

Modernizing Applications with IBM CICS

Russell Bonner

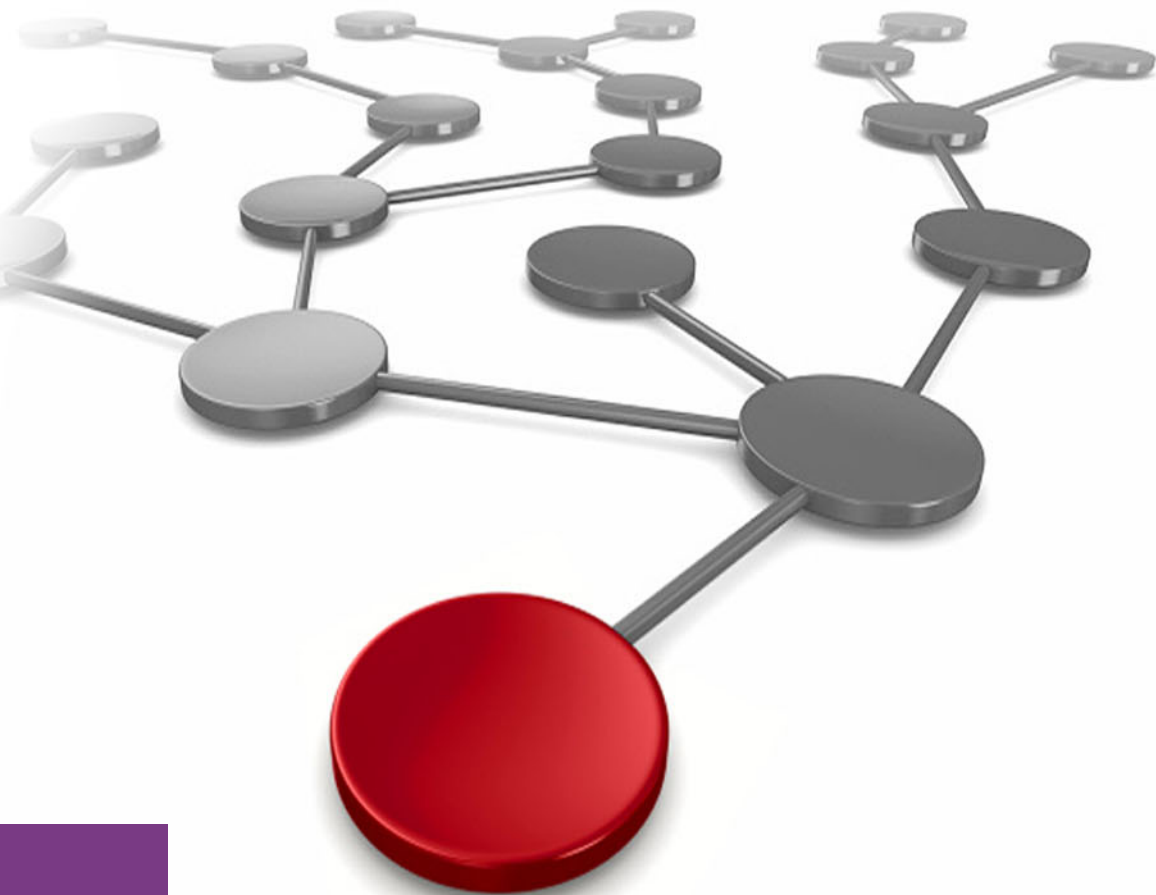
Sophie Green

Ezriel Gross

Jim Harrison

Debra Scharfstein

Will Yates



IBM Z



IBM Redbooks

Modernizing Applications with IBM CICS

December 2020

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

First Edition (December 2020)

