGNOSBIxUR_1",
        "groups": {
        }
    },
    {
        "source": "Scaling Policy",
        "target": "Business Intelligence",
        "attributes": {
        },
        "type": "policy.ScalingPolicyofCognosBI",
        "id": "policy.ScalingPolicyofCognosBI_1",
        "groups": {
        }
    }
],
"description": "Application template for IBM Business Intelligence Pattern
using a scaling policy and an existing IBM Tivoli Directory Server user registry
(LDAP namespace)",
"app_type": "application",
"locked": false,
"pattern_type": "ibmbi",
"version": "1.0"

```

```

    },
    "layers": [
      {
        "id": "layer",
        "nodes": [
          "Business Intelligence",
          "IBM Tivoli Directory Server",
          "Scaling Policy"
        ]
      }
    ]
  }
}

```

By comparing Figure 2-1 on page 18 and Example 2-1 on page 18, you can see that they actually describe the same pattern from two different views. The VAB view provides a simplified method of interaction, rather than having to edit and manipulate the code in the `appmodel.json` file.

In short, although the pattern looks simple, it is usually not. There are many things that the pattern engine handles to implement a successful deployment and utilization of a virtual application pattern (VAP).

High-level overview of a virtual application transformation

These are the steps you follow to implement a virtual application transformation:

1. Pre-deployment

Before the deployment of a virtual application, the plug-ins to be used in its pattern type must be defined in the `config.json` file contained in the plug-in archive. See 2.2, “Basic plug-in overview” on page 21 for information about the actual structure and layout of a plug-in, and where the `config.json` file is located.

The various middleware packages, hardware, plug-in ID, and the pattern type for the plug-in to be associated with are all defined in this file.

2. Deployment

After you have saved the pattern configuration that you want in the VAB, that data is stored as an application model in the `appmodel.json` file. This file is stored in the Storehouse of the management module of PureApplication System.

The application model is composed of components, links, and policies. This application model goes through several iterations and progressive steps to achieve deployment:

- a. Convert the application model into an unresolved topology. Do not include host names or identifying information, just the generalities such as disk space, what middleware to install, how much RAM and CPU, and so on.

In this step, Component and Link transforms use Java or the Apache Velocity files `<template_name>.vm` to set these values.

- b. Take the unresolved topology and use the definitions in the `config.json` file to help transform that unresolved topology into a resolved topology. The `appmodel` is used to map key components to the parts provided by the plug-ins. The packages (binary files) are defined in the `config.json` file.
- c. After the resolved topology document has been created, the pattern engine uses the resolved topology document, reserves and provisions the required resources, and writes the final topology document to use for deployment of the actual VMs that comprise the virtual application.

- d. Deploy the VMs into a cloud using the final topology document.
- e. Conduct the middleware and software installation and configuration, and then start the software on the deployed VMs. These actions are completed using a set of lifecycle management scripts that are defined for each component, such as the `start.py`, `install.py`, and `configure.py` Python scripts.

Figure 2-2 illustrates the steps described to realize a virtual application from beginning to end.

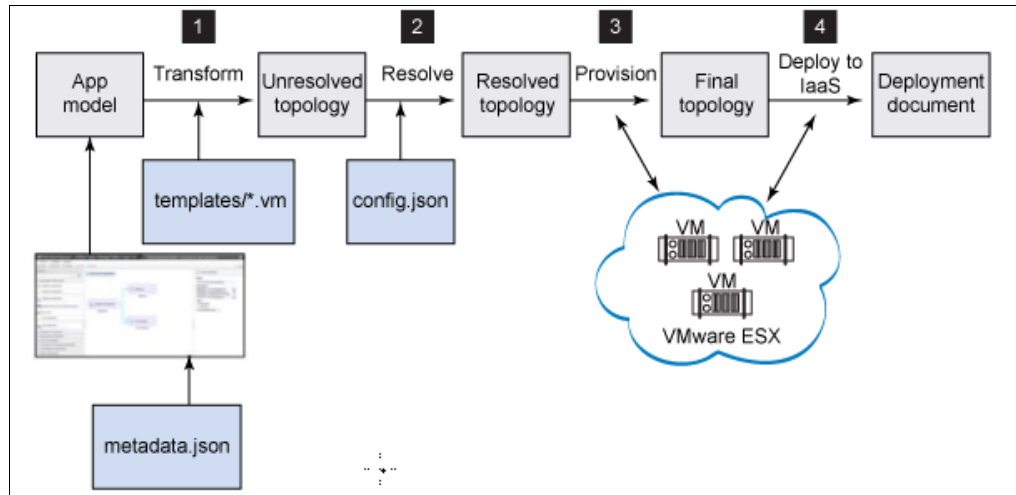


Figure 2-2 Flowchart diagram of the deployment process of a virtual application

2.2 Basic plug-in overview

Plug-ins are specifically structured archives of files that offer components, links, and policies to the pattern engine and operations for later lifecycle management of the virtual application. The pattern engine represents these in the virtual application WebUI as units to be added, deleted, or modified by dragging to create VAPs. The plug-ins define various attributes of the deployment of the actual VMs comprising the pattern and other virtual application lifecycle events.

Here are the various parts that can be included in a plug-in:

<code>parts/<plug-in_id>.tgz</code>	This compressed tape archive (TAR) file (.tgz) describes the artifacts to be installed by the workload agent. These artifacts and files are used to communicate with the PureApplication System console about how the virtual application will be managed on the actual VMs.
<code>nodeparts/<plug-in_id>.tgz</code>	Contains artifacts to be installed by the activation script. The workload agent is itself a node part.
<code>appmodel/tweak.json</code>	Contains code for changing a virtual application in the VAB.
<code>appmodel/metadata.json</code>	Contains the specific links, components, and policies that the plug-in shows to the user in the VAB.
<code>appmodel/operation.json</code>	Contains code for changing a virtual application in the VAB.
<code>bundles/<plug-in_id>.tgz</code>	Specifies the Java archive (.jar) file that contains the transformers, scanners, and provisioners of the plug-in.
<code>config.json</code>	Contains the plug-in configuration file information.

Important: The `config.json` file is the *only* required file for a plug-in. The other files are used as needed.

Plug-ins are covered in more detail in Chapter 4, “Plug-in Development Kit” on page 45. This chapter includes steps for plug-in development, and for virtual application lifecycle scripting.

2.3 Transforms

Transformers are kernel services that convert the virtual application model and its logical descriptions into the topology document that the pattern engine can use for the deployment of the VMs in the virtual application. The actual process is referred to as a *transformation*.

The entire topology document is a JavaScript Object Notation (JSON) object document. The topology fragments are JSON objects as well.

Transforms are multi-part operations, and there are basically two types:

- ▶ Template-based transforms
- ▶ Java implementations

Template-based transforms

The most common transform is implemented as a template of a JSON document. *Dependent objects* are used for links, and *topology fragments* are used for components. Both the PureApplication System W1500 and W1700 series embed Apache Velocity 1.7 as their template engine. The user guide for Apache Velocity is located at the following website:

<http://velocity.apache.org/engine/devel/user-guide.html>

You can also see the Javadoc in the Plug-in Development Kit (PDK), which can be downloaded from the following website:

http://pic.dhe.ibm.com/infocenter/psappsys/v1r0m0/topic/com.ibm.ipas.doc/iwd/pgt_installpdk.html

There is a `javadoc.zip` compressed file inside the `pdk.zip`.

Each plug-in has a specific file structure in which information about the plug-in’s deployment and lifecycle can be recorded and referenced by policies, links, or components. The component document’s name must match the ID of the component, policy, and link, as defined in the `appmodel/metadata.json` file of the plug-in.

Specified component properties in template files use path values relative to the plug-in root, as illustrated in Example 2-2.

Example 2-2 Name value and the relative path values

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0" name="starget">
  <implementation
class="com.ibm.maestro.model.transform.template.TemplateTransformer"/>
  <service>
    <provide interface="com.ibm.maestro.model.transform.TopologyProvider"/>
  </service>
  <property name="component.template" type="String"
```

```

value="templates/starget_component.vm"/>
    <property name="link.template" type="String"
value="templates/starget_link.vm"/>
</src:component>

```

The pattern engine sets the data in an Apache Velocity context to enable the use of variables that can be referenced. Continuing with the `<starget>` example, you can see the use of variables in the `starget_component.vm` file. The `$attributes` are defined in the `appmodel/metadata.json` file of the plug-in, as illustrated in Example 2-3.

Example 2-3 The appmodel/metadata.json file of the plug-in

```

{
  "vm-templates": [
    {
      "scaling": {
        "min": 1,
        "max": 1,
      },
      "name": "${prefix}-starget",
      "roles": [
        {
          'parms': {
            "st1": "$attributes.st1"
          },
          "type": "starget",
          "name": "starget"
        }
      ]
    }
  ]
}

```

Link transformations use a *depends array of the source role* to store the generated JSON data (see Example 2-4).

Example 2-4 The sample usage of a link transformer

```

## Link templates render the depends objects to be added to the source role.

## sourceRole is required to locate the source of the link. Value is the type of
## the source role.
#set( $sourceRole = "ssource" )

## sourcePackages is an optional array. Values in the array are added to the
## packages of the
## vm-template that is hosting the source role.
#set( $sourcePackages = ["pkg2"] )

## Obtain a tuple related to the matching target role:
## target.template == vm-template that holds the target role; target.role == role
## String argument is the type of the target role.
#set( $target = $provider.getMatchedRole($targetFragment, "starget") )

## Validate target. If not found, throw HttpException
#if( $target == $null )

```

```

        $provider.throwHttpException("Target Role starget not found.")
    #end

    [
        {
            "role": "${target.template.name}.${target.role.name}",
            "type": "starget",
            "parms": {
                "s11": "$attributes.s11"
            }
        }
    ]

```

Both the `if_value` and the `if_else_value` (for which there can be a null value) can be used to create conditional statements, as shown in Example 2-5 and in Example 2-6 on page 25.

Example 2-5 Sample of if_value used to create conditional events

```

{
    "vm-templates": [
        {
            "scaling":{
                "min": 1,
                "max": 1
            },
            "name": "${prefix}-sssource",
            "roles": [
                {
                    'parms': {
## Handling optional attributes:
## macro syntax: #macro( if_value $map $key $format_str )
## String value:
                        #if_value( $attributes, "ss_s", '"ss_s": "$value",' )
## Number value:
                        #if_value( $attributes, "ss_n", '"ss_n": $value,' )
## Boolean value:
                        #if_value( $attributes, "ss_b", '"ss_b": $value,' )
## Missing value -- will not render:
                        #if_value( $attributes, "not_defined", '"not_defined":
"$value",' )

## For artifacts, Inlet may send app model with absolute URLs for artifacts;
## other request paths might invoke with relative URLs.
## So use provider.generateArtifactPath(), which invokes URI.resolve() that
handles both cases.

## Handling required attributes; throws an exception if the attribute is
null/empty/not defined
                        "ss_f": "$provider.generateArtifactPath( $applicationUrl,
${attributes.ss_s} )",

## Handling range value (ss3)
                        "ss_r_min": "$attributes.ss_r.get(0)",
                        "ss_r_max": "$attributes.ss_r.get(1)",

## Handling policies: spolicy is defined; not_policy is not

```

```

#set( $spattrs = $provider.getPolicyAttributes($component, "spolicy") )
    #if_value( $spattrs, "sp1", '"sp1": "$value",' )
    #if_value( $spattrs, "not_defined", '"not_defined":
"$value",' )

#set( $npattrs = $provider.getPolicyAttributes($component, "no_policy") )
    #if_value( $npattrs, "np1", '"np1": "$value",' )

## Handling required config parms; throws an exception if the parm is
null/empty/not defined
    "cp1": "$config.cp1"
    },
    "type": "ssource",
    "name": "ssource"
  }
]
}
]
}

```

Example 2-6 Sample of if_else_value

```

"Users" : #if_else_value($attributes, "Users", $attributes.Users.serialize(),
[]),
# Render a formatted string if the mapped value exists and is not empty.
#macro( if_value $map $key $format )

# Render a formatted string if the mapped value exists and is not empty, else a
different string.
#macro( if_else_value $map $key $format_if, $format_else )

```

Lastly, Java static classes can be used in templates, which can be useful if other static methods are used on these classes in the template. Just as in Example 2-6, the `$attributes` are defined in the `appmodel/metadata.json` file of the plug-in, as shown in Example 2-7.

Example 2-7 Java static class used in a template

```

#set( $Math = $provider.getClassForName("java.lang.Math") )
"ss_r_math_max":$Math.max($attributes.ss_r.get(0), $attributes.ss_r.get(1)),

ss_r

```

Java implementations

For cases where templates are not sufficient, Java implementations can be used. Java implementations can generate the JSON documents with the included JSON APIs (`com.ibm.json.java.*`), or by modifying templates. Another preferred option is to use templates, and to enhance them with Java functions.

Template transforms can be enhanced with Java code. Template transforms can be quite useful. If you need Java code, do as much as you can with the template first, then add Java methods that you start from your template, as shown in the following steps:

1. Create your Java class and have it extend `TemplateTransformer`.
2. Add your Java methods. The public methods can be started from your template by using `$provider.myMethod()`. You can pass parameters into the Java methods.

3. Update your Open Services Gateway Initiative (OSGi) component document to set the implementation class to your new Java class, not TemplateTransformer. There are two methods for updating the Apache Velocity context:
 - a. The Apache Velocity context is available in the VelocityContext method. Therefore, you can pass it into a Java method by using \$context.
 - b. Implement the protected VelocityContext createContext(<String applicationUrl>, <JSONObject component>) method. In your method, call super.createContext(<applicationUrl>, <component>). This returns the Apache Velocity context, VelocityContext, to which you can add your custom objects.

The Java JSON APIs can be used to generate the required JSON fragments.
4. Just as in the Template implementations, the ID must match the policy, link, and component that are defined in the appmodel/metadata.json file of the plug-in (see Example 2-8).

Example 2-8 Value of name must match the ID of the links and policies of that plug-in

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0" name="WAR">
  <implementation
class="com.ibm.maestro.model.transform.was.WARTransformer"/>
  <service>
    <provide
interface="com.ibm.maestro.model.transform.TopologyProvider"/>
  </service>
</scr:component>
```

5. To implement component and link transformations by overriding their corresponding methods, Java implementations can use the extension of com.ibm.maestro.model.transform.TopologyProvider, as shown in Example 2-9.

Example 2-9 Link and Component example of Java implementation

```
public JSONObject transformComponent(
    String vmTemplateNamePrefix,
    String applicationUrl,
    JSONObject applicationComponent,
    Transformer transformer)
    throws Exception {
    return new JSONObject();
}

public void transformLink(
    JSONObject sourceFragment,
    JSONObject targetFragment,
    String applicationUrl,
    JSONObject applicationLink,
    Transformer transformer)
    throws Exception {
}
```

6. You can invoke templates by using these methods of the TopologyProvider class, as shown in Example 2-10 on page 27.

Example 2-10 TopologyProvider method invocation

```
public static JSONArtifact renderTemplateToJSON(Bundle b, String template,
String logTag, Context context) throws HttpException;
```

```
public static String renderTemplate(Bundle b, String template, String logTag,
Context context) throws HttpException;
```

7. Example 2-11 illustrates how you can use the WebSphere Application Server in a transform from a template to generate the topology fragment for the server instance.

Example 2-11 WebSphere Application Server TopologyProvider method invocation

```
protected void activate(ComponentContext context){
    _bundle = context.getBundleContext().getBundle();
}

@Override
public JSONObject transformComponent(String prefix, String applicationUrl,
JSONObject component, Transformer transformer) throws Exception {
    JSONObject topology;
    JSONObject scalingPolicy = getPolicy(component, "ScalingPolicyofWAS");
    String vmTemplateName = prefix + "-was";

    if (scalingPolicy == null) {
        VelocityContext context = new VelocityContext();
        context.put(TemplateTransformer.PREFIX, prefix);
        context.put(TemplateTransformer.APPLICATION_URL, applicationUrl); //
Value ends with a slash.
        context.put(TemplateTransformer.COMPONENT, component);

        JSONObject attributes = (JSONObject) component.get("attributes");
        context.put(TemplateTransformer.ATTRIBUTES, new RequiredMap(attributes));
        context.put(TemplateTransformer.CONFIG, new
RequiredMap(getConfigParms()));
        context.put(TemplateTransformer.PROVIDER, this);

        String logTag = "WAS:templates/SingleWAS.vm";
        topology = (JSONObject) renderTemplateToJSON(_bundle,
"templates/SingleWAS.vm", logTag, context);
    }
```

2.4 Methods for use in virtual application lifecycle management

Plug-ins are also used to define other aspects of the deployment and virtual application lifecycle. Storage space can be defined as a template. The template can even be named and referenced later in the topology document, with more specific attributes in a different array (see Example 2-12). The parameters for the size of the storage, the type of file system, and the type of mounting are all described for the additional storage in a vm-template.

Example 2-12 The db2-storage id is assigned to the storage template

```
{
    "vm-templates": [...],
    "storage-templates": [
        {
```

```

        "parms":{
            "size":4,
            "format":"ext3",
            "type":"auto"
        },
        "name":"db2-storage"
    }
]
}

```

Using these extensibility features, you can design plug-ins for specific deployment environments, or use the existing VAPs if you are developing less-specific applications. Example 2-13 defines the specific hard disk drive (HDD) and file system location to which to attach the storage shown in Example 2-12 on page 27. The vm-template is referenced via the storage-ref tag in another array.

Example 2-13 The storage-ref tag references the db2-storage template to define file system attributes

```

"storage":[
    {
        "storage-ref":"db2-storage",
        "device":"\\dev\\sdb",
        "mount-point":"\\home\\db2inst1"
    }
]

```

The pattern engine also provides limited support for OSGi services inside plug-ins. The following specific PureApplication System interfaces can be implemented:

TopologyProvider	Contains process components, links, and policies to create the unresolved topology document.
TopologyProcessor	Updates the unresolved topology document.
RegistryProvider	Provides a shared service. A replacement service for the deprecated PostProvisioner.
ServiceProvisioner	Provisions a resource from an external or shared service, such as the external or shared service client.
PostProvisioner	Looks at the topology document after it has been finalized and stored. The topology document is written once. A deprecated interface, PostProvisioner was replaced by RegistryProvider.
AppBindingService	Scans component artifacts in the appmodel.json file. For example, the type-webapp WebSphere Application Server plug-in scans the application. It returns data, such as the data source used by the application, from a given application, such as enterprise archive (EAR), web archive (WAR), or another application not based on Java Platform, Enterprise Edition (Java EE).

Important: Review the Javadoc shipped with the PDK for exact details of implementing these service interfaces for that specific version of the PDK. Changes might occur from version to version.

2.4.1 Maestro module for Python

There is also a Maestro module that can be imported into Python scripts by including the import statement in the Python script, as shown in Example 2-14. After it is imported, the Maestro package and the tools can be used.

Example 2-14 Import statement

```
import maestro
```

2.4.2 Maestro global parameters

You can also include the virtual applications global parameters used by Maestro for the deployment and lifecycle of the virtual application (see Example 2-15).

Example 2-15 Global parameters

```
role = maestro.role
peers = maestro.peers
node = maestro.node
parms = maestro.parms
role_status = maestro.role_status

def isQuorum( r=role ):
    return r.get('QUORUM') == 'QUORUM'

logger.debug('quorum leader: %s', isQuorum())

if role_status != 'INSTALLING' and role_status != 'CONFIGURING' and role_status !=
'INITIAL' \
    and not isQuorum(role) and not maestro.node['template']['scaling']['max'] ==
1:
    for p in peers:
        if isQuorum(peers[p]):
            # update local SSL key
            break
```

2.5 Pattern types

Pattern types are a collection of related plug-ins consisting of components, links, and policies for deploying virtual applications and the actions governing their lifecycle. Virtual application domains are defined by pattern types.

For example, the WebSphere Community Edition Sample Pattern Type contains plug-ins. These plug-ins are relative to deployments of simple or complicated web or Java EE applications on the WebSphere Community Server. Deployment is dependent on certain other features, such as middleware, being installed.

Different pattern types have different dependencies, and different middleware requirements that are related to those dependencies. The subject of pattern types and their relation to plug-ins, especially with regard to their structure and developmental requirements, is covered in greater detail in later chapters of this book.



Application pattern models

This chapter introduces the concept of virtual application patterns (VAPs), and provides guidance on VAP model design. This chapter includes the following main topics:

- ▶ Virtual application pattern terminology
- ▶ Virtual application pattern model design

3.1 Virtual application pattern terminology

There are a lot of terms in IBM PureApplication System regarding VAPs. Some of these terms are similar, and can cause confusion. This section provides a clear definition and explanation of the terminology.

3.1.1 Virtual application

A *virtual application* is an application that runs on virtual infrastructure. The application software with *Just Enough Operating System (JeOS)* is combined inside a virtual machine (VM) container to maximize the application performance. Virtual applications focus the user on the application requirements versus the virtual images and topology.

3.1.2 Virtual application pattern

IBM PureApplication System provides a generic framework for designing, deploying, and managing virtual applications. A build modeled for a specific virtual application, with all of the application artifacts and quality of service levels, is called a *virtual application pattern*.

This kind of pattern is more application-centric, and usually describes all of the application artifacts and their relationships without showing any underlying middleware details. Users create this kind of pattern to focus more on their application. The framework describes the suitable infrastructure to host the application during deployment.

To see VAP details, follow these steps:

1. In the Workload console of PureApplication System, go to **Pattern** → **Virtual Applications**. All of the VAPs with a certain pattern type are listed in the left pane.
2. Select one pattern, and the generic pattern information, such as description, author, users, and a preview, displays in the pattern overview pane on the right.

3.1.3 Virtual application instance

A deployed virtual application is called a *virtual application instance*. Each virtual application instance is linked to a VAP. One VAP can have multiple instances when it is deployed multiple times.

To see virtual application instance details, follow these steps:

1. In the Workload console of PureApplication System, go to **Instances** → **Virtual Applications**. All of the virtual application instances display in the left pane.
2. Select one instance, and the linked VAP displays in the From Pattern Attribute pane on the right.

3.1.4 Virtual Application Builder

PureApplication System provides a web-based console, called *Virtual Application Builder (VAB)*, for editing VAPs. In the VAB, you can connect pattern components and apply policies.

To edit a virtual application, follow these steps:

1. In the Workload console of PureApplication System, go to **Pattern** → **Virtual Applications**.

2. Select one pattern on the left pane, click **Open** in the toolbar, and the virtual application builder opens. You can also click **New** to create a new VAP to open the VAB.
3. There are three tab views of your VAP in VAB:
 - Diagram tab
 - List View tab
 - Source tab

On the Diagram tab shown in Figure 3-1, all of the available components for the current pattern type are listed by their application categories in the palette on the left. You can drag the component onto the canvas.

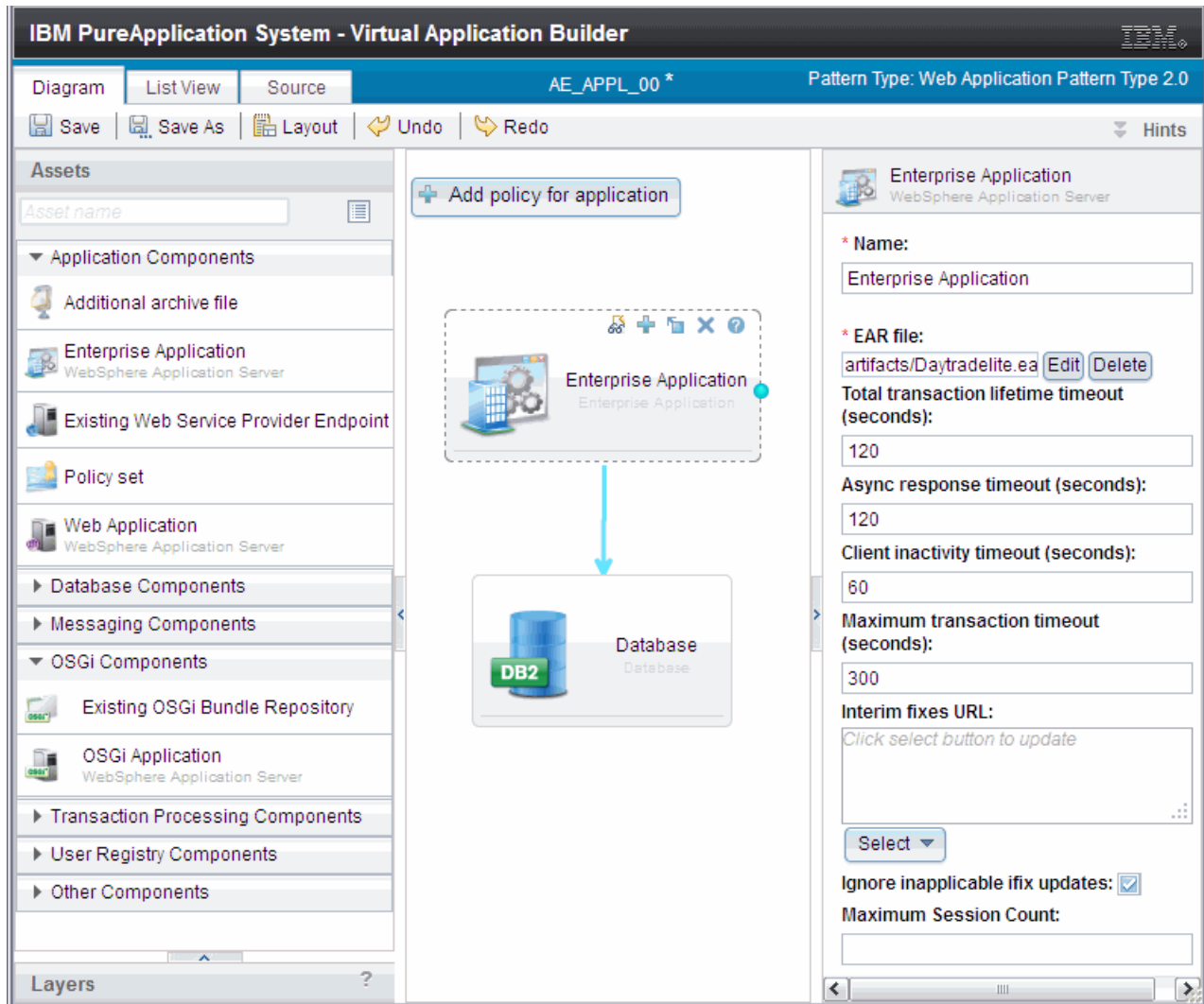


Figure 3-1 VAB Diagram view

4. You can create a link by clicking (and holding) a blue button on a component on the canvas, and dragging it to the other component to be linked. A directed link (a line with an arrowhead) appears on the canvas as you drag.
5. When you click a component, link, or policy, the predefined attributes are shown in the right pane. The available actions for a component are shown on the component itself:
 - a. Add policies to a component.
 - b. Switch to mini view.

- c. Delete the component.
 - d. Get help information about the component.
6. You can add a pattern-level policy by clicking **Add policy for application** in the upper-left corner of the canvas. The policy is added to all applicable components in this pattern. If you later add the same policy with a different attribute value on a single component, that policy will overwrite the pattern-level policy.
 7. On the List View tab shown in Figure 3-2, you can see your VAP and its components, links, and policies in a list. You cannot add or delete any pattern model elements in the List View. The predefined attributes are shown when you expand each item, and you can edit or change the attribute value there. The List View gives you a collected view of all of the pattern model element attributes.

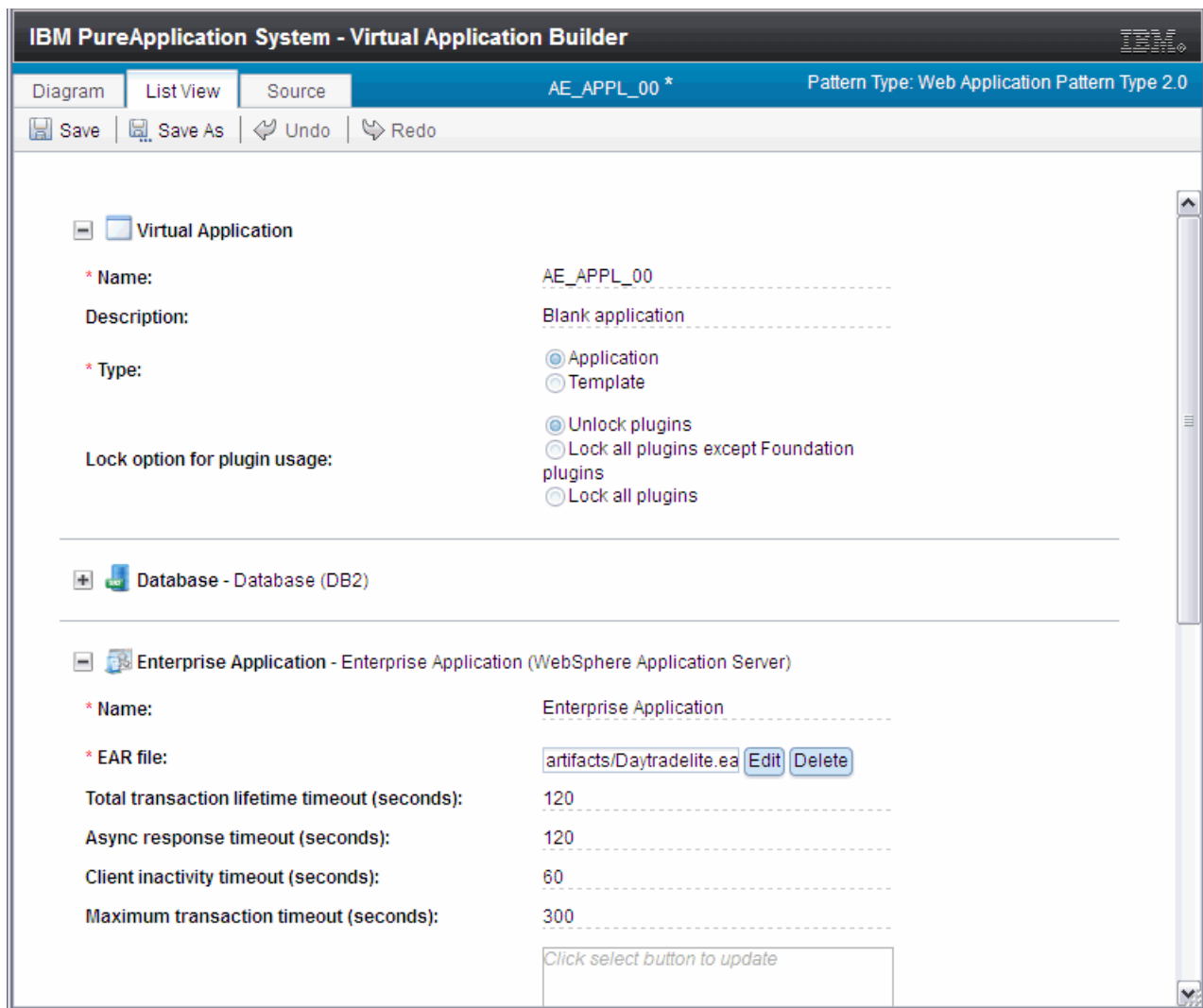


Figure 3-2 VAB List view

- On the Source tab shown in Figure 3-3, you see the source file for your VAP. This is the JavaScript Object Notation file (appmodel1.json) that describes the elements included in this pattern. This file is the input for kernel services to start the transformation, and to generate the final deployment of your pattern. There is no editing action available in this view.

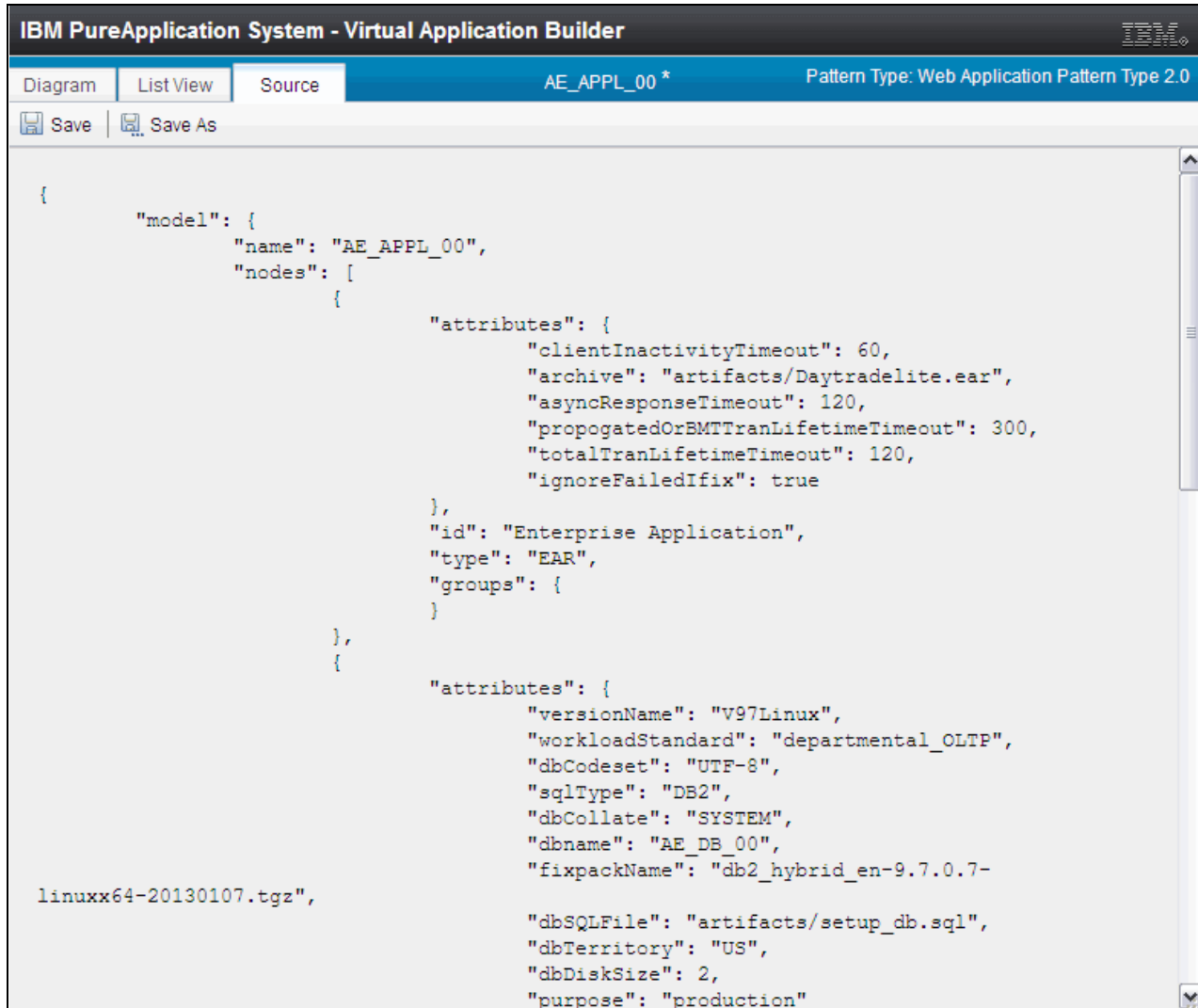


Figure 3-3 VAB Source view

3.1.5 Virtual application template

The *virtual application template* is a set of pre-configured components and links used to simplify and standardize the creation of a VAP. Every time you create a new VAP, you can select a blank application or an existing template to start with.

If you need to repeatedly create patterns and fill attributes, saving a template as a base is useful. If you have Create new catalog content permission, you can perform these actions:

- In the Workload console of PureApplication System, go to **Catalog** → **Virtual Application Templates**. You can see all of the virtual application templates by pattern type.
- You can create, delete, import, export, or edit a virtual application template.

3. In the VAB, after editing an application pattern, you can also save it as a template. Click **Save As** in the toolbar and select the **Save as application template** option.
4. You can deploy a virtual application template directly, and specify any required settings at deployment time.

3.1.6 Virtual application layer

The *virtual application layer* provides a way for you to control complexity, and to reuse virtual applications. By default, a VAP consists of one layer when you first create it. You can set up multiple layers by adding separate layers, or by importing other VAPs as a reference layer.

You cannot change anything for the imported pattern. Anything changed in the original pattern will be reflected in your created pattern.

To work with the virtual application layer, follow these steps:

1. In the VAB, expand **Layers** to view the layers of the VAP, as shown in Figure 3-4.

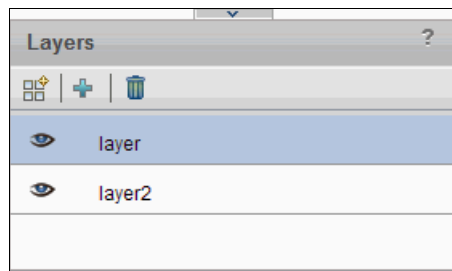


Figure 3-4 Virtual application layer management

2. The virtual application layer fits the requirement of typical development and operation scenarios, in which the developer creates and maintains the original VAP.
3. Testers import the original pattern as a reference layer, and add their layers for their specific non-functional requirements.
4. The functional changes developers make in the original pattern are automatically reflected in the pattern testers' reference layer.

3.1.7 Virtual application pattern type

The *virtual application pattern type* is a collection of plug-ins that defines components, links, and policies. It is grouped with configuration files that are packaged in a compressed tape archive (TAR) file (.tgz), and used to build a set of VAPs. A pattern type defines a virtual application domain.

For example, a *process automation pattern type* defines a domain in which business process applications are deployed. It includes components for process application artifacts. These components have attributes for the appropriate archive file, to be specified during construction of the VAP.

The virtual application pattern type is similar to that shown in 1.3, “Patterns” on page 7. All VAPs, virtual application instances, and virtual application templates are grouped by pattern type. Each of them must have only one pattern type.

The available component for your VAP construction is constrained by the pattern type that you select:

1. In the Workload console of PureApplication System, go to **Cloud** → **Pattern Types**. You can see all of the pattern types installed in the current system, as shown in Figure 3-5.
2. The two-digit version pattern type is a higher-level version for a pattern type, and it can include multiple four-digit version pattern types as its sub-items. When you create your VAP, you must select a two-digit version pattern type. It uses the latest four-digit version pattern type and its plug-ins by default. If you lock all of the plug-ins in that pattern, it will always use the specified four-digit version pattern type and plug-ins.

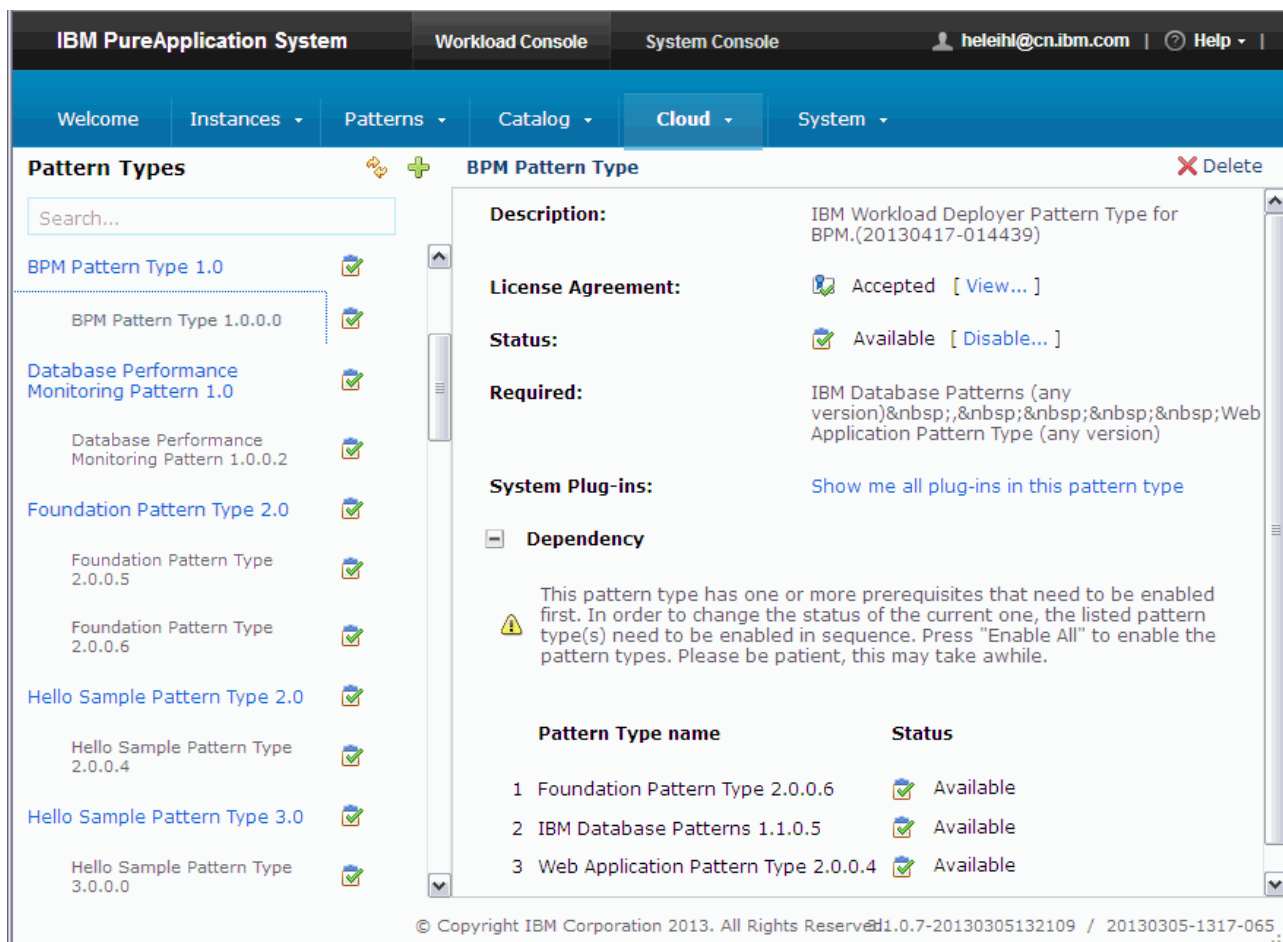


Figure 3-5 Virtual application pattern types

3. Select a four-digit version pattern type in the left pane, and the detailed information for that pattern type displays. You can accept the license and enable this pattern type.
4. All of the prerequisite pattern types are listed in the Dependency section, but if any of those are not installed or enabled, this pattern type cannot be enabled.
5. Click the **Show me all plug-ins in this pattern type** link, and all of the plug-ins under this pattern type are shown.
6. You can install a new pattern type by clicking the plus sign (+) on the toolbar, and then specifying your pattern type .tgz file.
7. You can also delete the selected pattern type by clicking **Delete** on the toolbar.

When you delete a pattern type, all of the plug-ins that claim this pattern type as primary are also deleted. Also notice if the pattern type is in use, which means that you created some VAPs with that pattern type. You might not be able to delete that pattern type unless you install the unlock plug-in.

3.1.8 Virtual application pattern plug-in

A *virtual application pattern plug-in* is a plug-in project that defines the model parts of a VAP, in addition to the underlying implementation that makes the parts deployable in the cloud. Each plug-in project must contain a `config.json` file.

Components, links, and policies are the most user-visible parts a plug-in can contribute, but there are other capabilities that a plug-in includes (for example, the appropriate lifecycle scripts to manage the virtual application through its various lifecycle events after its deployment). Plug-ins are grouped into pattern type by claiming their primary pattern type.

To see all of the installed plug-ins for a specified pattern type in the current system, follow these steps:

1. In the Workload console of PureApplication System, go to **Cloud** → **System Plug-ins**, as shown in Figure 3-6.

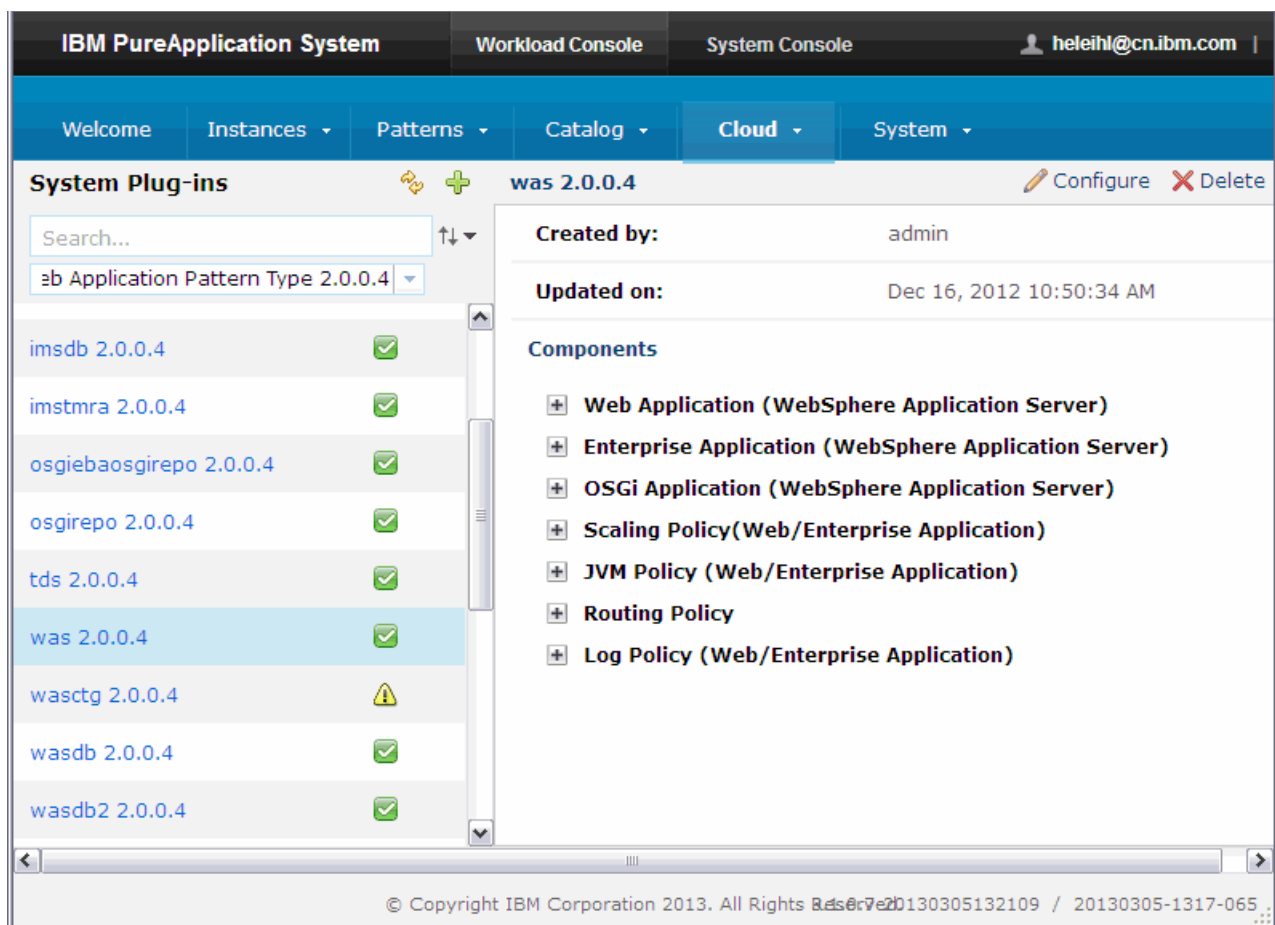


Figure 3-6 Virtual application pattern plug-ins

2. Select a plug-in on the left pane, and all of the components, links, and policies defined in that plug-in are displayed.

3. Expand a component, and you can see its detailed information, including all of the attributes defined in that component.
4. You can install or delete a plug-in by clicking the corresponding button on the toolbar. You can also configure a plug-in if the plug-in developer defined any configurable parameters.

3.2 Virtual application pattern model design

The *virtual application pattern model* is composed of three major elements: components, links, and policies. The implementation of these elements is in plug-in projects, and is grouped into pattern types. Designing the virtual application pattern model is actually the process of identifying components, links, and policies for your virtual application, and organizing them into appropriate plug-in and pattern type projects.

The goal is to keep your model simple, flexible, and reusable. Figure 3-7 shows the relationship between these elements. This is a typical output of your model design.

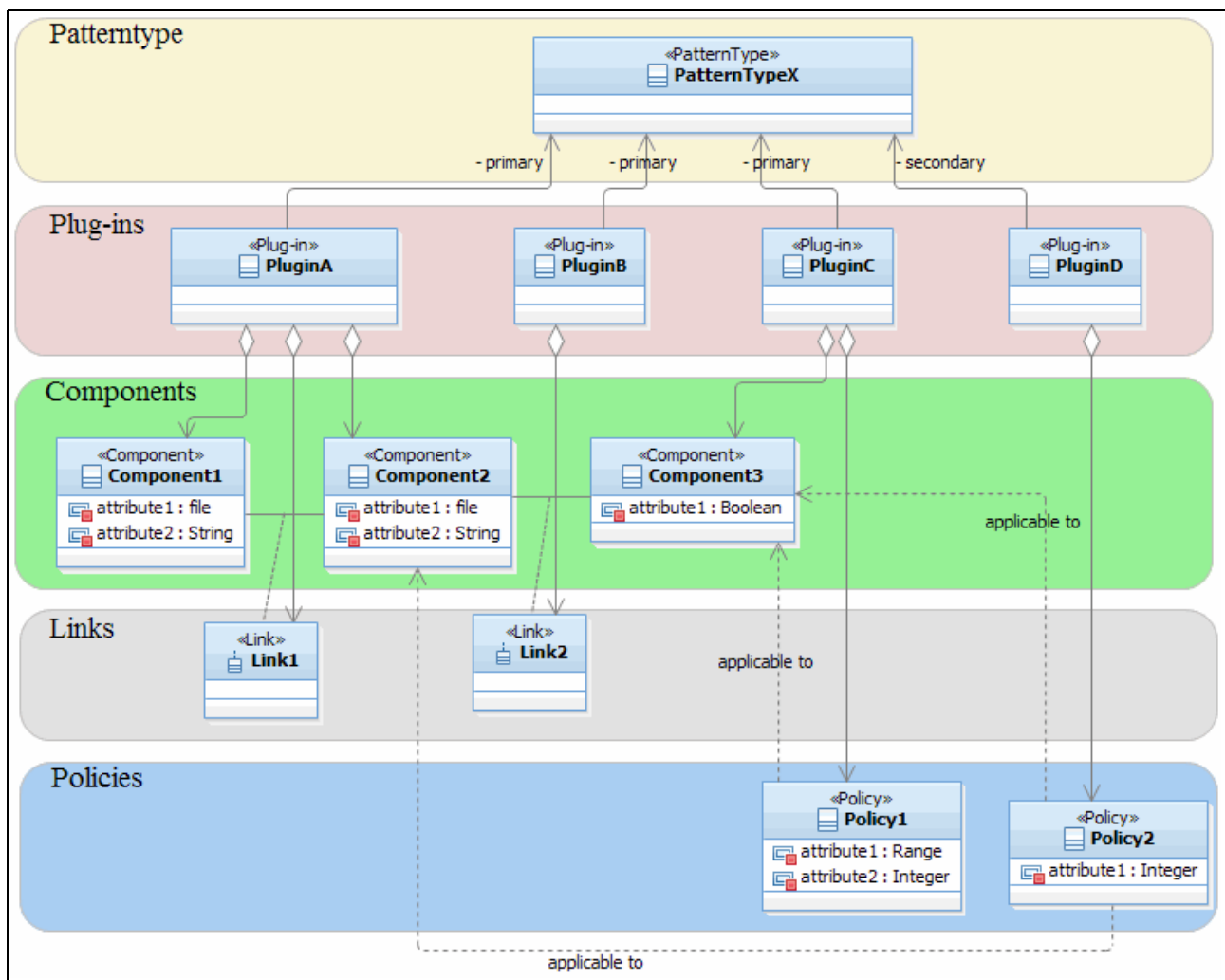


Figure 3-7 Virtual application pattern model overview

3.2.1 Prerequisites

PureApplication System uses the unified default image for the VAP deployment. It supports 64-bit hypervisors and images. For PureApplication System on Power, the default image is IBM OS Image for AIX® Systems V2.0. For PureApplication System on x86, the default image is IBM OS image for Red Hat Linux Systems V2.0.0.1.

You can also build your own Microsoft Windows image on x86 as the default image for the VAP. To build an image, you need to use the IBM Image Construction and Composition Tool (ICCT). ICCT can be downloaded from the Download Tooling toolbar on the Workload console.

Before you design and build your virtual application pattern model, always check whether PureApplication System base OS images provide the capabilities required by your middleware and application.

3.2.2 Planning your virtual application

When you want to model your application as a VAP, you need to consider your application domain. Your application could be any of the following types:

- ▶ A simple web application
- ▶ A web application that needs database access
- ▶ A web application that talks with a business process
- ▶ A mobile application that calls a business process with embedded business rule invocation and analytics reports

You might need high-availability or auto-scaling capability for your application, and these factors affect the design of your virtual application model. Collect the functional and non-functional requirements for your application as early as possible, before you start your virtual application pattern model design.

Keep in mind that your virtual application pattern model design is not just for a single, specific application. When it is designed and implemented, it can fit any application in the same domain. Therefore, take variability into consideration to make your virtual application pattern model flexible enough for reuse.

Each application is composed of one or more application artifacts, which are the deployable pieces of your application. When planning your virtual application, perform the following activities:

- ▶ Identify all of your application artifacts.
- ▶ Identify any underlying middleware and OS requirements.
- ▶ Draw the relationship between artifacts.
- ▶ Describe the necessary quality of service level for your application and artifacts.

You can create a diagram to describe your application details, as shown in Figure 3-8 on page 41.

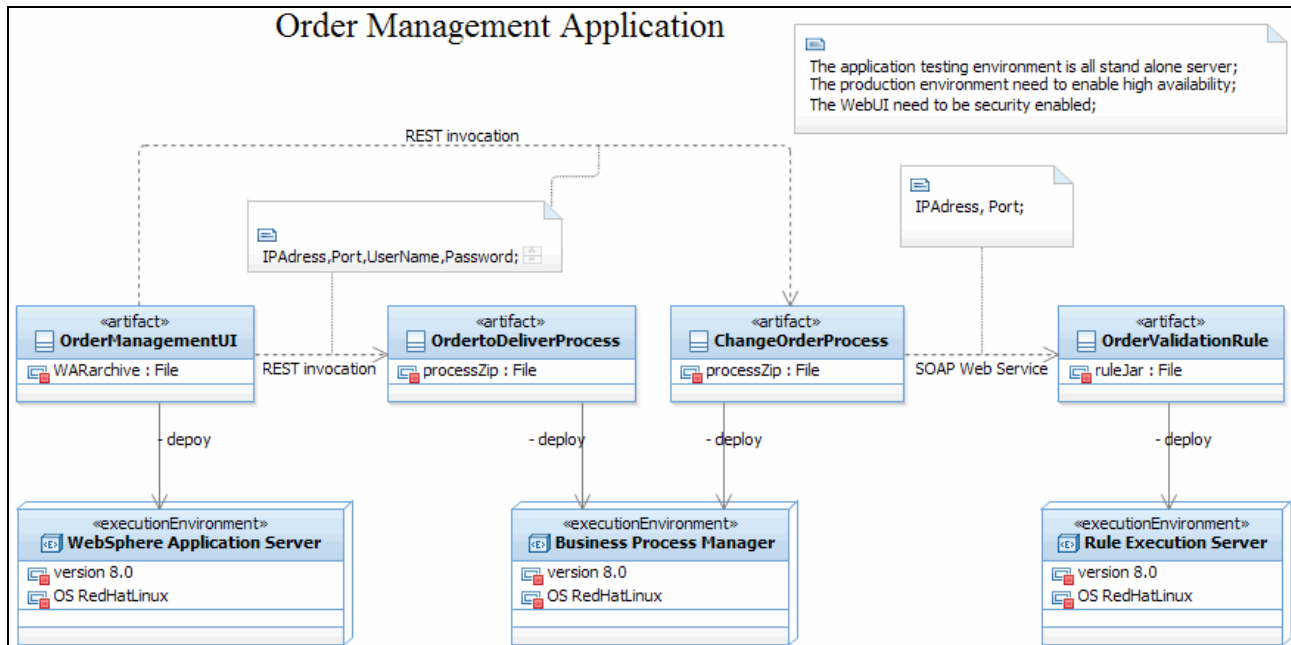


Figure 3-8 Virtual application planning

3.2.3 Identifying component and underlying middleware capabilities

A *component* represents an application artifact, such as a web archive (WAR) file, and its attributes, such as a maximum transaction timeout.

During your virtual application planning, you already identified all of the application artifacts, so you have a list of candidate components. For application artifacts of the same type, such as the two process applications shown in Figure 3-8, use the same component to represent them.

If there is another type of process application with an enterprise archive (.ear) file, think about whether you need to identify another component, or use the same component and handle the file type difference inside your component implementation code.

The implementation of a component in a plug-in project transforms the component into a deployable template that the pattern engine can recognize, and deploy to the cloud. One component can be transformed into multiple VMs, or multiple components can be transformed into one VM.

Usually, the component implementation includes these elements:

- ▶ The definition of operating system (OS) and resource information
- ▶ The installation of middleware
- ▶ The deployment of application artifacts

If there is no specific requirement, do not show the middleware topology at the component level.

3.2.4 Identifying links

A *link* is a connection (or relationship) between two components. For example, if a web application starts a database query, an outgoing link from the web application component to the database component defines this dependency.

The implementation of a link in a plug-in project is comparable to a configuration. The web application-to-database link is the data source configuration, and the process application-to-rule service link is the environment variable configuration.

The link between the process application and the rule service adds the target information, such as IP address and port number, into the source configuration. To use the link, your application implementation needs to follow some standard, so that the configuring can take effect during application run time.

3.2.5 Identifying policy

A *policy* represents a quality of service level for application artifacts in the virtual application. Policies can be applied globally, at the pattern level, or specified for individual components. For example, a logging policy defines logging settings, and a scaling policy defines criteria for dynamically adding or removing resources from the virtual application.

Defining the application non-functional requirements (NFR) as a policy is possible. Usually, a policy cannot change the application artifacts, but it can change the underlying topology of middleware.

3.2.6 Classifying pattern type and plug-in

After you identified components, links, and policies, you need to organize these elements in appropriate *plug-in* and *pattern type* projects. You can put everything in one plug-in and one pattern type, but you can also host each element in a separate plug-in project and a pattern type. Understanding the packaging of pattern type and plug-ins, and their relationship in PureApplication System, will help you make the correct decisions about classifying your model elements.

Plug-in projects claim one primary pattern type, and zero-to-many secondary pattern types, in their `config.json` files. Model elements defined in this plug-in are visible to those pattern types.

As shown in Example 3-1, the `plugin.com.ibm.test` plug-in claims `patternTypeA` as the primary pattern type, and `patternTypeB` and `patternTypeC` as secondary pattern types. When you create a VAP with `patternTypeA`, `patternTypeB`, or `patternTypeC`, you can use all of the components, links, and policies defined in this plug-in.

If you want the model elements in a plug-in visible to all of the pattern types, you can specify asterisk-colon-asterisk ("`*:*`") as the secondary pattern type, with the asterisks enclosed in double quotation marks.

Example 3-1 Plug-in project config.json file

```
{
  "name": "plugin.com.ibm.test",
  "version": "1.0.0.0",
  "patternTypes": {
    "primary": {
```



```

        "patterntypeA": "1.0"
    },
    "secondary": {
        "patterntypeB": "1.0"
        "patterntypeC": "1.0"
    }
},
"packages": {
    "TEST": [
        {
            "parts": [
                {
                    "part": "parts\\test.scripts.tgz"
                }
            ]
        }
    ]
},
"parms": {
},
"files": [
    "\\test\\test.zip"
]
}

```

If you separate your model elements with different plug-in and pattern types, make sure that you have one pattern type that contains all of the elements required for you to create your composite application pattern.

A plug-in project is packaged with its primary pattern type, and all of the files defined in the plug-in are also packaged into the pattern type. As shown in Example 3-1 on page 42, the `test.zip` compressed file is packaged into the `patterntype.tgz` file.

Usually, the files put into `patterntype.tgz` are the software installation binary files, which can be large. Therefore, when you have multiple plug-in projects that claim the same primary pattern type, you also need to think about the size of your pattern type, and the resources required to upload it. For more information about file packaging in plug-in projects, see 8.4, “How to manage binary files” on page 169.



Plug-in Development Kit

This chapter describes how to use the Plug-in Development Kit (PDK) to create custom virtual application patterns (VAPs). This chapter describes the following topics:

- ▶ Overview of the Plug-in Development Kit
- ▶ The Workload Plug-in Development perspective
- ▶ The Workload Plug-in Runtime perspective
- ▶ Setting up the PDK environment
- ▶ Create a simple plug-in project with the PDK

4.1 Overview of the Plug-in Development Kit

The PDK enables you to create custom content that you can add to a VAP in the IBM PureApplication System. You can create your own custom content by creating plug-ins and pattern types. The custom content can be third-party software, or an enhancement to existing software that is already available as a VAP.

The PDK includes an Eclipse plug-in that you can use to develop the pattern types and plug-ins, which you can later use to create your own VAPs.

To use the PDK Eclipse plug-in, you need to have Eclipse version 3.6 or later (32-bit) and Java SE6 (32-bit).

You can find the steps to set up your Eclipse environment with the PDK in the IBM PureApplication System Version 1.0 Information Center:

<http://pic.dhe.ibm.com/infocenter/psappsys/v1r0m0/index.jsp>

On that web page, go to **Working with Virtual Applications** → **Working with virtual application pattern plug-ins** → **Plug-in development guide** → **Plug-ins for development** → **Developing plug-ins in Eclipse**.

As part of the Eclipse plug-in, the PDK provides the following two perspectives:

- ▶ The Workload Plug-in Development perspective, which you use to develop plug-ins.
- ▶ The Workload Plug-in Runtime perspective, which you use to debug and test a deployment by connecting to an ongoing deployment in PureApplication System.

These perspectives are described in detail later in this chapter.

4.2 The Workload Plug-in Development perspective

The Workload Plug-in Development perspective helps you in your development by providing project skeletons for new plug-ins and pattern type projects. With the PDK, you can add new content within your plug-ins, such as parts, node parts, roles, components, links, policies, and so on.

In the Workload Plug-in Development perspective, you can complete the following tasks:

- ▶ Create a new plug-in project or a pattern type project. The project skeleton is created automatically.
- ▶ Import existing plug-in or pattern type projects into your workspace.
- ▶ Visually define components, links, and policies, and define their attributes.
- ▶ Visually define parts, node parts, and roles. The required folder structures are automatically created in the project.

To go to the Workload Plug-in Development perspective, follow these steps:

1. Click **Open Perspective** → **Other** in your Eclipse workspace to launch the Open Perspective window, as shown in Figure 4-1 on page 47.
2. Select **Workload Plug-in Development** and click **OK**.

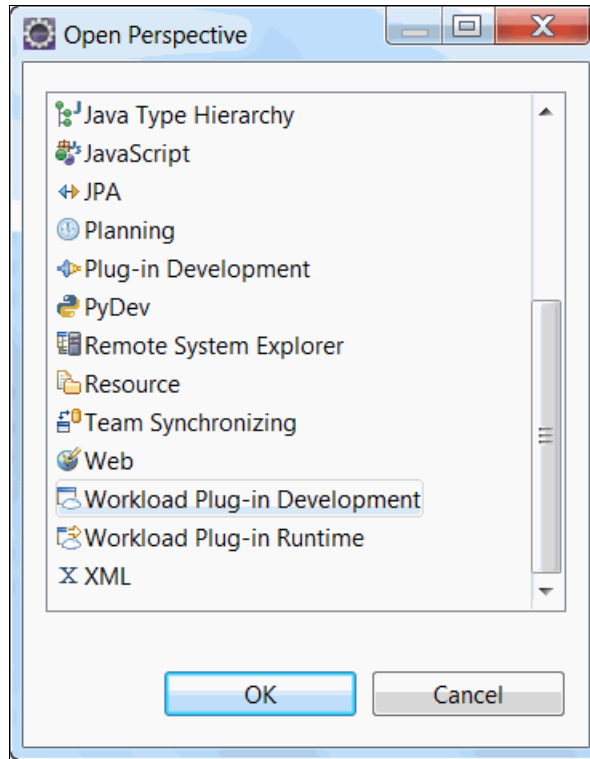


Figure 4-1 The Workload Plug-in Development perspective

4.2.1 Structure of a Workload Pattern Type project

You use a pattern type to create a logical grouping of a set of related plug-ins.

You can create a new pattern type project by clicking **File** → **New** → **Other** → **IBM Workload Pattern Type project**.

Figure 4-2 shows a typical folder structure for a pattern type project.



Figure 4-2 Folder structure for a pattern type project

The main file of interest in a pattern type project is the JavaScript Object Notation file (patterntype.json).

Figure 4-3 shows an overview of a pattern type.

The screenshot shows a web-based configuration interface for a pattern type. The window title is 'patterntype.json'. The main section is titled 'Overview' and contains several expandable panels:

- Pattern Type Basic Information:** Contains input fields for Name (patterntype.simple), Short Name (patterntype.simple), Version (1.0.0.0), Description, Status, and a checked checkbox for Builder.
- Prerequisites:** A table with columns 'Short Name' and 'Version'. It includes 'Add...' and 'Remove' buttons.
- Plug-in Projects:** A table with columns 'Name' and 'Version'. It shows one entry: 'plugin.com.ibm.simple.sa...' with version '1.0.0.0'. It includes a '+' icon and 'Add...' and 'Remove' buttons.
- Categories:** A table with columns 'ID', 'Label', and 'Description'. It shows one entry: 'Applic...' with label 'SimpleApp' and description 'A simple App'. It includes 'Add...' and 'Remove' buttons.

At the bottom, there is a tab bar with 'Overview' and 'patterntype.json'.

Figure 4-3 Overview of a pattern type

Example 4-1 shows the JavaScript Object Notation (JSON) code for the pattern type.

Example 4-1 *patterntype.json*

```
{
  "name": "patterntype.simple",
  "shortname": "patterntype.simple",
  "version": "1.0.0.0",
  "description": "A simple pattern type",
  "status": "",
  "prereqs": {
  },
  "categories": [
    {
      "id": "application",
      "label": "Application",
      "description": ""
    }
  ],
  "builder": true
}
```

You can use the pattern type to define the name, short name, and a version. You can also define the following things in the `patterntype.json` file:

- ▶ **Dependencies:** You can define a dependency on another pattern type using the `prereqs` element.
- ▶ **Categories:** Use the `categories` element to define groupings of components. Later, when you create the component in the plug-in project, you can assign one of the categories defined here.
- ▶ **Builder:** If you set the `builder` element to `true`, your pattern type is made available to the Virtual Application Builder (VAB) to display as a separate pattern type. If you are only extending an existing pattern type, and you do not want your pattern type to be made visible in the VAB, you can set this element to `false`.

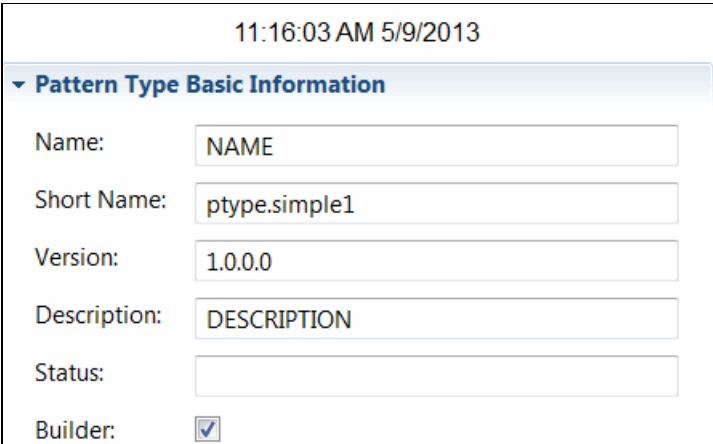
In addition to the `patterntype.json`, you can find the following folders and files in the Pattern Type project:

- ▶ The `locales` folder contains subfolders for each locale that you support.
- ▶ Each subfolder contains a `messages.json` file.

You can insert your translated messages in the `messages.json` file for each locale, and in the `messages.json` file directly under the `locales` folder, which is the default.

For example, you can define the Name and Description. You can name the translated elements of the `patterntype.json` file in the `messages.json` file under it.

The `messages.json` file is a property file with name and value pairs for each translatable variable. For example, Figure 4-4 shows the Name (NAME) and Description (DESCRIPTION), and these can be in translated format in each of the `messages.json` files.



11:16:03 AM 5/9/2013

▼ Pattern Type Basic Information

Name:

Short Name:

Version:

Description:

Status:

Builder: ☒

Figure 4-4 Pattern type with NAME and DESCRIPTION translated

Example 4-2 shows the corresponding `messages.json` file under the `locales/en/` folder. It has the NAME and DESCRIPTION expanded.

Example 4-2 `patterntype/locales/en/messages.json` file

```
{
  "NAME"      : "Simple Pattern Type",
  "DESCRIPTION": "IBM Workload Deployer Pattern Type for a simple component."
}
```

The `licenses` folder contains the license text in HTML format. If you create the `licenses` folder and the content in `<locale>.html` under it, the PureApplication System console requires you to accept the license to enable the pattern type, as shown in Figure 4-5.



patterntype.simple	
Description:	Simple Pattern Type project for demo
License Agreement:	 Accepted [View...]
Status:	 Unavailable [Enable...]
System Plug-ins:	Show me all plug-ins in this pattern type

Figure 4-5 Pattern type with license acceptance enabled

Important: Licenses are always defined for a pattern type. You cannot have separate licensing terms for plug-ins. All of the plug-ins that are defined for a pattern type can be under the same licensing terms.

4.2.2 Structure of a Workload Plug-in project

A plug-in project contains all of the code for the interface elements through components, links, and policies, and the implementation code through packages, parts, and node parts.

To create a new Workload Plug-in project, follow these steps:

1. Click **File** → **New** → **Other** → **IBM Workload Plug-in project**.

The PDK creates a skeleton structure for the plug-in project, one that you can edit and enhance as needed. Figure 4-6 on page 51 shows a typical structure for a Workload Plug-in project.

2. The main content of the Workload Plug-in project is under the `plugin` folder. The `plugin` folder contains the following files and folders:
 - The `config.json` file contains the configuration information of the plug-in project.
 - The `appmodel` folder defines all of the items that are made visible in the VAB. This folder contains the following three files:
 - In the `metadata.json` file, you define the component, links, and policies.
 - The `operation.json` file exposes actions that can be started on a virtual application after deployment.
 - The `tweak.json` file contains configuration attributes that can be changed during virtual application run time.
 - The `parts` folder contains subfolders for creating artifacts for the Python lifecycle scripts.
 - The `templates` folder contains the Apache Velocity template files. You can use the PDK to define Apache Velocity template patterns in the `.vm` format. These files transform your metadata into the Topology document. If you are using Java-based transformations, you do not need these files.
 - The `OSGi` folder contains the Open Services Gateway Initiative (OSGi) Service component in the `.xml` format. Use this component to map the components, links, and policies defined in the `metadata.json` to the transformers (see Figure 4-6 on page 51).

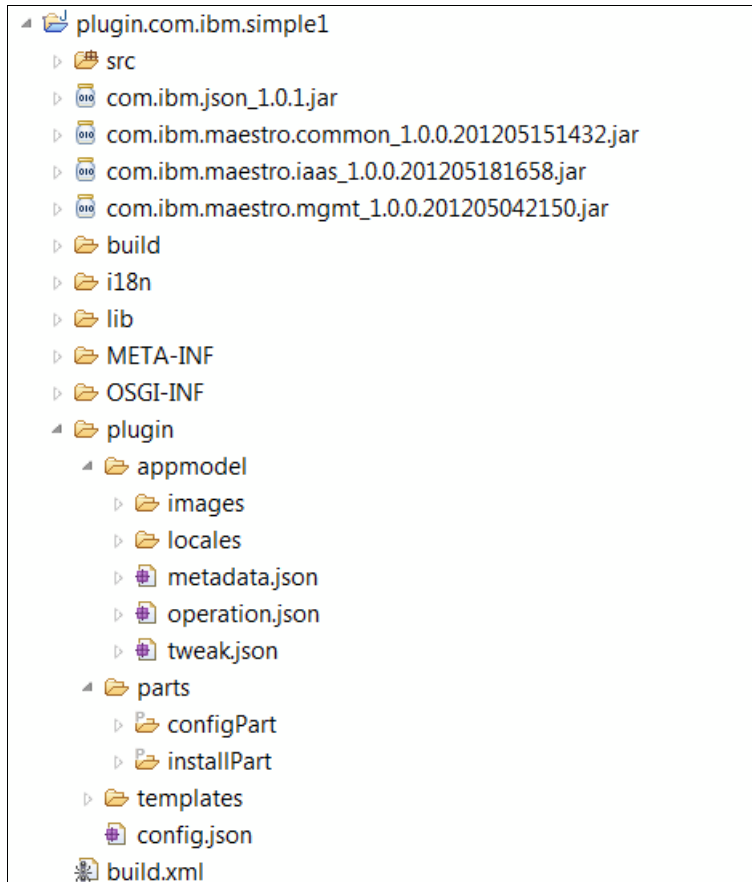


Figure 4-6 Project structure of a Workload Plug-in project

3. Select `config.json` from the Project Explorer window to see a quick overview of the package and metadata information for your project, as shown in Figure 4-7 on page 52. Use this view to define the folder structure required for parts, node parts, or roles.
4. To add a new node, node part, or role, click the plus sign (+) icon on the upper right of the Package pane, as shown in Figure 4-7 on page 52. From the pop-up window, you can select either part, node part, or role.

Important: This action only adds the required directory structure under the `plugin/parts` folder. You have to define the required entries in the `config.json` file separately.

5. The Overview window also has a metadata section that shows the component, links, and policies. You can click **Add**, which takes you to the Application Model view. There, you can define new components, links, or policies.

Figure 4-7 shows the Overview tab of the project.

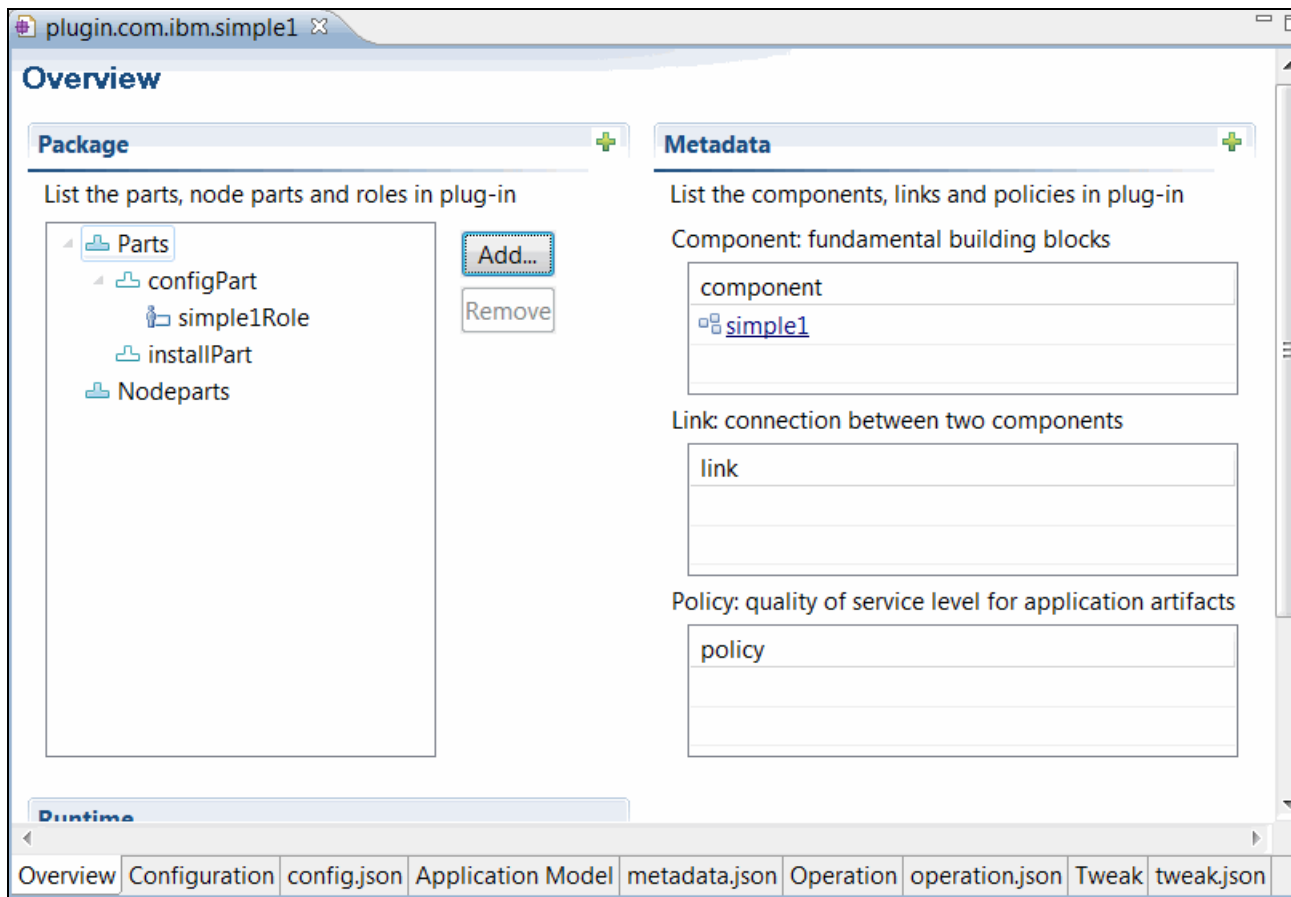


Figure 4-7 Overview of the Workload Plug-in project

Configuration view: The config.json file

To view the configured options in a GUI, you can go to the Configuration view by selecting the Configuration tab in the Overview.

Figure 4-8 on page 53 shows the configuration view. You can use this view to perform the following functions:

- ▶ Modify the basic plug-in information, such as the name and version number.
- ▶ Add secondary pattern types, if you want your component to be visible under other pattern types.
- ▶ Define plug-in files that are specific to the software product, such as configuration files or install files that are to be included with the plug-in. Defining plug-in files creates a new entry as a files element in the config.json file.
- ▶ Create plug-in parameters in which you can define variables that can be used later in the part or role lifecycle scripts.

To perform any of these actions, you can click **Add** next to the respective section and add the required details.

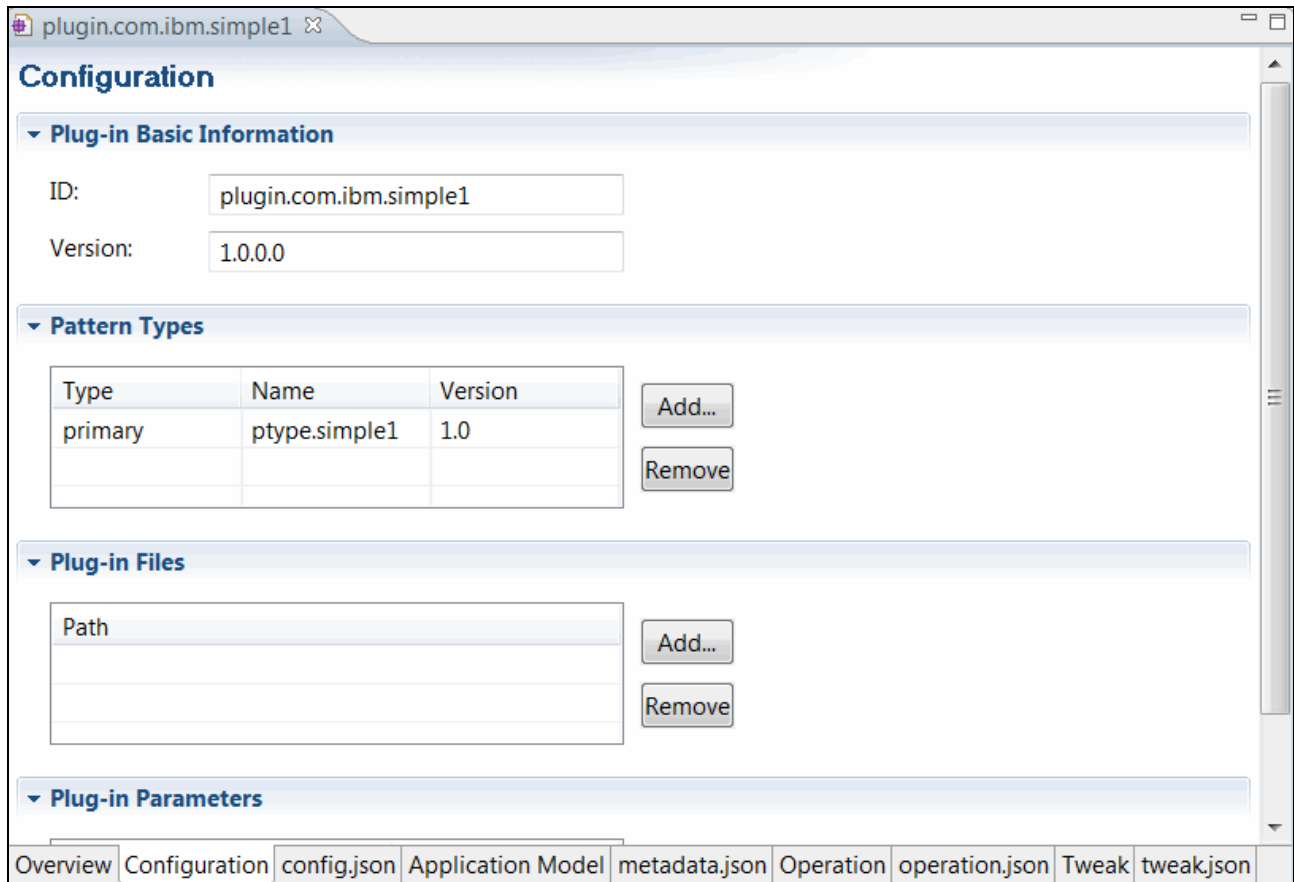


Figure 4-8 The Configuration tab of the plug-in

Example 4-3 shows the `config.json` file corresponding to the Configuration view in Figure 4-8. You can directly add or modify the `config.json` file to change the configuration. For example, the files section of the code corresponds to the Plug-in Files section of the Configuration view, and the `parms` section of the code corresponds to the Plug-in Parameters section of the Configuration view.

Example 4-3 A sample `config.json` file

```
{
  "name": "plugin.com.ibm.simple1",
  "version": "1.0.0.0",
  "patterntypes": {
    "primary": {
      "ptype.simple1": "1.0"
    }
  },
  "packages": {
    "simple1": [
      {
        "parts": [
          {
            "part": "parts\\installPart.tgz",
            "parms": {
              "installDir": "\\opt\\IBM\\sample"
            }
          }
        ]
      }
    ]
  }
}
```

```

    },
    {
      "part": "parts\\configPart.tgz",
      "parms": {
        "installDir": "\\opt\\IBM\\sample"
      }
    }
  ]
}
],
"parms": {
}
}

```

To logically split functionality into meaningful pieces, you can define more than one package in the packages section of the config.json file. Note that you have to edit the config.json file to add the packages. You cannot do this from the Configuration view.

When you define the configuration for your plug-in, you can choose all of the pattern types under which your component is made visible when the VAB is used. You can choose from the following options:

- ▶ *Isolate your components within your own pattern type*, so that only your components are visible when you open the VAB for your pattern type. To do this, define your plug-in with only one primary pattern type.
- ▶ *Extend an existing pattern type to your own pattern type*, so that all of the components of the existing pattern type are also displayed under your pattern type. You need to use the linked element in the config.json file to implement this option. Example 4-4 makes all of the components of the patterntype.other visible when you open the VAB for the patterntype.simple.

Example 4-4 The linked option in config.json

```

"patterntypes": {
  "primary": {
    "patterntype.simple": "1.0"
  },
  "linked": {
    "patterntype.other": "1.0"
  }
}

```

- ▶ *Add your plug-in to an existing pattern type*, so that when a user opens the VAB for the existing pattern type, your components are also available to use. To do this, you need to set the secondary pattern type to the pattern type that you are adding your component to, as shown in Example 4-5. In addition to this, if you do not want your primary pattern type to be available as an option in the VAB, you have to set builder to false in your primary pattern type JSON file, patterntype.json. Adding a secondary pattern makes your components visible under the existing pattern type.

Example 4-5 Add a secondary pattern type in config.json

```

"patterntypes": {
  "primary": {
    "patterntype.simple": "1.0"
  }
}

```

```
    },  
    "secondary": [  
      {  
        "patternType.other": "1.0"  
      }  
    ]  
  }  
}
```

Application Model view: The metadata.json file

In the Application Model view, you create the components, links, and policies. These are made available to the VAB console. To go to the application model, click the **Application Model** tab on the Overview page.

In the Application Model view, you can create new components, links, and policies by clicking **Add** in the Metadata List section. Each of the components, links, or policies can have their own attributes, which you define using the Attributes section.

The Category option determines the category under which your component is visible in the VAB palette.

Figure 4-9 shows the Application Model view of a project. This project has defined a single component called `simple1`. It has one attribute called `name` of type `string`. Based on the Category selection, the `simple1` component is visible under the application section in the palette.

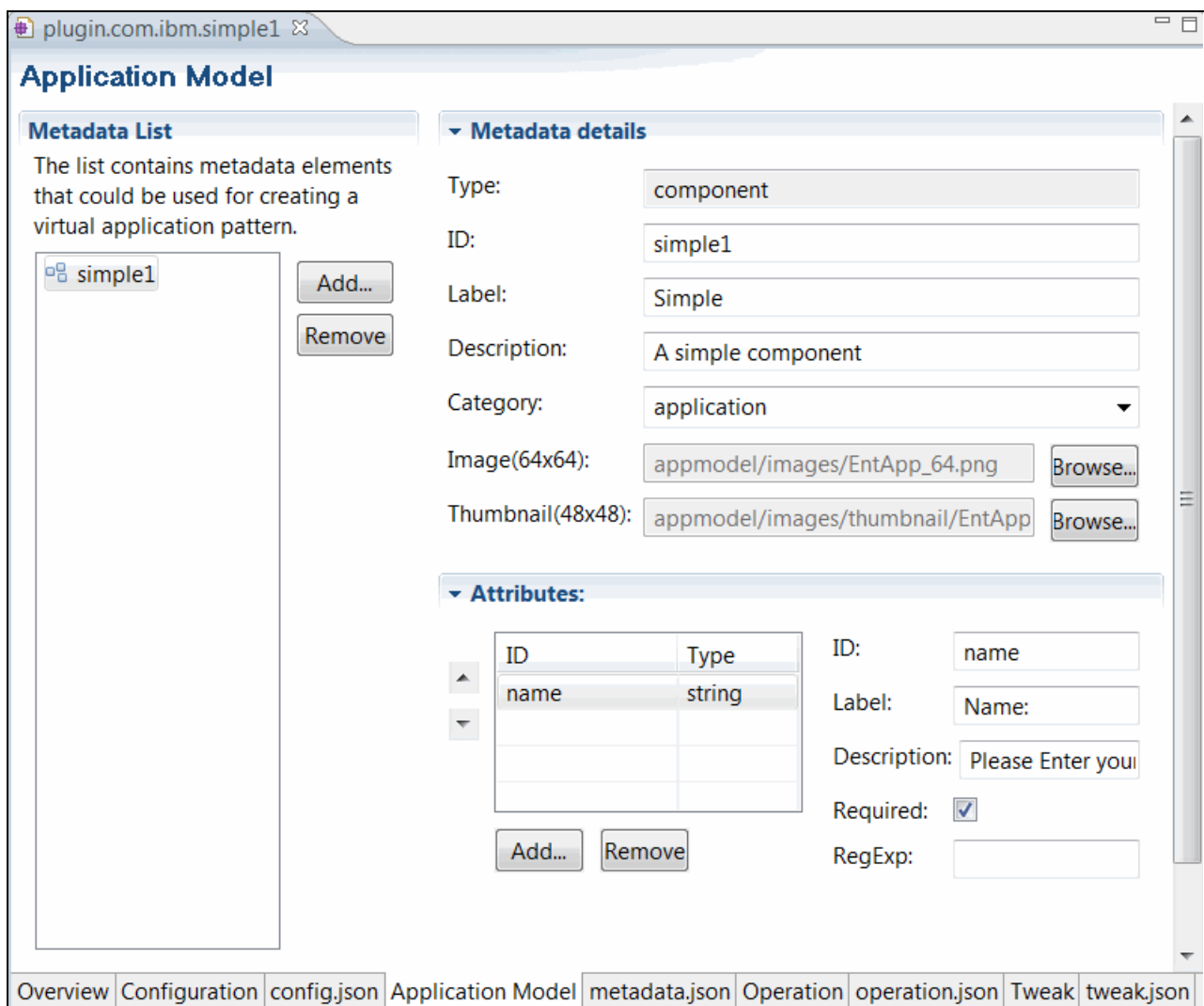


Figure 4-9 The Application Model view

Example 4-6 shows the same application model in the `metadata.json` file. You can view the `metadata.json` file directly, and modify any of the entries that you have already created.

Example 4-6 A sample metadata.json file

```
[
  {
    "id": "simple1",
    "type": "component",
    "image": "appmodel\\images\\EntApp_64.png",
    "thumbnail": "appmodel\\images\\thumbnail\\EntApp_48.png",
    "attributes": [
      {
        "id": "name",
```

```

        "type": "string",
        "required": true,
        "label": "Name:",
        "description": "Please Enter your name"
    }
],
"label": "Simple",
"description": "A simple component",
"category": "application"
}
]

```

Operation view: The operation.json file

Use the Operation tab to add entries to the list in the operation.json file. The operation.json file contains a list of operations that can be called during the virtual application run time. With this option, you can define new operations that can be started on a deployed virtual application instance. For instance, a maintenance operation, such as restarting the application run time, can be defined as an operation.

The operations defined here get displayed in the Operation tab of the Virtual Application Console in the PureApplication System GUI. To see these operations, follow these steps:

1. Click **Manage** on a deployed instance. A new window opens.
2. Click **Operations** on the menu to take you to the Operations tab.
3. You can then navigate to your role to see the operations that you defined. Figure 4-10 shows an Operation view.
4. Add operations by clicking **Add** and supplying details, such as the Role where the operation has to be defined.

Operation

Operation List
The list contains operations that could be invoked during virtual application runtime.

- WASCE.RESTART
- WASCE.configuration

Operation Form

ID: RESTART

Label: Restart WAS CE instance

Description: Run the shutdown and startup command against WAS CE

Category:

Script: operation.py restart

Attributes:

ID	Type

ID:

Label:

Description:

Required: ☐

Overview Configuration config.json Application Model metadata.json Operation operation.json Tweak tweak.json

Figure 4-10 The Operation view

Example 4-7 shows the equivalent `operation.json` file. The JSON entry script defines the script to be run whenever the operation is started through the Operations tab in the Virtual Application Console at run time.

Example 4-7 A sample `operation.json` entry

```
{
  "WASCE": [
    {
      "id": "RESTART",
      "label": "Restart WAS CE instance",
      "script": "operation.py restart",
      "description": "Run the shutdown and startup command against WAS CE"
    },
    {
      "id": "configuration",
      "label": "Update Configuration",
      "description": "Update configuration of WAS CE",
      "script": "change.py"
    }
  ]
}
```

Tweak view: The `tweak.json` file

On the Tweak tab, you can add entries into the `tweak.json` file. The `tweak.json` file contains a list of configuration attributes that can be changed during the virtual application run time. Use this option to define which configuration attributes (as defined in the `metadata.json` file) can be modified in a deployed virtual application instance.

The entries mentioned in the `tweak.json` file get displayed on the Operations tab of the Virtual Application Console at run time.

Figure 4-11 on page 59 shows a simple Tweak entry. You can use the Tweak view to add new entries into the `tweak.json` file. You need to specify the details, such as the ID and the Role where this action has to be defined. There is also the option of referring the action to an existing application model attribute that is defined in the `metadata.json` file. You can do this by using the Reference option, and specifying the Reference type, Reference Target, and Reference ID.

Tweak

Configuration List

The list contains configuration attributes that could be changed during virtual application runtime.

WASCE.ARCHIVE

Add

Remove

Configuration Details

ID:

ARCHIVE

Role:

WASCE

Label:

Update WAR package

Description:

Upload a new WAR package to replace the c

Type:

file

Reference:

☐

Reference type:

Reference Target:

Reference ID:

Overview

Configuration

config.json

Application Model

metadata.json

Operation

operation.json

Tweak

tweak.json

Figure 4-11 The Tweak view

Example 4-8 shows the equivalent `tweak.json` file. This entry can result in the archive being modifiable through the Configuration section of the Operations tab of the Virtual Application Console at run time.

Example 4-8 A sample `tweak.json` entry

```
[
  {
    "id": "WASCE.ARCHIVE",
    "ref-component": "WARCE",
    "label": "WAR File",
    "description": "Specifies the web application (*.war) to be uploaded.",
    "ref-id": "archive"
  }
]
```

Parts

The parts in a plug-in project define a folder structure for scripts for the parts, roles, and dependencies.

Figure 4-12 shows a sample structure of the parts. If you are in the Workload Plug-in Development perspective, you can see these components:

- ▶ Parts are displayed as a folder icon with the letter P.
- ▶ Roles are displayed as a folder icon with the letter R.
- ▶ Dependencies are displayed as a folder icon with the letter D.
- ▶ All of the lifecycle Python scripts are displayed as a sheet icon with an asterisk (*).

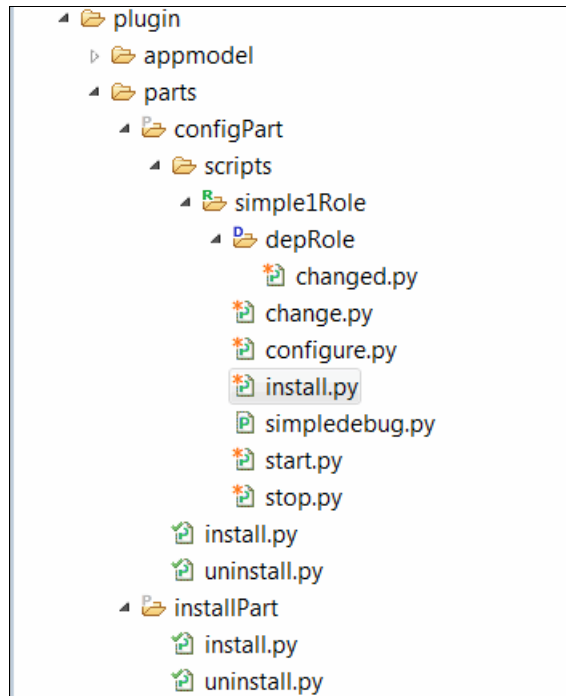


Figure 4-12 Parts folder structure in the Project Explorer

A part can contain the `install.py` and `uninstall.py` Python lifecycle scripts. These scripts are normally used for saving the software binary files, and for setting up for the installation or configuration of the software in later steps.

Lifecycle scripts

The actual software installation and the configuration for the virtual application are created in the lifecycle scripts defined under the roles and dependencies.

A role can have the following lifecycle scripts:

- ▶ The `install.py` script is a lifecycle script that contains all of the install code, which is started during the initial deployment.
- ▶ The `configure.py` script contains the configuration of the installed software, which is started during the initial deployment after `install.py`.
- ▶ The `start.py` script contains commands for starting the instance that is now configured, and is started during the initial deployment after `start.py`.
- ▶ The `stop.py` script contains commands for stopping the instance.
- ▶ The `changed.py` script is a lifecycle script that is started by the pattern engine when there is a change in the state of the peer role. Implement this script if you need to run some actions whenever a peer role changes its state.

- ▶ The `suspend.py` script runs during an upgrade of a pattern at run time. This lifecycle script is used to suspend the role, and is started by the pattern engine when the virtual machine (VM) is shutting down.
- ▶ The `checkpoint.py` script runs during the upgrade of a pattern at run time. The plug-in developer uses this lifecycle script to do a checkpoint during upgrade, and a backup of the role and related data can be taken. This is applicable for a scenario where a restore is required, either due to an upgrade failure, or when a rollback is initiated.
- ▶ The `resume.py` script runs during an upgrade of a pattern at run time. This lifecycle script is meant to bring the role back to a `RUNNING` state, which is started by the pattern engine when the stopped VM is rebooted again.

A dependency can have the same set of lifecycle scripts. The difference here is the order in which these lifecycle scripts are started.

Deployment

During deployment, the order of execution is shown in the following steps:

1. For each part defined in the VM template, run the `install.py` file of the part.
2. For each role defined, run the lifecycle scripts in the following order:
 - a. `<Role>/install.py`
 - b. `<Role>/<dependency>/install.py` (for each dependency)
 - c. `<Role>/configure.py`
 - d. `<Role>/<dependency>/configure.py` (for each dependency)
 - e. `<Role>/start.py`

Note that, although each of the scripts in a role runs serially, the roles are initiated concurrently.

You can, however, configure the script to run the scripts in a role concurrently, with specific restrictions (`maestro.*` variables are read-only).

Lifecycle events

The following lifecycle scripts are started based on certain events occurring during the lifecycle of the deployed virtual application instance. These scripts let you control the order of execution:

- ▶ The `changed.py` script under the role reacts to any changes in peer roles. You can use this script to implement actions that can occur when a peer role changes. For instance, a workload manager can take the necessary action when a new instance of the application server comes up.
- ▶ The `changed.py` script under the `<role>/<dependency>` folder reacts to any changes to the role dependency. You might need to wait for a different or dependent role (possibly in a different VM instance) to finish starting before your configuration takes place. In that case, you can create a dependency on that role in your transformer, and create the `<role>/<dependency>/changed.py` script to have those dependent commands.

OSGi folder

The OSGi service component is located in the OSGi folder. Create a new OSGi service component:

1. Click **File** → **New** → **Other** → **OSGi Service Component**. A new window opens.
2. Enter the name. The name has to match with the component, link, or policy for which the service is meant.

3. You have the choice to select either the template-based transformer or the Java-based transformer.
4. Figure 4-13 shows the creation of a template-based transformer. Specify the Component vm-template file and the Link vm-template file to be created.

Skeleton files for the component and link are created. These files are created in the `templates` folder that is explained in a later section in this chapter.

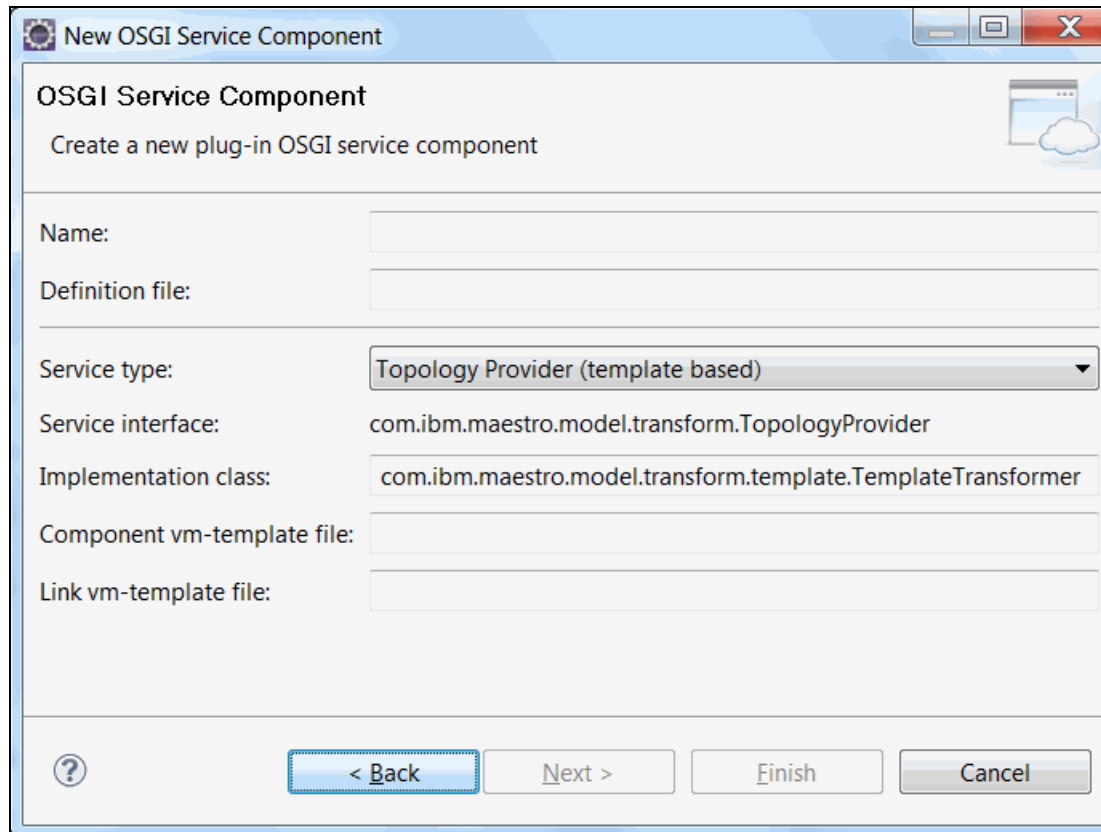


Figure 4-13 Create a new OSGi service component for template-based transformer

Figure 4-14 on page 63 shows the window for creating a new OSGi component for a Java-based transformer. You need to specify the class that you want to create as part of the Java implementation. The PDK creates the `.java` file as an extension to the `TopologyProvider` class. You have to implement one or both of the `transformComponent` and the `transformLink` methods, as required by your project.

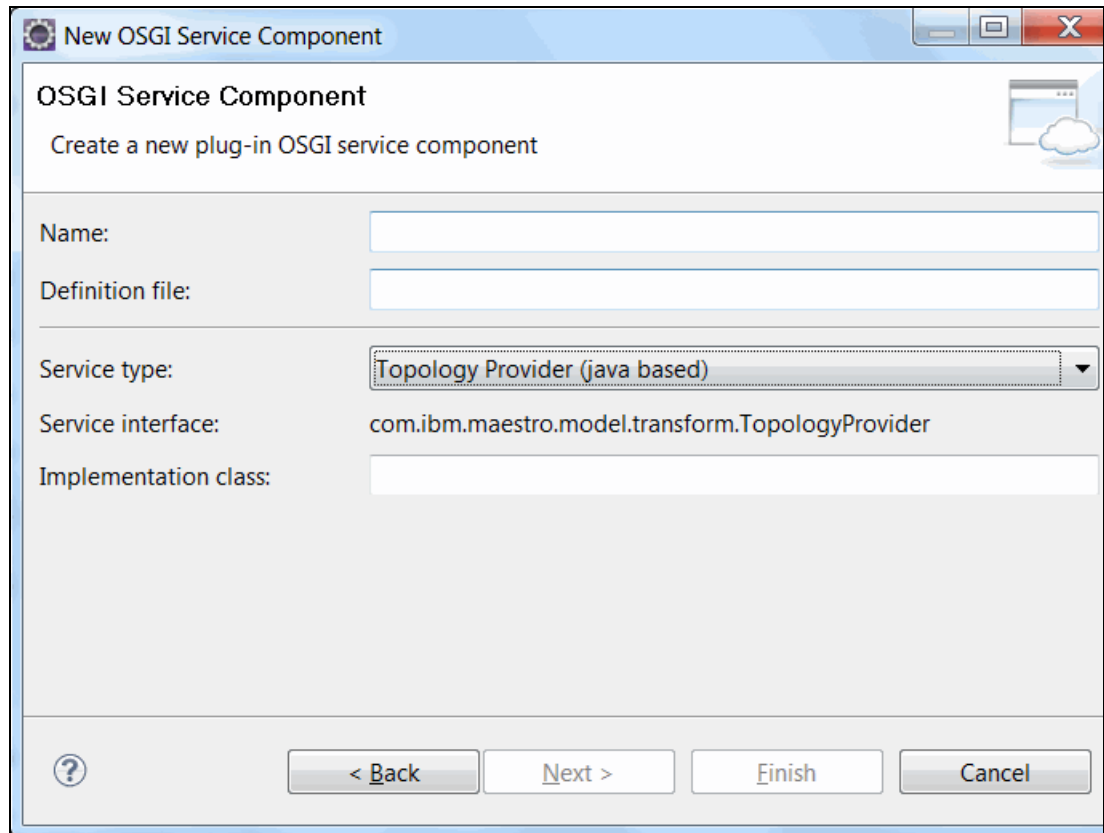


Figure 4-14 Create a new OSGi service component for a Java-based transformer

Templates

When you create a plug-in, you have a choice about how to define the transformer, either using the VM templates, or using Java code by implementing the `TopologyProvider` API. If you use the VM template-based transformation, a file with a `.vm` extension (the VM template) has to be created under the `templates` folder. Example 4-9 shows a simple VM template file.

Example 4-9 A simple VM template file

```
{
  "vm-templates": [
    {
      "name"      : "${prefix}-simple",
      "packages" : [ "simple1" ],
      "roles"    : [
        {
          "plugin": "$provider.PluginScope",
          "type"  : "SIMPLE",
          "name"  : "SIMPLE",
          "parms" : {
            "name" : "$attributes.name"
          }
        }
      ]
    }
  ]
}
```

4.2.3 Upload the pattern type and plug-in from the PDK

You can upload your pattern type and plug-in projects directly from the Eclipse integrated development environment (IDE) using the PDK.

You first need to configure the PDK with IBM PureApplication System:

1. Click **Preferences** → **IBM Workload Plug-in**.
2. Complete the details, such as the IP of the PureApplication System console, your user name, and your password, as shown in Figure 4-15.

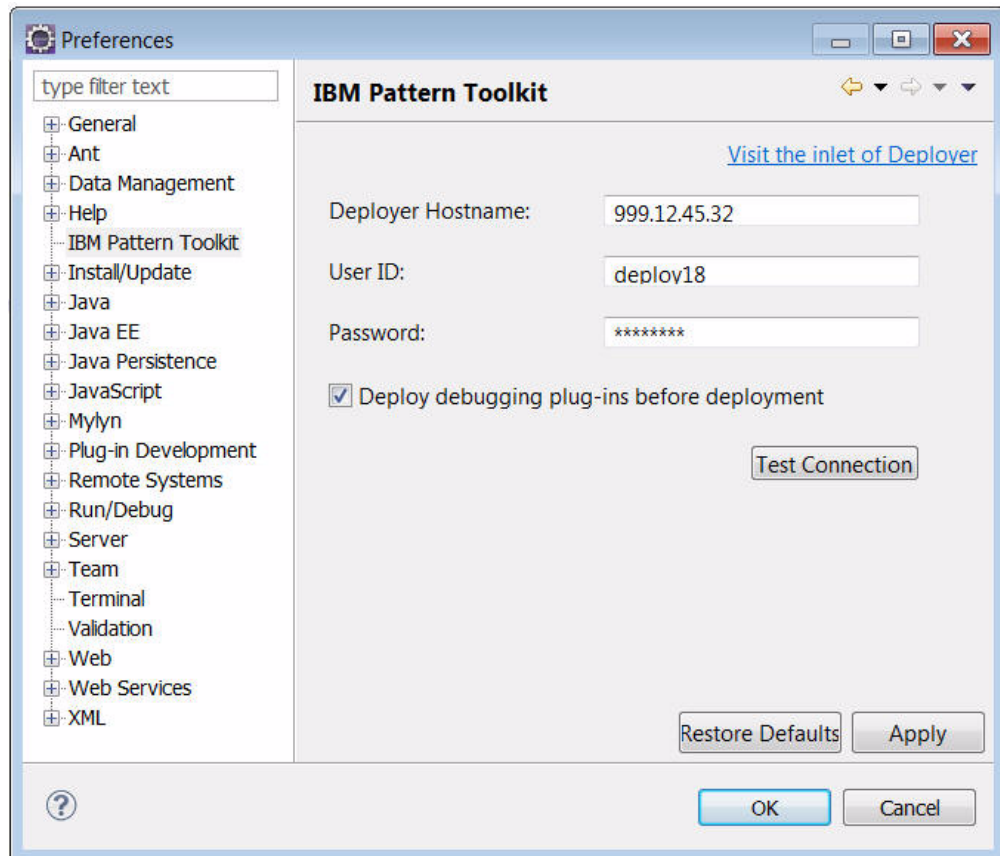


Figure 4-15 Configure PDK with PureApplication System

3. To upload the pattern type project into PureApplication System, right-click the pattern type project and select **IBM Pattern Toolkit Build**, and then click **Install/update to deployer**, as shown in Figure 4-16 on page 65.

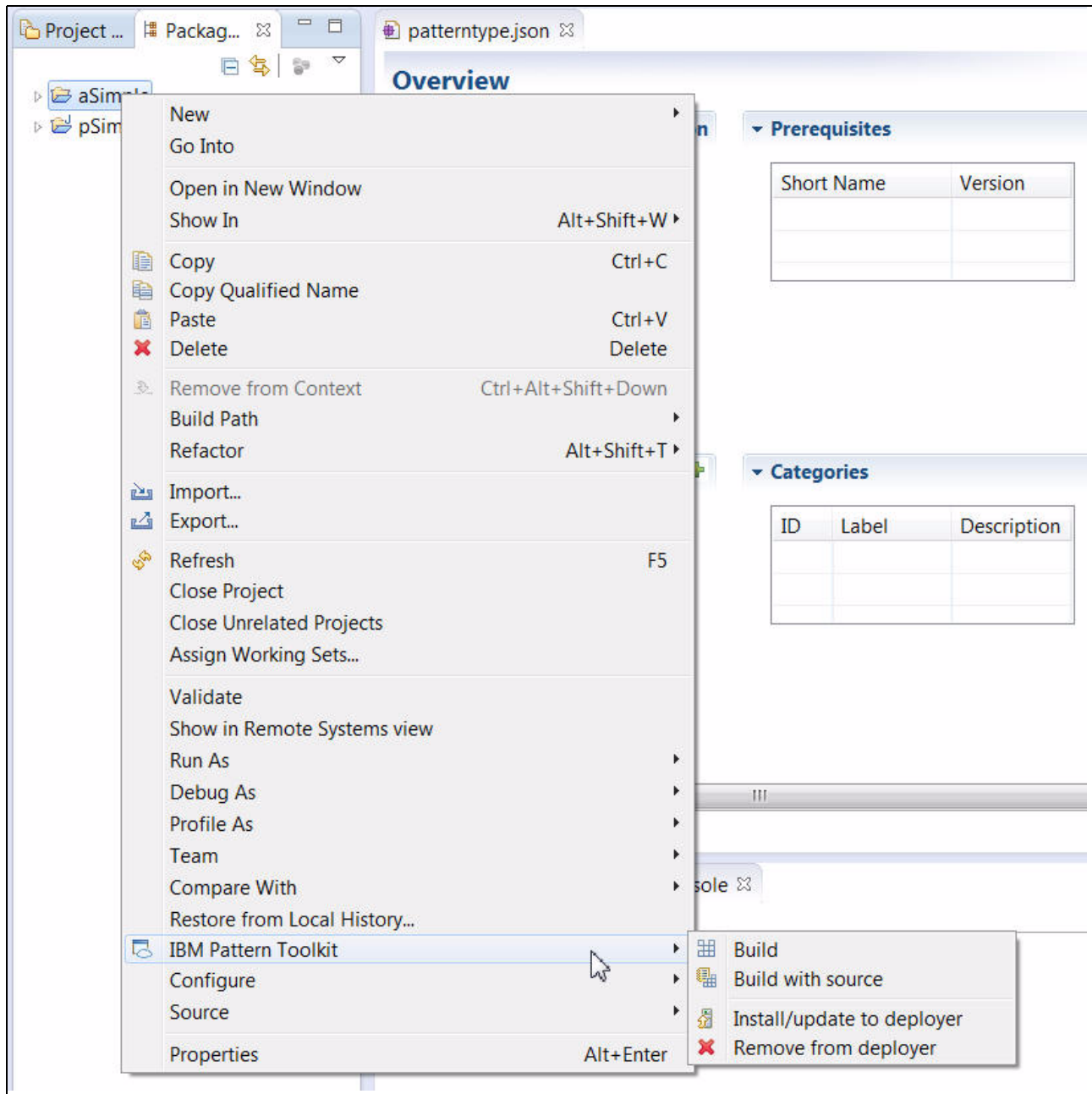


Figure 4-16 Upload the pattern type and plug-in projects from the Eclipse IDE

4.3 The Workload Plug-in Runtime perspective

You can use the Workload Plug-in Runtime perspective to debug and test your deployed pattern directly from the Eclipse IDE. With this option, you can connect to a deployed instance of a virtual application, and see the status of the deployment in the Eclipse workspace. This perspective is useful during plug-in development, because you can debug the deployed plug-in from the Eclipse IDE itself.

You also have the additional option of modifying any failed Python files, and uploading them into the deployed instance to enable live debugging on the deployed instance.

Follow these steps to switch to the Workload Plug-in Runtime perspective:

1. Click **Open Perspective** → **Other** in your Eclipse workspace to launch the Open Perspective window.
2. Select the **Workload Plug-in Runtime** perspective and click **OK**.

Figure 4-17 shows the Workload Plug-in Runtime perspective on the Eclipse workspace, where a connection has been created to a deployed instance.

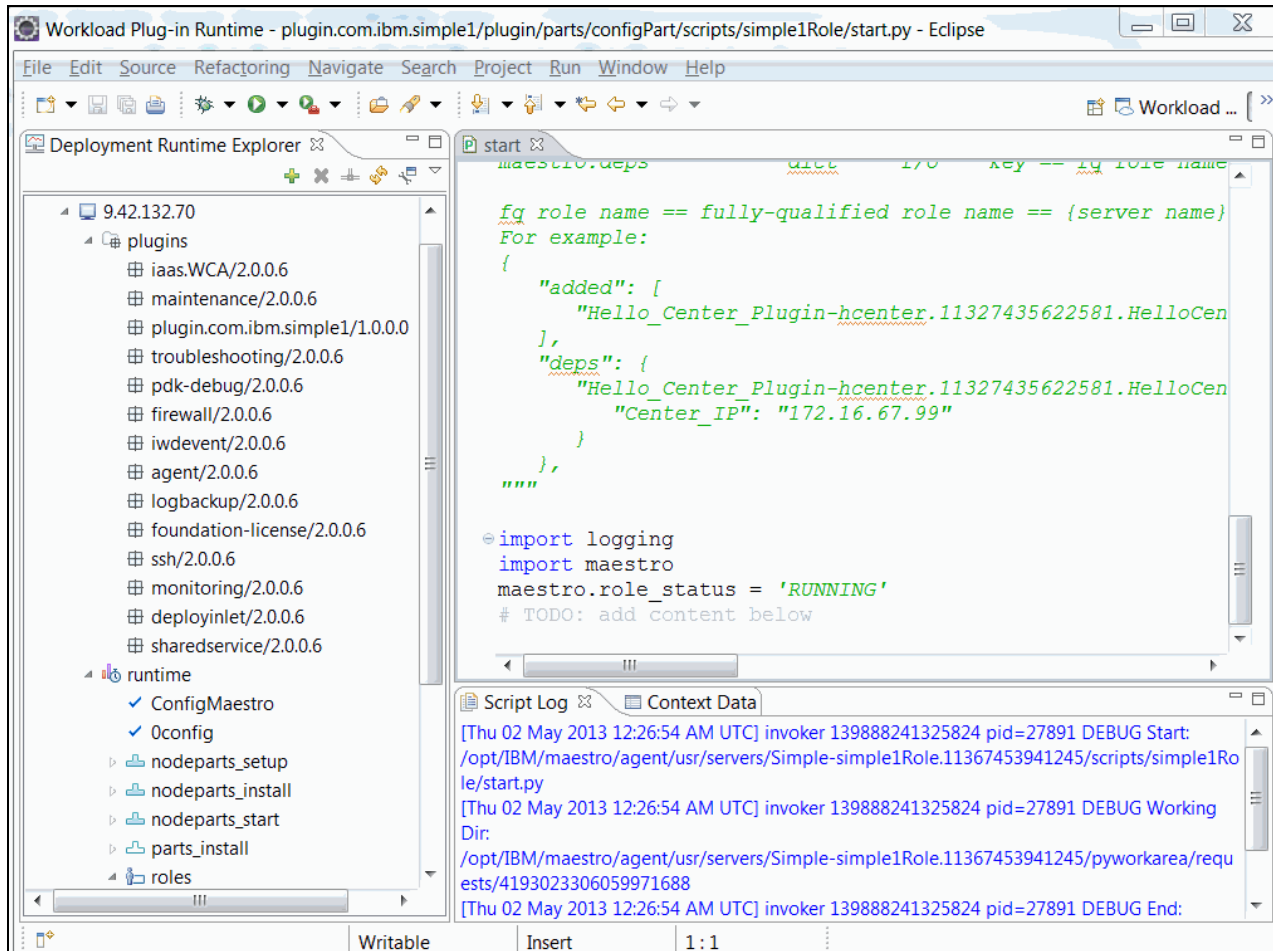


Figure 4-17 The Workload Runtime perspective

You will learn all about debugging using the Workload Plug-in Runtime perspective in Chapter 7, “Debugging and testing” on page 149.

4.4 Setting up the PDK environment

For the latest version of the PDK, and the steps to download it, install the PDK and set it up. See the PureApplication System binary files Information Center at the following website:

http://pic.dhe.ibm.com/infocenter/psappsys/v1r0m0/topic/com.ibm.ipas.doc/iwd/pgt_installpdk.html

4.5 Create a simple plug-in project with the PDK

For step-by-step instructions on creating a simple pattern with the PDK, watch the video at the following website:

http://youtu.be/gCQIQfU_3tY

The video shows you how to create a simple pattern type and a plug-in. The video also shows you how to create a VAP and deploy the pattern in PureApplication System.



Part 2

Creating and implementing an application pattern model

This part takes you through an example of building your own custom pattern type using development tools that come as part of IBM PureApplication System. The following topics are covered:

- ▶ Chapter 5, “Case study” on page 71
- ▶ Chapter 6, “Implementing the model” on page 87
- ▶ Chapter 7, “Debugging and testing” on page 149
- ▶ Chapter 8, “Leading practices for plug-in design and implementation” on page 165



Case study

This chapter describes a real application build for the healthcare industry, and the motivation to build a composite virtual application pattern (VAP) for this application. It takes you through the detailed steps of the pattern model design, and defines the final pattern types and plug-ins ready for implementation.

This chapter covers the following main topics:

- ▶ A business problem for a life science company
- ▶ The spend record processing application
- ▶ Composite application pattern for Promotional Spend Compliance

5.1 A business problem for a life science company

The prescription pharmaceutical market in the US is a USD100-billion dollar business, and spends about \$30 billion annually on face-to-face promotions, journal advertising, samples, and direct-to-consumer marketing. Many studies have shown the effect of gifts, even small ones, on doctors' behaviors. This effect can drive up drug costs, and sometimes even put patients at risk.

5.1.1 Background

Signed into law in March 2010, the Physician Payment Sunshine Act and various existing state laws require pharmaceutical and medical device manufacturers to publicly report gifts and payments made to physicians and teaching hospitals.

All US manufacturers (and other entities under common ownership) of drug, device, biologics, and medical supplies covered under Medicare, Medicaid, or the State Children's Health Insurance Program (SCHIP) must report payments on an annual basis to the Department of Health and Human Services (HHS), which posts the information on a public website.

The healthcare reform law requires the disclosure of aggregate payments above \$100, whether via cash or in-kind transfers, to all covered recipients, specifically physicians and teaching hospitals. The following items are considered payments under this law:

- ▶ Compensation
- ▶ Food
- ▶ Entertainment or gifts
- ▶ Travel
- ▶ Consulting fees
- ▶ Honorariums
- ▶ Research funding or grants
- ▶ Education or conference funding
- ▶ Stocks or stock options
- ▶ Ownership or investment interest
- ▶ Royalties or licenses
- ▶ Charitable contributions
- ▶ Any other transfer of value described by the Secretary of HHS

Reporting companies are required to report the receiving physician's name, address, and national provider identifier, and the value, date, form, and nature of the payment, using standardized descriptions for the payment types listed. For each failure to report, fines of up to \$10,000 will be applied, not to exceed \$150,000 annually. For each knowing failure to report, fines of up to \$100,000 will be applied, not to exceed \$1,000,000 annually.

Starting on January 1, 2012, manufacturers must record all transfers of value. This information was to be reported to the HHS by March 31, 2013, and annually thereafter. HHS will then post this information on a publicly available, searchable online database as of September 30, 2013, and on June 30 of each year beginning thereafter.

5.1.2 Solution requirements

With the new requirements for promotional spending compliance regulations, the life science enterprise needs to provide an end-to-end solution composed of the data management, business process, monitoring, and reporting components that properly integrate, categorize, and report all money paid to physicians and teaching hospitals.

These are the main capabilities to accommodate the new requirements:

- Provide data integration with various sources, and gain efficiencies through proactive data monitoring and exception handling, to remain compliant with state and federal laws.
- Provide flexibility in dealing with evolving legislation.
- Provide business insights through deeper analytics.
- Select Software as a Service (SaaS)-hosted or in-house solutions based on the business justification of Return on Investment (ROI).
- Provide elastic scaling based on resource usage to reduce cost and maximize efficiency.
- Provide Agile solution lifecycle management capabilities.

5.1.3 Spend record processing use case

There are many use cases and functional requirements identified for the Promotional Spend Compliance (PSC) solution, but they are not the focus of this book. This chapter contains the key processes as an application example, and defines the virtual application pattern model for the PSC applications in the same domain, to prove the concept of an SaaS offering of this solution.

The key application is the spend record processing application. As shown in Figure 5-1, the spend record is loaded every day at a scheduled time, and then it goes through a validation check that confirms its completeness. If any essential content is missing, it goes back to the spend user for updating. When it becomes a valid spend record, it goes through a compliance check. If the spend record is compliant, the process is finished. Otherwise, it goes back to the spend user again for compliance confirmation.

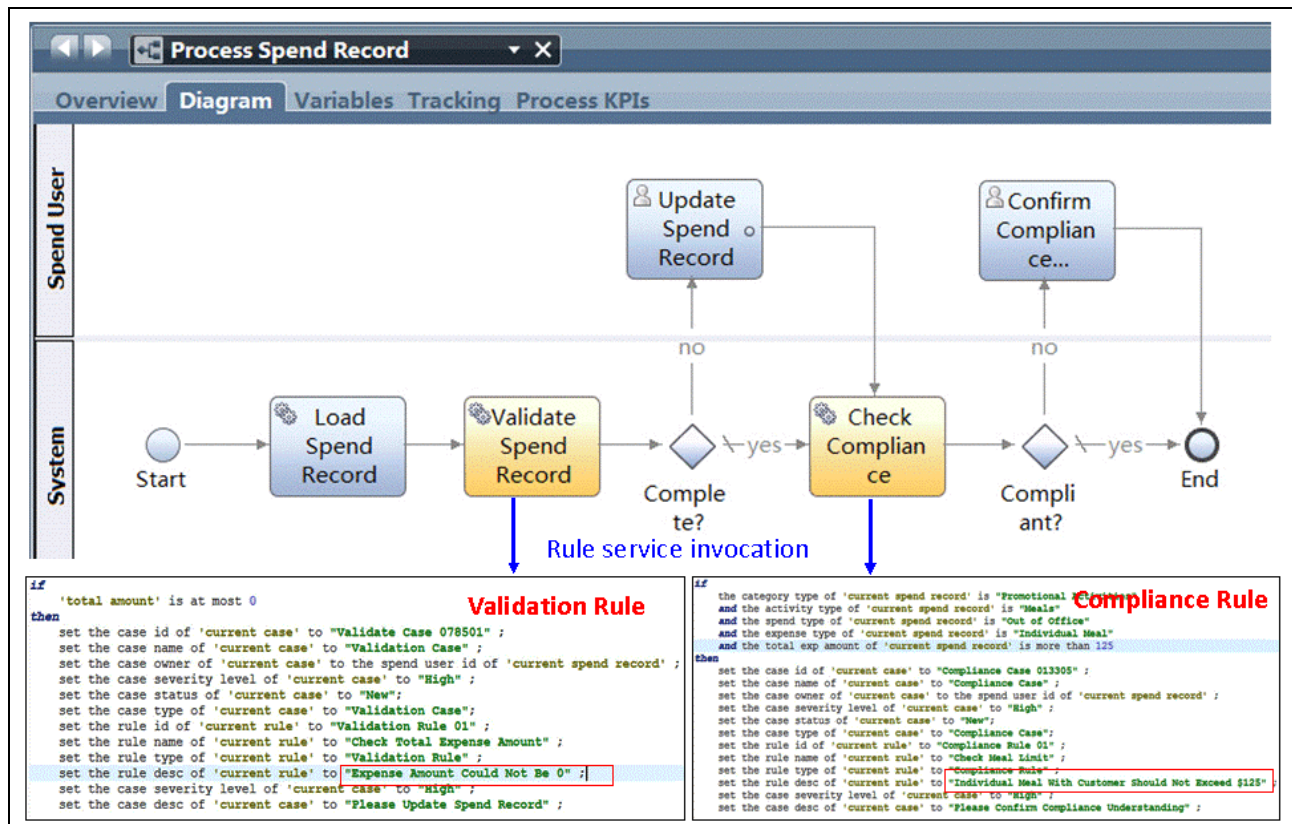


Figure 5-1 PSC Process Spend Record scenario

5.2 The spend record processing application

The spend record processing application needs to be flexible, expandable, and robust to ensure compliance with new and evolving federal and state regulations requiring companies to disclose payments made to healthcare providers and physicians. It provides the following capabilities:

- ▶ Load the spend record
- ▶ Spend record validation
- ▶ Spend record compliance check
- ▶ Invalid spend record handling
- ▶ Noncompliance case handling
- ▶ Notifications (to the spend user and the physician)

For better business process design and implementation, separate the business logic from the business process execution. This approach gives your system flexibility from the business perspective, and lower maintenance effort from the IT perspective. This chapter illustrates the spend record processing application with two major artifacts:

- ▶ The `PSCRuleApp.jar` file contains validation rules and compliance rules on WebSphere Operational Decision Management V8.0. The rule applications are shown as web services when deployed.
- ▶ The `pscV01.zip` file is for end-to-end business processes in IBM Business Process Manager V8.0 (BPM). The rule web services are involved inside the process implementation.

5.2.1 Validation and compliance rule application artifact

The `PSCRuleApp.jar` PSC rule application artifact is available to download with the additional web material that accompanies this book:

<ftp://www.redbooks.ibm.com/redbooks/SG248146>

WebSphere Operational Decision Management is used to develop and deploy the PSC rule application. WebSphere Operational Decision Management includes two main components:

- ▶ Decision Server, for developing and running decisions, and for event detection
- ▶ Decision Center, for more business-oriented decision management

In this case study, the focus is on the rule application deployment and execution, so it only includes the Decision Server component. As shown in Figure 5-2 on page 75, the rule project is created in the WebSphere Operational Decision Management Rule Designer integrated development environment (IDE), exported as a Java Archive (`.jar`) file, and then deployed to the execution server.

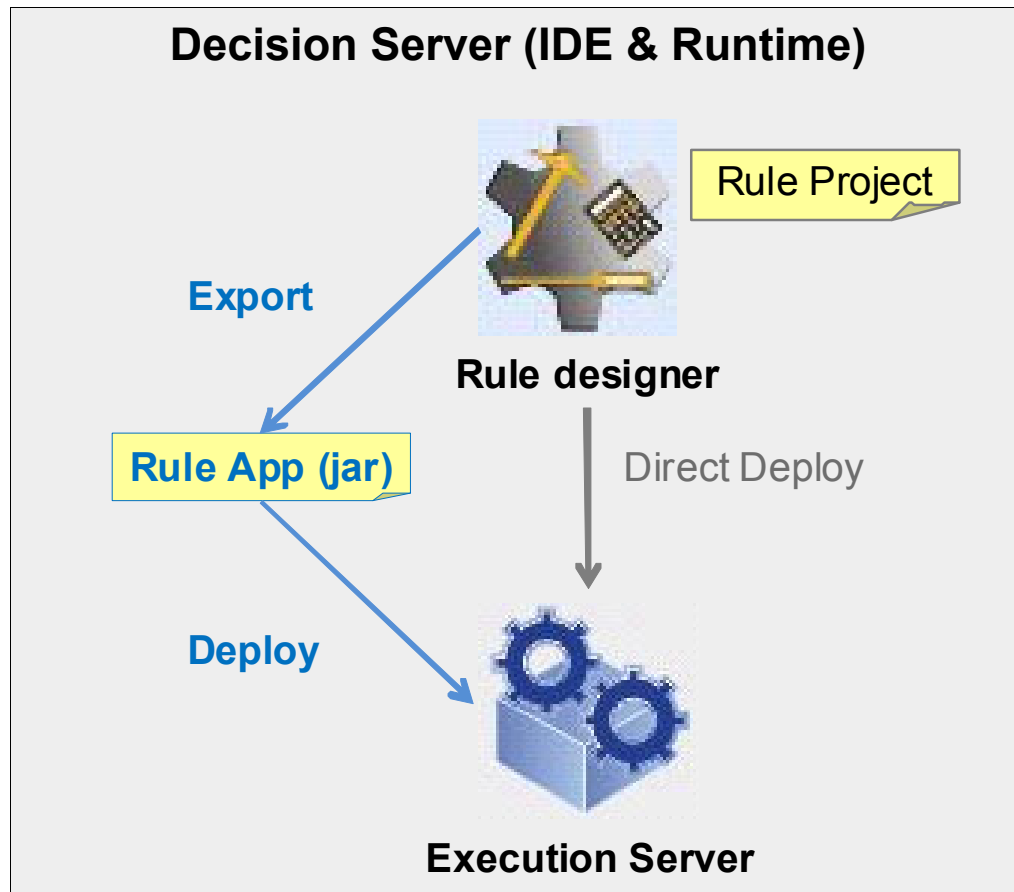


Figure 5-2 Rule application development and deployment in the Decision Server component

Figure 5-3 shows the details of the PSC rule project implemented in the Rule Designer IDE.

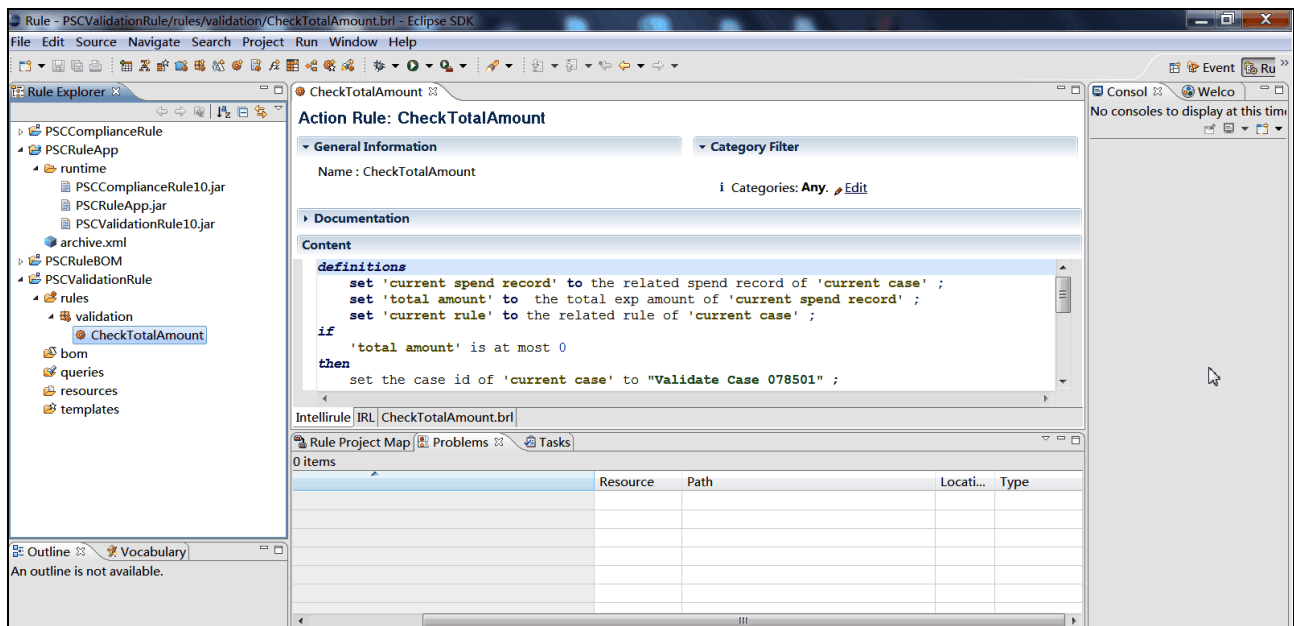


Figure 5-3 PSC rule implementation in WebSphere Operational Decision Management Rule Designer

For more information about the rule application and decision server, go to the WebSphere Operational Decision Management Information Center:

<http://pic.dhe.ibm.com/infocenter/dmanager/v8r0/index.jsp>

5.2.2 Spend record process application artifact

The pscV01.zip PSC process application artifact is available to download with the additional web material that accompanies this book:

<ftp://www.redbooks.ibm.com/redbooks/SG248146>

BPM is used to develop and deploy the PSC business process. BPM includes two major components:

- ▶ Process Center provides a unified BPM repository for process applications. It contains process models and service implementations, including any required supporting files. It supports process design and development by providing process authoring tools, such as Process Designer and Integration Designer.
- ▶ Process Server is the runtime environment that supports Business Process Model and Notation (BPMN) 2.0 and Business Process Execution Language (BPEL) processes.

In this case study, the PSC process is implemented as a BPMN process in the Process Designer tool, and then exported as a .zip file that can be deployed to the Process Server component, as shown in Figure 5-4.

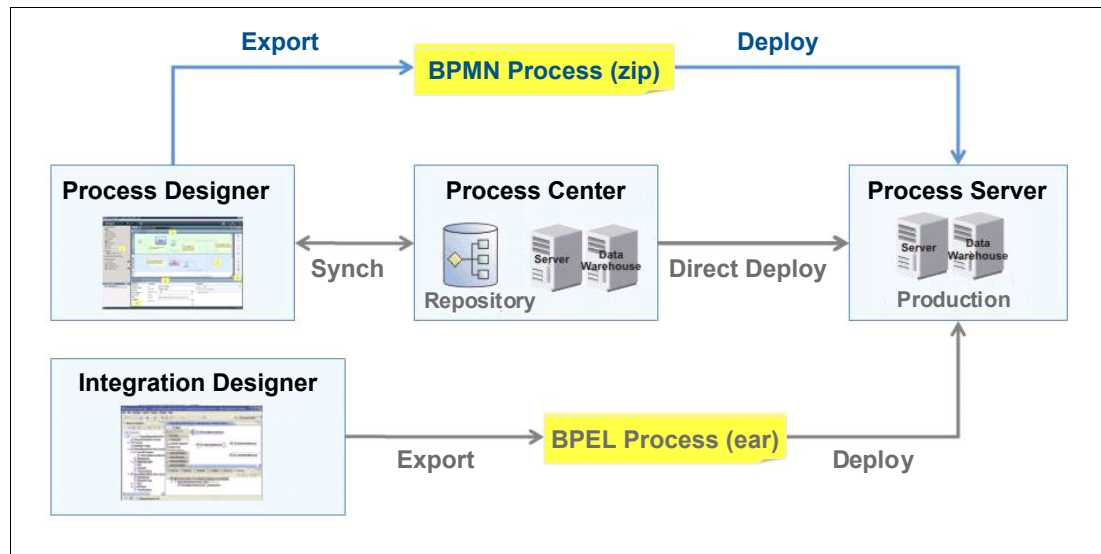


Figure 5-4 Process application development and deployment in BPM

There is another way to deploy the process application directly from Process Center to Process Server. You need to specify the process acronym and snapshot name.

Figure 5-5 on page 77 shows details of the PSC process illustrated in Process Designer.

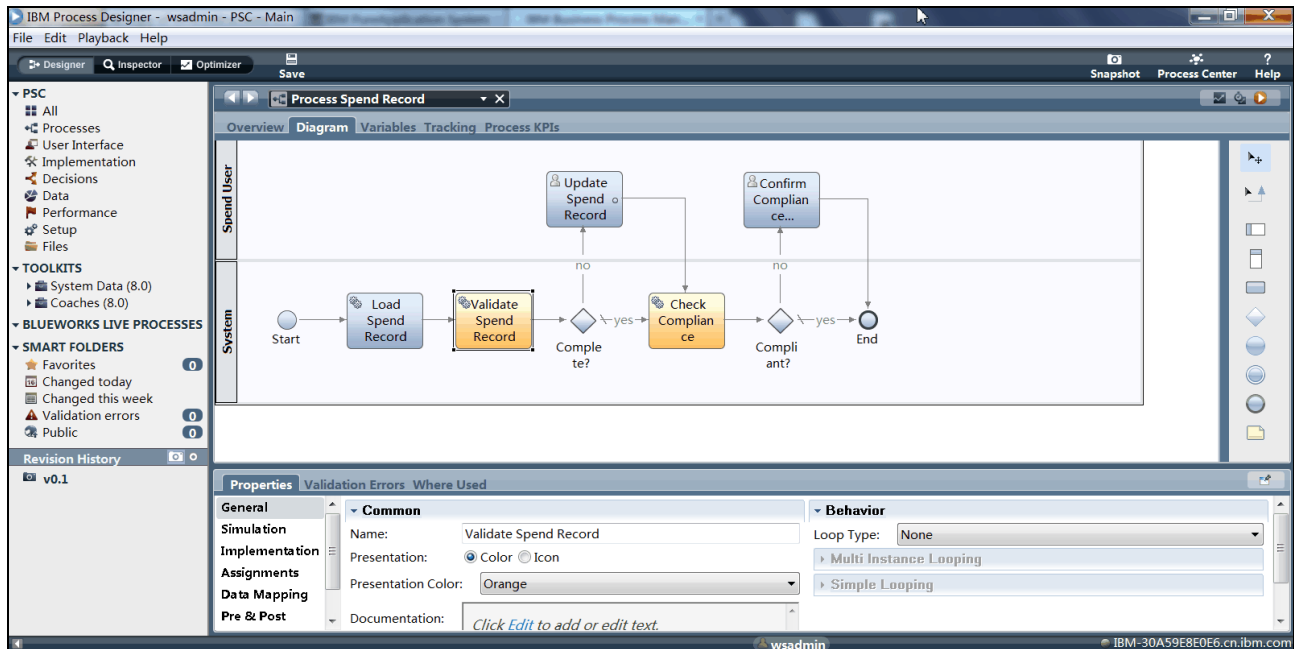


Figure 5-5 PSC process implementation in BPM Process Designer

For more information about business process development and management, go to the IBM Business Process Manager V8.0 Information Center:

<http://pic.dhe.ibm.com/infocenter/dmndhelp/v8r0mx/index.jsp>

5.2.3 Start validation and compliance rules in process

There are two typical approaches for starting a rule service in process. One is via an embedded decision service, and the other is via a normal web service.

Using an embedded decision service, you need to point to a Rule Execution Server (RES) with its credentials, and specify the corresponding rule application and rule set. You do not need to provide the detailed rule application deployment information, but you need to have the access to the RES.

This requires that all of your Rule Execution Servers and rule applications are stable when you develop your business process and integrate the servers and applications. Any later change, such as RES credentials, addresses, or port numbers, can significantly affect the integration, and need to be updated in the Process Designer tool.

Figure 5-6 illustrates starting a rule service in process using an embedded decision service.

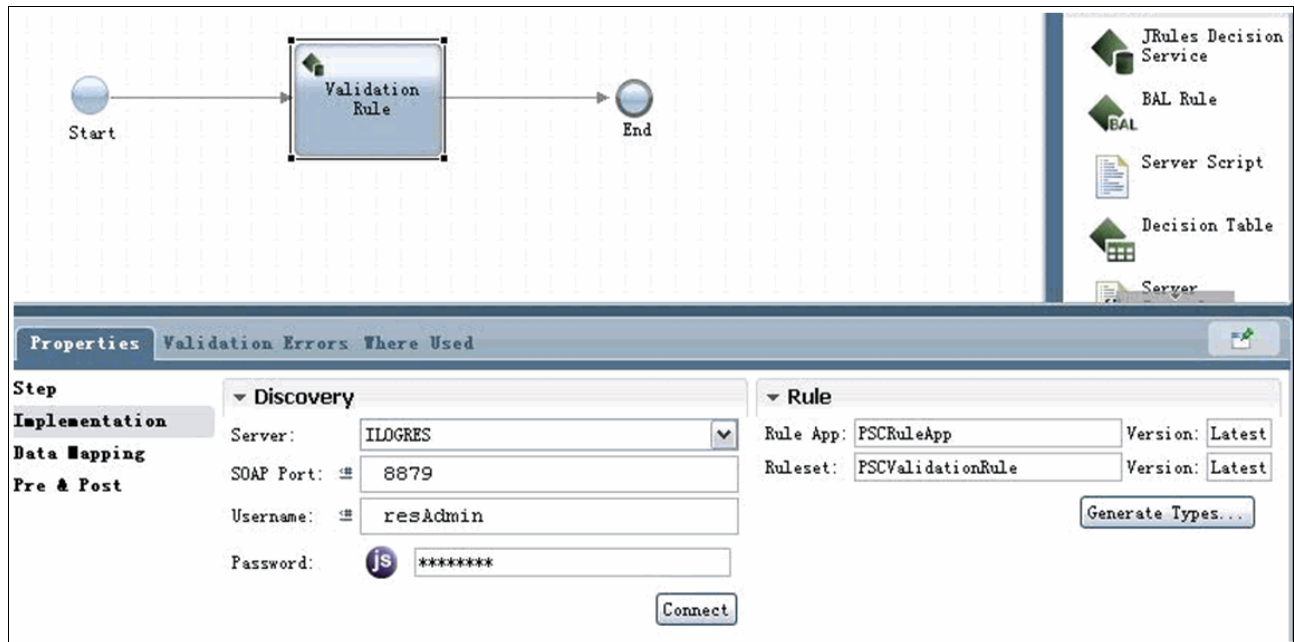


Figure 5-6 Process rule integration via embedded decision service

As shown in Figure 5-7 on page 79, using web services is treating the rule application as a normal web service. You specify the Web Services Description Language (WSDL) URL, and the endpoint address URL, for your rule service.

With this approach, you do not need to know the RES credentials. The RES already generated the WSDL for the rule service when you deployed the rule application, so you can just get it and import it into Process Designer. Furthermore, you can define environment variables for the actual RES address to make it configurable at run time.

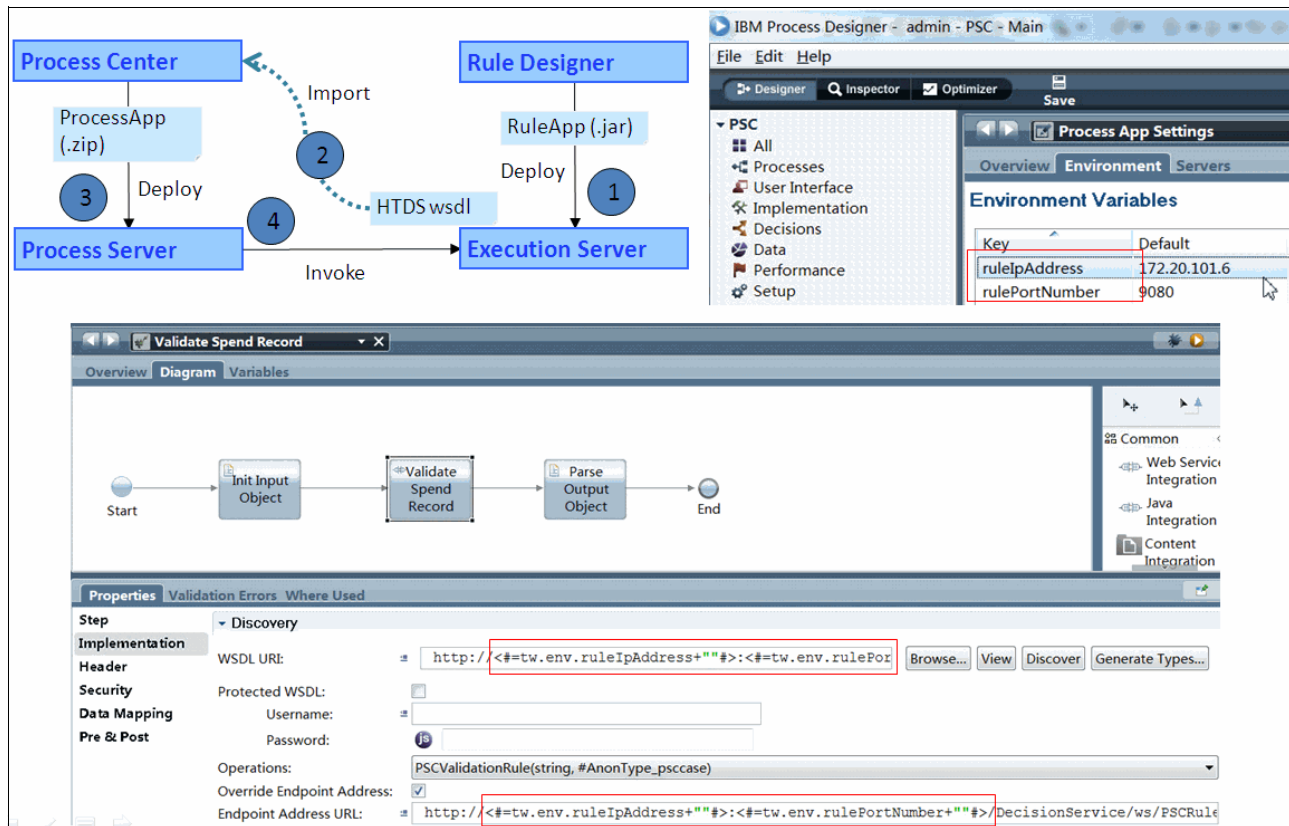


Figure 5-7 Process rule integration via normal web service

In the spend record processing application, the normal web service approach was implemented for the process and rule integration. This provides flexibility on the RES information configuration during run time.

This is especially important for defining VAPs for this application, because the RES information will never be known until it is deployed in the cloud. If you cannot find a way to configure this kind of information after deployment, it will be hard to define a composite VAP.

5.3 Composite application pattern for Promotional Spend Compliance

As described in 5.2, “The spend record processing application” on page 74, the spend record processing application is composed of two major artifacts, *the process* and *the rule*.

Figure 5-8 shows the virtual application planning diagram for this application. In addition to the functional requirements for this application, there are some non-functional requirements identified, such as elastic scaling, based on the usage of the resources and the ease of application lifecycle management.

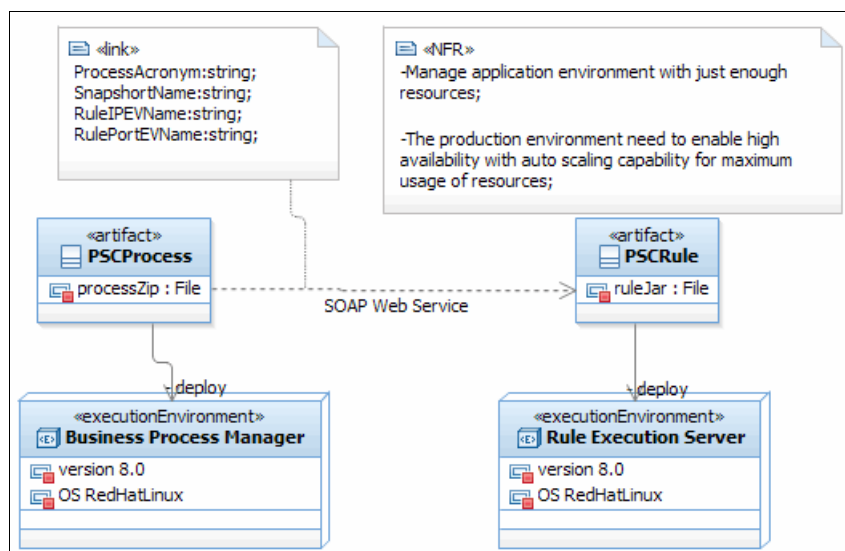


Figure 5-8 PSC virtual application

As shown in Figure 5-9 on page 81, the virtual application pattern model design contains the following elements:

- ▶ A process application component
- ▶ A rule application component
- ▶ A link representing the integration between the process and the rule
- ▶ A scaling policy for the auto scale
- ▶ A usage intent policy for the application lifecycle management

When you package these model elements, you define a process automation pattern type for the process management domain, and you define a rule execution pattern type for the rule management domain. You put the process application component in the process plug-in project, which belongs to the process automation pattern type.

You put the rule application component in the rule plug-in project, and then create a link between the process and the rule. The two policies in the rule plug-in project belong to the rule execution pattern type (see Figure 5-9 on page 81).

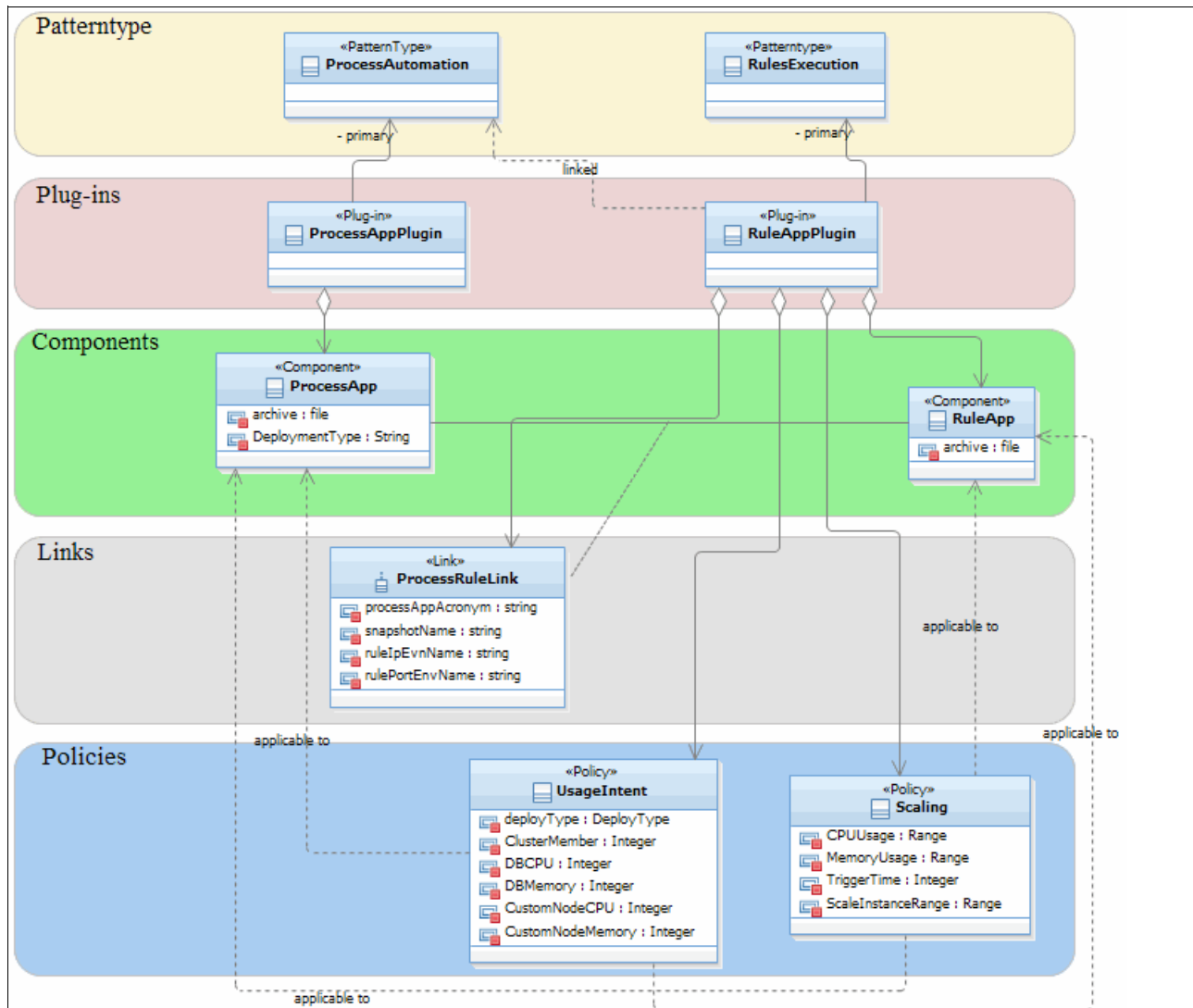


Figure 5-9 PSC virtual application pattern model design

The main reason you separate the two pattern types here is because you need to package the product binary files that install BPM and WebSphere Operational Decision Management in the plug-in and pattern type. Both products have large binary files. Therefore, keep them separate and smaller to avoid having a huge pattern type package that takes a long time to upload to IBM PureApplication System.

Furthermore, the process and rules are two separate business domains. The two pattern types can be used independently for a single process application or a single rule application, and do not necessarily have to be bound together.

You do not define a new composite pattern type for creating the VAP that contains elements in both pattern types, you just link the rule application plug-in to the process automation pattern type. As a result, when you create a VAP with the rule execution pattern type, the process component displays in the palette. There is more information about linked usage in 8.2, “Associating a plug-in with pattern types” on page 166.

5.3.1 Rule application component and transformed topology

The rule application component can be transformed to a topology that installs the RES, and deploys the specified rule application during the pattern deployment. Figure 5-10 shows the rule application component and its attribute in the virtual pattern builder.

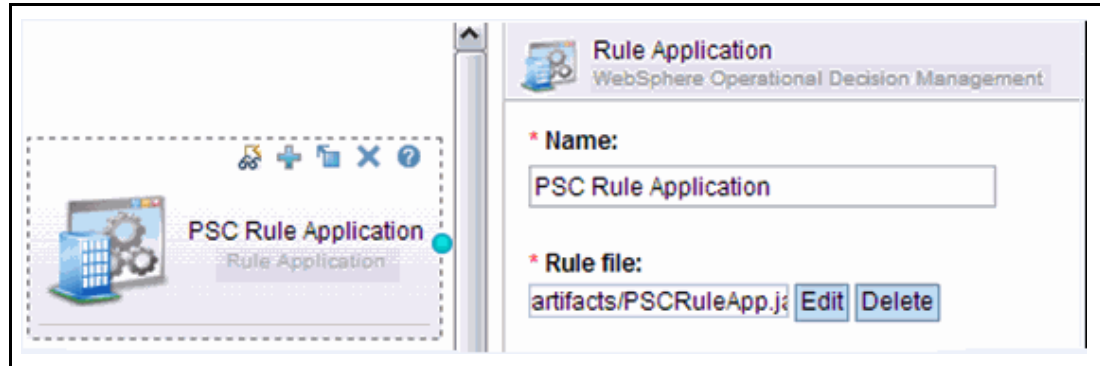


Figure 5-10 Rule application component in the virtual pattern builder

Two policies are defined for the rule application component:

- ▶ The usage intent policy, which defines four stages of the application lifecycle:
 - Development
 - Testing
 - Staging
 - Production
- ▶ The scaling policy

For development, this means that the rule application is deployed to a runtime environment for development purposes. Usually, that application is transformed into a stand-alone rule execution server, as shown in Figure 5-11.

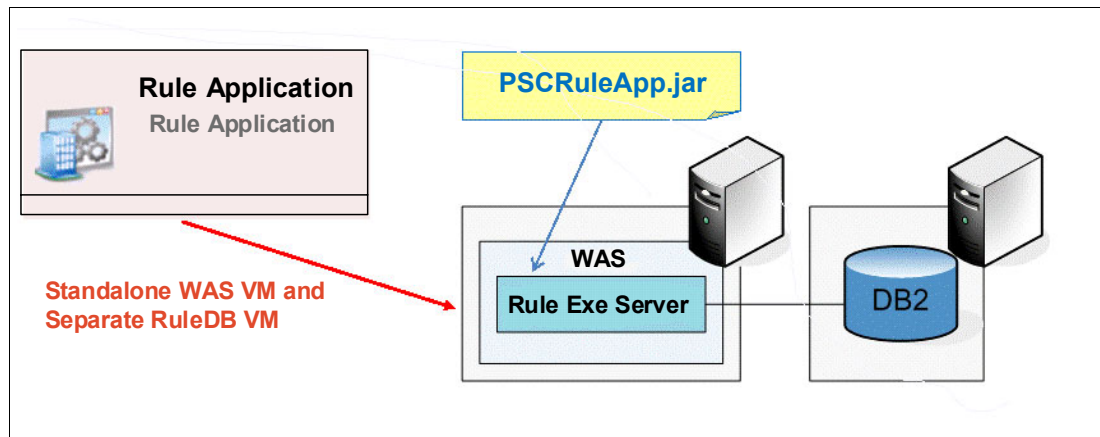


Figure 5-11 Rule application transformed topology with development usage intent

Two virtual machines (VMs) are created in the cloud:

- ▶ One is installed with IBM DB2® as the RES database.
- ▶ The other is installed with RES on a stand-alone WebSphere Application Server.

The rule application is deployed on the RES.

For testing, staging, and production, the rule application is deployed to a high-availability, enabled environment with different numbers of clustered members (to provide different quality of service levels). Figure 5-12 shows the transformed topology of a rule application with testing usage intent.

There are six VMs created in the cloud:

- ▶ Two HTTP server VMs
- ▶ A WebSphere Application Server Deployment Manager VM
- ▶ A RES database VM
- ▶ Two WebSphere Application Server Custom Node VMs composed as a cluster
- ▶ The RES installed on the cluster
- ▶ The rule application deployed on the RES

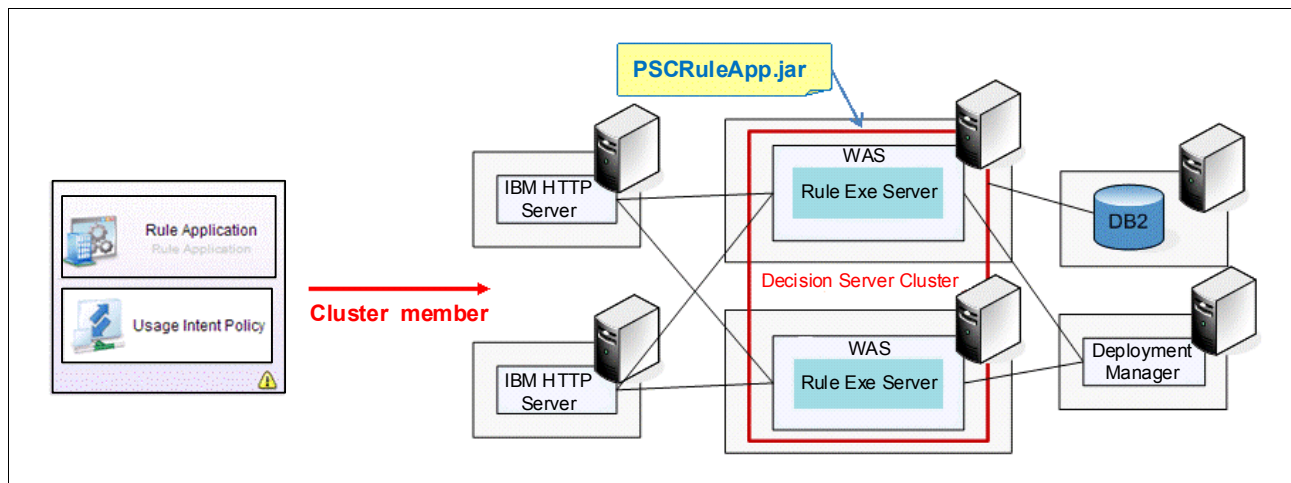


Figure 5-12 Rule application transformed topology with a testing usage intent

The auto-scaling policy is actually applied with testing, staging, and production usage intent. It is capable of adjusting cluster members, such as CPU and memory, based on the VM resource usage.

5.3.2 Process application component and transformed topology

The process application component can be transformed into a topology that installs the Process Server and deploys the process application during the pattern deployment.

Figure 5-13 shows the process application component and its attributes in the virtual pattern.

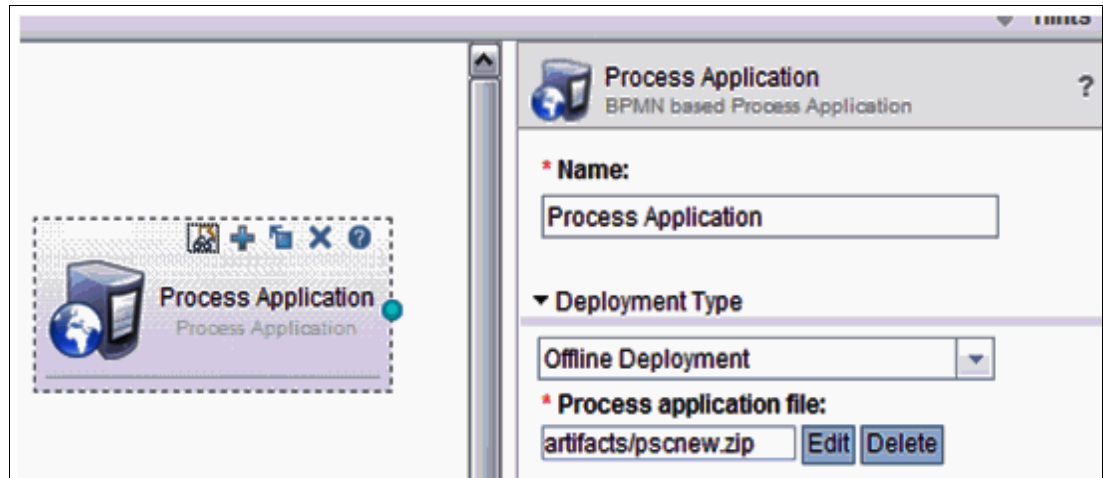


Figure 5-13 Process application component in the virtual pattern builder

Two policies, which have the same effect with rule application components, can also be applied to the process application component. Figure 5-14 shows the details of a transformed topology with different usage intent policy settings.

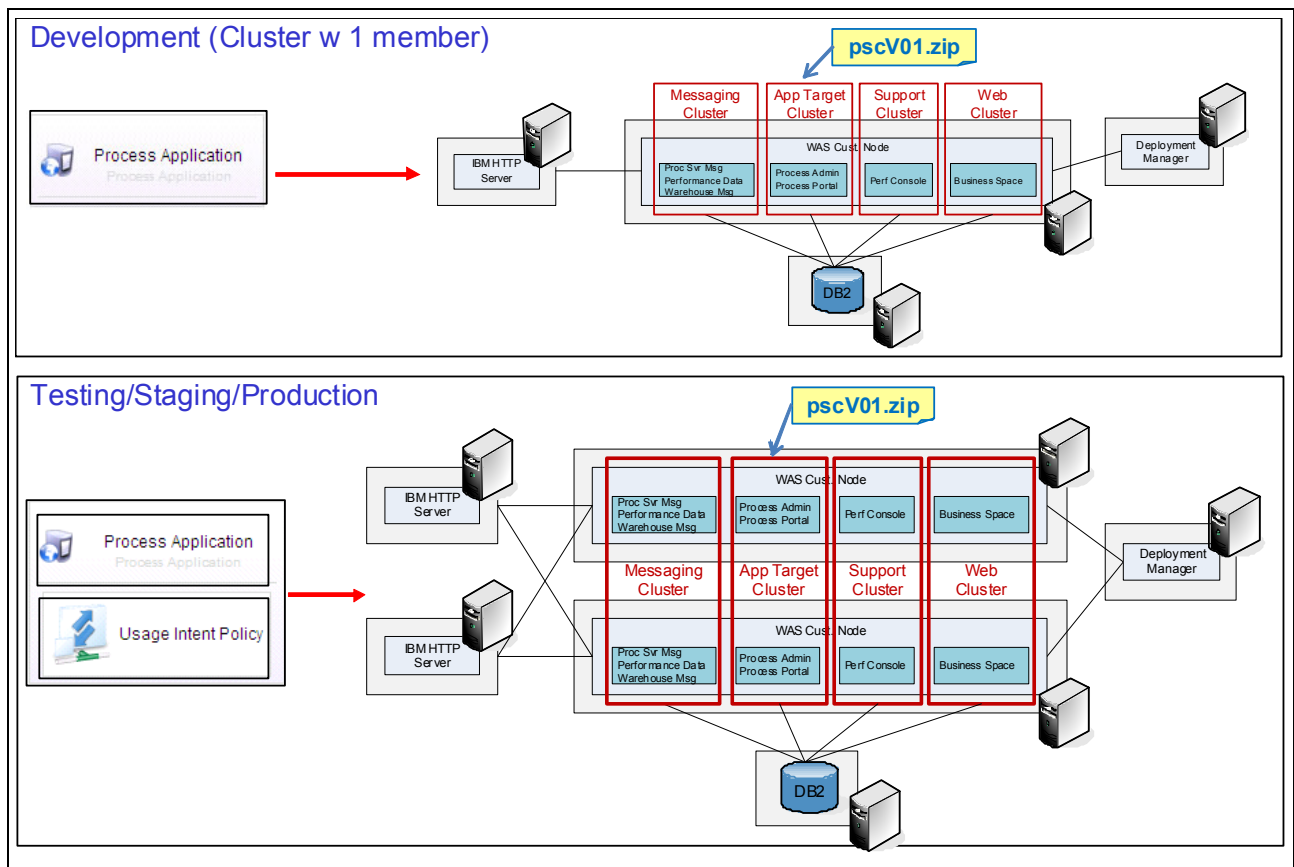


Figure 5-14 Process application component transformed topology with policy

5.3.3 Process and rule link transformed topology

As described in 5.2.3, “Start validation and compliance rules in process” on page 77, the process rule integration is implemented through normal web service integration. You define two environment variables, the rule service, the IP address, and the port number.

The transformed topology for this link is actually the configuration of getting the rule service, IP address, and port number from the transformed rule application component, and setting it to the Process Server to which the process application is deployed.



Implementing the model

This chapter introduces detailed information about how to implement the virtual application pattern model. This information includes how to define pattern types, plug-ins, components, links, and policies.

This chapter contains the following topics:

- ▶ The Business Rule Application pattern type
- ▶ The Business Process Application pattern type
- ▶ The business process and business rule link

6.1 The Business Rule Application pattern type

The process of creating pattern types involves multiple steps. The following sections explain each step.

6.1.1 Creating rule application components

Create rule application components by following the steps in this section.

Create a pattern type

Follow these steps to create a pattern type:

1. Create pattern type projects using the wizard (see Figure 6-1):
 - a. Create a pattern type project by selecting **File** → **New** → **Project** → **IBM Workload Plug-in Development** → **IBM Workload Pattern Type Project**.
 - b. Enter `patterntype.ruleexecution` as both the Project name and the Pattern type name, and `1.0.0.0` as the Pattern type version.
 - c. Click **Finish**.

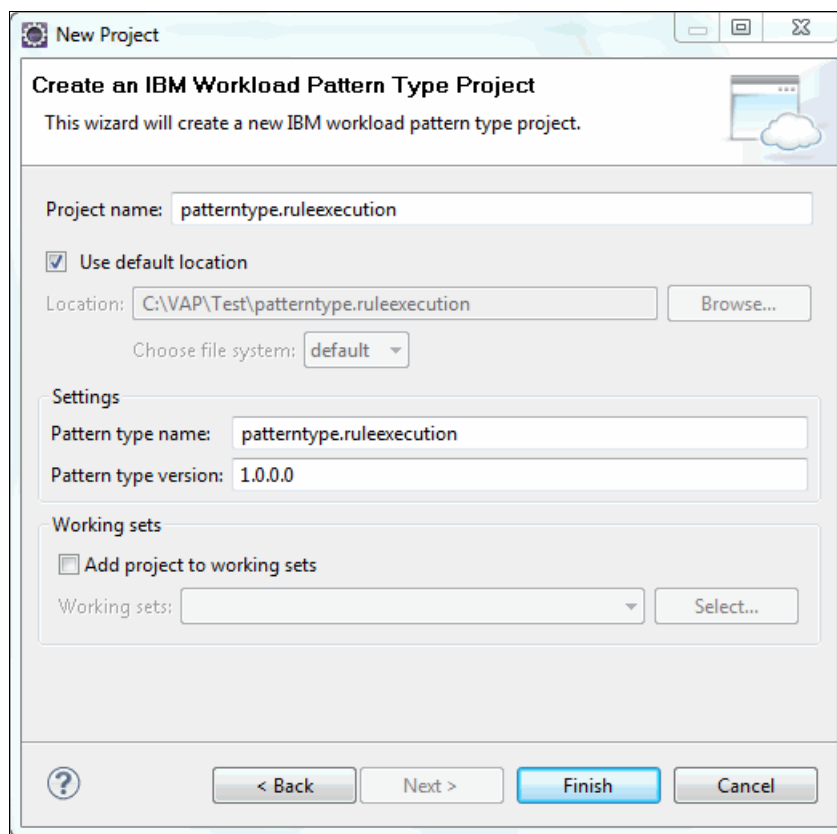


Figure 6-1 Create a pattern type

2. Update the `patterntpc.json` pattern type configuration file:
 - a. Open the `patterntype.json` file.
 - b. Add a category by clicking **Add** in the Categories section, as shown in Figure 6-2 on page 89.

patterntype.json Overview

Pattern Type Basic Information

Name:

Short Name:

Version:

Description:

Status:

Builder: ☐

Prerequisites

Short Name	Version

Add... Remove

Plug-in Projects

Name	Version

Categories

ID	Label	Description

Add... Remove

Overview patterntype.json

Figure 6-2 The patterntype.json file

- c. Complete the ID, Label, and Description, as shown in Figure 6-3. Click **OK**.

Add a Category

Add a Pattern Type Category.

ID:

Label:

Description:

? OK Cancel

Figure 6-3 Add a category

- d. Save the patterntype.json file by clicking **Save**.

Create a plug-in

Follow these steps to create a plug-in project:

1. Select **File** → **New** → **Project** → **IBM Workload Plug-in Development** → **IBM Workload Plug-in Project**.
2. Enter `plugin.com.ibm.wodm.res` as both the Project name and the Plug-in name (see Figure 6-4).
3. Click **Finish**.

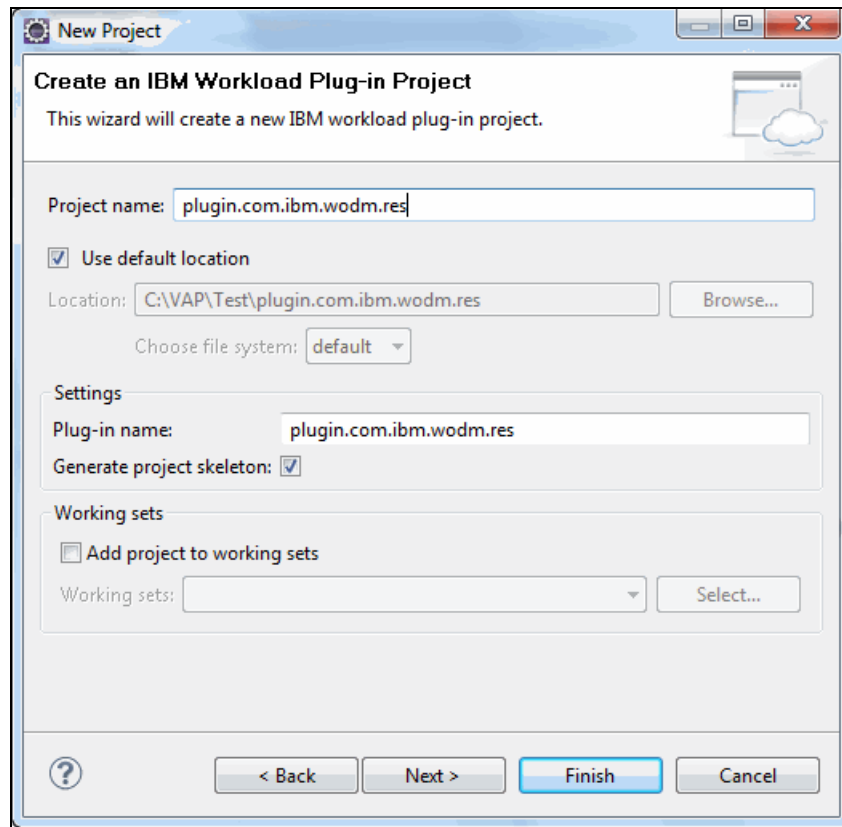


Figure 6-4 Create a plug-in

4. Update the `config.json` plug-in configuration file:
 - a. Open the `config.json` file and switch to the Configuration tab (see Figure 6-5 on page 91).
 - b. Remove the default pattern type.
 - c. Add a reference to the pattern type by clicking **Add** in the Pattern Types section.

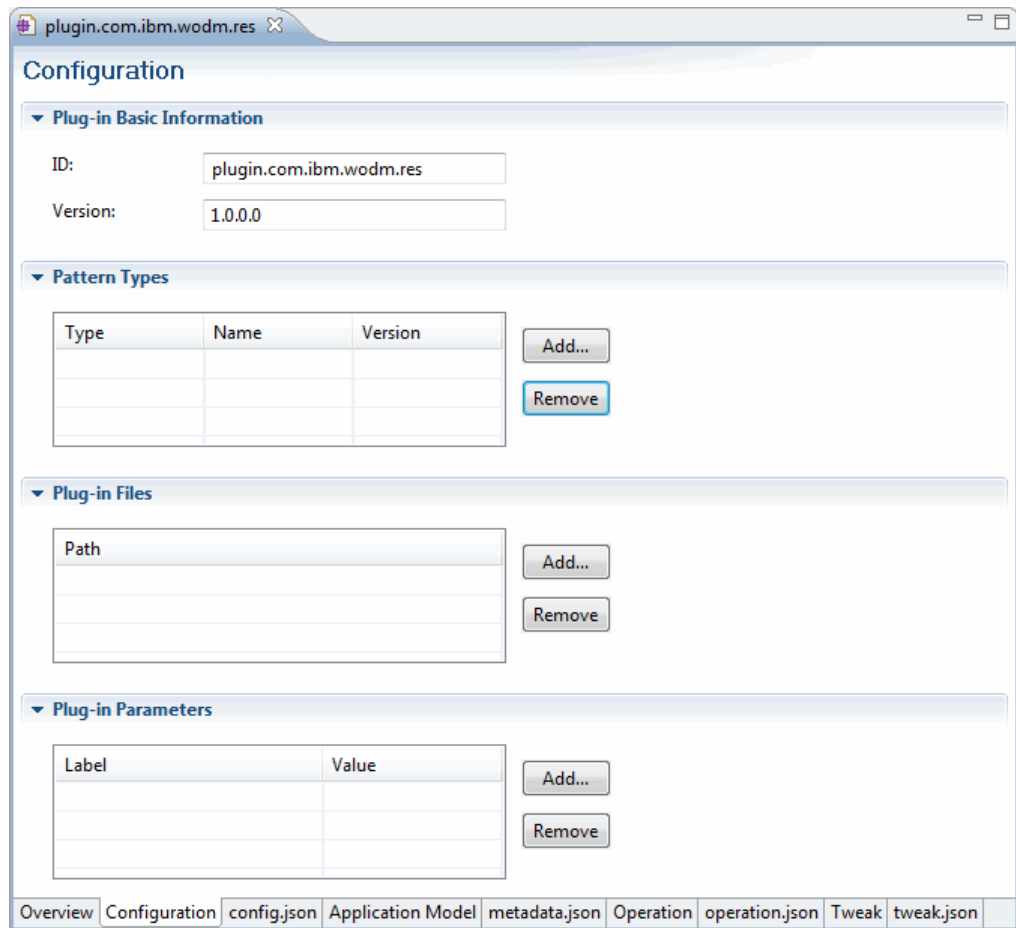


Figure 6-5 The `config.json` Configuration tab

- d. Add files by clicking **Add** in the Plug-in Files section.
- e. Select **primary** as the Type. Enter `patterntype.ruleexecution` as the Name, and 1.0 as the Version (see Figure 6-6). Click **OK**.

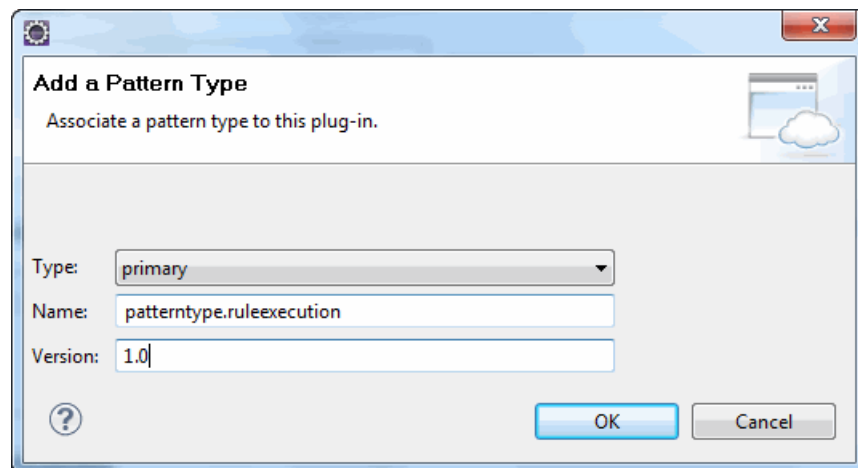


Figure 6-6 Add a pattern type

The pattern engine parses the config.json/files structure to create content in the storehouse. When you deploy the pattern instance, the part script on the VM can download software installation files from the storehouse.

Figure 6-7 lists the files in the storehouse.



Figure 6-7 Files in the storehouse

- f. Add a file by clicking **Add** in the Plug-in Files section.
- g. Enter /res/WS_DCSN_SVR-IM_REPO-V8.0_MP_ML.tar as the File path (see Figure 6-8). Click **OK**.

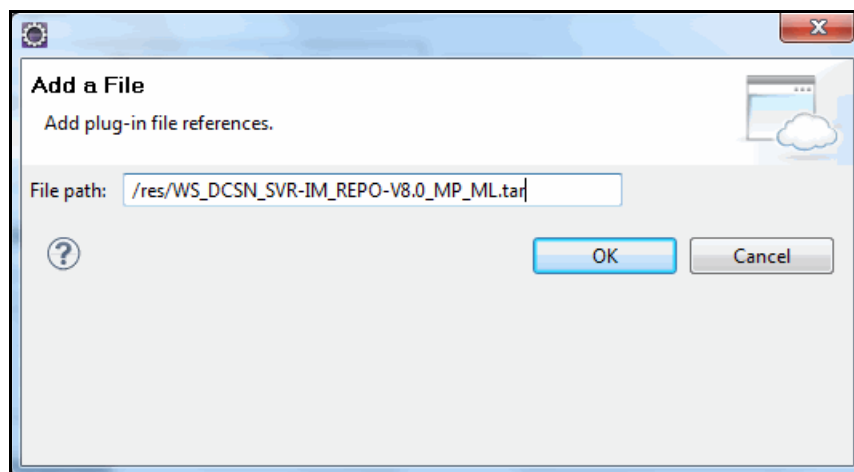


Figure 6-8 Add a file

- h. Add the following software installation file names to the plug-in files:
 - /res/WS_DCSN_SVR-IM_REPO-V8.0_MP_ML.tar
 - /res/CZM9KML.zip
 - /res/CZM9LML.zip
 - /res/CZM9MML.zip
 - /res/CZVG4ML.zip
 - /res/IBM_INSTALLATION_MGR_V1.5.3_LIN_ML.zip
 - /res/DB2_ESE_10_Linux_x86-64.tar.gz
 - /res/CZM91ML.zip
 - /res/CZM94ML.zip
 - /res/CZM95ML.zip
 - /res/CZMR9ML.zip
- i. Download the Business Rule Application pattern type software installation files listed in Table A-1 on page 178. Replace the files in the `patternype.ruleexecution-1.0.0.0.tgz\files\res` folder with the files that you downloaded.
- j. Add parameters by clicking **Add** in the Plug-in Parameters section.
- k. Add content to the plug-in parameters, as shown in Figure 6-9.

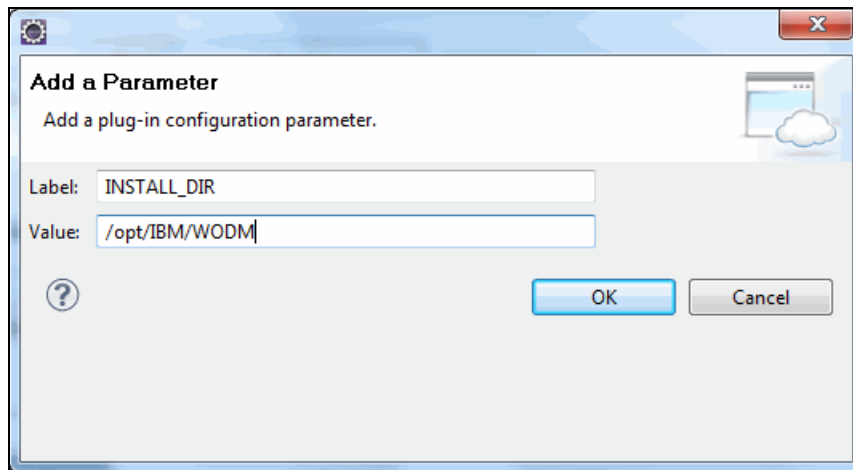


Figure 6-9 Add a parameter

- l. Enter the Labels and Values shown in Table 6-1.

Table 6-1 Parameter values

Label	Value
INSTALL_DIR	/opt/IBM/WODM
MINIMUM_MEMORY	512
PORT	9080
MINIMUM_DISK	300

- m. Click **OK**.
- n. Switch to the **config.json** tab and add content to the packages. In packages, you can use parameters defined in params in this format: "\$parametername".

Table 6-2 lists the packages, parts, and scripts defined in the rule plug-in. Remove the default package and add these packages to the config.json file. Example 6-1 on page 95 illustrates the config.json file.

Table 6-2 Rule plug-in packages, parts, and scripts

Package	Part	Script	Function
RESDB2Install	db2install	Part db2install install.py	Download the DB2 installation file from storehouse
		Role DB2Install lifecycle scripts	Install DB2
RESDB2Config	resdb2	Role RESDB2 lifecycle scripts	Configure the rule database
RESWASInstall	wasinstall	Part wasinstall install.py	Download the WebSphere Application Server installation file from the storehouse and install WebSphere Application Server
		Role WASStandalone lifecycle scripts	Configure the WebSphere Application Server stand-alone profile
		Role WASDmgr lifecycle scripts	Configure the WebSphere Application Server cluster Deployment Manager
		Role WASCustom lifecycle scripts	Configure the WebSphere Application Server Custom node
RESInstall	resinstall	Part RESInstall install.py	Download the WebSphere Decision Server installation file from the storehouse and install WebSphere Decision Server
RESConfigure	resConfigure	Role RESStandalone lifecycle scripts	Deploy WebSphere Decision Server application to WebSphere Application Server stand-alone
		Role RESDmgr lifecycle scripts	Deploy WebSphere Decision Server application to the WebSphere Application Server cluster
RESIHSInstall	ihinstall	Part ihinstall install.py	Download IBM WebSphere Application Server Supplements from storehouse
		Role IHSInstall lifecycle scripts	Install IBM HTTP Server

```
{
  "name": "plugin.com.ibm.wodm.res",
  "version": "1.0.0.0",
  "patterntypes": {
    "primary": {
      "patterntype.ruleexecution": "1.0"
    }
  },
  "packages": {
    "RESDB2Install": [
      {
        "requires": {
          "arch": "x86_64"
        },
        "parts": [
          {
            "part": "parts\\db2install.tgz",
            "parms": {
              "installDir": "$INSTALL_DIR"
            }
          }
        ]
      }
    ],
    "RESDB2Config": [
      {
        "requires": {
          "arch": "x86_64"
        },
        "parts": [
          {
            "part": "parts\\resdb2.tgz"
          }
        ]
      }
    ],
    "RESIHSInstall": [
      {
        "requires": {
          "arch": "x86_64"
        },
        "parts": [
          {
            "part": "parts\\ihsinstall.tgz",
            "parms": {
              "installDir": "$INSTALL_DIR"
            }
          }
        ]
      }
    ],
    "RESWASInstall": [
      {
        "requires": {
```

```

        "arch": "x86_64"
    },
    "parts": [
        {
            "part": "parts\\wasinstall.tgz",
            "parms": {
                "installDir": "$INSTALL_DIR"
            }
        }
    ]
},
"RESInstall": [
    {
        "requires": {
            "arch": "x86_64"
        },
        "parts": [
            {
                "part": "parts\\resinstall.tgz",
                "parms": {
                    "installDir": "$INSTALL_DIR"
                }
            }
        ]
    }
],
"RESConfigure": [
    {
        "requires": {
            "arch": "x86_64"
        },
        "parts": [
            {
                "part": "parts\\resconfigure.tgz",
                "parms": {
                    "installDir": "$INSTALL_DIR"
                }
            }
        ]
    }
],
},
"parms": {
    "MINIMUM_MEMORY": 512,
    "PORT": 9080,
    "MINIMUM_DISK": 300,
    "INSTALL_DIR": "\\opt\\IBM\\WODM"
},
"files": [
    "\\res\\CZM9KML.zip",
    "\\res\\CZM9LML.zip",
    "\\res\\CZM9MML.zip",
    "\\res\\CZVG4ML.zip",
    "\\res\\IBM_INSTALLATION_MGR_V1.5.3_LIN_ML.zip",

```

```

        "\res\DB2_ESE_10_Linux_x86-64.tar.gz",
        "\res\WS_DCSN_SVR-IM_REPO-V8.0_MP_ML.tar",
        "\res\CZM91ML.zip",
        "\res\CZM94ML.zip",
        "\res\CZM95ML.zip",
        "\res\CZXR9ML.zip"
    ]
}

```

5. Update the metadata.json plug-in metadata file:
 - a. Create a component by clicking **Add**.
 - b. Select **component** as the Type, and click **OK** (see Figure 6-10).

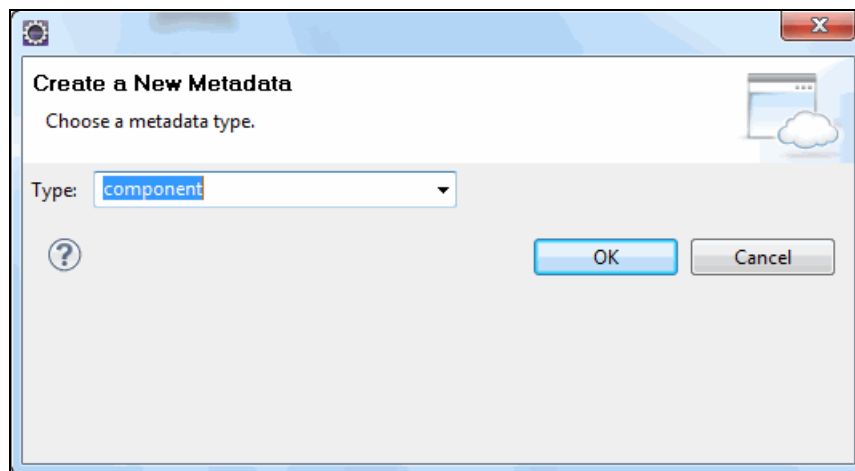


Figure 6-10 Create a new metadata

- c. Change the Component name to ruleApp.
 - d. Add a message to the plugin/appmodel/messages.json file.

You can add different language messages to individual messages.json files, which are stored in different language folders, such as de and en, under the locales folder (see Figure 6-11).

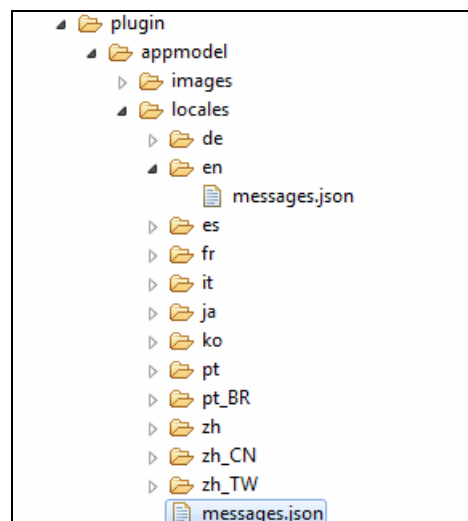


Figure 6-11 The messages.json file location

Example 6-2 illustrates the messages.json file.

Example 6-2 The messages.json file

```
{
  RULEAPP_LABEL:"Rule Application (WebSphere Operational Decision
  Management)",
  RULEAPP_DESCRIPTION:"A rule application (WebSphere Operational Decision
  Management) cloud component represents a managed business rule execution
  platform.",
  RULEAPP_ARCHIVE_LABEL:"Rule file",
  RULEAPP_ARCHIVE_DESCRIPTION:"Specifies the rule (*.jar) to be uploaded."
}
```

- e. In the Metadata details, enter RULEAPP_LABEL as the Label, RULEAPP_DESCRIPTION as the Description, and application as the Category.
- f. Add the images folder to the plugin/appmodel folder, and add image files to the images folder (see Figure 6-12).

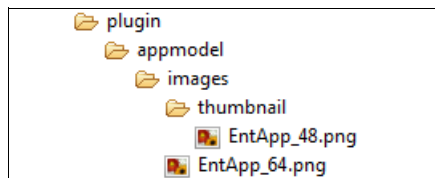


Figure 6-12 Images folder and image file

- g. In the Metadata details, set Images(64*64) to appmodel/images/EntApp_64.png, and set Thumbnail(48*48) to appmodel/images/thumbnail/EntApp_48.png.
- h. Create an attribute by clicking **Add** in the Attributes section.
- i. Select **file** as the Attribute type and click **OK** (see Figure 6-13).

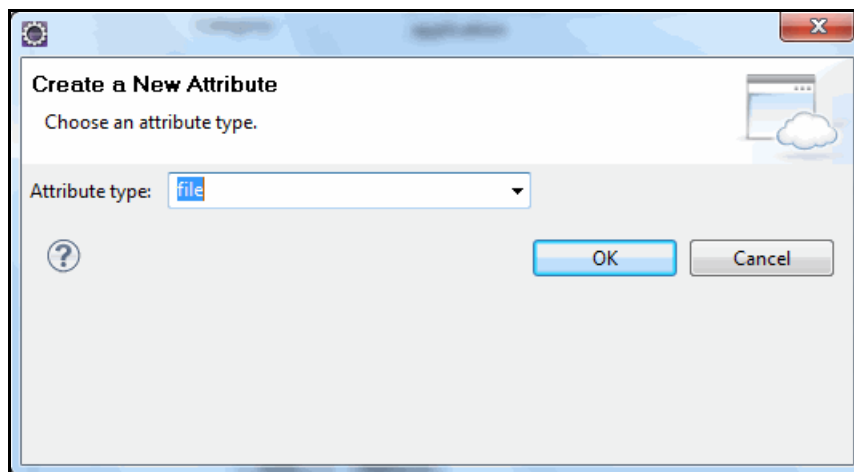


Figure 6-13 The Create a New Attribute window

- j. In the Attributes section, enter archive as the ID, RULEAPP_ARCHIVE_LABEL as the Label, and RULEAPP_ARCHIVE_DESCRIPTION as the Description. Select the **Required** check box (see Figure 6-14 on page 99).

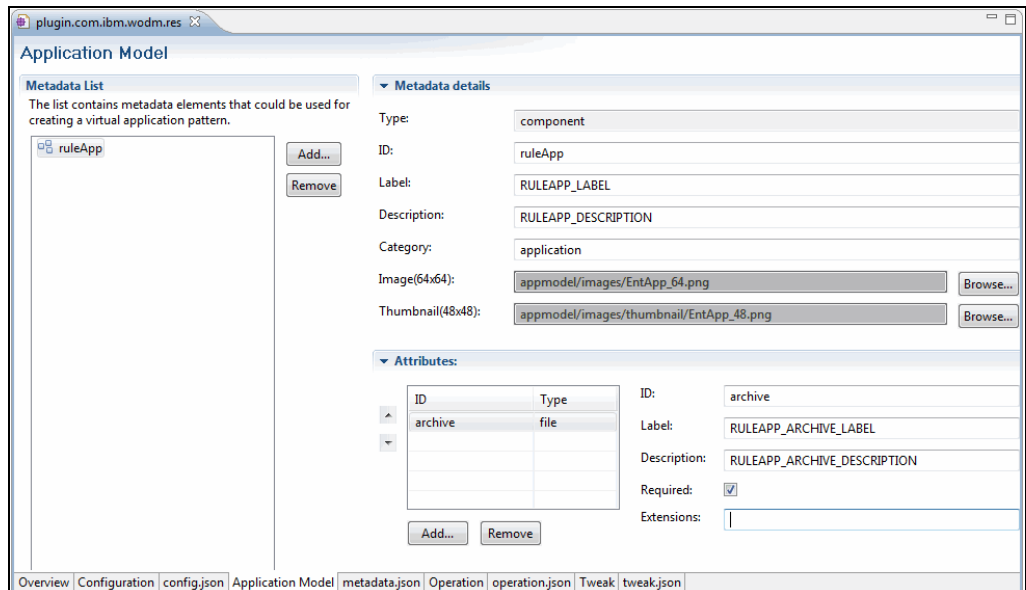


Figure 6-14 Component details

Example 6-3 shows a sample metadata.json file.

Example 6-3 Sample metadata.json file

```
[
  {
    "id": "ruleApp",
    "type": "component",
    "label": "RULEAPP_LABEL",
    "description": "RULEAPP_DESCRIPTION",
    "category": "application",
    "thumbnail": "appmodel\\images\\thumbnail\\EntApp_48.png",
    "image": "appmodel\\images\\EntApp_64.png",
    "attributes": [
      {
        "id": "archive",
        "type": "file",
        "required": true,
        "label": "RULEAPP_ARCHIVE_LABEL",
        "description": "RULEAPP_ARCHIVE_DESCRIPTION"
      }
    ]
  }
]
```

- k. Add Service-Component: OSGI-INF/res.xml to the MANIFEST.MF tab of the plugin.com.ibm.wodm.res file (see Figure 6-15).

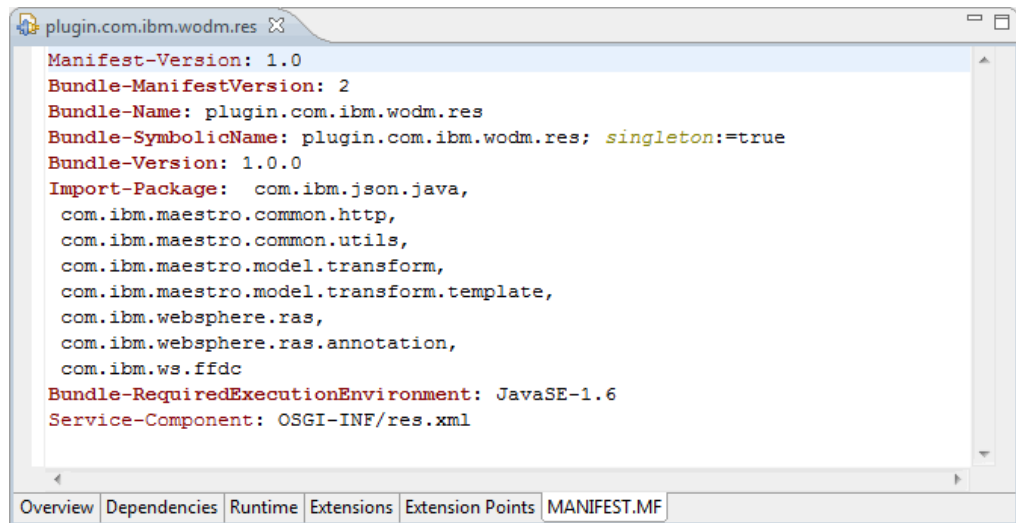


Figure 6-15 The MANIFEST.MF tab

6. Create a res.xml file in the OSGI-INF folder (see Example 6-4):
- Enter ruleApp as the component name.
 - Enter com.ibm.maestro.model.transform.res.RESTransformer as the implementation class in the res.xml file.

Example 6-4 The res.xml file

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
name="ruleApp">
  <implementation
class="com.ibm.maestro.model.transform.res.RESTransformer"/>
<service>
<provide interface="com.ibm.maestro.model.transform.TopologyProvider"/>
</service>
</scr:component>
```

6.1.2 Define a transformer

This section uses the rule stand-alone topology as an example of how to create topology in a transformer.

Figure 6-16 on page 101 shows the rule stand-alone topology.

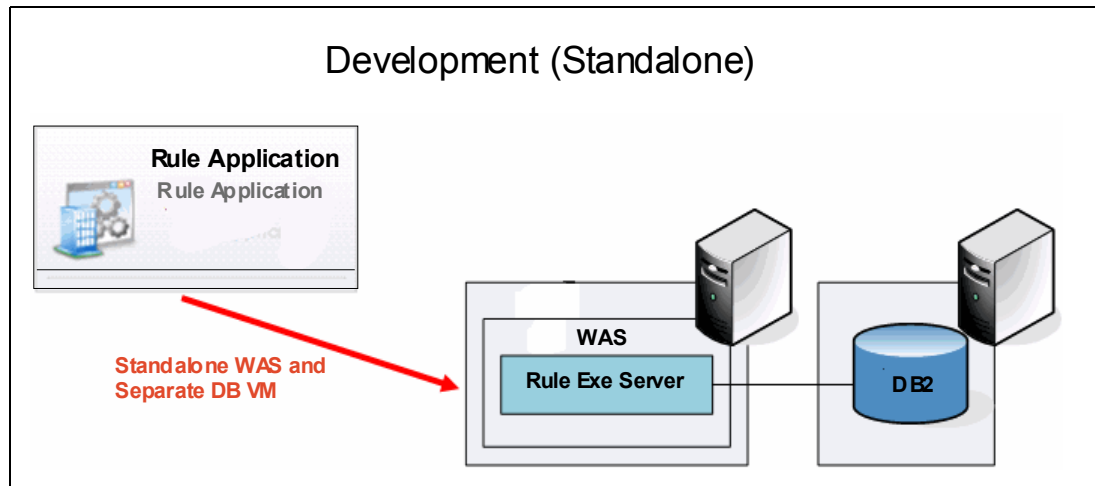


Figure 6-16 Rule stand-alone topology

Figure 6-17 shows a sample rule stand-alone topology.

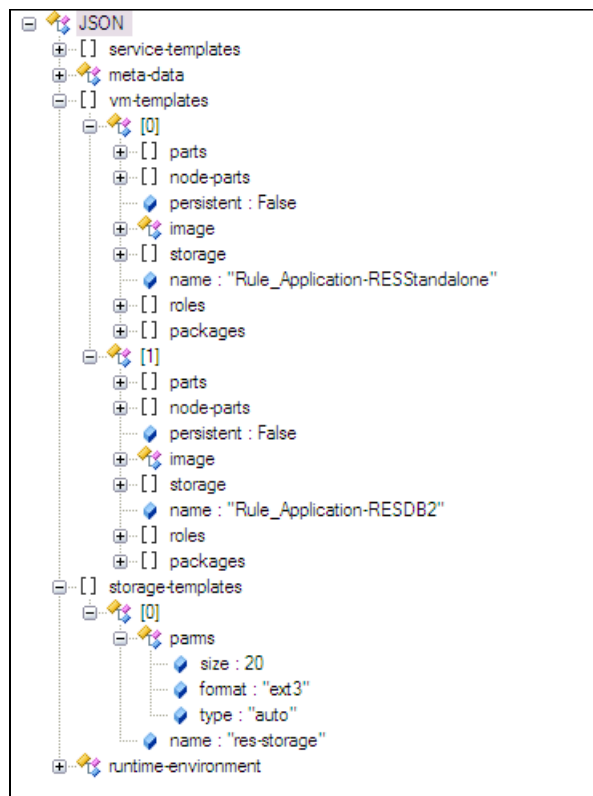


Figure 6-17 The topology.json rule stand-alone topology

Table 6-3 lists the content to add to the topology.json file.

Table 6-3 Rule stand-alone topology content

vm-template name	packages	roles	storage
xxxRuleComponent-RESStandalone	RESWASInstall RESInstall RESConfigure	WASStandalone RESStandalone	res-storage

vm-template name	packages	roles	storage
xxxRuleComponent-RESDB2	RESDB2Install RESDB2Config	DB2Install RESDB2	res-storage

Figure 6-18 shows role dependency in a rule stand-alone topology.

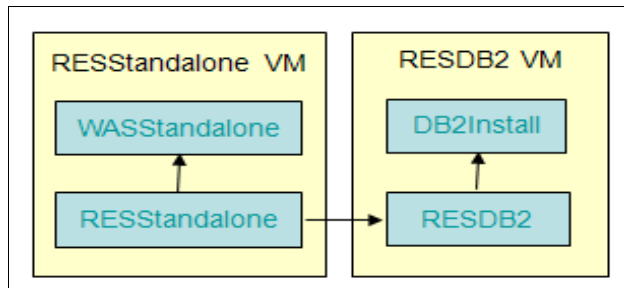


Figure 6-18 Role dependency in stand-alone topology

Figure 6-19 shows the role dependency structure in the file system.



Figure 6-19 Role dependency

To create a transformer, follow these steps:

1. Create a `RESTRansformer` transformer class in the `com.ibm.maestro.model.transform.res` package, and set `RESTRansformer` to extend `com.ibm.maestro.model.transform.TopologyProvider`.
2. Override the `transformComponent` method. Create a topology object as the return object of the `transformComponent` method. Add vm-templates to the topology object to define the VM property (see Example 6-5 and Example 6-6).

Example 6-5 Define constants in Constants.java

```
public static final String VM_TEMPLATE = "vm-templates";
```

Example 6-6 Create topology object in RESTRansformer.java

```
JSONObject templates = new JSONObject();  
JSONArray vmTemplates = new JSONArray();  
templates.put(Constants.VM_TEMPLATE, vmTemplates);  
return templates;
```

3. Add storage-templates to the topology object to add new disks to the VM later.

Figure 6-20 shows a storage-template sample, which is also illustrated in Example 6-7 and Example 6-8.

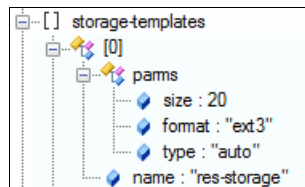


Figure 6-20 A storage-template sample

Example 6-7 Define constants in Constants.java

```
public static final String STORAGE_TEMPLATE = "storage-templates";  
public static final String NAME = "name";  
public static final String RES_STORAGE = "res-storage";  
public static final String PARMS = "parms";  
public static final String SIZE = "size";  
public static final String FORMAT = "format";  
public static final String TYPE = "type";
```

Example 6-8 Create storage templates in RESTRansformer.java

```
JSONArray storageTemplates = new JSONArray();  
templates.put(Constants.STORAGE_TEMPLATE, storageTemplates);  
JSONObject storageTemplate = new JSONObject();  
storageTemplates.add(storageTemplate);  
storageTemplate.put(Constants.NAME, Constants.RES_STORAGE);  
JSONObject storageParms = new JSONObject();  
storageTemplate.put(Constants.PARMS, storageParms);  
storageParms.put(Constants.SIZE, 20);  
storageParms.put(Constants.FORMAT, Constants.EXT3);  
storageParms.put(Constants.TYPE, Constants.AUTO);
```

4. Define the vm-template rule stand-alone topology. Put the vm-template pattern engine transform into one or multiple VMs with the same configuration:
 - a. Define the vm-template name (see Example 6-9).
 - b. Add packages to the vm-template (see Example 6-10).

Example 6-9 Define constants in Constants.java

```
public static final String NAME = "name";
public static final String HYPHEN = "-";
public static final String PACKAGES = "packages";
public static final String PACKAGE_WAS_INSTALL = "RESWASInstall";
public static final String PACKAGE_RES_INSTALL = "RESInstall";
public static final String PACKAGE_RES_CONFIGURE = "RESConfigure";
```

Example 6-10 Define vm-template and add packages in RSTransformer.java

```
JSONObject vm = new JSONObject();
vmTemplates.add(vm);
vm.put(Constants.NAME, vmTemplateNamePrefix + Constants.HYPHEN +
Constants.ROLE_RES_STANDALONE_NAME);
JSONArray packages = new JSONArray();
vm.put(Constants.PACKAGES, packages);
packages.add(Constants.PACKAGE_WAS_INSTALL);
packages.add(Constants.PACKAGE_RES_INSTALL);
packages.add(Constants.PACKAGE_RES_CONFIGURE);
```

- c. In the vm-template (see Example 6-11 on page 105 and Example 6-12 on page 105), define storage that refers to storageTemplates to add new disks to VMs.

Figure 6-21 shows storage that refers to a storage-template.

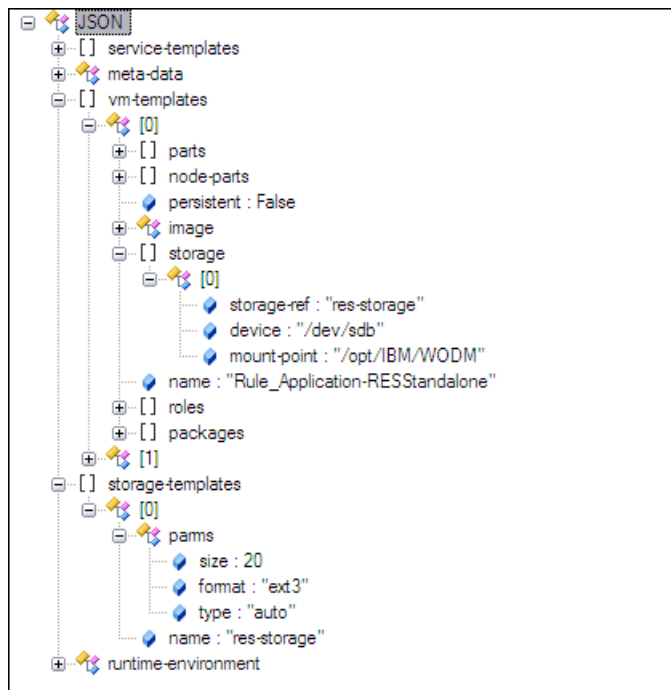


Figure 6-21 Storage referring to a storage template

Example 6-11 Define constants in Constants.java

```
public static final String STORAGE = "storage";
public static final String STORAGE_REF = "storage-ref";
public static final String RES_STORAGE = "res-storage";
public static final String MOUNT_POINT = "mount-point";
public static final String PARMS_INSTALL_DIR = "INSTALL_DIR";
```

Example 6-12 Add a new disk to a VM in RESTransformer.java

```
JSONArray storages = new JSONArray();
vm.put(Constants.STORAGE, storages);
JSONObject storage = new JSONObject();
storages.add(storage);
storage.put(Constants.STORAGE_REF, Constants.RES_STORAGE);
// retrieve INSTALL_DIR parameter from config.json using
// getConfigParm(parameter name) method
storage.put(Constants.MOUNT_POINT, getConfigParm(Constants.PARMS_INSTALL_DIR));
```

- d. Add roles to the rule stand-alone vm-template, as shown in Figure 6-22.

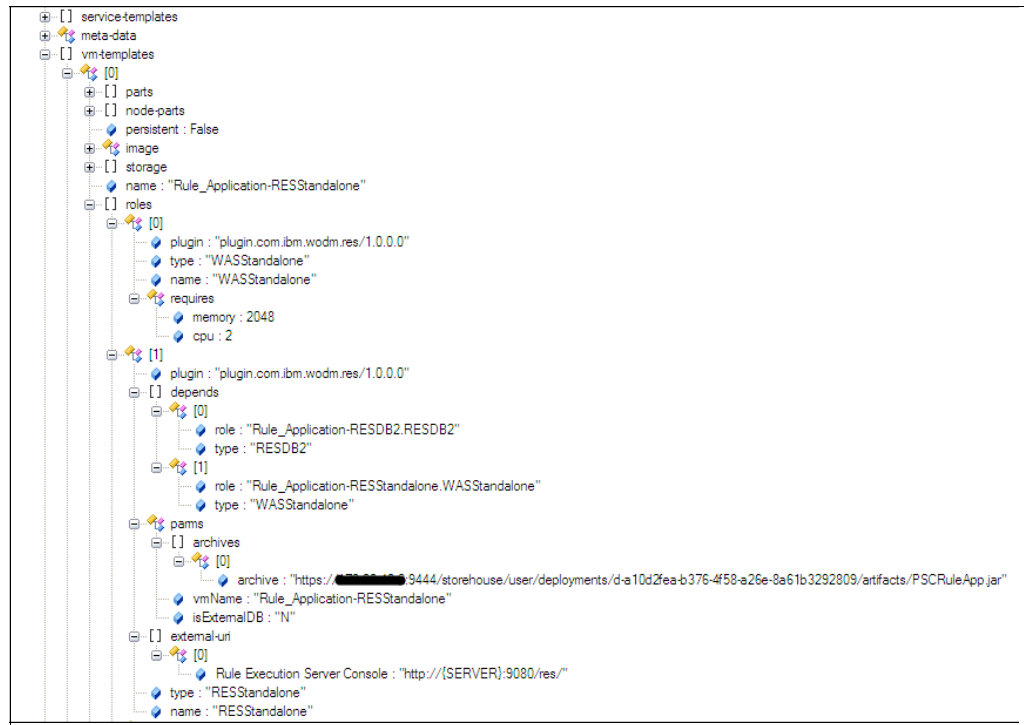


Figure 6-22 Rule stand-alone vm-template roles

Follow these steps to add roles:

- i. Add the WASStandalone role to the rule stand-alone vm-template. Define the role as WASStandalone (see Example 6-13 and Example 6-14).

Example 6-13 Define constants in Constants.java

```
public static final String ROLES = "roles";
public static final String PLUGIN = "plugin";
public static final String NAME = "name";
public static final String TYPE = "type";
```

Example 6-14 Define the WASStandalone role in RESTransformer.java

```
JSONArray roles = new JSONArray();
vm.put(Constants.ROLES, roles);
JSONObject roleWASStandalone = new JSONObject();
roles.add(roleWASStandalone);
roleWASStandalone.put(Constants.PLUGIN, getPluginScope());
roleWASStandalone.put(Constants.NAME, Constants.ROLE_WAS_STANDALONE_NAME);
roleWASStandalone.put(Constants.TYPE, Constants.ROLE_WAS_STANDALONE_NAME);
```

- ii. Define the VM CPU and memory in the requires property (see Example 6-15 and Example 6-16).

Example 6-15 Define constants in Constants.java

```
public static final String REQUIRES = "requires";
public static final String MEMORY = "memory";
public static final String CPU = "cpu";
int memoryCustomNode = 2;
int memoryDB = 2048;
```

Example 6-16 Define CPU and memory in RESTransformer.java

```
JSONObject requires = new JSONObject();
roleWASStandalone.put(Constants.REQUIRES, requires);
requires.put(Constants.MEMORY, memoryCustomNode);
requires.put(Constants.CPU, cpuCustomNode);
```

- iii. Add the RESStandalone role to the stand-alone rule vm-template. Define the RESStandalone role (see Example 6-17 and Example 6-18).

Example 6-17 Define constants in Constants.java

```
public static final String PLUGIN = "plugin";
public static final String NAME = "name";
public static final String TYPE = "type";
public static final String ROLE_RES_STANDALONE_NAME = "RESStandalone";
public static final String EXTERNAL_URL = "external-uri";
public static final String RESCONSOLE = "http://{SERVER}:9080/res/";
```

Example 6-18 Define the RESStandalone role in RESTransformer.java

```
JSONObject roleRESStandalone = new JSONObject();
roles.add(roleRESStandalone);
roleRESStandalone.put(Constants.PLUGIN, getPluginScope());
```

```

roleRESStandalone.put(Constants.NAME,
Constants.ROLE_RES_STANDALONE_NAME);
roleRESStandalone.put(Constants.TYPE,
Constants.ROLE_RES_STANDALONE_NAME);
roleRESStandalone.put(Constants.EXTERNAL_URL, generateExternalUris());

private Object generateExternalUris() {
JSONArray externalUris = new JSONArray();
final String consoleUrl = Constants.RESCONSOLE;
JSONObject csUri = new JSONObject();
csUri.put(Constants.RES_CONSOLE, consoleUrl);
externalUris.add(csUri);
return externalUris;
}

```

- iv. Define the role parameters (see Example 6-19 and Example 6-20).

Example 6-19 Define constants in Constants.java

```

public static final String PARMS = "parms";
public static final String ARCHIVES = "archives";
public static final String ARCHIVE = "archive";
public static final String VM_NAME = "vmName";
public static final String HYPHEN = "-";
public static final String ROLE_RES_STANDALONE_NAME = "RESStandalone";
public static final String ATTRIBUTES = "attributes";

```

Example 6-20 Define role parameters in RESTransformer.java

```

JSONObject attributes = (JSONObject)
applicationComponent.get(Constants.ATTRIBUTES);
JSONObject parms = new JSONObject();
roleRESStandalone.put(Constants.PARMS, parms);
JSONArray archiveArray = new JSONArray();
parms.put(Constants.ARCHIVES, archiveArray);
Object archive = attributes.get(Constants.ARCHIVE);
if (archive != null) {
JSONObject archiveObject = new JSONObject();
archiveObject.put(Constants.ARCHIVE, applicationUrl + archive);
archiveArray.add(archiveObject);
}
parms.put(Constants.VM_NAME, vmTemplateNamePrefix + Constants.HYPHEN +
Constants.ROLE_RES_STANDALONE_NAME);

```

- v. Define the constants (see Example 6-21).

Example 6-21 Define constants in Constants.java

```

public static final String DEPENDS = "depends";
public static final String HYPHEN = "-";
public static final String ROLE_RES_DB2_NAME = "RESDB2";
public static final String ROLE = "role";
public static final String TYPE = "type";
public static final String ROLE_RES_STANDALONE_NAME = "RESStandalone";
public static final String POINT = ".";

```

- vi. Define the role dependency and add role dependency from RESandalone to RESDB2 and WASandalone (see Example 6-22).

Example 6-22 Define role dependency in RESTransformer.java

```
JSONArray dependsStandalone = new JSONArray();
roleRESstandalone.put(Constants.DEPENDS, dependsStandalone);
JSONObject dependDB = new JSONObject();
dependsStandalone.add(dependDB);
String dependDBRole = vmTemplateNamePrefix + Constants.HYPHEN +
Constants.ROLE_RES_DB2_NAME + Constants.POINT +
Constants.ROLE_RES_DB2_NAME;
dependDB.put(Constants.ROLE, dependDBRole);
dependDB.put(Constants.TYPE, Constants.ROLE_RES_DB2_NAME);
JSONObject dependWASstandalone = new JSONObject();
dependsStandalone.add(dependWASstandalone);
String dependWASstandaloneRole = vmTemplateNamePrefix + Constants.HYPHEN
+ Constants.ROLE_RES_STANDALONE_NAME + Constants.POINT +
Constants.ROLE_WAS_STANDALONE_NAME;
dependWASstandalone.put(Constants.ROLE, dependWASstandaloneRole);
dependWASstandalone.put(Constants.TYPE,
Constants.ROLE_WAS_STANDALONE_NAME);
```

- 5. Define the db2 vm-template rule:
 - a. Define the vm-template name (see Example 6-23).

Example 6-23 Define constants in Constants.java

```
public static final String NAME = "name";
public static final String HYPHEN = "-";
public static final String ROLE_RES_DB2_NAME = "RESDB2";
public static final String PACKAGES = "packages";
public static final String PACKAGE_DB2_INSTALL = "RESDB2Install";
public static final String PACKAGE_RES_DB2 = "RESDB2Config";
```

- b. Add packages to the vm-template (see Example 6-24).

Example 6-24 Define the vm-template name and add packages in RESTransformer.java

```
JSONObject vmDB = new JSONObject();
vmTemplates.add(vmDB);
vmDB.put(Constants.NAME, vmTemplateNamePrefix + Constants.HYPHEN +
Constants.ROLE_RES_DB2_NAME);
JSONArray packagesDB = new JSONArray();
vmDB.put(Constants.PACKAGES, packagesDB);
packagesDB.add(Constants.PACKAGE_DB2_INSTALL);
packagesDB.add(Constants.PACKAGE_RES_DB2);
```

- c. In the vm-template, define storage that refers to storage templates to add new disks to the VM (see Example 6-25 and Example 6-26 on page 109).

Example 6-25 Define constants in Constants.java

```
public static final String STORAGE = "storage";
public static final String STORAGE_REF = "storage-ref";
public static final String RES_STORAGE = "res-storage";
```

```
public static final String MOUNT_POINT = "mount-point";
public static final String PARMS_INSTALL_DIR = "INSTALL_DIR";
```

Example 6-26 Define storage in vm-template in RESTransformer.java

```
JSONArray storagesDB = new JSONArray();
vmDB.put(Constants.STORAGE, storagesDB);
JSONObject storageDB = new JSONObject();
storagesDB.add(storageDB);
storageDB.put(Constants.STORAGE_REF, Constants.RES_STORAGE);
storageDB.put(Constants.MOUNT_POINT,
getConfigParm(Constants.PARMS_INSTALL_DIR));
```

- d. Add roles to the rule db2 vm-template (as shown in Figure 6-23):

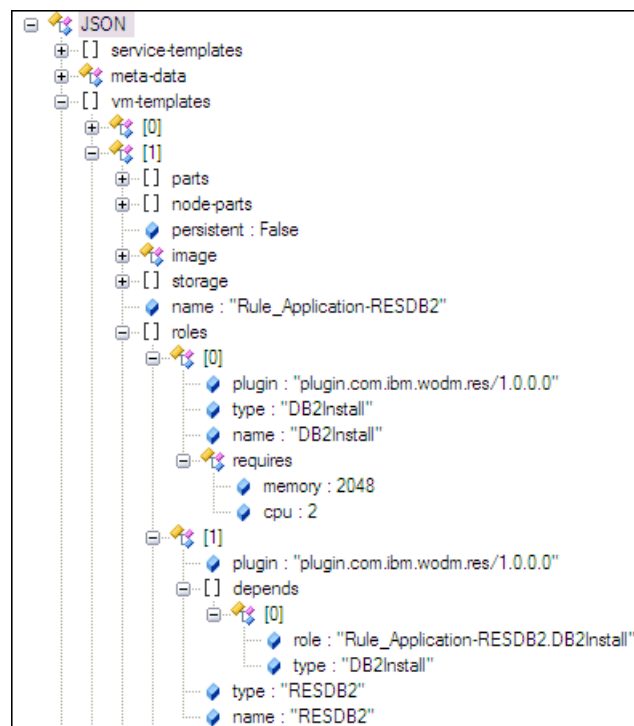


Figure 6-23 The db2 rule vm-template roles

- i. Add the DB2Install role to the rule db2 vm-template. Define the DB2Install role (see Example 6-27 and Example 6-28).

Example 6-27 Define constants in Constants.java

```
public static final String ROLES = "roles";
public static final String PLUGIN = "plugin";
public static final String NAME = "name";
public static final String ROLE_DB2_INSTALL_NAME = "DB2Install";
public static final String TYPE = "type";
```

Example 6-28 Define the DB2Install role in RESTransformer.java

```
JSONArray rolesDB = new JSONArray();
```

```
vmDB.put(Constants.ROLES, rolesDB);
JSONObject roleDB = new JSONObject();
rolesDB.add(roleDB);
roleDB.put(Constants.PLUGIN, getPluginScope());
roleDB.put(Constants.NAME, Constants.ROLE_DB2_INSTALL_NAME);
roleDB.put(Constants.TYPE, Constants.ROLE_DB2_INSTALL_NAME);
```

- ii. Define the VM CPU and memory in the requires property (see Example 6-29 and Example 6-30).

Example 6-29 Define constants in Constants.java

```
public static final String REQUIRES = "requires";
public static final String MEMORY = "memory";
public static final String CPU = "cpu";
int cpuDB = 2;
int memoryDB = 2048;
```

Example 6-30 Define VM CPU and memory in RESTransformer.java

```
JSONObject requiresDB = new JSONObject();
roleDB.put(Constants.REQUIRES, requiresDB);
requiresDB.put(Constants.MEMORY, memoryDB);
requiresDB.put(Constants.CPU, cpuDB);
```

- iii. Add the RESDB2 role to the rule stand-alone vm-template (see Example 6-31 and Example 6-32).

Example 6-31 Define constants in Constants.java

```
public static final String PLUGIN = "plugin";
public static final String NAME = "name";
public static final String TYPE = "type";
public static final String ROLE_RES_DB2_NAME = "RESDB2";
```

Example 6-32 Define the RESDB2 role in RESTransformer.java

```
JSONObject roleRESDB2 = new JSONObject();
rolesDB.add(roleRESDB2);
roleRESDB2.put(Constants.PLUGIN, getPluginScope());
roleRESDB2.put(Constants.NAME, Constants.ROLE_RES_DB2_NAME);
roleRESDB2.put(Constants.TYPE, Constants.ROLE_RES_DB2_NAME);
```

- iv. Define the role dependency, and add the role dependency from RESDB2 to DB2Install (see Example 6-33 and Example 6-34).

Example 6-33 Define constants in Constants.java

```
public static final String DEPENDS = "depends";
public static final String HYPHEN = "-";
public static final String ROLE_RES_DB2_NAME = "RESDB2";
public static final String POINT = ".";
public static final String ROLE_DB2_INSTALL_NAME = "DB2Install";
public static final String ROLE = "role";
public static final String TYPE = "type";
public static final String ROLE_DB2_INSTALL_NAME = "DB2Install";
```

Example 6-34 Add role dependency from RESDB2 to DB2Install in RESTransformer.java

```
JSONArray resdb2Depends= new JSONArray();
roleRESDB2.put(Constants.DEPENDS, resdb2Depends);
JSONObject dependsRESDB2Install = new JSONObject();
resdb2Depends.add(dependsRESDB2Install);
String dependDB2InstallRole = vmTemplateNamePrefix + Constants.HYPHEN +
Constants.ROLE_RES_DB2_NAME + Constants.POINT +
Constants.ROLE_DB2_INSTALL_NAME;
dependsRESDB2Install.put(Constants.ROLE, dependDB2InstallRole);
dependsRESDB2Install.put(Constants.TYPE, Constants.ROLE_DB2_INSTALL_NAME);
```

Define the part and role lifecycle scripts

This section introduces how to define the part and role lifecycle scripts.

Figure 6-24 lists the rule plug-in parts and role lifecycle script structures.

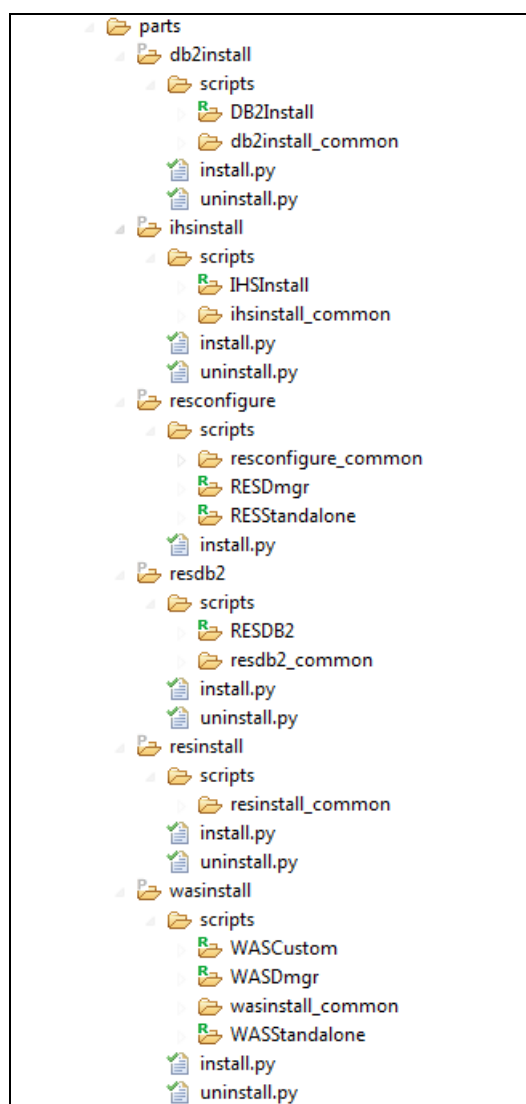


Figure 6-24 Rule plug-in part and role lifecycle script structure

Table 6-4 lists the packages, parts, and scripts defined in the rule plug-in.

Table 6-4 Rule plug-in package, part, and script

Package	Part	Script	Function
RESDB2Install	db2install	Part db2install install.py	Download DB2 installation file from storehouse
		Role DB2Install lifecycle scripts	Install DB2
RESDB2Config	resdb2	Role RESDB2 lifecycle scripts	Configure the rule database

Package	Part	Script	Function
RESWASInstall	wasinstall	Part wasinstall install.py	Download the WebSphere Application Server installation file from the storehouse and install WebSphere Application Server
		Role WASStandalone lifecycle scripts	Configure WebSphere Application Server stand-alone profile
		Role WASDmgr lifecycle scripts	Configure WebSphere Application Server cluster Deployment Manager
		Role WASCustom lifecycle scripts	Configure WebSphere Application Server Custom node
RESInstall	resinstall	Part RESInstall install.py	Download the WebSphere Decision Server installation file from the storehouse and install WebSphere Decision Server
RESConfigure	resConfigure	Role RESStandalone lifecycle scripts	Deploy WebSphere Decision Server application to WebSphere Application Server stand-alone
		Role RESDmgr lifecycle scripts	Deploy WebSphere Decision Server application to WebSphere Application Server Cluster
RESIHSInstall	ihinstall	Part ihsinstall install.py	Download IBM WebSphere Application Server Supplements from storehouse
		Role IHSInstall lifecycle scripts	Install IBM HTTP Server

Figure 6-25 shows the part and role lifecycle script run sequence in the rule standalone topology.

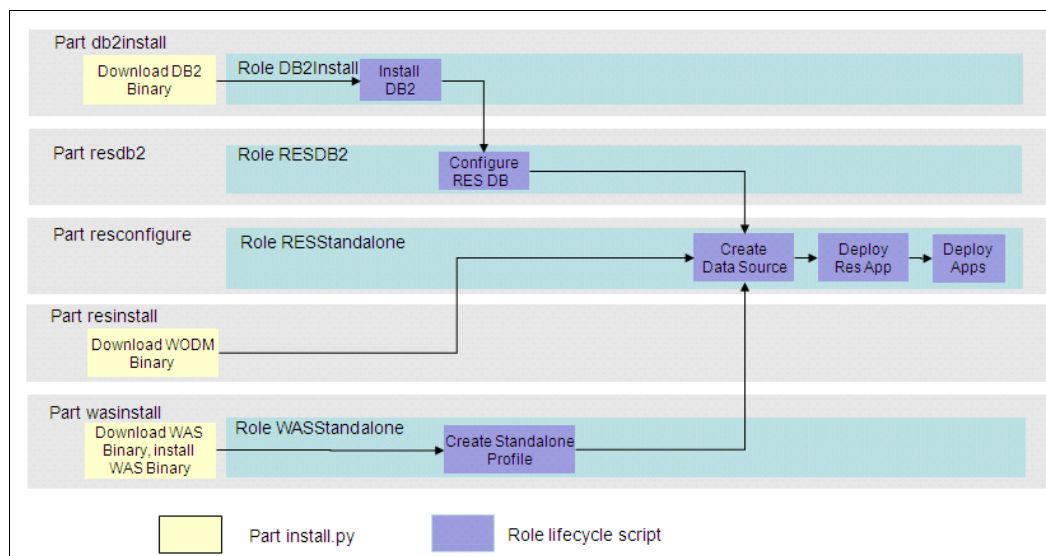


Figure 6-25 Part and role lifecycle script run sequence

To add the part and role lifecycle scripts, follow these steps:

1. Define the db2install part:
 - a. Create the part (see Example 6-35):
 - i. Create a part folder named db2install under the plugin/parts folder, which includes install.py.
 - ii. Add content to the plugin/parts/db2install/install.py part lifecycle script.

Example 6-35 The plugin/parts/db2install/install.py script

```
maestro.install_scripts('scripts')

installDir = maestro.parms['installDir']
maestro.node['parts']['RESDB2Install']['installDir'] = installDir
downloadDir = installDir + '/download'

# download DB2 installation file
installerUrl = urlparse.urljoin(maestro.filesurl,
'/storehouse/admin/files/res/DB2_ESE_10_Linux_x86-64.tar.gz')
maestro.download(installerUrl, downloadDir +
'/DB2_ESE_10_Linux_x86-64.tar.gz')
```

- b. Define the role lifecycle script (see Example 6-36 on page 115):
 - i. Add the DB2Install role to the plugin/parts/db2install/scripts folder, and add install.py and start.py to the DB2Install role folder.
 - ii. Add content to the role lifecycle script under the plugin/parts/db2install/scripts/DB2Install folder.
 - iii. Add content to the role lifecycle script plugin/parts/db2install/scripts/DB2Install/install.py.

Example 6-36 The plugin/parts/db2install/scripts/DB2Install/install.py role lifecycle script

```
installDir = maestro.node['parts']['RESDB2Install']['installDir']
scriptCommonFolder = scriptdir + '/db2install_common'
downloadDir = installDir + '/download'
softwareDir = installDir + '/software'

# install db2
scriptFolderDir = scriptdir + '/db2install_common/installDB2/'
rc = maestro.trace_call(logger, [scriptFolderDir + 'installDB2.sh',
downloadDir, scriptCommonFolder, softwareDir])
maestro.check_status(rc, 'install DB2 error')

# export db2 ip, port, username, password when db2 is installed
successfully
maestro.export['DB2InstallIP'] = maestro.node['instance']['public-ip']
maestro.export['DB2InstallPort'] = '50000'
maestro.export['DB2InstallUsername'] = 'db2inst1'
maestro.export['DB2InstallPassword'] = 'password'
```

- iv. Add content to the plugin/parts/db2install/scripts/DB2Install/start.py role lifecycle script (see Example 6-37).

Example 6-37 Content for plugin/parts/db2install/scripts/DB2Install/start.py

```
# open port through firewall
maestro.firewall.open_tcpin(dport=50000)
maestro.firewall.open_tcpout(dport=50000)
maestro.firewall.open_tcpin(dport=1521)
maestro.firewall.open_tcpout(dport=1521)

# change role status to RUNNING
maestro.role_status = 'RUNNING'
```

2. Define the resdb2 role:

- a. Create the resdb2 part folder under plugin/parts.
- b. Define the role lifecycle script:
 - i. Add the RESDB2 role to the plugin/parts/resdb2/scripts folder.
 - ii. Define the role dependency.
 - iii. Create a role dependency folder called DB2Install under plugin/parts/resdb2/scripts/RESDB2, because the role RESDB2 depends on the role DB2Install.
- iv. Add content to plugin/parts/resdb2/scripts/RESDB2/DB2Install/changed.py. If the RESDB2 role gets the parameter exported by the DB2Install role, RESDB2 starts to configure the database (see Example 6-38).

Example 6-38 Content in plugin/parts/resdb2/scripts/RESDB2/DB2Install/changed.py

```
myrole = None
if len(maestro.updated) >= 1:
    myrole = maestro.updated[0]
elif len(maestro.added) >= 1:
    myrole = maestro.added[0]
```

```

if myrole and existFlag():
    logger.debug('This is a restart operation')
elif myrole and maestro.deps[myrole]['DB2InstallIP'] and not existFlag():
    # config WebSphere Decision Server DB
    scriptdir = maestro.node['scriptdir']
    scriptCommonFolder = scriptdir + '/resdb2_common'
    scriptFolderDir = scriptdir + '/resdb2_common/configureDB2/'
    rc = maestro.trace_call(logger, [scriptFolderDir + 'configureDB2.sh',
    scriptCommonFolder])

    # exprt parameter
    dbParam = {}
    dbParam['dbip'] = maestro.deps[myrole]['DB2InstallIP']
    dbParam['dbport'] = maestro.deps[myrole]['DB2InstallPort']
    dbParam['dbusername'] = maestro.deps[myrole]['DB2InstallUsername']
    dbParam['dbpassword'] = maestro.deps[myrole]['DB2InstallPassword']
    maestro.export['DB2Param4RES'] = dbParam

    # change role status
    maestro.role_status = 'RUNNING'

```

3. Define the wasinstall part:

a. Create the part:

- i. Create the wasinstall part folder under plugin/parts, which includes install.py.
- ii. Add content to the plugin/parts/wasinstall/install.py part lifecycle script (see Example 6-39).

Example 6-39 Content in the plugin/parts/wasinstall/install.py part lifecycle script

```

installDir = maestro.parms['installDir']
if not 'RESWASInstall' in maestro.node['parts']:
    maestro.node['parts']['RESWASInstall'] = {}
maestro.node['parts']['RESWASInstall']['installDir'] = installDir

downloadDir = installDir + '/download'
softwareDir = installDir + '/software'

# download software installation file
installerUrl = urlparse.urljoin(maestro.filesurl,
'/storehouse/admin/files/res/IBM_INSTALLATION_MGR_V1.5.3_LIN_ML.zip')
maestro.download(installerUrl, downloadDir +
'/IBM_INSTALLATION_MGR_V1.5.3_LIN_ML.zip')

installerUrl = urlparse.urljoin(maestro.filesurl,
'/storehouse/admin/files/res/CZM9KML.zip')
maestro.download(installerUrl, downloadDir + '/CZM9KML.zip')

installerUrl = urlparse.urljoin(maestro.filesurl,
'/storehouse/admin/files/res/CZM9LML.zip')
maestro.download(installerUrl, downloadDir + '/CZM9LML.zip')

installerUrl = urlparse.urljoin(maestro.filesurl,
'/storehouse/admin/files/res/CZM9MML.zip')

```

```

maestro.download(installerUrl, downloadDir + '/CZM9MML.zip')

installerUrl = urlparse.urljoin(maestro.filesurl,
'/storehouse/admin/files/res/CZVG4ML.zip')
maestro.download(installerUrl, downloadDir + '/CZVG4ML.zip')

scriptCommonFolder = scriptdir + '/wasinstall_common'
logger.debug(scriptCommonFolder)

# install IBM Installation Manager
scriptFolderDir = scriptdir + '/wasinstall_common/installIM/'
rc = maestro.trace_call(logger, [scriptFolderDir + 'installIM.sh',
downloadDir, softwareDir])

# install WebSphere Application Server
scriptFolderDir = scriptdir + '/wasinstall_common/installWAS/'
rc = maestro.trace_call(logger, [scriptFolderDir +
'installWAS4Cluster.sh', downloadDir, scriptCommonFolder, softwareDir])

```

b. Define the role lifecycle script:

- i. Add the WASStandalone role to the plugin/parts/wasinstall/scripts folder, and add install.py and start.py to the WASStandalone role folder.
- ii. Add content to the role lifecycle script under the plugin/parts/wasinstall/scripts/WASStandalone folder.
- iii. Add content to the plugin/parts/wasinstall/scripts/WASStandalone/install.py role lifecycle script (see Example 6-40).

Example 6-40 Content in plugin/parts/wasinstall/scripts/WASStandalone/install.py

```

scriptdir = node['scriptdir']
moutPoint = maestro.node['parts']['RESWASInstall']['installDir']
softwareHome = moutPoint + '/software'
scriptCommonFolder = scriptdir + '/wasinstall_common'

# create WebSphere Application Server stand-alone profile
scriptFolderDir = scriptdir + '/wasinstall_common/installWAS/'
rc = maestro.trace_call(logger, [scriptFolderDir +
'createStandaloneProfile.sh', scriptCommonFolder, softwareHome])

```

- iv. Add content to the plugin/parts/wasinstall/scripts/WASStandalone/start.py role lifecycle script (see Example 6-41).

Example 6-41 Content in plugin/parts/wasinstall/scripts/WASStandalone/start.py

```

# open port through firewall
maestro.firewall.open_tcpin(dport=2809)
maestro.firewall.open_tcpout(dport=2809)
maestro.firewall.open_tcpin(dport=8879)
maestro.firewall.open_tcpout(dport=8879)
maestro.firewall.open_tcpin(dport=9100)
maestro.firewall.open_tcpout(dport=9100)
maestro.firewall.open_tcpin(dport=9401)
maestro.firewall.open_tcpout(dport=9401)
maestro.firewall.open_tcpin(dport=9403)

```

```

maestro.firewall.open_tcpout(dport=9403)
maestro.firewall.open_tcpin(dport=9402)
maestro.firewall.open_tcpout(dport=9402)
maestro.firewall.open_tcpin(dport=9060)
maestro.firewall.open_tcpout(dport=9060)
maestro.firewall.open_tcpin(dport=9080)
maestro.firewall.open_tcpout(dport=9080)
maestro.firewall.open_tcpin(dport=9043)
maestro.firewall.open_tcpout(dport=9043)
maestro.firewall.open_tcpin(dport=9443)
maestro.firewall.open_tcpout(dport=9443)
maestro.firewall.open_tcpin(dport=5060)
maestro.firewall.open_tcpout(dport=5060)
maestro.firewall.open_tcpin(dport=5061)
maestro.firewall.open_tcpout(dport=5061)
maestro.firewall.open_tcpin(dport=7276)
maestro.firewall.open_tcpout(dport=7276)
maestro.firewall.open_tcpin(dport=7286)
maestro.firewall.open_tcpout(dport=7286)
maestro.firewall.open_tcpin(dport=5558)
maestro.firewall.open_tcpout(dport=5558)
maestro.firewall.open_tcpin(dport=5578)
maestro.firewall.open_tcpout(dport=5578)
maestro.firewall.open_tcpin(dport=9633)
maestro.firewall.open_tcpout(dport=9633)
maestro.firewall.open_tcpin(dport=80)
maestro.firewall.open_tcpout(dport=80)
maestro.firewall.open_tcpin(dport=443)
maestro.firewall.open_tcpout(dport=443)
maestro.firewall.open_tcpin(dport=50000)
maestro.firewall.open_tcpout(dport=50000)
maestro.firewall.open_tcpin(dport=1521)
maestro.firewall.open_tcpout(dport=1521)

```

4. Define the reinstall part:

a. Create the part:

- i. Create the `reinstall` part folder under the `plugin/parts` folder, which includes `install.py`.
- ii. Add content to the `plugin/parts/reinstall/install.py` part lifecycle script (see Example 6-42).

Example 6-42 Content in plugin/parts/reinstall/install.py

```

installDir = maestro.parms['installDir']
maestro.node['parts']['RESInstall']['installDir'] = installDir
softwareDir = installDir + '/software'
downloadDir = installDir + '/download'

# download Websphere Decision Server installation file
installerUrl = urlparse.urljoin(maestro.filesurl,
'/storehouse/admin/files/res/WS_DCSN_SVR-IM_REPO-V8.0_MP_ML.tar')
maestro.download(installerUrl, downloadDir +
'/WS_DCSN_SVR-IM_REPO-V8.0_MP_ML.tar')

```

```
# install WebSphere Decision Server
scriptFolderDir = scriptdir + '/resinstall_common/installWODM/'
rc = maestro.trace_call(logger, [scriptFolderDir + 'installWODM.sh',
downloadDir, scriptCommonFolder, softwareDir, im_folder])
```

5. Define the resconfigure part:

- a. Create the resconfigure part folder under plugin/parts.
- b. Define the role lifecycle script (see Example 6-43):
 - i. Add the RESStandalone role in the plugin/parts/resconfigure/scripts folder, and add install.py to the RESStandalone role folder.
 - ii. Add content to the role lifecycle script under the plugin/parts/resconfigure/scripts/RESStandalone folder.
 - iii. Add content to the plugin/parts/resconfigure/scripts/RESStandalone/install.py role lifecycle script.

Example 6-43 Content in plugin/parts/resconfigure/scripts/RESStandalone/install.py

```
scriptdir = node['scriptdir']
archives = parms['archives']
installDir = maestro.node['parts']['RESConfigure']['installDir']
softwareDir = installDir + '/software'

# download rule application file from Storehouse
for archive in archives:
    archive_content = archive['archive']
    archive_name = archive_content.rsplit('/')[-1]
    archive_file = os.path.join(scriptdir, archive_name)
    maestro.download(archive_content, archive_file)
    role['archive_file'].append(archive_file)
```

- iv. Define the role dependency (see Example 6-44). RESStandalone depends on RESDB2. Create the RESDB2 role dependency folder under the plugin/parts/resdb2/scripts/RESStandalone folder. Add content to plugin/parts/resdb2/scripts/RESStandalone/RESDB2/changed.py.

Example 6-44 Content in .../RESStandalone/RESDB2/changed.py

```
myrole = None
if len(maestro.updated) >= 1:
    myrole = maestro.updated[0]
elif len(maestro.added) >= 1:
    myrole = maestro.added[0]

# check if role RESDB2 export DB2 parameters
if myrole and 'DB2Param4RES' in maestro.deps[myrole] and not existFlag():
    role['resdbIsReady'] = 'true'
    dbParam = maestro.deps[myrole]['DB2Param4RES']
    role['dbParam'] = dbParam
# check if role WASStandalone run successfully
if role.has_key('wasStandaloneIsReady') and role['wasStandaloneIsReady'] == 'true':
```

```

# configure WebSphere Decision Server and install rule application to
WebSphere Decision Server
configResStandalonePart(role['wasParam']['PROFILE_HOME'],role['wasParam']
['CELL_NAME'],role['wasParam']['NODE_NAME'],role['dbType'],
role['dbParam']['dbip'], role['dbParam']['dbport'],
role['dbParam']['dbusername'], role['dbParam']['dbpassword'],
role['dbInstName'],role['jdbcName'])

def configResStandalonePart(profile_home, cell_name, node_name, db_type,
db_hostname, db_port, db_username, db_password, db_instance_name,
jdbc_driver_name):
role = maestro.role
scriptdir = maestro.node['scriptdir']
scriptCommonFolder = scriptdir + '/resconfigure_common'
scriptFolderDir = scriptCommonFolder + '/configureWODM/'
scriptFolderDir = scriptdir + '/resconfigure_common/installRuleApp/'

# configure WebSphere Decision Server
rc = maestro.trace_call(logger, [scriptFolderDir + 'ConfigODM.sh',
scriptCommonFolder,softwareDir, profile_home, cell_name, node_name,
db_hostname, db_port, db_username, db_password, db_type,
db_instance_name, jdbc_driver_name])

# install rule application .jar file to WebSphere Decision Server
for archive in role['archive_file']:
rc = maestro.trace_call(logger, [scriptFolderDir + 'installRuleApp.sh',
scriptCommonFolder, archive, softwareDir])

# export parameters
maestro.export['resIp'] = maestro.node['instance']['public-ip']
maestro.export['resPort'] = '9080'
maestro.export['vmName'] = maestro.parms['vmName']

# change role status
maestro.role_status = 'RUNNING'

```

- v. RESStandalone depends on WASInstallS. Create the WASInstallS role dependency folder under the plugin/parts/resdb2/scripts/RESStandalone folder. Add content to plugin/parts/resdb2/scripts/RESStandalone/WASInstallS/changed.py (see Example 6-45).

Example 6-45 Content in .../RESStandalone/WASInstallS/changed.py

```

myrole = None
if len(maestro.updated) >= 1:
myrole = maestro.updated[0]
elif len(maestro.added) >= 1:
myrole = maestro.added[0]

if myrole and 'WASEnvIsReady' in maestro.deps[myrole] and not
existFlag():
role['wasStandaloneIsReady'] = 'true'
wasParam = maestro.deps[myrole]['WASParameters']
role['wasParam'] = wasParam

```

```

if role.has_key('resdbIsReady') and role['resdbIsReady'] == 'true':
    configResStandalonePart(role['wasParam']['PROFILE_HOME'],role['wasParam']
    ['CELL_NAME'],role['wasParam']['NODE_NAME'],role['dbType'],
    role['dbParam']['dbip'], role['dbParam']['dbport'],
    role['dbParam']['dbusername'], role['dbParam']['dbpassword'],
    role['dbInstName'],role['wasParam']['JDBC_DRIVER'],role['jdbcName'])

def configResStandalonePart(profile_home, cell_name, node_name, db_type,
db_hostname, db_port, db_username, db_password, db_instance_name,
jdbc_driver_dir,jdbc_driver_name):
    role = maestro.role
    scriptdir = maestro.node['scriptdir']

    # configure WebSphere Decision Server
    scriptFolderDir = script_common_dir + '/configureWODM/'
    rc = maestro.trace_call(logger, [scriptFolderDir + 'ConfigODM.sh',
    script_common_dir,softwareDir, profile_home, cell_name, node_name,
    db_hostname, db_port, db_username, db_password, db_type,
    db_instance_name, jdbc_driver_dir, jdbc_driver_name])

    # install rule application to WebSphere Decision Server
    scriptFolderDir = script_common_dir + '/installRuleApp/'
    for archive in role['archive_file']:
        rc = maestro.trace_call(logger, [scriptFolderDir + 'installRuleApp.sh',
        script_common_dir, archive, softwareDir])

    # export parameters
    maestro.export['resIp'] = maestro.node['instance']['public-ip']
    maestro.export['resPort'] = '9080'
    maestro.export['vmName'] = maestro.parms['vmName']
    maestro.export['WODM_DEPLOYMENT_STATUS'] = 'ready'

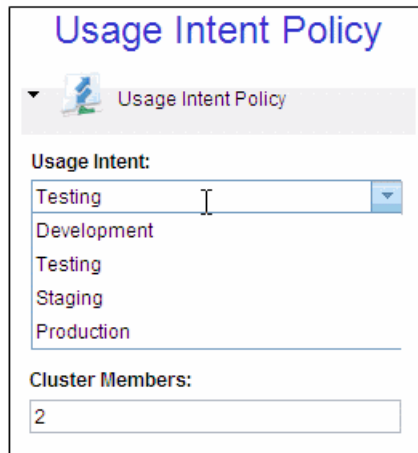
    # change role status
    maestro.role_status = 'RUNNING'

```

6.1.3 Create a usage intent policy

You can add a usage intent policy to manage application components, and to generate different topologies by setting different properties in the usage intent policy.

Figure 6-26 shows a usage intent policy attribute.



The image shows a configuration window titled "Usage Intent Policy". It has a dropdown menu for "Usage Intent" with options: Testing, Development, Testing, Staging, and Production. Below this is a "Cluster Members" field with the value "2".

Figure 6-26 Usage intent policy attribute

Figure 6-27 lists the stand-alone topology when the deployment type is Development.

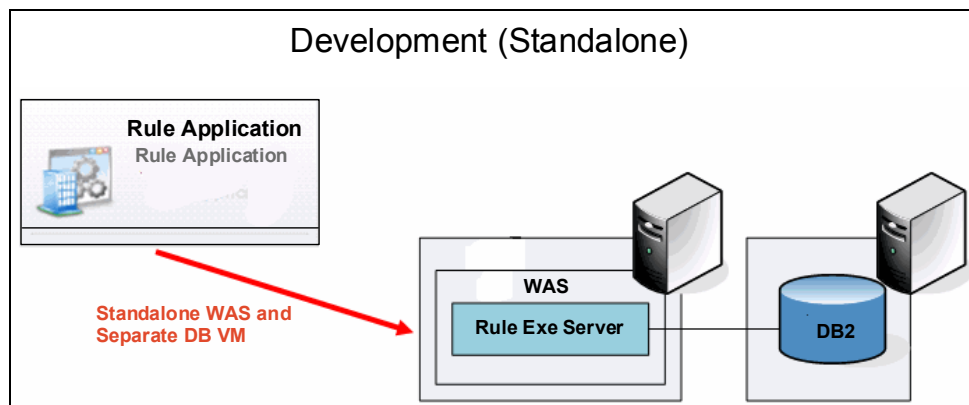


Figure 6-27 Stand-alone topology

Figure 6-28 lists the cluster topology when the deployment type is Testing, Staging, or Production.

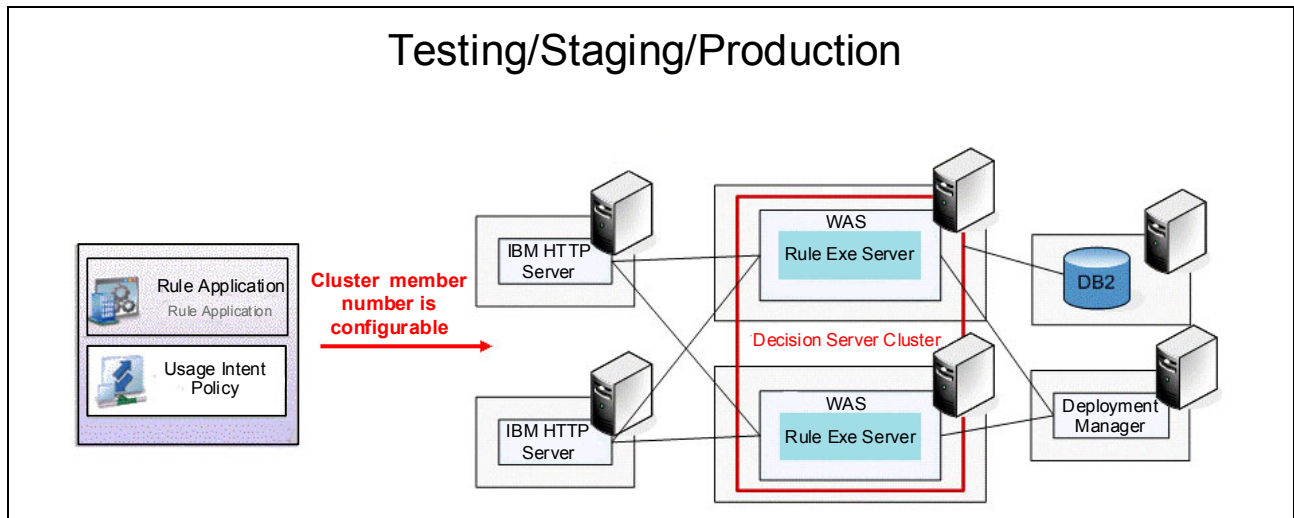


Figure 6-28 Cluster topology

Figure 6-29 shows role dependency in the rule stand-alone topology.

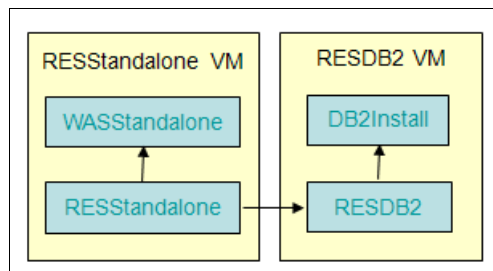


Figure 6-29 Role dependency in the rule stand-alone topology

Figure 6-30 shows role dependency in the rule cluster topology, and Table 6-5 shows the usage intent deployment types.

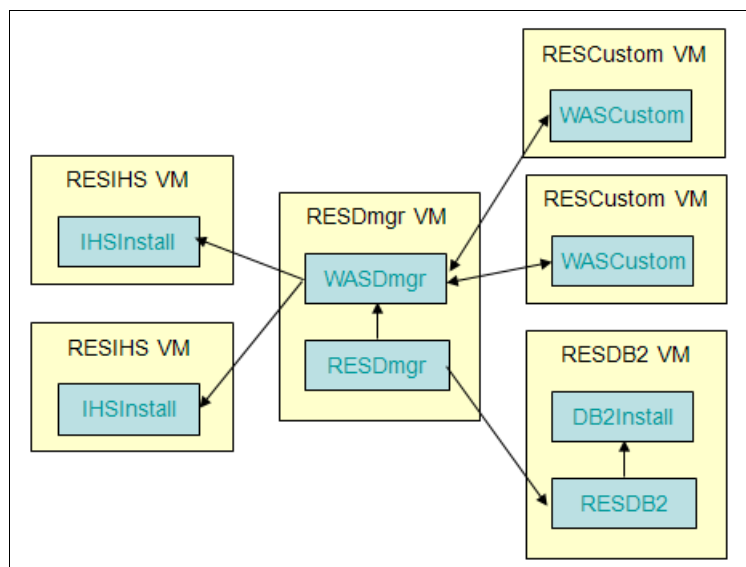


Figure 6-30 Role dependency in rule cluster topology

Table 6-5 Usage intent policy deployment types

Deployment type	Topology type	Virtual machines
Development	Stand-alone	<ul style="list-style-type: none"> ▶ One DB2 VM ▶ One WebSphere Decision Server Stand-alone VM
Testing	Cluster	<ul style="list-style-type: none"> ▶ One DB2 VM ▶ One WebSphere Decision Server deployment manager VM ▶ Two WebSphere Decision Server custom node VMs ▶ Two IBM HTTP Servers
Staging	Cluster	<ul style="list-style-type: none"> ▶ One DB2 VM ▶ One WebSphere Decision Server deployment manager VM ▶ Three WebSphere Decision Server custom node VMs ▶ Two IBM HTTP Servers
Production	Cluster	<ul style="list-style-type: none"> ▶ One DB2 VM ▶ One WebSphere Decision Server deployment manager VM ▶ Four WebSphere Decision Server custom node VMs ▶ Two IBM HTTP Servers

Create usage intent policies by performing the following tasks.

Create a usage intent policy plug-in

Follow these steps to create a usage intent policy plug-in:

1. Create the plug-in project:
 - a. Create a plug-in by selecting **File** → **New** → **Project** → **IBM Workload Plug-in Development** → **IBM Workload Plug-in Project**.
 - b. Enter `plugin.com.ibm.policy.usage` as both the Project name and the Plug-in name.
 - c. Click **Finish**.
2. Update the `config.json` plug-in configuration file, and set the primary pattern type to `"patterntype.ruleexecution": "1.0"` (see Example 6-46).

Example 6-46 The config.json file

```
{
  "name": "plugin.com.ibm.policy.usage",
  "version" : "1.0.0.0",
  "patterntypes": {
    "primary": {
      "patterntype.ruleexecution": "1.0"
    },
    "secondary": [
      {
        "": ""
      }
    ]
  },
  "packages": {
    "PLUGIN.COM.IBM.POLICY.USAGE": []
  },
  "parms": {
  }
}
```

3. Add the `usagePolicy` usage intent policy to `metadata.json`, and set attributes for the usage intent policy:
 - When the development type is `Testing`, set `testingClusterMember` as the cluster custom node number.
 - When the development type is `Staging`, set `stagingClusterMember` as the cluster custom node number.
 - When the development type is `Production`, set `productionClusterMember` as the cluster custom node number.

Figure 6-31 lists usage intent policy definitions, and Example 6-47 illustrates the metadata.json file.

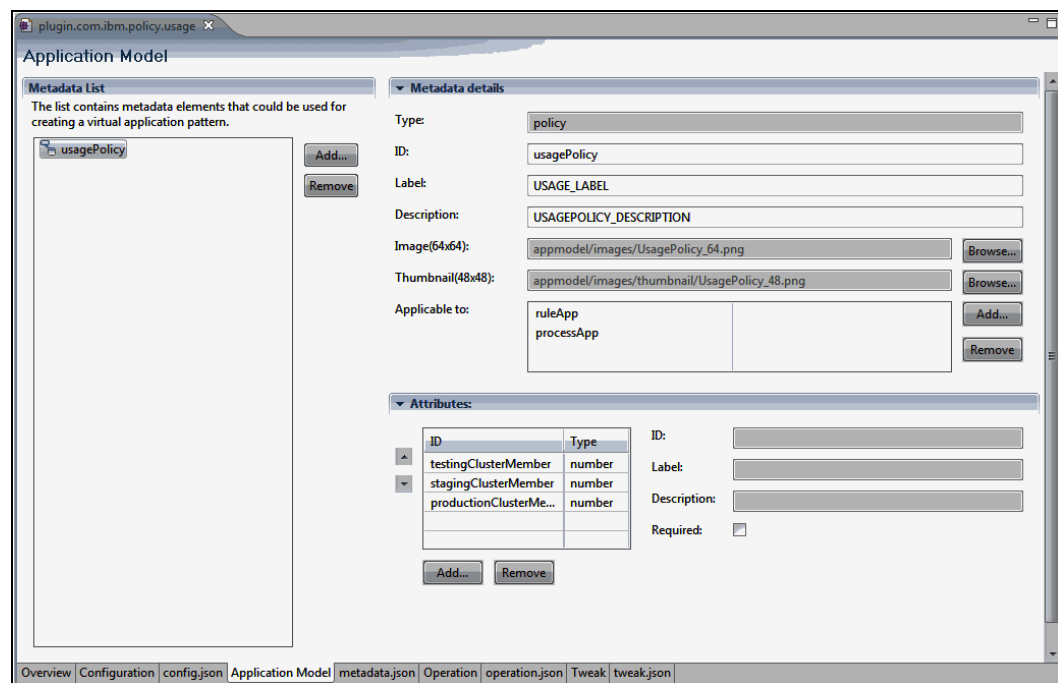


Figure 6-31 Usage intent policy

Example 6-47 The metadata.json file

```
[
  {
    "id": "usagePolicy",
    "type": "policy",
    "applicableTo": [
      "ruleApp",
      "processApp"
    ],
    "thumbnail": "appmodel\\images\\thumbnail\\UsagePolicy_48.png",
    "image": "appmodel\\images\\UsagePolicy_64.png",
    "label": "USAGE_LABEL",
    "description": "USAGEPOLICY_DESCRIPTION",
    "groups": [
      {
        "category": "DEPLOYMENT_TYPE",
        "id": "Development",
        "label": "DEVELOPMENT_LABEL",
        "defaultValue": true,
        "attributes": [
        ],
        "description": "DEPLOYMENT_TYPE_DEVELOPMENT_DESCRIPTION"
      },
      {
        "category": "DEPLOYMENT_TYPE",
        "id": "Testing",
        "label": "TESTING_LABEL",
        "defaultValue": false,

```

```

        "attributes": [
            "testingClusterMember"
        ],
        "description": "DEPLOYMENT_TYPE_TESTING_DESCRIPTION"
    },
    {
        "category": "DEPLOYMENT_TYPE",
        "id": "Staging",
        "label": "STAGING_LABEL",
        "defaultValue": false,
        "attributes": [
            "stagingClusterMember"
        ],
        "description": "DEPLOYMENT_TYPE_STAGING_DESCRIPTION"
    },
    {
        "category": "DEPLOYMENT_TYPE",
        "id": "Production",
        "label": "PRODUCTION_LABEL",
        "defaultValue": false,
        "attributes": [
            "productionClusterMember"
        ],
        "description": "DEPLOYMENT_TYPE_PRODUCTION_DESCRIPTION"
    }
],
"attributes": [
    {
        "id": "testingClusterMember",
        "type": "number",
        "required": false,
        "label": "CLUSTER_MEMBER_LABEL",
        "description": "CLUSTER_MEMBER_DESCRIPTION",
        "sampleValue": 2,
        "max": 3000,
        "min": 1,
        "invalidMessage": "CLUSTER_MEMBER_INVALIDMESSAGE"
    },
    {
        "id": "stagingClusterMember",
        "type": "number",
        "required": false,
        "label": "CLUSTER_MEMBER_LABEL",
        "description": "CLUSTER_MEMBER_DESCRIPTION",
        "sampleValue": 3,
        "max": 3000,
        "min": 1,
        "invalidMessage": "CLUSTER_MEMBER_INVALIDMESSAGE"
    },
    {
        "id": "productionClusterMember",
        "type": "number",
        "required": false,
        "label": "CLUSTER_MEMBER_LABEL",
        "description": "CLUSTER_MEMBER_DESCRIPTION",
    }
]

```

```

        "sampleValue": 4,
        "max": 3000,
        "min": 1,
        "invalidMessage": "CLUSTER_MEMBERE_INVALIDMESSAGE"
    }
}
}
]
]

```

Update the transformer

Use the usage intent policy in RESTransformer to update the Business Rule Application pattern topology (see Example 6-48 and Example 6-49).

Example 6-48 Define constants in Constants.java

```

public static final String USAGEPOLICY = "usagePolicy";
public static final String GROUPS = "groups";
public static final String DEVELOPMENT = "Development";
public static final String TESTING = "Testing";
public static final String STAGING = "Staging";
public static final String PRODUCTION = "Production";
public static final String TST_CLUSTER_MEMBER = "testingClusterMember";
public static final String STG_CLUSTER_MEMBER = "stagingClusterMember";
public static final String PRD_CLUSTER_MEMBER = "productionClusterMember";
public static final String NAME = "name";
public static final String HYPHEN = "-";
public static final String VM_SUFFIX_CUSTOM = "RESCustom";
public static final String PACKAGES = "packages";
public static final String PACKAGE_WAS_INSTALL = "RESWASInstall";
public static final String PACKAGE_RES_INSTALL = "RESInstall";
public static final String SCALLING = "scaling";
public static final String MIN = "min";
public static final String MAX = "max";

```

Example 6-49 Define topology in RESTransformer.java

```

// get policy
JSONObject usagePolicy = getPolicy(applicationComponent, Constants.USAGEPOLICY);

if (usagePolicy == null) {
    isCluster = false;
} else {
    JSONObject usagePolicyAttributes = (JSONObject)
    usagePolicy.get(Constants.ATTRIBUTES);

// get group attribute
JSONObject group = (JSONObject) usagePolicy.get(Constants.GROUPS);

if (group != null) {
    boolean isDevelopment = (Boolean) group.get(Constants.DEVELOPMENT);
    boolean isTesting = (Boolean) group.get(Constants.TESTING);
    boolean isStaging = (Boolean) group.get(Constants.STAGING);
    boolean isProduction = (Boolean) group.get(Constants.PRODUCTION);
}
}

```

```

if (isDevelopment) {
// if Deployment type is Development, set isCluster to false
isCluster = false;
else {

if (isTesting) {
// if Deployment type is Tesing, set isCluster to true, retrieve cluster number
value from attribute and set min and max value
isCluster = true;
clusterMember = (Long) usagePolicyAttributes.get(Constants.TST_CLUSTER_MEMBER);
minValue = clusterMember.intValue();
maxValue = clusterMember.intValue();
else if (isStaging) {
// if Deployment type is Staging, set isCluster to true, retrieve cluster number
value from attribute and set min and max value
isCluster = true;
clusterMember = (Long) usagePolicyAttributes.get(Constants.STG_CLUSTER_MEMBER);
minValue = clusterMember.intValue();
maxValue = clusterMember.intValue();
else if (isProduction) {
// if Deployment type is Production, set isCluster to true, retrieve cluster
number value from attribute and set min and max value
isCluster = true;
clusterMember = (Long) usagePolicyAttributes.get(Constants.PRD_CLUSTER_MEMBER);
minValue = clusterMember.intValue();
maxValue = clusterMember.intValue();
}
}
}
}

if(isCluster == false){
// define stand-alone topology

}else{
// define cluster topology

// define custom node virtual machine
JSONObject vmCustom = new JSONObject();
vmTemplates.add(vmCustom);
vmCustom.put(Constants.NAME, vmTemplateNamePrefix + Constants.HYPHEN +
Constants.VM_SUFFIX_CUSTOM);

JSONArray packagesCustom = new JSONArray();
vmCustom.put(Constants.PACKAGES, packagesCustom);
packagesCustom.add(Constants.PACKAGE_WAS_INSTALL);
packagesCustom.add(Constants.PACKAGE_RES_INSTALL);

// set min and max value in scaling property to define custom node number
JSONObject scaling = new JSONObject();
vmCustom.put(Constants.SCALLING, scaling);
scaling.put(Constants.MIN, minValue);
scaling.put(Constants.MAX, maxValue);
}

```

6.1.4 Create a scaling policy

This section provides information about how to define CPU-based auto scaling in IBM PureApplication System.

Figure 6-32 lists scaling policy definitions.

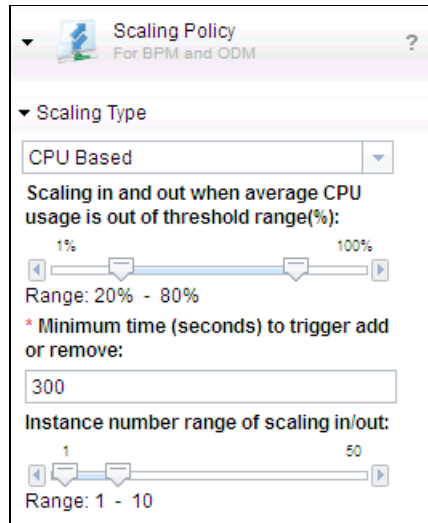


Figure 6-32 Scaling policy

Create a scaling policy by performing the following tasks.

Create a plug-in

To create a scaling policy, first create a plug-in:

1. Create a plug-in project:
 - a. Create a plug-in by selecting **File** → **New** → **Project** → **IBM Workload Plug-in Development** → **IBM Workload Plug-in Project**.
 - b. Enter `plugin.com.ibm.policy.scaling` as both the Project name and the Plug-in name.
 - c. Click **Finish**.
2. Update the `config.json` plug-in configuration file, and set the primary pattern type to `"pattern.type.ruleexecution": "1.0"` (see Example 6-50).

Example 6-50 Update `config.json`

```
{
  "name": "plugin.com.ibm.policy.scaling",
  "version": "1.0.0.0",
  "pattern.types": {
    "primary": {
      "pattern.type.ruleexecution": "1.0"
    },
    "secondary": [
      {
        "pattern.type.ruleexecution": "1.0"
      }
    ]
  }
}
```



```

    },
    "packages": {
    },
    "parms": {
    }
}

```

3. Update the metadata.json file, and define the scaling policy (see Figure 6-33 and Example 6-51).

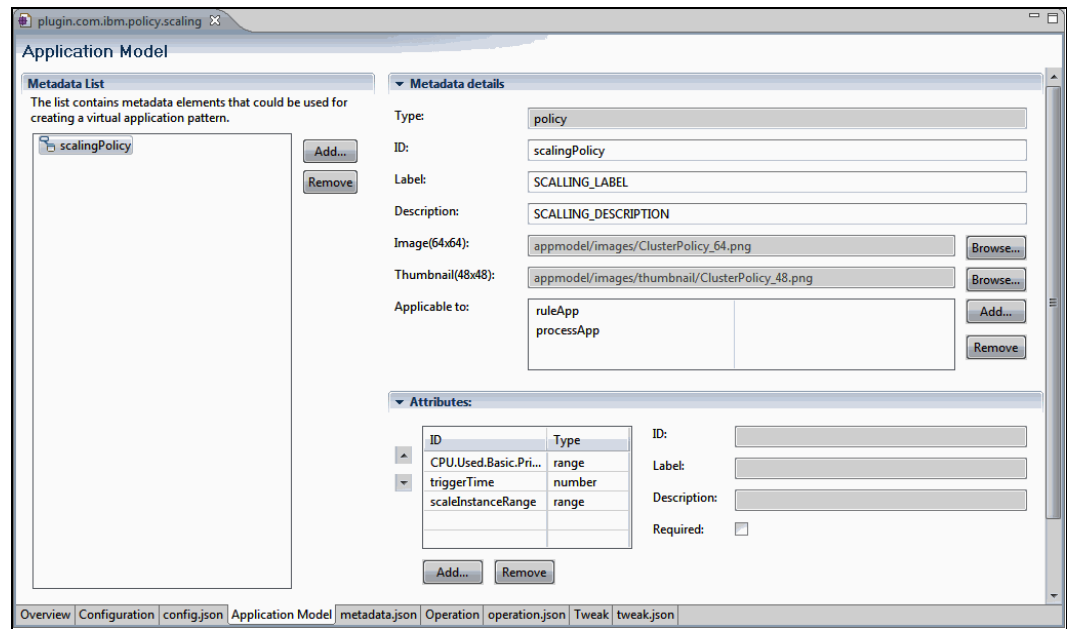


Figure 6-33 Scaling policy definition

Example 6-51 The metadata.json file

```

[
  {
    "id": "scalingPolicy",
    "type": "policy",
    "transformAs": "linked",
    "applicableTo": [
      "ruleApp",
      "processApp"
    ],
    "label": "SCALLING_LABEL",
    "thumbnail": "appmodel\\images\\thumbnail\\ClusterPolicy_48.png",
    "image": "appmodel\\images\\ClusterPolicy_64.png",
    "description": "SCALLING_DESCRIPTION",
    "groups": [
      {
        "category": "SCALE_POLICY_TYPE",
        "id": "Basic",
        "label": "SCALE_POLICY_TYPE_BASIC",
        "defaultValue": true,
        "attributes": [
          "CPU.Used.Basic.Primitive",

```

```

        "triggerTime",
        "scaleInstanceRange"
    ],
    "description": "SCALE_POLICY_TYPE_BASIC_DESCRIPTION"
}
],
"attributes": [
{
    "id": "CPU.Used.Basic.Primitive",
    "label": "SCALE_ATTRIBUTES_CPU",
    "type": "range",
    "displayType": "percentage",
    "required": false,
    "max": 100,
    "min": 1,
    "sampleValue": [
        20,
        80
    ],
    "description": "SCALE_ATTRIBUTES_CPU_DESCRIPTION"
},
{
    "id": "triggerTime",
    "label": "TRIGGER_TIME_LABEL",
    "type": "number",
    "max": 1800,
    "min": 30,
    "required": true,
    "sampleValue": 300,
    "invalidMessage": "TRIGGER_TIME_INVALIDMESSAGE",
    "description": "TRIGGER_TIME_DESCRIPTION"
},
{
    "id": "scaleInstanceRange",
    "label": "SCALE_INSTANCE_RANGE_LABEL",
    "type": "range",
    "min": 1,
    "max": 50,
    "required": false,
    "sampleValue": [
        1,
        10
    ],
    "description": "SCALE_INSTANCE_RANGE_DESCRIPTION"
}
]
}
]

```

-
4. Add the Service-Component: OSGI-INF/scalingPolicyTransform.xml to MANIFEST.MF.
 5. Create a scalingPolicyTransform.xml file in the OSGI-INF folder, and set the component name and implementation class in that file (see Example 6-52 on page 133).

Example 6-52 The scalingPolicyTransform.xml file

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
name="scalingPolicy">
  <implementation
class="com.ibm.maestro.model.transform.scaling.ScalingPolicyTransformer"/>
<service>
<provide interface="com.ibm.maestro.model.transform.TopologyProvider"/>
</service>
</scr:component>
```

6. Create a ScalingPolicyTransformer class to transform the scaling policy.

Figure 6-34 shows the scaling property in topology.json that was added by the ScalingPolicyTransformer.

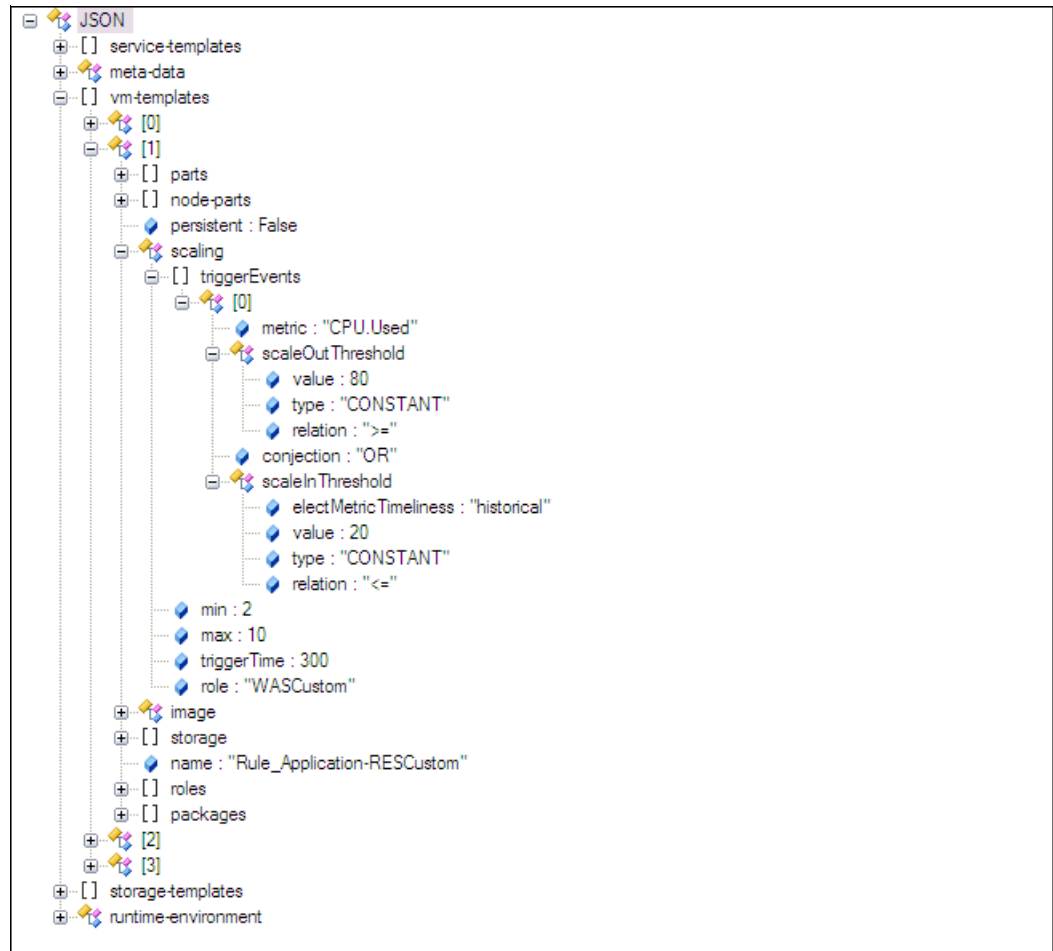


Figure 6-34 Scaling property

- a. Override the transformComponent method, and create a scaling object in this method (see Example 6-53 and Example 6-54).

Example 6-53 Define constants in Constants.java

```
public static final String GROUPS = "groups";
public static final String BASIC = "Basic";
public static final String CPU_USED_BASIC_PRIMITIVE =
    "CPU.Used.Basic.Primitive";
public static final String TRIGGERTIME = "triggerTime";
public static final String SCALEINSTANCERANGE = "scaleInstanceRange";
public static final String MIN = "min";
public static final String MAX = "max";
public static final String SCALING = "scaling";
public static final String PRIMITIVE = "Primitive";
public static final String CONJECTION = "conjection";
public static final String OR = "OR";
public static final String CONSTANT = "CONSTANT";
public static final String TYPE = "type";
public static final String RELATION = "relation";
public static final String LESS = "<=";
public static final String BIGGER = ">=";
public static final String VALUE = "value";
public static final String HISTORICAL = "historical";
public static final String ELECTMETRICTIMELINESS = "electMetricTimeliness";
public static final String SCALEINTHRESHOLD = "scaleInThreshold";
public static final String SCALEOUTTHRESHOLD = "scaleOutThreshold";
public static final String ROLE = "role";
```

Example 6-54 Create a scaling object in ScalingPolicyTransformer.java

```
Long triggerTime = null;
JSONArray scaleInstanceRange=null;
String[] metrics = {};
if((Boolean)((JSONObject)scalingNode.get(Constants.GROUPS)).get(Constants.BASIC)){
    metrics = new String[]{Constants.CPU_USED_BASIC_PRIMITIVE};
    triggerTime = (Long)policyAttributes.get(Constants.TRIGGERTIME);
    scaleInstanceRange =
        (JSONArray)policyAttributes.get(Constants.SCALEINSTANCERANGE);
    scalingRoot.put(Constants.MIN, scaleInstanceRange.get(0));
    scalingRoot.put(Constants.MAX, scaleInstanceRange.get(1));
}

scalingRoot.put(Constants.TRIGGERTIME, triggerTime);
JSONArray triggerEvents = new JSONArray();
scalingRoot.put(Constants.TRIGGEREVENTS, triggerEvents);
for(String metric : metrics){
    if(policyAttributes.get(metric) != null){
        JSONObject triggerEvent = new JSONObject();

        String[] tokens = metric.split("\\.");
        if(tokens[tokens.length -1].equals(Constants.PRIMITIVE)){
            triggerEvent.put(Constants.METRIC, tokens[0] + "." + tokens[1]);
            triggerEvent.put(Constants.CONJECTION, Constants.OR);
        }
    }
}
```

```

JSONArray thresholds = (JSONArray) policyAttributes.get(metric);

JSONObject scaleInThreshold = new JSONObject();
scaleInThreshold.put(Constants.TYPE, Constants.CONSTANT);
scaleInThreshold.put(Constants.RELATION, Constants.LESS);
scaleInThreshold.put(Constants.VALUE, thresholds.get(0));
scaleInThreshold.put(Constants.ELECTMETRICTIMELINESS, Constants.HOSTORICAL);

JSONObject scaleOutThreshold = new JSONObject();
scaleOutThreshold.put(Constants.TYPE, Constants.CONSTANT);
scaleOutThreshold.put(Constants.RELATION, Constants.BIGGER);
scaleOutThreshold.put(Constants.VALUE, thresholds.get(1));

triggerEvent.put(Constants.SCALEINTHRESHOLD, scaleInThreshold);
triggerEvent.put(Constants.SCALEOUTTHRESHOLD, scaleOutThreshold);
}
triggerEvents.add(triggerEvent);
}
}

```

- b. Override the transformLink method, and add a scaling object to the vm-template (see Example 6-55 and Example 6-56).

Example 6-55 Define constants in Constants.java

```

public static final String SCALING_POLICY="scaling-policy";
public static final String ROLE = "role";
public static final String SCALING = "scaling";

```

Example 6-56 Add a scaling object to vm-template in ScalingPolicyTransformer.java

```

JSONObject scalingRoot = (JSONObject)target.get(Constants.SCALING_POLICY);
MatchedRole matchedRole = getMatchedRole(source, roleName);
JSONObject roleScalingRoot=JSONObject.parse(scalingRoot.toString());
roleScalingRoot.put(Constants.ROLE, roleName);
vmTemplate.put(Constants.SCALING,roleScalingRoot);

```

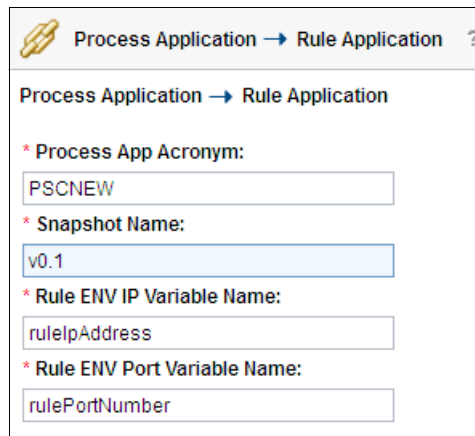
6.2 The Business Process Application pattern type

IBM provides a Business Process Application pattern type that you can deploy to PureApplication System. Download the Business Process Application pattern type software installation files listed in Table A-2 on page 179. Replace the installation files in the pattern.type.processautomation-1.0.0.0.tgz/files/bpm folder with the files that you downloaded.

6.3 The business process and business rule link

This section provides information about how to integrate business processes and rules by creating a link between a business process and a business rule. The link includes scripts that update the Business Process Manager database (BPMDb) to integrate processes with rules.

Figure 6-35 shows the business process and rule link definitions.



Process Application → Rule Application ?

Process Application → Rule Application

* Process App Acronym:

* Snapshot Name:

* Rule ENV IP Variable Name:

* Rule ENV Port Variable Name:

Figure 6-35 Business process and rule link

Figure 6-36 shows the environment variables defined in the sample business process.

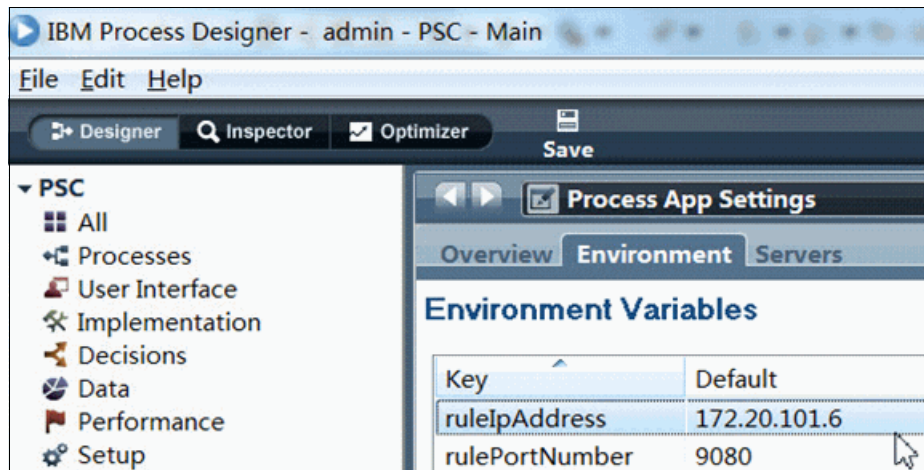


Figure 6-36 Environment variables

Figure 6-37 shows how the process uses environment variables to start the rule service.

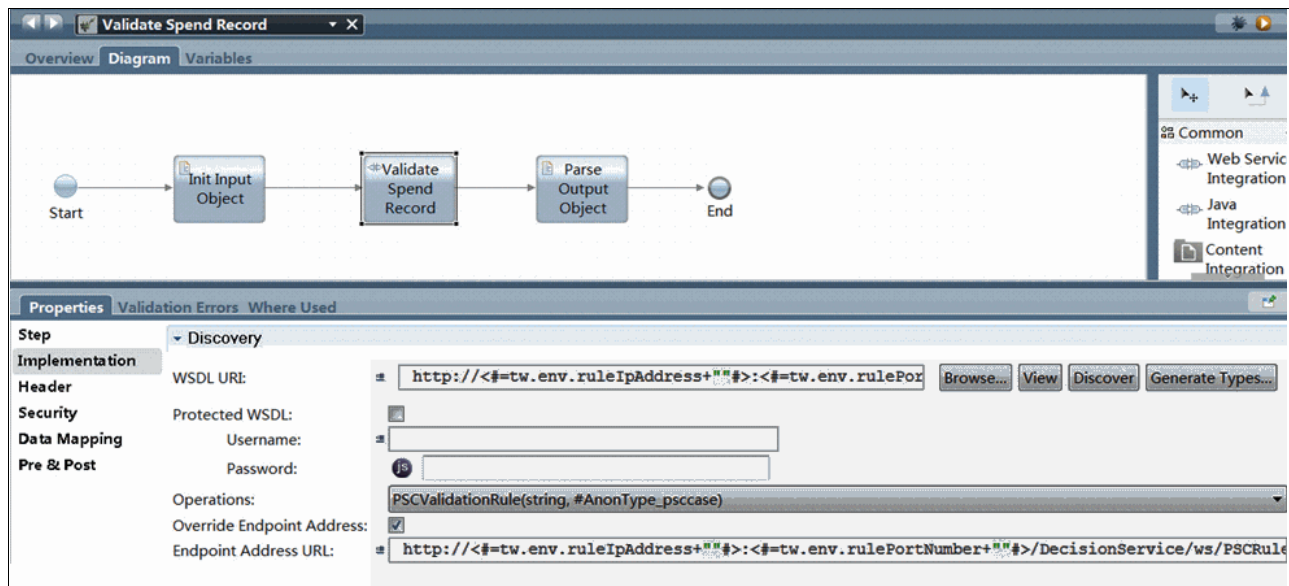


Figure 6-37 Process invoke rule service

Example 6-57 shows how to add environment variable content to BPMDB.

Example 6-57 Update BPMDB Structured Query Language (SQL)

```
INSERT INTO LSW_ENV_VAR_VAL
(
    ENV_VAR_VAL_ID,
    SNAPSHOT_ID,
    TIP,
    ENV_VAR_ID,
    VALUE,
    NAME,
    LAST_MODIFIED,
    LAST_MODIFIED_BY_USER_ID
)
VALUES ((SELECT CASE
    WHEN MAX(LSW_ENV_VAR_VAL.ENV_VAR_VAL_ID) IS NULL THEN 1
    ELSE MAX(LSW_ENV_VAR_VAL.ENV_VAR_VAL_ID) + 1
END
FROM LSW_ENV_VAR_VAL),
(SELECT LSW_SNAPSHOT.SNAPSHOT_ID
FROM LSW_SNAPSHOT
WHERE LSW_SNAPSHOT.NAME = 'SNAPSHOTNAME'
AND PROJECT_ID = (SELECT PROJECT_ID
FROM LSW_PROJECT
WHERE SHORT_NAME = 'PROCESSACRONYM')),
'F',
(SELECT LSW_ENV_VAR.ENV_VAR_ID
FROM LSW_ENV_VAR
WHERE LSW_ENV_VAR.NAME = 'IPN'),
'IPVAL',
'IPN',
( CURRENT_TIMESTAMP ),
9
);
```

Create the link between processes and rules by performing these tasks.

Create a plug-in project

To create a link, first create a plug-in project:

1. Create a plug-in project:
 - a. Create a plug-in by selecting **File** → **New** → **Project** → **IBM Workload Plug-in Development** → **IBM Workload Plug-in Project**.
 - b. Enter `plugin.com.ibm.bpm.psres` as both the Project name and the Plug-in name.
 - c. Click **Finish**.
2. Update the `config.json` plug-in configuration file (see Example 6-58):
 - a. Set the primary pattern type to `"patternType.ruleexecution": "1.0"`.
 - b. Create two packages to integrate the process cluster with the business rule stand-alone and cluster.

Example 6-58 The config.json file

```
{
  "name": "plugin.com.ibm.bpm.psres",
  "version": "1.0.0.0",
  "patternTypes": {
    "primary": {
      "patternType.ruleexecution": "1.0"
    },
    "linked": {
      "patternType.processautomation": "1.0"
    }
  },
  "packages": {
    "PSDmgrRESStandalone": [
      {
        "parts": [
          {
            "part": "parts\\psdmgrresstandalone.scripts.tgz"
          }
        ]
      }
    ],
    "PSDmgrRESMgr": [
      {
        "parts": [
          {
            "part": "parts\\psdmgrresmgr.scripts.tgz"
          }
        ]
      }
    ]
  },
  "parms": {
  }
}
```

3. Create a PSRES link in the metadata.json file:
 - a. In the Metadata details section, select **link** as the Type.
 - b. Enter PSRES as the ID.
 - c. Enter PSRES_LABEL as the Label.
 - d. Enter PSRES_DESC as the Description.
 - e. Click **Add** to add processApp as the Source, and ruleApp as the Target.

Figure 6-38 lists the PSRES link definition, and Example 6-59 shows an example of the metadata.json file.

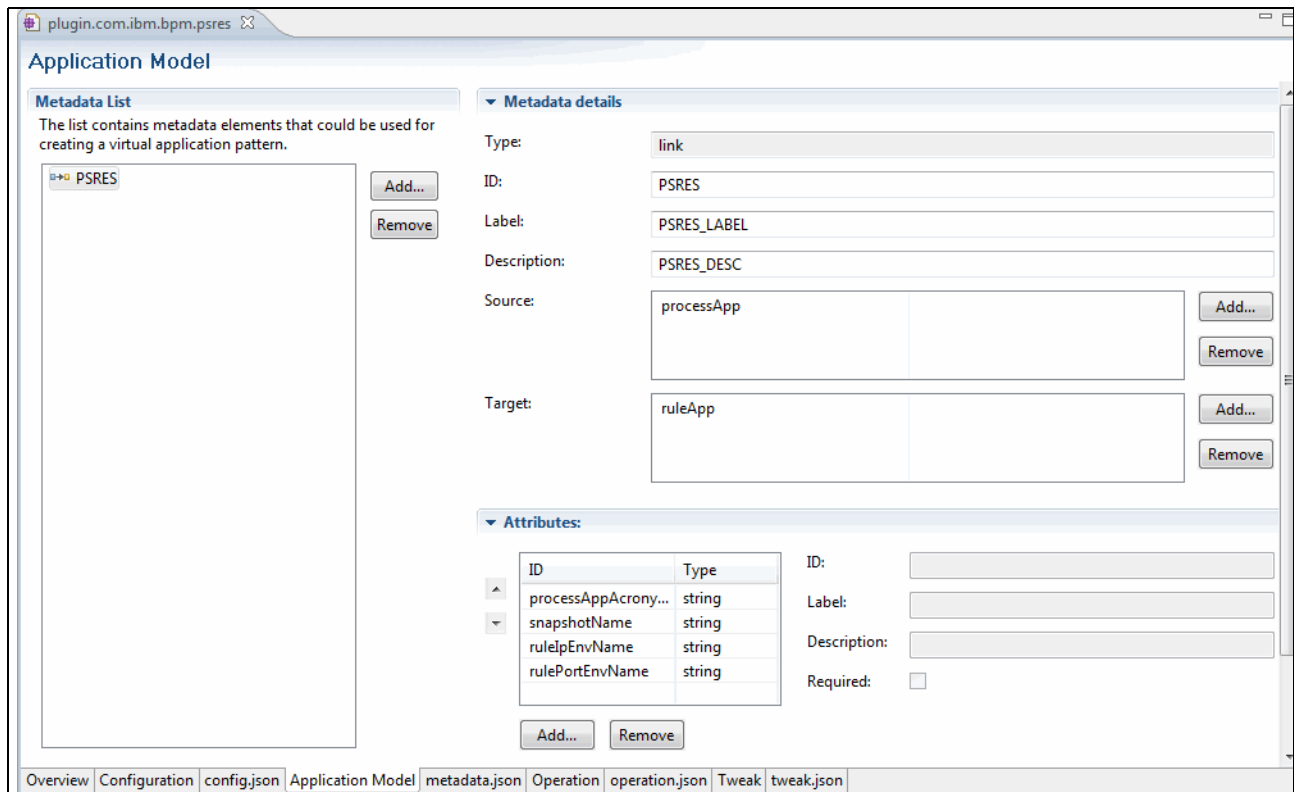


Figure 6-38 Link definition

Example 6-59 The metadata.json file

```
[
  {
    "id": "PSRES",
    "type": "link",
    "source": [
      "processApp"
    ],
    "target": [
      "ruleApp"
    ],
    "label": "PSRES_LABEL",
    "description": "PSRES_DESC",
    "attributes": [
      {
        "id": "processAppAcronym",
        "type": "string",
```

```

        "required": true,
        "label": "PROCESS_APP_ACRONYM_LABEL",
        "description": "PROCESS_APP_ACRONYM_DESC"
    },
    {
        "id": "snapshotName",
        "type": "string",
        "required": true,
        "label": "SNAPSHOT_NAME_LABEL",
        "description": "SNAPSHOT_NAME_DESC"
    },
    {
        "id": "ruleIpEnvName",
        "type": "string",
        "required": true,
        "label": "RULE_IP_ENV_NAME_LABEL",
        "description": "RULE_IP_ENV_NAME_DESC"
    },
    {
        "id": "rulePortEnvName",
        "type": "string",
        "required": true,
        "label": "RULE_PORT_ENV_NAME_LABEL",
        "description": "RULE_PORT_ENV_NAME_DESC"
    }
  ]
}
]

```

4. Add Service-Component: OSGI-INF/psres.xml to MANIFEST.MF.
5. Create a psres.xml file in the OSGI-INF folder, and set the component name and implementation class in psres.xml (see Example 6-60).

Example 6-60 The psres.xml file

```

<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0" name="PSRES">
  <implementation
class="com.ibm.maestro.model.transform.psres.PSRESTransformer"/>
<service>
<provide interface="com.ibm.maestro.model.transform.TopologyProvider"/>
</service>
</scr:component>

```

Define a transformer

Follow these steps to define a transformer:

1. Create a Transformer named PSRESTransformer.
2. Override the transformLink method.
3. Update the PSDmgr vm-template content:
 - a. Add packages to the vm-template (see Example 6-61 on page 141 and Example 6-62 on page 141).

Example 6-61 Define constants in Constants.java

```
public static final String PACKAGES = "packages";
public static final String PACKAGE_PSDMGR_RESDMGR = "PSDmgrRESdmgr";
public static final String PACKAGE_PSDMGR_RESSTANDALONE =
"PSDmgrRESstandalone";
```

Example 6-62 Add packages to the vm-template

```
MatchedRole source = getMatchedRole(sourceFragment, targetRoleType_PS);
JSONArray packages = (JSONArray) source.template.get(Constants.PACKAGES);
if (packages == null) {
    packages = new JSONArray();
    source.template.put(Constants.PACKAGES, packages);
}
//check if package already exist
boolean pkgExist = false;
for(int i = 0; i < packages.size(); ++i){
    String pkgName = String.valueOf(packages.get(i));
    if(Constants.PACKAGE_PSDMGR_RESDMGR.equals(pkgName)
    || Constants.PACKAGE_PSDMGR_RESSTANDALONE.equals(pkgName)){
        pkgExist = true;
        break;
    }
}
if(!pkgExist){
    if(isDmgr){
        packages.add(Constants.PACKAGE_PSDMGR_RESDMGR);
    }else{
        packages.add(Constants.PACKAGE_PSDMGR_RESSTANDALONE);
    }
}
```

- b. Create the PSRES role. Set the PSRES role dependency to PSDmgr and RESStandalone, or to RESDmgr (see Example 6-63 and Example 6-64).

Example 6-63 Define constants in Constants.java

```
public static final String PLUGIN = "plugin";
public static final String NAME = "name";
public static final String TYPE = "type";
public static final String ROLE_PSRES_NAME = "PSRES";
public static final String DEPENDS = "depends";
public static final String POINT = ".";
public static final String TARGET_ROLE_TYPE_STD = "RESStandalone";
public static final String TARGET_ROLE_NAME_STD = "RESStandalone";
public static final String TARGET_ROLE_TYPE_DMGR = "RESdmgr";
public static final String TARGET_ROLE_NAME_DMGR = "RESdmgr";
```

Example 6-64 Create the PSRES role

```
JSONObject targetRole = null;
JSONObject targetTemplate = null;
boolean isDmgr = false;
```

```

for (Iterator iterator = templates.iterator(); iterator.hasNext();) {
    targetTemplate = (JSONObject) iterator.next();
    JSONArray roles = (JSONArray) targetTemplate.get(Constants.ROLES);
    for (Iterator iterator1 = roles.iterator(); iterator1.hasNext();) {
        targetRole = (JSONObject) iterator1.next();
        String type = (String) targetRole.get(Constants.TYPE);
        // find the vm-template that contains target role
        if (targetRoleTypeStd.equals(type)) {
            foundTargetRole = true;
            targetRoleType_RES = Constants.TARGET_ROLE_TYPE_STD;
            targetRoleName_RES = Constants.TARGET_ROLE_NAME_STD;
            targetVmName_RES = (String) targetTemplate.get(Constants.NAME);
            break;
        } else if (targetRoleTypeDmgr.equals(type)) {
            foundTargetRole = true;
            isDmgr = true;
            targetRoleType_RES = Constants.TARGET_ROLE_TYPE_DMGR;
            targetRoleName_RES = Constants.TARGET_ROLE_NAME_DMGR;
            targetVmName_RES = (String) targetTemplate.get(Constants.NAME);
            break;
        }
    }
    if(foundTargetRole){
        break;
    }
}

boolean isFoundPS = false;
boolean isFoundPSRES = false;
for (Iterator iterator = templates.iterator(); iterator.hasNext();) {
    JSONObject template = (JSONObject) iterator.next();
    JSONArray roles = (JSONArray) template.get(Constants.ROLES);

    for (Iterator iterator1 = roles.iterator(); iterator1.hasNext();) {
        JSONObject role = (JSONObject) iterator1.next();
        String type = (String) role.get(Constants.TYPE);
        // find the vm-template that contains targetRoleTypePS role
        if (targetRoleTypePS.equals(type)) {
            targetRoles = roles;
            targetTemplate_PS = template;
            targetRole_PS = role;
            isFoundPS = true;
        } else if (targetRoleTypePSRES.equals(type)) {
            targetRole_PSRES = role;
            isFoundPSRES = true;
        }
    }
}

targetRoles.add(rolePSRES);
rolePSRES.put(Constants.PLUGIN, getPluginScope());
rolePSRES.put(Constants.NAME, Constants.ROLE_PSRES_NAME);
rolePSRES.put(Constants.TYPE, Constants.ROLE_PSRES_NAME);
JSONArray depends = new JSONArray();

```

```

rolePSRES.put(Constants.DEPENDS, depends);

// Add PSRES depends on RESStandalone/RESMgr
JSONObject dependRES = new JSONObject();
depends.add(dependRES);
dependRES.put(Constants.TYPE, targetRoleType_RES);
String roleRES = targetTemplate.get(Constants.NAME) + Constants.POINT +
targetRoleName_RES;
dependRES.put(Constants.ROLE, roleRES);

//Add PSRES depends on PS
JSONObject dependPS = new JSONObject();
depends.add(dependPS);
dependPS.put(Constants.TYPE, targetRoleType_PS);
String rolePSFullName = targetTemplate_PS.get(Constants.NAME) +
Constants.POINT + targetRoleName_PS;
dependPS.put(Constants.ROLE, rolePSFullName);

```

- c. Set parameters for the PSRES role (see Example 6-65 and Example 6-66).

Example 6-65 Define constants in Constants.java

```

public static final String ATTRIBUTES = "attributes";
public static final String PARMS = "parms";
public static final String PARA_TYPE_KEY = "type";
public static final String PARA_LINK_TYPE_VALUE = "PS-RES";
public static final String LINK_PROPERTY_PARAMETERS = "parameter";
public static final String TARGET_VM_NAME = "targetVmName";
public static final String LINK_PROPERTIES = "linkProperties";

```

Example 6-66 Set parameters for the PSRES role

```

// Add parameters
JSONObject attributes = (JSONObject) link.get(Constants.ATTRIBUTES);
JSONObject parms = null;
Object parmsObject = rolePSRES.get(Constants.PARMS);
if (parmsObject == null) {
parms = new JSONObject();
rolePSRES.put(Constants.PARMS, parms);
}else{
parms = (JSONObject) parmsObject;
}

attributes.put(Constants.PARA_TYPE_KEY, Constants.PARA_LINK_TYPE_VALUE);
JSONObject attrWrapper = new JSONObject();
attrWrapper.put(Constants.LINK_PROPERTY_PARAMETERS, attributes);
attrWrapper.put(Constants.TARGET_VM_NAME, targetVmName_RES);

JSONArray linkPropertyArray = null;
Object linkPropertyObject = parms.get(Constants.LINK_PROPERTIES);
if(linkPropertyObject==null){
linkPropertyArray = new JSONArray();
}else{
linkPropertyArray = (JSONArray) linkPropertyObject;
}

```

```
linkPropertyArray.add(attrWrapper);
parms.put(Constants.LINK_PROPERTIES, linkPropertyArray);
```

Define the part and lifecycle scripts

To add the part and role lifecycle scripts, follow these steps:

1. Define the psdmgrresdmgr.scripts part:
 - a. Create a part folder called psdmgrresdmgr.scripts under plugin/parts.
 - b. Define the role lifecycle script (see Example 6-67):
 - i. Add the PSRES role in the plugin/parts/psdmgrresdmgr.scripts/scripts folder.
 - ii. Add the PSDmgr role dependency in the plugin/parts/psdmgrresdmgr.scripts/scripts/PSRES folder, and add content to plugin/parts/psdmgrresdmgr.scripts/scripts/PSRES/PSDmgr/changed.py.

Example 6-67 The .../PSRES/PSDmgr/changed.py file

```
if myrole and 'BPM_DEPLOYMENT_STATUS' in maestro.deps[myrole] and
maestro.deps[myrole]['BPM_DEPLOYMENT_STATUS'] == 'ready':
    role['BPM_DEPLOYMENT_STATUS'] = 'ready'
    role['BPMDB_HOSTNAME'] = maestro.deps[myrole]['BPMDB_HOSTNAME']
    role['BPMDB_PORT'] = maestro.deps[myrole]['BPMDB_PORT']
    role['BPMDB_USERNAME'] = maestro.deps[myrole]['BPMDB_USERNAME']
    role['BPMDB_PASSWORD'] = maestro.deps[myrole]['BPMDB_PASSWORD']

if role.has_key('WODM_DEPLOYMENT_STATUS'):
if role['WODM_DEPLOYMENT_STATUS'] == 'ready':
    updateDBInfo()

def updateDBInfo():
    linkProperties = parms['linkProperties']
    update_folder = com_dir + '/updateENV'
    db_updater = update_folder + '/updateENVValue.sh'

    for linkItem in linkProperties:
        linkProperty = linkItem['parameter']
        link_type = linkProperty['type']
        if link_type == 'PS-RES':
            processAppAcronym = linkProperty['processAppAcronym']
            snapshotName = linkProperty['snapshotName']
            ruleIpEnvName = linkProperty['ruleIpEnvName']
            rulePortEnvName = linkProperty['rulePortEnvName']
            targetVMName = linkItem['targetVMName']
            vmNameValue = targetVMName

    #get DB info.
    db_hostname=role['BPMDB_HOSTNAME']
    db_port=role['BPMDB_PORT']
    db_username=maestro.deps[myrole]['BPMDB_USERNAME']
    db_password=maestro.deps[myrole]['BPMDB_PASSWORD']
```

```
maestro.trace_call(logger, [db_updater, update_folder, 'DB2',
db_hostname, db_port, db_username, db_password, processAppAcronym,
snapshotName, ruleIpEnvName, role[vmNameValue + '_resIp'],
rulePortEnvName, role[vmNameValue + '_resPort']])
```

```
maestro.role_status = 'RUNNING'
```

- iii. Add the RESDmgr role dependency in the plugin/parts/psdmgrresdmgr.scripts/scripts/PSRES folder, and add content to plugin/parts/psdmgrresdmgr.scripts/scripts/PSRES/RESDmgr/changed.py (see Example 6-68).

Example 6-68 The .../PSRES/RESDmgr/changed.py file

```
if myrole and 'resIp' in maestro.deps[myrole] and 'resPort' in
maestro.deps[myrole] and 'vmName' in maestro.deps[myrole]:
role['vmName'] = maestro.deps[myrole]['vmName']
vmNameValue = role['vmName']
role[vmNameValue + '_resIp'] = maestro.deps[myrole]['resIp']
role[vmNameValue + '_resPort'] = maestro.deps[myrole]['resPort']
if role.has_key('resVmList');
logger.debug('resVmList:')
logger.debug(role['resVmList'])
else:
role['resVmList'] = []

role['resVmList'].append(role['vmName'])
#get link number with type PS-RES
psres_len = 0
for link_item in parms['linkProperties']:
if link_item['parameter']['type'] == 'PS-RES':
psres_len = psres_len + 1
if len(role['resVmList']) >= psres_len:
role['WODM_DEPLOYMENT_STATUS'] = 'ready'

if role.has_key('BPM_DEPLOYMENT_STATUS'):
if 'ready' == role['BPM_DEPLOYMENT_STATUS'] and 'ready' ==
role['WODM_DEPLOYMENT_STATUS']:
updateDBInfo()

def updateDBInfo():
update_folder = com_dir + '/updateENV'
db_updater = update_folder + '/updateENVValue.sh'
for linkItem in linkProperties:
linkProperty = linkItem['parameter']
link_type = linkProperty['type']
if link_type == 'PS-RES':
processAppAcronym = linkProperty['processAppAcronym']
snapshotName = linkProperty['snapshotName']
ruleIpEnvName = linkProperty['ruleIpEnvName']
rulePortEnvName = linkProperty['rulePortEnvName']
targetVMName = linkItem['targetVMName']
vmNameValue = targetVMName
#get DB info.
db_hostname=role['BPMDB_HOSTNAME']
```

```

db_port=role['BPMDB_PORT']
db_username=maestro.deps[myrole]['BPMDB_USERNAME']
db_password=maestro.deps[myrole]['BPMDB_PASSWORD']
maestro.trace_call(logger, [db_updater, update_folder, 'DB2',
db_hostname, db_port, db_username, db_password, processAppAcronym,
snapshotName, ruleIpEnvName, role[vmNameValue + '_resIp'],
rulePortEnvName, role[vmNameValue + '_resPort']])

maestro.role_status = 'RUNNING'

```

- iv. Define the psdmgrresstandalone.scripts part.
- c. Create the psdmgrresstandalone.scripts part folder under plugin/parts.
- d. Define the role lifecycle script:
 - i. Add the PSRES role in the plugin/parts/psdmgrresstandalone.scripts/scripts folder.
 - ii. Add the PSDmgr role dependency in the plugin/parts/psdmgrresstandalone.scripts/scripts/PSRES folder, and add content to plugin/partspsdmgrresstandalone.scripts/scripts/PSRES/PSDmgr/changed.py (see Example 6-69).

Example 6-69 The .../psdmgrresstandalone.scripts/scripts/PSRES/PSDmgr/changed.py file

```

if myrole and 'BPM_DEPLOYMENT_STATUS' in maestro.deps[myrole] and
maestro.deps[myrole]['BPM_DEPLOYMENT_STATUS'] == 'ready':
    role['BPM_DEPLOYMENT_STATUS'] = 'ready'
    role['BPMDB_HOSTNAME'] = maestro.deps[myrole]['BPMDB_HOSTNAME']
    role['BPMDB_PORT'] = maestro.deps[myrole]['BPMDB_PORT']
    role['BPMDB_USERNAME'] = maestro.deps[myrole]['BPMDB_USERNAME']
    role['BPMDB_PASSWORD'] = maestro.deps[myrole]['BPMDB_PASSWORD']

if role.has_key('WODM_DEPLOYMENT_STATUS'):
    if role['WODM_DEPLOYMENT_STATUS'] == 'ready':
        updateDBInfo()

def updateDBInfo():
    linkProperties = parms['linkProperties']
    update_folder = com_dir + '/updateENV'
    db_updater = update_folder + '/updateENVValue.sh'

    for linkItem in linkProperties:
        linkProperty = linkItem['parameter']
        link_type = linkProperty['type']
        if link_type == 'PS-RES':
            processAppAcronym = linkProperty['processAppAcronym']
            snapshotName = linkProperty['snapshotName']
            ruleIpEnvName = linkProperty['ruleIpEnvName']
            rulePortEnvName = linkProperty['rulePortEnvName']
            targetVMName = linkItem['targetVMName']
            vmNameValue = targetVMName

    #get DB info.
    db_hostname=role['BPMDB_HOSTNAME']
    db_port=role['BPMDB_PORT']

```



```

db_username=maestro.deps[myrole]['BPMDB_USERNAME']
db_password=maestro.deps[myrole]['BPMDB_PASSWORD']
maestro.trace_call(logger, [db_updater, update_folder, 'DB2',
db_hostname, db_port, db_username, db_password, processAppAcronym,
snapshotName, ruleIpEnvName, role[vmNameValue + '_resIp'],
rulePortEnvName, role[vmNameValue + '_resPort']])

```

```

maestro.role_status = 'RUNNING'

```

- iii. Add the RESStandalone role dependency in the plugin/parts/psdmgrresstandalone.scripts/scripts/PSRES folder, and add content to plugin/parts/psdmgrresstandalone.scripts/scripts/PSRES/RESStandalonechanged.py (see Example 6-70).

Example 6-70 The .../scripts/PSRES/RESStandalonechanged.py file

```

if myrole and 'resIp' in maestro.deps[myrole] and 'resPort' in
maestro.deps[myrole] and 'vmName' in maestro.deps[myrole]:
    role['vmName'] = maestro.deps[myrole]['vmName']
    vmNameValue = role['vmName']
    role[vmNameValue + '_resIp'] = maestro.deps[myrole]['resIp']
    role[vmNameValue + '_resPort'] = maestro.deps[myrole]['resPort']
    if role.has_key('resVmList');
    logger.debug('resVmList:')
    logger.debug(role['resVmList'])
else:
    role['resVmList'] = []

role['resVmList'].append(role['vmName'])
#get link number with type PS-RES
psres_len = 0
for link_item in parms['linkProperties']:
    if link_item['parameter']['type'] == 'PS-RES':
        psres_len = psres_len + 1
    if len(role['resVmList']) >= psres_len:
        role['WODM_DEPLOYMENT_STATUS'] = 'ready'

if role.has_key('BPM_DEPLOYMENT_STATUS'):
    if 'ready' == role['BPM_DEPLOYMENT_STATUS'] and 'ready' ==
role['WODM_DEPLOYMENT_STATUS']:
        updateDBInfo()

def updateDBInfo():
    update_folder = com_dir + '/updateENV'
    db_updater = update_folder + '/updateENVValue.sh'
    for linkItem in linkProperties:
        linkProperty = linkItem['parameter']
        link_type = linkProperty['type']
        if link_type == 'PS-RES':
            processAppAcronym = linkProperty['processAppAcronym']
            snapshotName = linkProperty['snapshotName']
            ruleIpEnvName = linkProperty['ruleIpEnvName']
            rulePortEnvName = linkProperty['rulePortEnvName']
            targetVMName = linkItem['targetVMName']

```

```
vmNameValue = targetVMName
#get DB info.
db_hostname=role['BPMDB_HOSTNAME']
db_port=role['BPMDB_PORT']
db_username=maestro.deps[myrole]['BPMDB_USERNAME']
db_password=maestro.deps[myrole]['BPMDB_PASSWORD']
maestro.trace_call(logger, [db_updater, update_folder, 'DB2',
db_hostname, db_port, db_username, db_password, processAppAcronym,
snapshotName, ruleIpEnvName, role[vmNameValue + '_resIp'],
rulePortEnvName, role[vmNameValue + '_resPort']])

maestro.role_status = 'RUNNING'
```



Debugging and testing

This chapter provides information about the options that are available for debugging your plug-in and testing a pattern.

This chapter describes the following topics:

- ▶ The debug and unlock plug-ins
- ▶ Troubleshooting the transformer
- ▶ Using the Workload Plug-in Runtime perspective
- ▶ Manually debug the scripts

7.1 The debug and unlock plug-ins

The Plug-In Development Kit (PDK) contains the debug plug-in and the unlock plug-in as built-in components for development and debugging purposes.

The *debug plug-in* (*pdk-debug*) is meant for development, testing, and debugging purposes. It helps in debugging lifecycle scripts and topology documents in a plug-in. The debug plug-in provides a debug component. You have to insert this component into your pattern whenever you want to enable your pattern for debugging. The debug component provides two modes:

- ▶ Mock deployment
- ▶ Deployment for manual debugging

This chapter describes these options in detail later on.

The *unlock plug-in* (*plugin-unlock*) is also a useful plug-in during development. By default in IBM PureApplication System, you cannot delete a plug-in if it is being used by a deployed pattern. Using the unlock plug-in, you can override this setting and delete plug-ins even when they are in use.

The debug and unlock plug-ins are included with the PDK. You can find the `pdk-debug-x.x.x.x.tgz` and the `plugin.unlock-x.x.x.x.tgz` files in the folder where you have extracted the PDK.

7.1.1 Mock deployment

After you have created your plug-in, it is a good idea to perform a mock deployment. The debug plug-in has a Mock deployment mode for this purpose. The debug plug-in enabled in Mock deployment mode runs all of the transformations and generates the topology documents in the storehouse, but does not actually create a real virtual machine (VM) deployment in PureApplication System.

Figure 7-1 on page 151 shows **Mock deployment** selected as the Debug mode. You can provide a comma-separated List of ServiceProvisioner types to run if your plug-in is dependent on shared services.

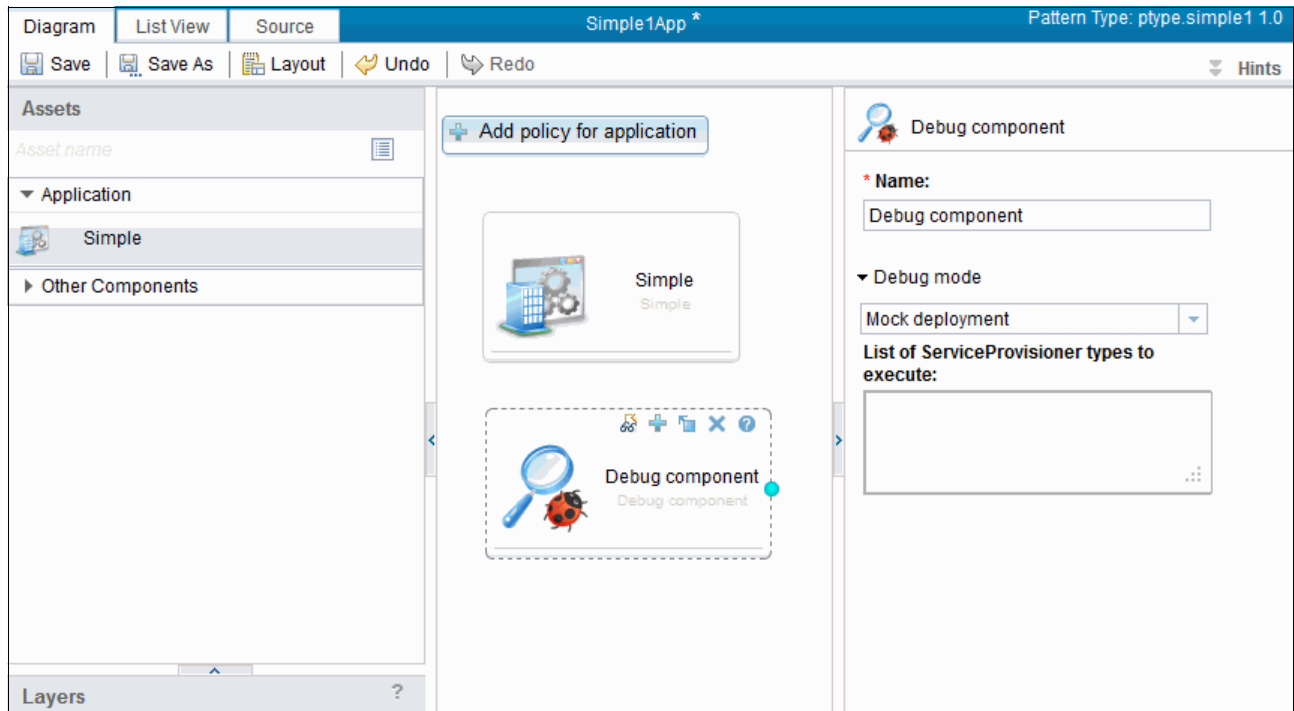


Figure 7-1 Debug Plug-in with mock deployment mode selected

If the deployment in this mode succeeds, you can examine the results:

1. Go to **System** → **Storehouse Browser**.
2. Browse through the storehouse browser, and select **User** → **deployments** → **<Your deployment ID>**.
3. Click **topology.json**, and then click **Get Contents**.
4. Use your favorite JavaScript Object Notation (JSON) viewer to go through the generated **topology.json** file to determine if there are any mistakes.

Figure 7-2 shows a sample topology.json file. Examine the sections for parts, roles, and packages to see if they are correct. If your plug-in uses shared services, check the service-templates section to verify them as well.

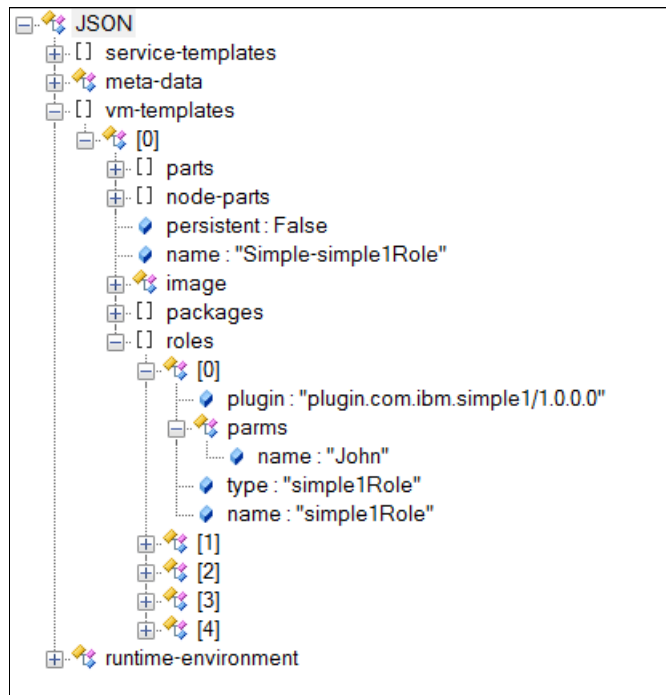


Figure 7-2 A sample topology.json object

The advantage of using this mode is that you can save time and resources by avoiding unnecessary VM deployments. It enables a faster debug cycle, where you can ensure that your topology documents are created as expected before you actually deploy a VM instance.

7.1.2 Deployment for manual debugging

When you want to debug manually, you have the following options:

1. Select the **Deployment for manual debugging** Debug mode when you need to debug the Python scripts in the deployed pattern. You can debug pretty much all of your plug-in scripts using this method.

The scripts can be part install scripts, node part scripts, role lifecycle scripts, or dependency scripts. Figure 7-3 on page 153 shows the Debug component in a sample pattern set to Deployment for manual debugging.

2. Select the **Resumable on script error** option to suspend the VM deployment when an error is detected in the scripts. You can fix the script at run time and resume the script execution to completion.

You can do this manually, as described in 7.4, “Manually debug the scripts” on page 162. To do this, you log in to the instance and fix and run the scripts manually.

You can also fix the script directly from the Eclipse integrated development environment (IDE), as described in 7.3, “Using the Workload Plug-in Runtime perspective” on page 154.

3. Select the **Record sensitive data** option if you want the sensitive inputs to be written into log files.

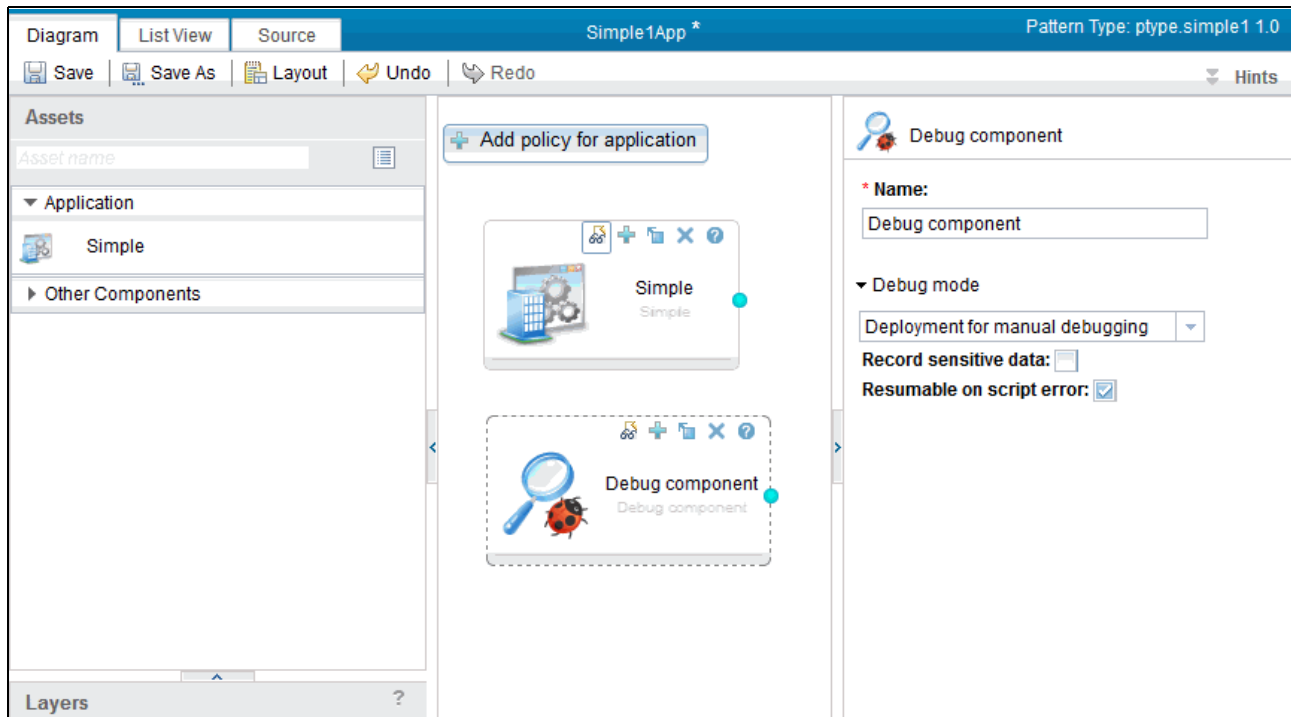


Figure 7-3 Debug component with mode in Deployment for manual debugging

7.2 Troubleshooting the transformer

Any output or debug messages that you print in your transformer code are available in the kernel service log files.

To view the kernel service log files, follow these steps:

1. Click **System** → **Troubleshooting**.
2. On the page that opens, click **Kernel Service log file**. You can see the console.log.<number> file, the trace.log.<number> file, and the ffdc.log.<number> file.

If you have errors in your transformer, you can examine these files for possible clues. For example, for the error in Figure 7-4 on page 154, the ffdc.log.<number> file contains the exception message, as shown in Example 7-1.

Example 7-1 Exception message in kernel service logs

```
-----Start of DE processing----- = [5/3/13 20:04:51:880 UTC], key =
com.ibm.maestro.common.http.HttpException: statusCode: 500 : CWZPL1014X: Cannot
find TopologyProvider for type simple1
com.ibm.maestro.mgmt.resources.AppMgmtResource 1107
Exception = com.ibm.maestro.common.http.HttpException
Source = com.ibm.maestro.mgmt.resources.AppMgmtResource
probeid = 1107
Stack Dump = com.ibm.maestro.common.http.HttpException: statusCode: 500 :
CWZPL1014X: Cannot find TopologyProvider for type simple1
at
com.ibm.maestro.model.transform.internal.TransformServiceImpl.getTopologyProvid
er(TransformServiceImpl.java:487)
```

```

        at
com.ibm.maestro.model.transform.internal.TransformServiceImpl.transformComponent(
TransformServiceImpl.java:528)
        at
com.ibm.maestro.model.transform.internal.TransformServiceImpl.transformAppModel(
TransformServiceImpl.java:408)
        at
com.ibm.maestro.model.transform.internal.TransformServiceImpl.generateTopology(
TransformServiceImpl.java:139)
        at
com.ibm.maestro.mgmt.resources.AppMgmtResource.deploy(AppMgmtResource.java:1156
)
        .....
        .....

```

In Figure 7-4, the error shows that the name defined in the Open Services Gateway Initiative (OSGi) Service component did not match with the name defined in the metadata.json file. As a result, the transformation service cannot find the TopologyProvider mentioned in the OSGi component.

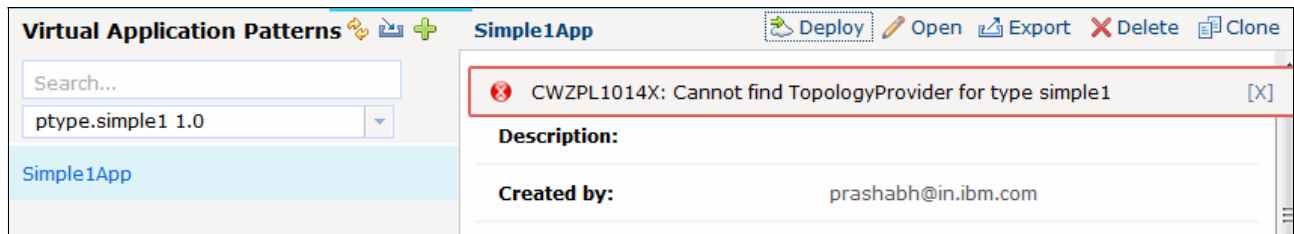


Figure 7-4 Error in resolving the topology

Figure 7-5 shows another error that occurred in the transformer. The error is that the package called *simple2* has no candidates. It seems that the *simple2* package is defined in the VM template (or TopologyProvider code if it is implemented through Java code), but there is no corresponding package defined in the config.json file. So, the pattern engine is unable to find the parts or roles for the *simple2* package in the config.json file.

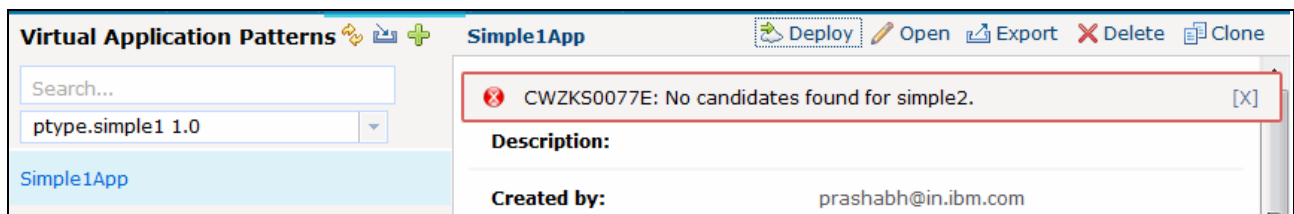


Figure 7-5 Error in the created package

In general, it is a good practice to use the debug component in Mock deployment mode until your transformers are running correctly, so that the topology that you create is correct.

7.3 Using the Workload Plug-in Runtime perspective

The Workload Plug-in Runtime perspective is an Eclipse perspective that is available in the Plug-in Development Kit (PDK). This perspective is specifically meant for debugging a deployed virtual pattern at run time directly from your Eclipse IDE.

The Workload Plug-in Runtime perspective is useful for ongoing development, unit test, and debugging of Python scripts.

The following section provides information about how you can use the Workload Plug-in Runtime perspective to debug a virtual application pattern (VAP) when it is deployed.

7.3.1 Upload the plug-in from your Eclipse workspace

After you have developed, built, and uploaded your plug-in into PureApplication System, follow these steps:

1. To build the pattern type, right-click your pattern type project and select **IBM Pattern Toolkit** → **Build**.
2. To upload the pattern directly from the PDK, right-click your pattern type project and select **IBM Pattern Toolkit** → **Install/Update to deployer**.
3. Note that you first need to configure the IBM Pattern Toolkit preferences through the **Preferences** → **IBM Pattern Toolkit**. Make sure that you enable the **Deploy debugging plug-ins before deployment** option. This option uploads the following two plug-ins before deploying your pattern:
 - The debug plug-in. This plug-in has the required code to suspend or resume the Python scripts (parts scripts, role lifecycle scripts, or dependency scripts) at a point of failure. It also enables you to fix a failed Python script, and resume the deployment from the failed step.
 - The unlock plug-in. By default, PureApplication System does not enable you to delete a plug-in that is detected to be in use. If a pattern is deployed using your plug-in, you cannot delete it. The unlock plug-in disables this feature so that you can delete and re-import the latest version of your plug-in during development.

If the plug-in already exists when you try to upload it, you are asked to confirm if you want to overwrite the existing plug-in. Click **Yes** and the plug-in is uploaded into PureApplication System.

7.3.2 Create and deploy a virtual application pattern

If you have to debug your deployment, you need to perform the following steps when you create the pattern:

1. Log in to the PureApplication System console and create a new pattern that you want to test.

Figure 7-6 on page 156 shows a simple pattern with a single component called *Simple*. It accepts an attribute called *Name*.

2. Import the debug and unlock plug-ins into PureApplication System. These plug-ins are included in the PDK. They are available in the PDK directory after you accept the PDK license:
 - a. Import the debug plug-in:
 - i. On the Workload Console, click **Cloud** → **System Plug-ins**.
 - ii. Click the Plus icon (+) on the top of the left frame.
 - iii. Browse to the PDK folder, and select **pdk-debug-x.x.x.x.tgz**.
 - b. Follow the same procedure to import the `plugin.unlock-x.x.x.x.tgz` plug-in into PureApplication System.

3. Adding the debug component into the pattern is important as well. It is available in the Other Components pane. The debug component ensures that the required code for debugging is available in the deployed virtual application instance.

Figure 7-6 shows a simple application for debugging.

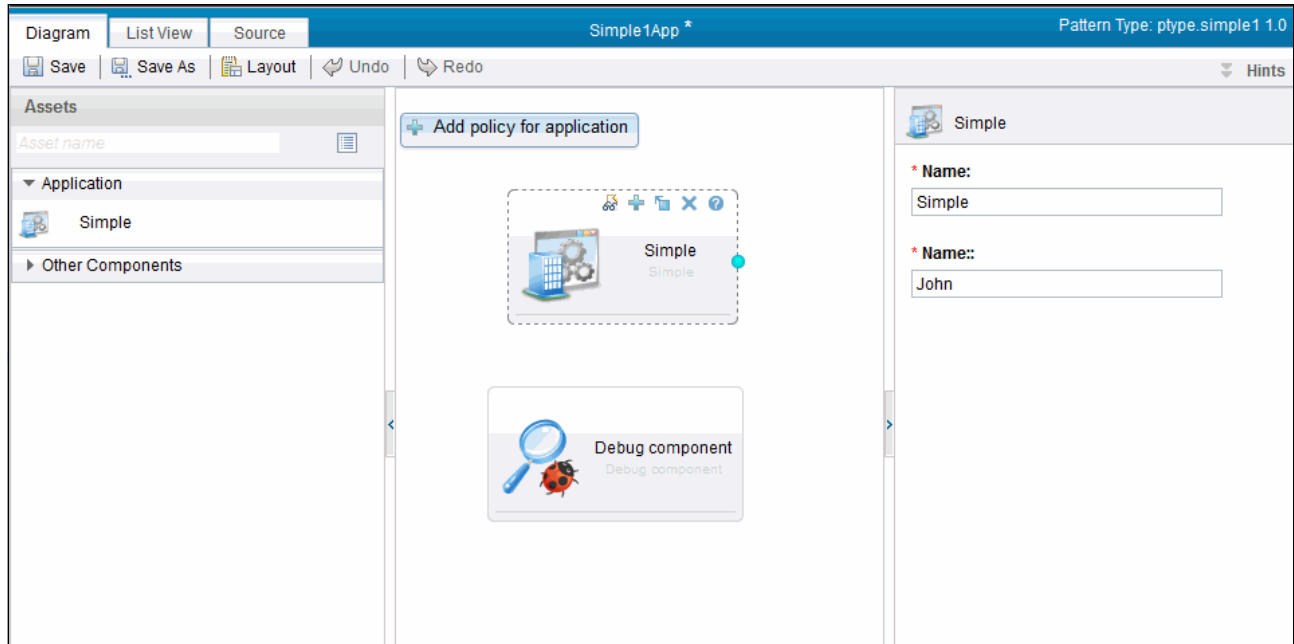


Figure 7-6 A simple pattern for debugging

4. In the Debug component pane, choose the following options, as shown in Figure 7-7:
 - a. Select **Deployment for manual debugging** as the Debug mode.
 - b. Select the **Resumable on script error** option.

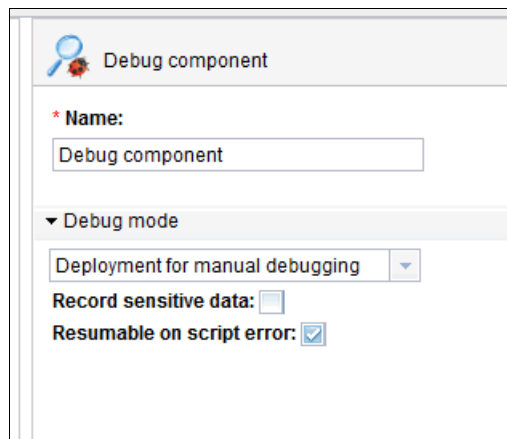


Figure 7-7 Attributes of the Debug component

5. Save the pattern that you just created.
6. Deploy the pattern.
7. When you deploy the pattern, provide a Secure Shell (SSH) key. This is required for the Eclipse IDE to connect to the deployed instance for debugging purposes.

You can choose to generate the private-public key pair and download the private key, or you can choose to supply your own public key for the deployment.

7.3.3 Connect to the deployed pattern

After you have deployed the pattern, connect to the deployed pattern from the Eclipse IDE through the Workload Plug-in Runtime perspective:

1. Switch to the Workload Plug-in Runtime perspective and click the Plus icon (+) in the Deployment Runtime explorer. A new window opens, as shown in Figure 7-8.
2. The IDE displays all deployed instances on PureApplication System that are configured in the environment. Alternatively, you can choose to manually enter a deployed VM.
3. Click **Test** to verify if the connection details are correct, and then click **Next**.

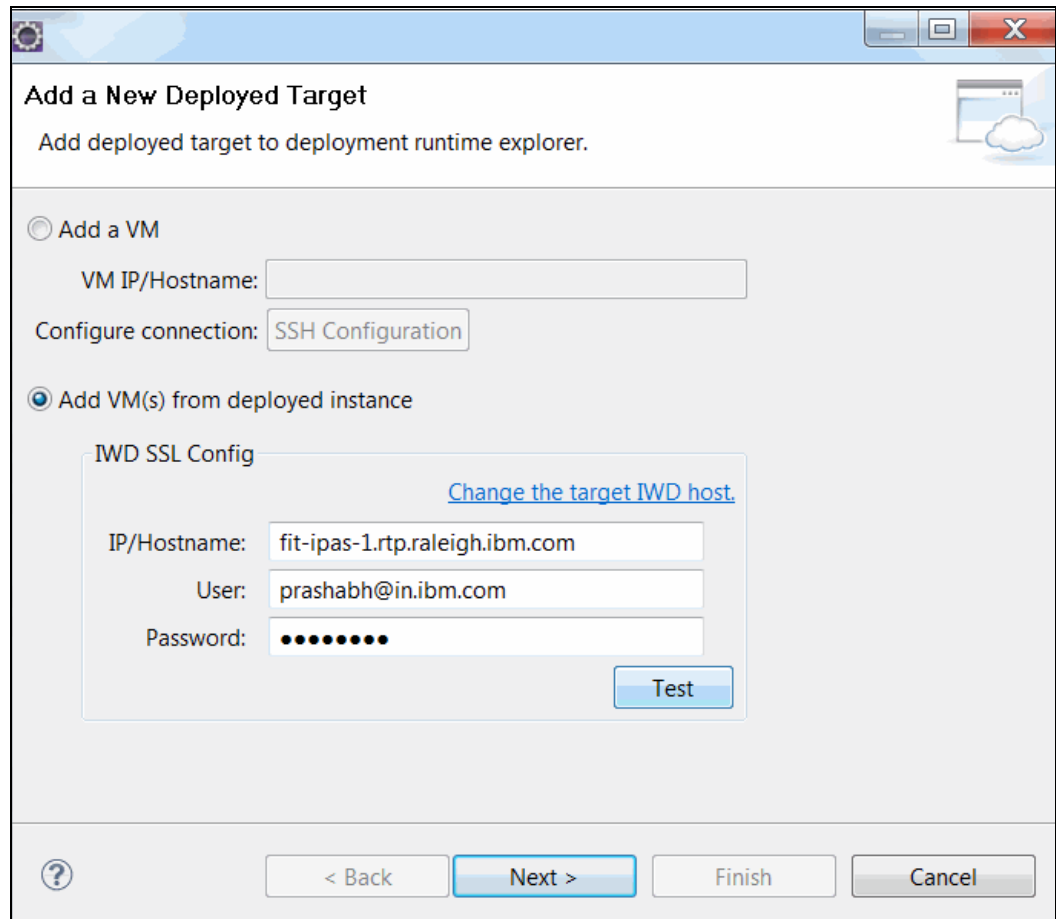


Figure 7-8 The Add a New Deployed Target window

4. A window showing the deployed instance opens, as shown in Figure 7-9. Choose your deployment from the list of targets and select **SSH Configuration**.

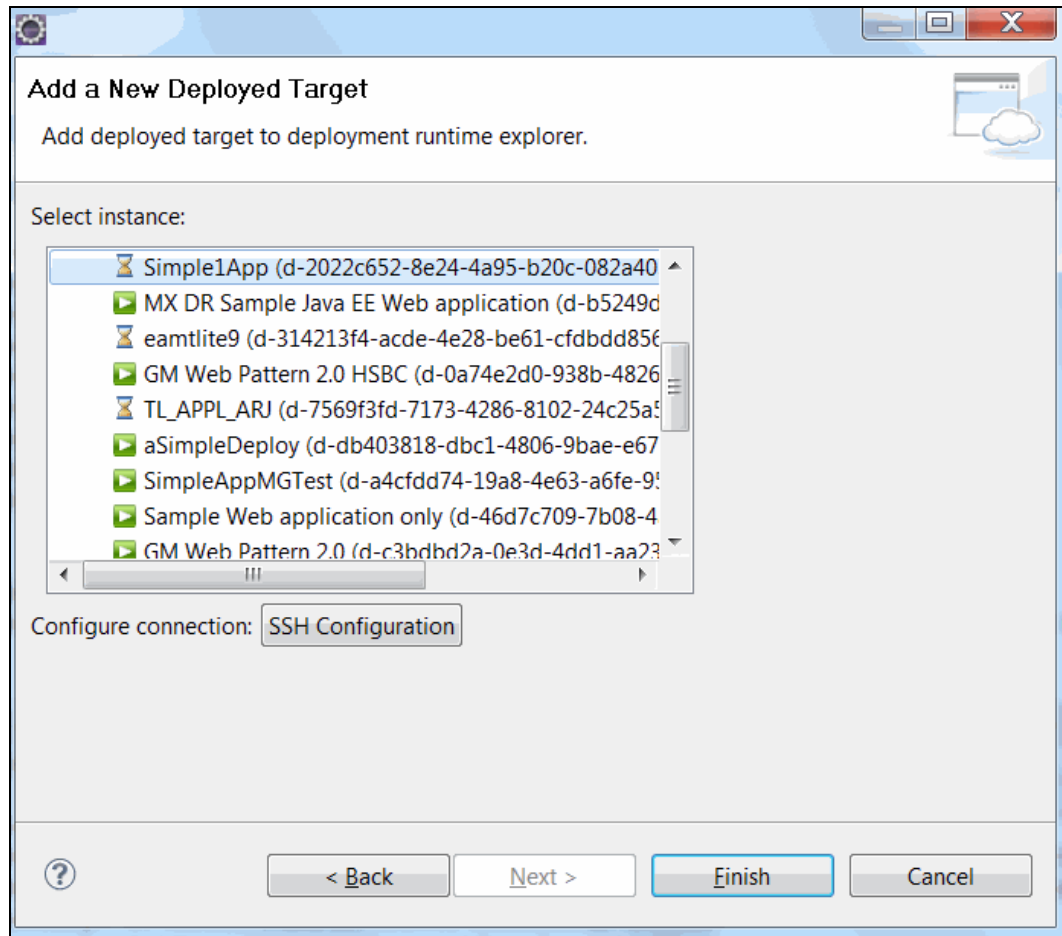


Figure 7-9 List of target VMs to which you can connect

5. The SSH Configuration is used by the Eclipse IDE during debugging. The Eclipse IDE connects to the deployed VM to upload changed scripts using this SSH key. In the SSH Configuration window, shown in Figure 7-10 on page 159, you have the option to select the mode:
 - *Key mode* means that you provide the private key of your public-private key pair.
 - *User/Password mode* means that you supply the username and password to access the deployed VM instance. For this option, you will have to log in once to the deployed VM instance, and set a user ID and a password.
6. Select the **Auto monitor running VM/Instance status** option. This means that you can track the live status of the deployment. Click **OK**, and then click **Finish**.

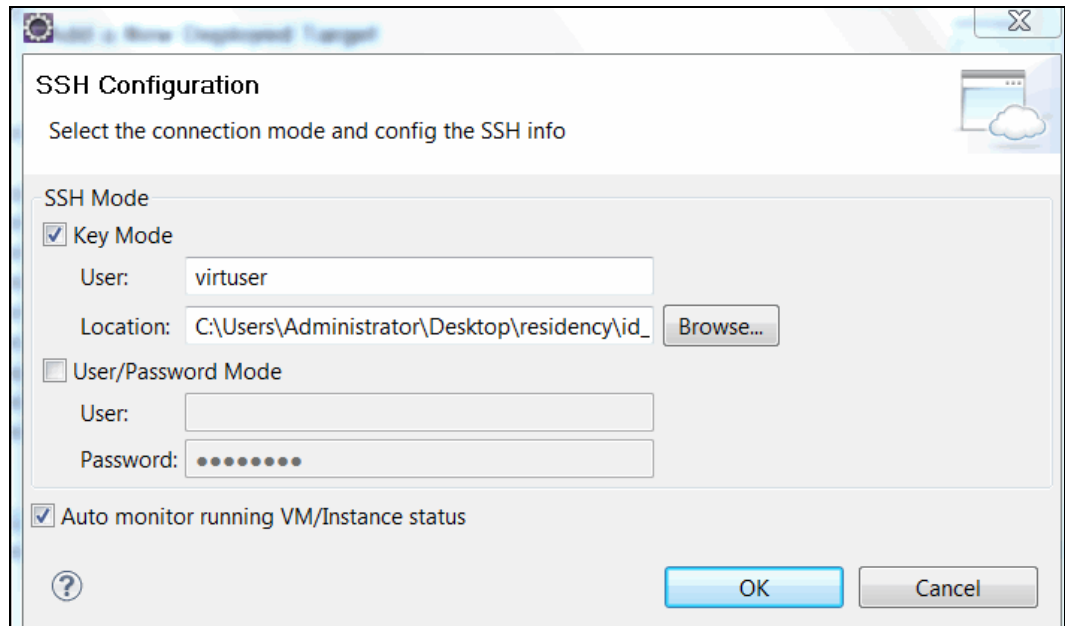


Figure 7-10 SSH Configuration options

7. On the Deployment Runtime Explorer window of your Eclipse IDE, you can see an item named *IP Unassigned*, as shown in Figure 7-11.

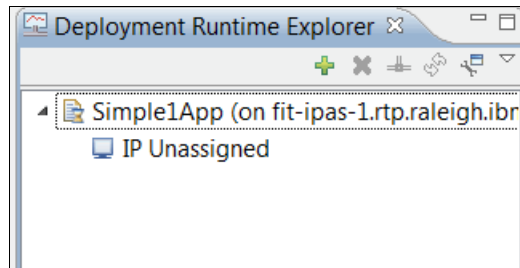


Figure 7-11 Deployment state in IP Unassigned

8. After the IP is assigned to the VM, and the deployment progresses, you will see that the status gets updated in the Eclipse IDE. Figure 7-12 shows an IP on the deployment. As the deployment progresses further, the details become visible in the Deployment Runtime Explorer.

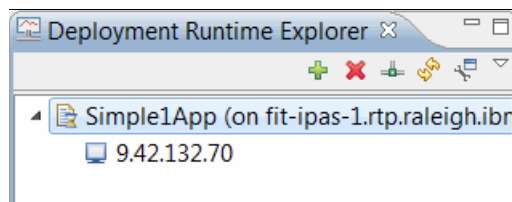


Figure 7-12 IP assigned now updated in the Eclipse IDE

7.3.4 Resume from the point of failure

After you have connected to the deployed pattern, you can verify if there was a failure. If there was, it is displayed in the IDE.

The Deployment Runtime Explorer shows the point of failure on the install tab in the upper-right pane, and the Script Log view in the lower-right pane, as shown in Figure 7-13. You can see the exact error that occurred in the Python script. In this example, the `parms[name]` is wrong.

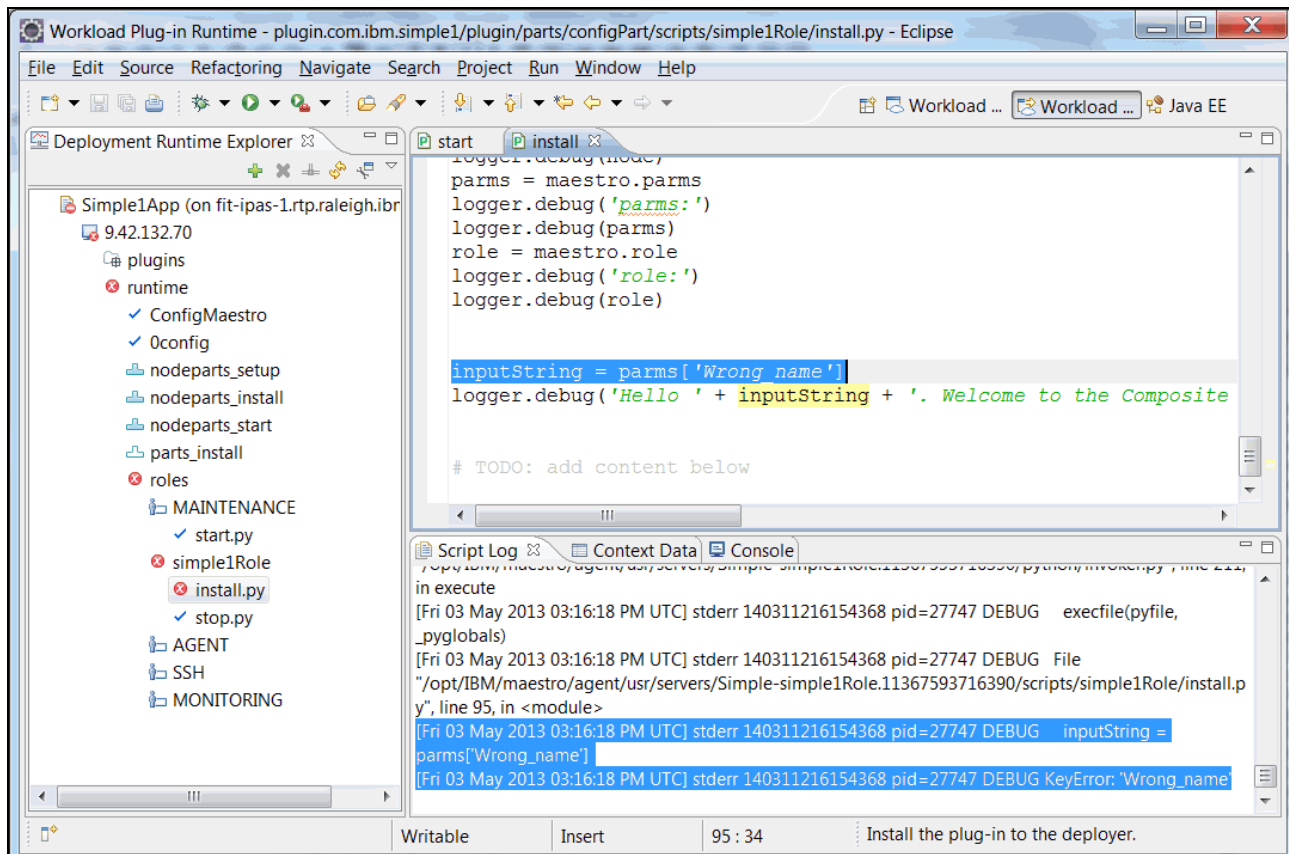


Figure 7-13 Error in a lifecycle script

After the error has been fixed, you can upload the updated script directly from the IDE into the deployed instance, and then resume with the deployment of the instance:

1. Right-click the script that you have just updated, and select **Upload and Resume**, as shown in Figure 7-14 on page 161.
2. You can rerun the failed scripts until all errors are fixed.

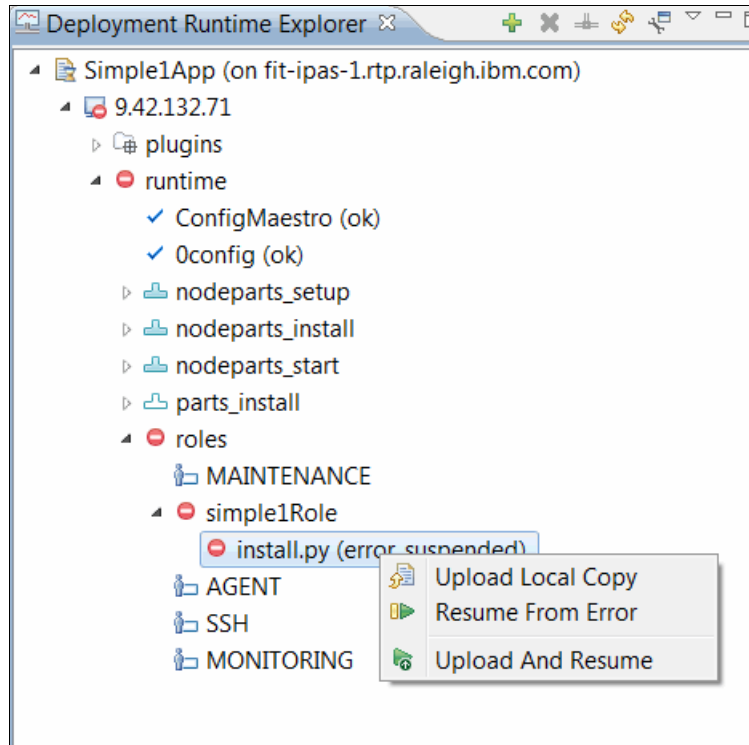


Figure 7-14 Upload locally made changes, and resume deployment

3. If you want to rerun all of the lifecycle scripts under a role, right-click that role and select **Restart from install.py**, as shown in Figure 7-15.

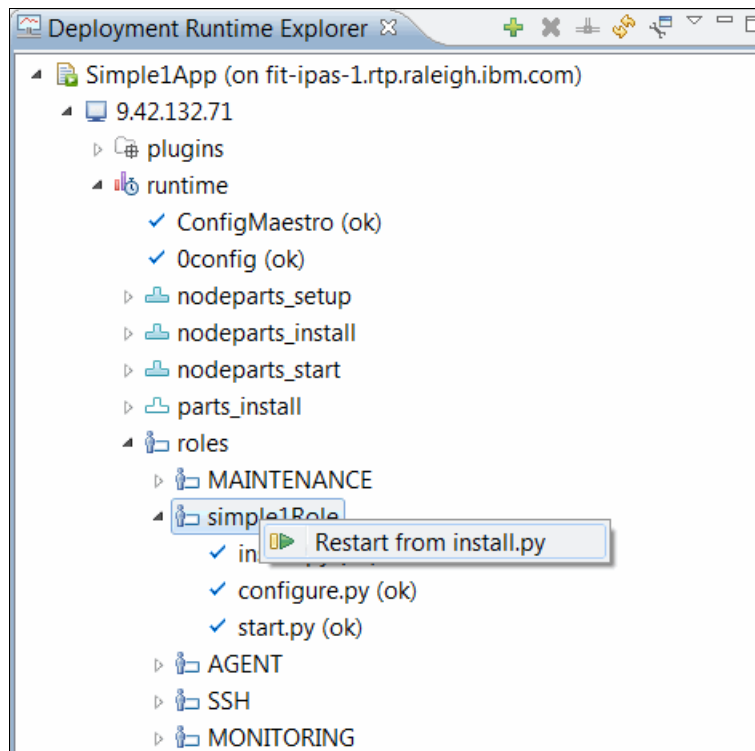


Figure 7-15 Restart the lifecycle of a role

Note that when you upload and resume a failed deployment, only the deployed VM is updated with the new scripts. If your changes have to be reflected into a new pattern deployment, you must upload your pattern type again.

7.4 Manually debug the scripts

In addition to debugging through the Eclipse IDE, as described in 7.3, “Using the Workload Plug-in Runtime perspective” on page 154, you also have the option to debug your scripts manually by logging in to the deployed virtual instance.

Enable manual debugging

To enable manual debugging, you need to have the debug component in your pattern:

1. Make sure that you have selected **Deployment for manual debugging** as the Debug mode for your debug component.
2. If you want the deployment to suspend when an error in the deployment scripts occurs, select the **Resumable on script error** option.
3. During deployment, make sure that you supply your SSH public key (or generate one through the console). You need the corresponding private key to log in to the deployed instance later.
4. After you deploy your pattern, if there is a failure in the parts install scripts, role lifecycle scripts, or dependency scripts, you will see a red error message in the Virtual Application Console, as shown in Figure 7-16.

At this point, the deployment is suspended.



Status:	Launching  Step 4 of 4  80% (Preparing middleware)
Using Environment profile:	Pure Enablement EP
Priority:	High
In cloud group:	Shared Cloud Group
Referenced shared services:	System Monitoring
Pattern type:	ptype.simple1 1.0
From pattern:	Simple1App
Error message:	The execution is suspended at script: /opt/IBM/maestro/agent/usr/servers/Simple-simple1Role.11367688837162/scripts/simple1Role/install.py

Figure 7-16 Deployment suspended at the point of failure

Fix errors and resume deployment

You can fix the errors in your deployment scripts on the VM, and then resume the deployment:

1. Log in to the VM instance as virtuser using your Private SSH key.
2. Switch to the root user.
3. Set up your environment for rerunning the scripts:
 - /0config/nodepkgs/common/scripts/pdk-debug/setEnv.sh

4. If you have made changes in the role lifecycle scripts, you cannot rerun the fixed scripts as is. You need to change to a particular directory and run the role lifecycle scripts. Run the **reqDirs.sh** command.
5. Find the folder directory that matches with the script that you need to rerun. For example, if the script that failed is `install.py` in the `simple1Role` folder, the output in Example 7-2 shows that you need to run the following command:

```
cd $NODEDIR/pyworkarea/requests/947052170504088311
```
6. If you need to change the input parameters that are supplied to the script from the topology document, you can open the `in.json` file in the same directory.
7. To manually rerun the script, run the following script:

```
$NODEDIR/scripts/simple1Role/install.py
```


 Make sure that you use the appropriate role type.
8. The `out.json` output file contains the output JSON object.

Example 7-2 Output from reqDirs.sh

```
$NODEDIR = /opt/IBM/maestro/agent/usr/servers/Simple-simple1Role.11367688837162
$NODEDIR/partsInstall/0/install.py RequestDir:
$NODEDIR/pyworkarea/requests/77732031642914658
$NODEDIR/python/log_injector.py RequestDir:
$NODEDIR/pyworkarea/requests/7055831409416548244
...
...
...
...
$NODEDIR/python/log_injector.py RequestDir:
$NODEDIR/pyworkarea/requests/7166240584618064648
$NODEDIR/scripts/simple1Role/install.py RequestDir:
$NODEDIR/pyworkarea/requests/947052170504088311
$NODEDIR/scripts/SSH/start.py RequestDir:
$NODEDIR/pyworkarea/requests/8693672242009793243
$NODEDIR/scripts/AGENT/start.py RequestDir:
$NODEDIR/pyworkarea/requests/4745983025238758677
$NODEDIR/scripts/simple1Role/configure.py RequestDir:
$NODEDIR/pyworkarea/requests/1825877938271024049
$NODEDIR/scripts/MAINTENANCE/start.py RequestDir:
$NODEDIR/pyworkarea/requests/1362709691172844145
$NODEDIR/scripts/MONITORING/start.py RequestDir:
$NODEDIR/pyworkarea/requests/4752115321385616215
$NODEDIR/scripts/MONITORING/change.py RequestDir:
$NODEDIR/pyworkarea/requests/1900404093440706917
$NODEDIR/scripts/simple1Role/start.py RequestDir:
$NODEDIR/pyworkarea/requests/8220960768490826789
```

9. After you fix the error in the script, run the **resume.sh** command to resume the script.

Important: When you run **resume.sh**, the pattern deployment resumes from the failed script, and the failed script is rerun. For example, if the `install.py` originally failed at line 30 with an exception, on resumption the whole of `install.py` reruns. Therefore, you have to undo the execution result from lines 1 - 30 manually before resuming.

10. Alternatively, if you need to rerun the node part scripts, set the environment using the **setEnv.sh** script, as mentioned in step 3 on page 162, and then run one of the following commands in the command-line interface (CLI):

```
cd /0config/nodepkgs/common/install (if it is an installation node part script)
```

or

```
cd /0config/nodepkgs/common/start (if it is a node part start script)
```

11. If it is a shell script, run the script itself on the directory.

12. If it is a .py script, run the following script:

```
runScript.sh <script-name>
```



Leading practices for plug-in design and implementation

This chapter highlights leading practices resulting from practical experience and feedback from the development team. By adopting these practices, you can reduce the time required to develop and deploy your virtual pattern types and plug-ins.

This chapter includes the following sections:

- ▶ Before you start
- ▶ Associating a plug-in with pattern types
- ▶ Plug-in design hints and tips
- ▶ How to manage binary files
- ▶ Using persistent VMs
- ▶ Platform consideration
- ▶ Using versions to accelerate development
- ▶ Naming convention
- ▶ How to set an endpoint URL to the pattern

8.1 Before you start

Before starting plug-in development, you need to review the following checklists and make sure that you have all of the information necessary to better plan and manage your plug-in. These checklists represent generic guidelines that you can use as appropriate. Add your own items based on your specific situation to ensure full coverage of your plug-in requirements.

8.1.1 Basic checklist

This basic checklist contains items that you need to address before actual development, to ensure that you have everything you need to develop a successful pattern:

- ☐ What are the name and prefix of the pattern?
- ☐ What is the size of the binary files that you need to install?
- ☐ What platform do you plan to support: Intel or Power?
- ☐ What existing middleware does it need to integrate with (for example, WebSphere Application Server and DB2 versions)?
- ☐ Is there a topology diagram?
- ☐ Are there existing install and configuration scripts that are already tested on the platform?
- ☐ How many virtual machines (VMs) do you want to create (minimum and maximum)?
- ☐ What are the User Interface (UI) components (plug-ins), and the links between those components (optional)?
- ☐ What are the properties for these UI components?
- ☐ Have you created a visual application pattern skeleton with placeholders for scripts?
- ☐ What are the keys for connecting to the VMs via Secure Shell (SSH) keys?

8.1.2 Advanced checklist

After you are done with the basic checklist, you might need to consider more advanced items. Use the following checklist to ensure that you have covered the majority of what you need:

- ☐ Do you need scaling? Which part of your software is scalable?
- ☐ What additional policies are required?
- ☐ What application templates will be useful?

8.2 Associating a plug-in with pattern types

A plug-in can be associated with a single primary pattern type, and with one or more secondary pattern types. There are also linked and prerequisite options that provide additional flexibility. The following sections provide more details about each option.

8.2.1 Primary pattern type

The `pattern types` element in Figure 8-1 on page 167 is used to associate a plug-in with pattern types. For a plug-in to be usable, its primary pattern type must be enabled within the PureApplication System GUI.

```

{
  "name": "pluginSimple",
  "version": "1.0.0.0",
  "patterntypes": {
    "primary": {
      "patterntypeSimple": "1.0"
    },
    "linked": {
      "webapp": "2.0"
    }
  }
},

```

Figure 8-1 Primary and linked association (config.json)

8.2.2 Secondary pattern type

You can associate the plug-in with one or more secondary pattern types. As depicted in Figure 8-2, the plug-in also declares secondary association with the IBM Web Application Pattern Version 2.0, which means that the enabled plug-in is also available for creating IBM Web Application Pattern workloads. At a minimum, you must specify one primary pattern type or one primary and one secondary pattern type.

Hint: If you need your plug-in to show up in the Virtual Application Builder (VAB) for all pattern types, set the secondary element as shown in the following example:

```
"secondary": [{ "*":"*"}]
```

```

{
  "name": "pluginSimple",
  "version": "1.0.0.0",
  "patterntypes": {
    "primary": {
      "patterntypeSimple": "1.0"
    },
    "secondary": {
      "webapp": "2.0"
    }
  }
}

```

Figure 8-2 Primary and secondary (config.json)

8.2.3 Linked option

The linked option depicted in Figure 8-1 provides additional flexibility. If a plug-in declares a linked relationship with a pattern type, it associates that linked pattern type to the plug-in's primary pattern type. Therefore, when you import the plug-in, any plug-ins associated with the linked pattern type are automatically associated with the plug-in's primary pattern type.

If you have configured your `config.json` file as shown in Figure 8-1 on page 167, all of the plug-ins associated with IBM Web Application Pattern are also automatically associated with `patternTypeSimple`. Therefore, when using `patternTypeSimple`, you can also see the assets of IBM Web Application Pattern in the Resource Palette.

Furthermore, you can use the linked option to extend an existing pattern type with additional plug-ins, without having to change the pattern type itself. The linked option is used at the plug-in level, and is defined in the `config.json` plug-in project file.

The linked option is defined as a *loose dependency*, which means that you can import and enable a pattern type no matter if the linked pattern type exists.

8.2.4 Prerequisites

Prerequisites are used at the pattern type level, and are defined in the `patternType.json` file, as shown in Figure 8-3. When pattern type A requires pattern type B, it means that pattern type A can be imported, but cannot be enabled before pattern type B is imported and enabled. So if a pattern type cannot work without other components in other pattern types, you can use prerequisites, because they are a strong dependency.

```
{
  "name": "patternTypeSimple",
  "shortname": "patternTypeSimple",
  "version": "1.0.0.1",
  "description": "",
  "prereqs": {
    "foundation": "*"
  },
}
```

Figure 8-3 Prereqs association (`patternType.json`)

8.3 Plug-in design hints and tips

You can refer to the following list of hints and tips when you are designing your plug-in:

- ▶ Create plug-ins for components that belong to same pattern type, and that need to be combined to implement some functions.
- ▶ If two components are linked and belong to the same plug-in, add the link to the plug-in, create a separate plug-in for the link, or add the link to the plug-in that includes either component.
- ▶ If policies are used on the pattern level, define separate plug-ins for the policies. If policies are used on the component level, add policies to the plug-ins that include the component.
- ▶ Define separate plug-ins for shared services.
- ▶ Develop a set of generic plug-ins to support several components. This is a more flexible way to deliver truly general-purpose components that can be extended to other complex components.

- ▶ If the disk value needed is larger than the initial disk value of a VM, add the disk to the VM using `storage-template` in the `topology.json` file.
- ▶ Make sure that the VMs open all of the ports through the firewall that they need to communicate with other VMs.

Use `maestro.firewall.open_tcpin(dport=<PORT_NUMBER>)` and `maestro.firewall.open_tcpout(dport=<PORT_NUMBER>)`

8.4 How to manage binary files

The default approach that you can use to handle binary files used by a specific plug-in is to bundle the binary files within the plug-in. The software binary files are included in the plug-in archive file, and are available for use when the pattern type and the plug-in are enabled in the IBM PureSystems administrative console.

This approach works fine for small binary files. However, the approach is not suitable for large binary files. Especially during plug-in development, repeatedly uploading large binary files can be prohibitive, particularly with a slow network.

Also, it makes the software tightly integrated with the plug-in development and enablement processes. This adds extra size to the image, and it affects the version if it changes from time to time.

To expedite the development process, and to better manage the binary files, you can group all of the binary files for a pattern type in a separate plug-in. That way, you can achieve these benefits:

- ▶ Update the other plug-ins that have the actual scripts in them easily.
- ▶ Leave the plug-in with the binary files alone.
- ▶ Keep the content in a storehouse.

8.5 Using persistent VMs

There are two ways for a plug-in to mark a VM as persistent:

- ▶ Add the persistent property to the `vm-template`.
- ▶ Add the persistent property to the package configuration.

If a VM stops unexpectedly, the master agent recovers the failed VM. The action depends on the VM type. A persistent VM is rebooted. Other VMs are replaced. Set this attribute to `true` for VMs that have recoverable state information.

8.6 Platform consideration

IBM PureSystems plug-ins are by nature platform-neutral. In some implementations, the scripts are single-source, which means that they are written to run on both the Linux and AIX platforms.

You might find that the implementation between the two platforms is much too different. If the plug-in is required to support multiple platforms, rather than limiting support to one of them, the plug-in developers must design an execution plan to support multiple platforms.

Otherwise, the ongoing development, maintenance, and debugging of the plug-in can become difficult rather quickly.

The following areas are particularly important for you to consider:

- ▶ Managing supported versus non-supported platforms
- ▶ Organizing files and scripts for multiple platforms
- ▶ Writing platform-neutral and platform-specific scripts
- ▶ Offering Microsoft Windows support

The following sections describe each of these areas in more detail.

8.6.1 Managing supported versus non-supported platforms

The plug-in configuration file, `config.json`, has a `requires` section for each package instance. If a plug-in supports multiple platforms with the same set of files, there is no need to declare a platform.

Otherwise, the `arch` clause can be used, along with other attributes such as `memory` and `cpu`, to indicate that the package definition is limited to support a particular platform.

Figure 8-4 on page 171 shows a snippet of a sample `config.json` file showing how to add different entries for each target platform. The package has two definitions, each with independent system requirements and parts.


```

{
  "name": "pluginSimple",
  "version": "1.0.0.3",
  "patternTypes": {
    "primary": {
      "patternTypeSimple": "1.0"
    }
  },
  "packages": {
    "componentSimple1Pkg": [
      {
        "requires": {
          "arch": "x86_64",
          "memory": 4000,
          "cpu": 2
        },
        "parts": [
          {
            "part": "parts\\componentSimple1.scripts.linux.tgz",
            "parms": {
              "installDir": "/opt/cmpSimple",
              "LinuxBinary": "$All_Binary"
            }
          },
          {
            "part": "parts/linux.scripts.tgz"
          }
        ]
      },
      {
        "requires": {
          "arch": "ppc_64",
          "memory": 4000,
          "cpu": 2
        },
        "parts": [
          {
            "part": "parts\\componentSimple1.scripts.aix.tgz",
            "parms": {
              "installDir": "/opt/cmpSimple",
              "AIXBinary": "$All_Binary"
            }
          },
          {
            "part": "parts/AIX.scripts.tgz"
          }
        ]
      }
    ]
  },
  "parms": {
    "All_Binary": null
  }
}

```

Figure 8-4 A config.json file snippet showing how to add multiplatform support

8.6.2 Organizing files and scripts for multiple platforms

The folder structure in the plug-in is tightly related to the package definition. For easier maintenance, always consider using the same set of files and scripts for multiple platforms. Indicate the target platform in the directory names. Normally, `lifecycle.py` scripts are platform-independent, and are the same for each platform.

Figure 8-5 shows a sample plug-in file structure for the Linux and AIX platforms.

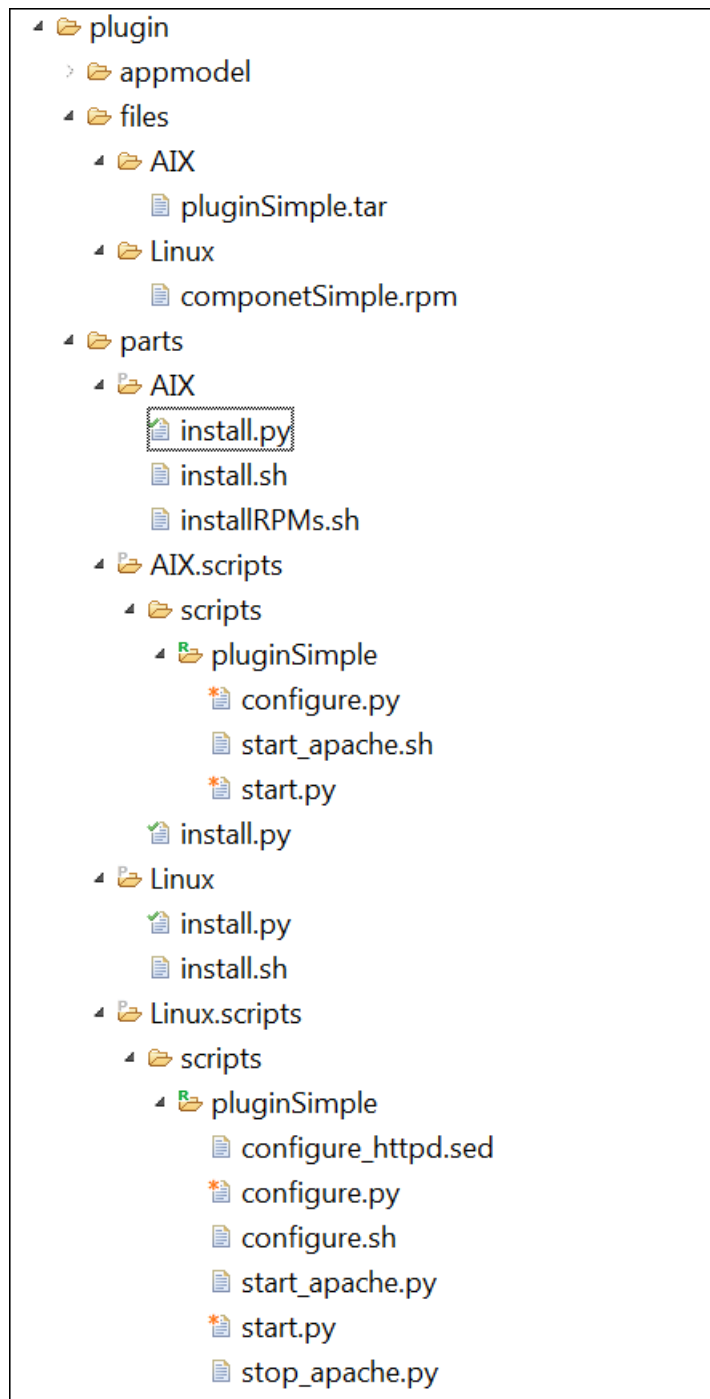


Figure 8-5 Sample file organization for multiplatform support

Note: The example shown in Figure 8-5 does not represent an actual file structure. It simply illustrates the idea of defining a different set of files for each supported platform.

The *Plug-in Development Guide* describes a well-defined structure for the Eclipse project that a plug-in implementation must follow.

For details about the structure, see the following website:

http://pic.dhe.ibm.com/infocenter/psappsys/v1r0m0/topic/com.ibm.ipas.doc/iwd/pgt_pluginovw.html

8.6.3 Writing platform-neutral and platform-specific scripts

The principle is to write the same set of scripts for both the Linux and AIX platforms wherever possible. For platform-dependent scripts, you need to write two sets of scripts, one for each platform. You need to give attention to the platform limitations in terms of commands and scripts.

8.6.4 Supporting Microsoft Windows

IBM PureApplication System V1.1 provides support for Microsoft Windows 2008 R2. You can create and deploy patterns with one or more components on Windows 2008 R2. You have to bring your own license design, which enables you to take advantage of existing investments.

Support is planned for the following patterns:

- ▶ Microsoft SQL Server 2008 R2 SP2
- ▶ Microsoft SharePoint 2010 SP1

8.7 Using versions to accelerate development

Each pattern type has a version in the `major.minor` format. During development, occasionally you want to deploy and compare multiple versions of a pattern type, or deploy new applications on IBM PureSystems with plug-ins.

For simplicity, you do not want to go through the pattern type cleanup cycle. So you can use a different major version to construct and install the pattern type to expedite the development process.

The plug-ins packaged in the pattern type also need to have a new major version, and to refer to the new pattern type version. For example, suppose that the currently installed pattern type version is 1.0.0.1, and the new pattern type version is 2.0.0.2. The plug-in can have a configuration such as the sample segment of the `config.json` file, shown in Figure 8-6.

```
{
  "name": "pluginSimple",
  "version": "2.0.0.2",
  "patterntypes": {
    "primary": {
      "patterntypeSimple": "2.0"
    }
  },
}
```

Figure 8-6 Plug-in version update configuration

8.8 Naming convention

In more complex environments, you manage multiple plug-in and pattern types. Therefore, you need to adopt a consistent naming convention to offer descriptive and meaningful names for your development artifacts.

There is no existing IBM rule for naming conventions in virtual pattern development for PureApplication System. Therefore, naming can follow internal guidelines, where each company or development team defines naming convention rules convenient to each team, and ensures that all team members abide by those rules.

Below are some guidelines for a naming convention for a project. You can use the same guidelines, or adapt them to suit your environment.

- For the pattern type project, you can start the name with `patterntype.xxxx`. The plug-in project can start with `plugin.xx.xx.xx`. It can use the name `plugin.com.company.<product abbreviation>`. For example, a file can be named `plugin.com.ibm.bpm`. This makes it clear in your workspace which file is a pattern type and which is a plug-in, as shown in Figure 8-7 on page 174.
- Usually, the link is a separate plug-in, and the name of a link is obvious, such as `plugin.xxx.xxx.<source product abb>.<target product abb>`. For example, the file could be named `plugin.com.ibm.bpm.psres` (versus `plugin.com.ibm.wasdb2`).

Figure 8-7 shows other examples of file names.

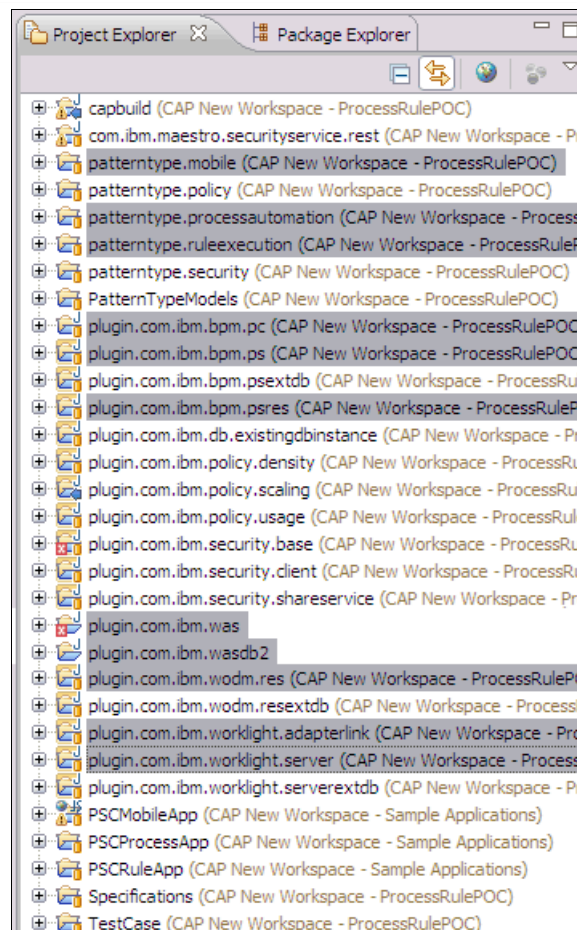


Figure 8-7 Patterntype and plug-in sample naming convention

- In the patterntype.json file, there are two attributes: Name and Short Name. Use a product name in the Name field that is also the pattern type name showed in PureApplication System.

The Short Name is the actual name referenced by the plug-in project. That needs to be a unique name. You should not have two different pattern types with same Short Name. See Figure 8-8 and Figure 8-9 for a real example.

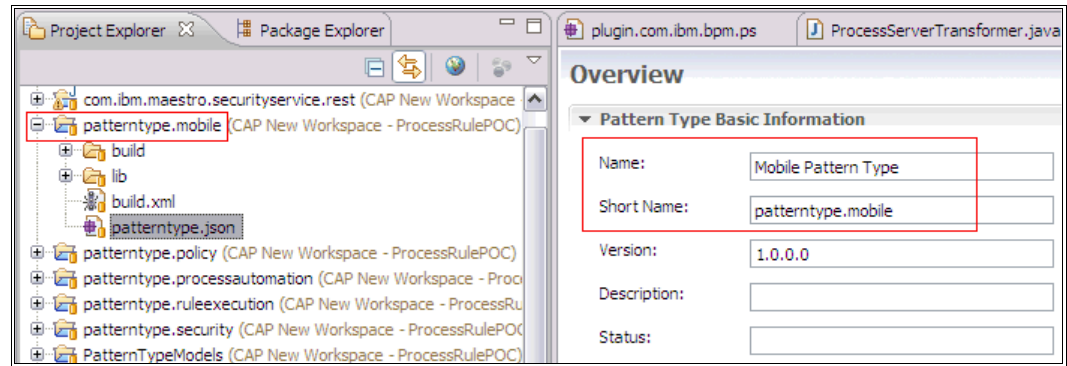


Figure 8-8 Pattern type Name and Short Name uses

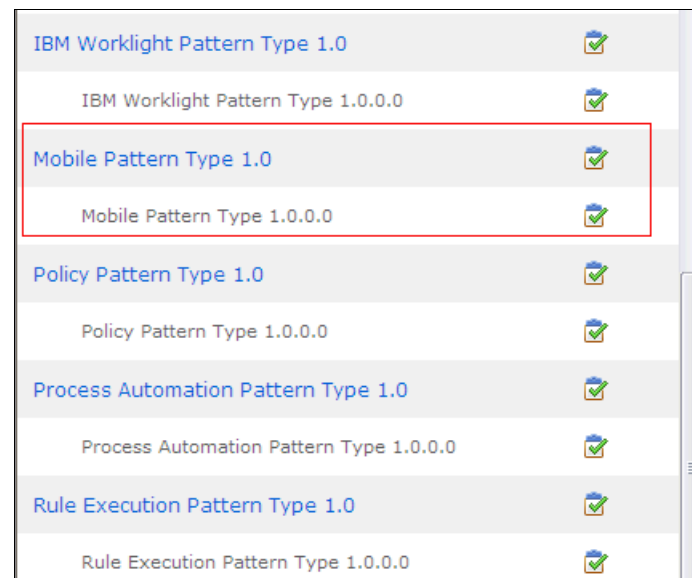


Figure 8-9 Pattern type descriptive name

- The component, link, and policies that you define in the metadata.json file have a unique ID, which is then used as the unique identifier for these elements in PureApplication System, as depicted in Figure 8-10.

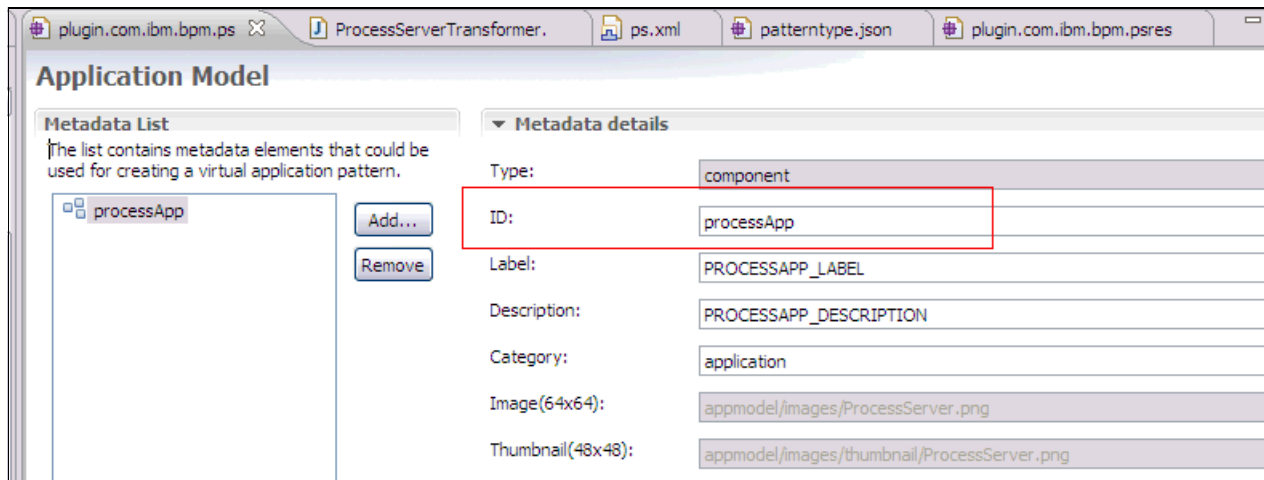


Figure 8-10 Setting the unique ID

8.9 How to set an endpoint URL to the pattern

When you deploy one of the built-in patterns, such as the WebApp pattern, you can see the URL (endpoint) next to the WebSphere Application Server role. You use this to access your deployed application, as shown in Figure 8-11.

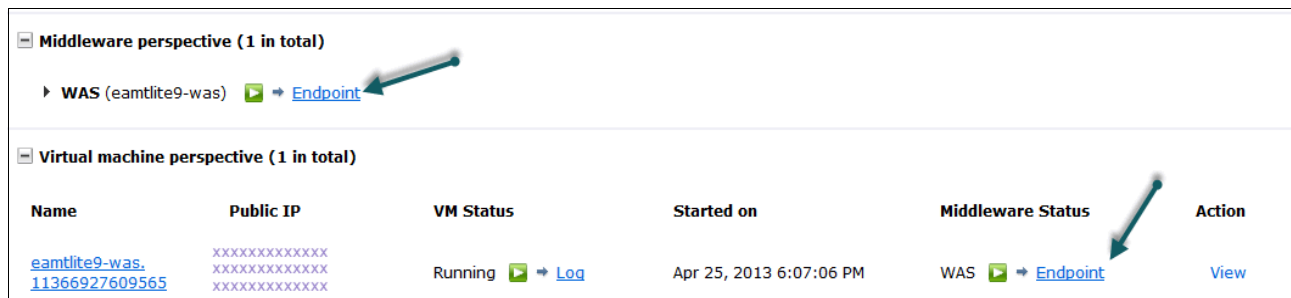


Figure 8-11 Endpoint view for a deployed pattern instance

If you want to add an endpoint to your pattern, follow these steps:

1. Locate the role in the roles array for which you want to add the link.
2. Insert a line similar to this example:

```
"external-uri": [{"ENDPOINT": "http://<SERVER>:8080/${attributes.appURL}"}]
```

The **<SERVER>** variable is replaced by the actual IP address of the server at deployment time.

You can also refer to other parameters defined in the tweak.json file by adding the parameter names between sets of braces, and they will be replaced by their values.



A

Additional material

This book refers to additional material that can be downloaded from the Internet, as described in the following sections.

Locating the web material

The web material associated with this book is available from the IBM Redbooks web server:

<ftp://www.redbooks.ibm.com/redbooks/SG248146>

Alternatively, you can go to the IBM Redbooks website:

<http://ibm.com/redbooks>

Select **Additional materials** and open the directory that corresponds with the IBM Redbooks form number, SG248146.

Using the web material

The additional web material that accompanies this book includes the following files:

- ▶ The Chapter6SourceCode.zip file. This is the source code for the Business Rule Application and Business Process Application pattern types.
- ▶ The patterntype.ruleexecution-1.0.0.0.tgz file. You can deploy the Business Rule Application pattern type to IBM PureApplication System by uploading this file.
- ▶ The patterntype.processautomation-1.0.0.0.tgz file. You can deploy the Business Process Application pattern type to PureApplication System by uploading this file.

Downloading and extracting the web material

To access the web material, follow these steps:

1. Create a subdirectory (folder) on your workstation, and download the contents of the web material file into this folder. Extract the sg248146.zip file into this folder to access the sample application files.
2. Download the software installation files listed in Table A-1 for the Business Rule Application pattern type from the IBM website:
<http://www.ibm.com>
3. Replace the files in the patterntype.ruleexecution-1.0.0.0.tgz\files\res folder.

Table A-1 Software installation files for the Business Rule Application pattern type

Software name	Description	File name
IBM WebSphere Application Server	WebSphere Application Server Network Deployment V8.0 (1 of 4) for Multiplatform, Multilingual	CZM9KML.zip
	WebSphere Application Server Network Deployment V8.0 (2 of 4) for Multiplatform, Multilingual	CZM9LML.zip
	WebSphere Application Server Network Deployment V8.0 (3 of 4) for Multiplatform, Multilingual	CZM9MML.zip
	WebSphere Application Server Network Deployment V8.0 (4 of 4) for Multiplatform, Multilingual	CZVG4ML.zip
IBM HTTP Server	WebSphere Application Server V8.0 Supplements (1 of 4) for Multiplatform, Multilingual	CZM91ML.zip
	WebSphere Application Server V8.0 Supplements (2 of 4) for Multiplatform, Multilingual	CZM94ML.zip
	WebSphere Application Server V8.0 Supplements (3 of 4) for Multiplatform, Multilingual	CZM95ML.zip
	WebSphere Application Server V8.0 Supplements (4 of 4) for Multiplatform, Multilingual	CZXR9ML.zip
IBM DB2	DB2 Enterprise Server Edition V10.1 for Linux on AMD64 and Intel EM64T systems (x64) Multilingual	DB2_ESE_10_Linux_x86-64.tar.gz
IBM Installation Manager	IBM Installation Manager V1.5.3 for Linux Multilingual	IBM_INSTALLATION_MGR_V1.5.3_LIN_ML.zip
WebSphere Decision Server	WebSphere Decision Server (IM Repository) V8.0 for Multiplatform, Multilingual	WS_DCSN_SVR-IM_REPO-V8.0_MP_ML.tar

4. Download the software installation files list in Table A-2 for the Business Process Application pattern type from the IBM website:
<http://www.ibm.com>
5. Replace the files in the patterntype.processautomation-1.0.0.0.tgz\files\bpm folder.

Table A-2 Software installation files for Business Process Application pattern type

Software Name	Description	File name
IBM Business Process Manager	IBM Business Process Manager Standard V8.0 for Linux 32 bit/64 bit (1 of 3) Multilingual	BPM_Std_V800_Linux_x86_1_of_3.tar.gz
	IBM Business Process Manager Standard V8.0 for Linux 32 bit/64 bit (2 of 3) Multilingual	BPM_Std_V800_Linux_x86_2_of_3.tar.gz
	IBM Business Process Manager Standard V8.0 for Linux 32 bit/64 bit (3 of 3) Multilingual	BPM_Std_V800_Linux_x86_3_of_3.tar.gz
IBM HTTP Server	WebSphere Application Server V8.0 Supplements (1 of 4) for Multiplatform, Multilingual	CZM91ML.zip
	WebSphere Application Server V8.0 Supplements (2 of 4) for Multiplatform, Multilingual	CZM94ML.zip
	WebSphere Application Server V8.0 Supplements (3 of 4) for Multiplatform, Multilingual	CZM95ML.zip
	WebSphere Application Server V8.0 Supplements (4 of 4) for Multiplatform, Multilingual	CZXR9ML.zip
IBM DB2	DB2 Enterprise Server Edition V10.1 for Linux on AMD64 and Intel EM64T systems (x64) Multilingual	DB2_ESE_10_Linux_x86-64.tar.gz
IBM Installation Manager	IBM Installation Manager V1.5.3 for Linux Multilingual	IBM_INSTALLATION_MGR_V1.5.3_LIN_ML.zip

Related publications

The publications listed in this section are considered particularly suitable for providing more detailed information about 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:

- ▶ *IBM PureApplication Systems Best Practices*, SG24-8145:
<http://www.redbooks.ibm.com/redbooks/SG248145>

Other Publications

The following book provides additional information about the topic of patterns in this document:

- ▶ Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994, ISBN 0201633612:
<http://amzn.to/1e101Ia>

Online resources

These websites are also relevant as further information sources:

- ▶ Hardware in IBM PureApplication System:
<http://www.ibm.com/developerworks/cloud/library/cl-ps-aim1302-hardwarepureapp/>
- ▶ User guide for Apache Velocity:
<http://velocity.apache.org/engine/devel/user-guide.html>
- ▶ PureApplication System Information Center:
http://pic.dhe.ibm.com/infocenter/psappsys/v1r0m0/topic/com.ibm.ipas.doc/iwd/pgt_installpdk.html
- ▶ WebSphere Operational Decision Management V8.0 Information Center:
<http://pic.dhe.ibm.com/infocenter/dmanager/v8r0/index.jsp>
- ▶ IBM Business Process Manager V8.0 Information Center:
<http://pic.dhe.ibm.com/infocenter/dmndhelp/v8r0mx/index.jsp>
- ▶ IBM PureApplication System Plug-in Development Guide and Information Center:
http://pic.dhe.ibm.com/infocenter/psappsys/v1r0m0/topic/com.ibm.ipas.doc/iwd/pgt_pluginovw.html

How to get Redbooks

You can search for, view, download, or order these documents and other Redbooks, Redpapers, Web Docs, drafts, and additional materials, at the following website:

<http://ibm.com/redbooks>

Help from IBM

IBM Support and downloads:

<http://ibm.com/support>

IBM Global Services:

<http://ibm.com/services>



Redbooks®

Creating Composite Application Pattern Models for IBM PureApplication System

Support elastic workloads using PureApplication System auto-scaling

This IBM Redbooks publication describes how IBM PureApplication System supports the creation of virtual systems and virtual applications. PureApplication System does so using a pattern model that enables you to take advantage of predefined, pre-configured, and proven middleware topologies and deployments.

Learn how to create your own custom pattern types

This book also presents an abstraction level that focuses on functional capabilities and applications, completely encapsulating the underlying middleware. It describes in detail the model and the associated frameworks in PureApplication System, as well as a methodology and approach toward designing and implementing a custom pattern model. This book shows concrete implementation examples that you can use when creating your own pattern model, paired with a collection of leading practices.

Build composite application patterns quickly and easily

This IBM Redbooks publication gives critical guidance to, and serves as a reference for, independent software vendors (ISVs) who want to create patterns for their packaged applications on PureApplication System. Clients who want to extend and enhance their existing patterns can also use this book.

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

SG24-8146-00

ISBN 0738438510