© Copyright International Business Machines Corporation 2020. 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	v
Trademarks	vi
Preface	vii
Accompanying education course	vii
Authors	viii
Now you can become a published author, too!	viii
Comments welcome	viii
Stay connected to IBM Redbooks	ix
Chapter 1. Introduction	1
1.1 CICS and the hybrid multi-cloud	2
1.2 Migrating to the hybrid multi-cloud	2
1.2.1 Maintaining the status quo	2
1.2.2 Using cloud-native applications	2
1.2.3 Modernizing existing applications	3
1.3 CICS Hello World COBOL example	3
Chapter 2. IBM CICS application development	5
2.1 Application development in CICS	6
2.1.1 Batch processing versus online transaction processing	6
2.1.2 Programming paradigm	6
2.1.3 Basic architecture of a CICS program	7
2.1.4 CICS resources	9
2.2 CICS sample application	10
2.3 CICS modernization	11
2.4 CICS built-in transactions	12
2.4.1 CICS Execute Command Interpreter	12
2.4.2 CICS Execution Diagnostic Facility	13
Chapter 3. Coding applications to run in IBM CICS	15
3.1 Introduction to the EXEC CICS application programming interface	16
3.2 CICS API example	17
3.3 COBOL translator	18
3.4 Response codes	20
Chapter 4. Programming an IBM CICS application in COBOL	21
4.1 Presentation logic	22
4.1.1 Communication area	22
4.1.2 First time through processing	24
4.1.3 Function evaluation	25
4.1.4 Field validation and link to PAYBUS	26
4.1.5 Checking the return code from the link	27
4.1.6 Remaining presentation logic processing	28
4.2 Business logic	29
4.2.1 COMMAREA and special processing	30
4.2.2 Request analysis	30
4.2.3 Updating a record	32
4.2.4 Adding a record	33

4.2.5 Deleting a request.	34
4.2.6 Browsing forward and backward.	34
Chapter 5. Modernization by using channels and containers	37
5.1 Examining the existing functions.	38
5.2 Introducing channels and containers	38
5.3 From COMMAREA to channels and containers	39
5.4 Working with CICS programs in Visual Studio Code.	42
5.5 CICS and Zowe.	44
Chapter 6. Modernizing applications with Java	47
6.1 Why use Java with CICS.	48
6.2 Writing CICS Java applications.	48
6.2.1 Hello World code sample	49
6.2.2 Moving the Payroll application to Java	49
6.3 Unit testing Java applications	51
6.3.1 Writing a basic unit test.	51
6.3.2 Mocking with CICS applications	52
6.3.3 JCICSX remoting	52
Chapter 7. Modern IBM CICS application programming features	55
7.1 Asynchronous programming	56
7.1.1 Asynchronous programming analogy	56
7.1.2 Asynchronous programming principles.	57
7.2 Event processing.	60
7.2.1 Event processing in CICS.	60
7.2.2 Event processing example	62
7.3 Link to WebSphere Liberty	63
Chapter 8. DevOps and IBM CICS	67
8.1 Introduction to DevOps	68
8.2 DevOps on IBM Z with CICS applications.	68
8.2.1 The integrated development environment and debugging	69
8.2.2 Source code management	70
8.2.3 Build solutions.	70
8.2.4 Pipeline automation	70
8.2.5 Unit testing	71
8.2.6 Integration testing with Galasa	71
8.2.7 Deployment.	71
8.2.8 Analysis.	72

Notices

This information was developed for products and services offered in the US. This material might be available from IBM in other languages. However, you may be required to own a copy of the product or product version in that language in order to access it.

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, MD-NC119, Armonk, NY 10504-1785, US

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 jurisdictions 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 provide in any way it believes appropriate without incurring any obligation to you.

The performance data and client examples cited are presented for illustrative purposes only. Actual performance results may vary depending on specific configurations and operating conditions.

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.

Statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

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 actual people or business enterprises 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. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation, registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at “Copyright and trademark information” at <http://www.ibm.com/legal/copytrade.shtml>

The following terms are trademarks or registered trademarks of International Business Machines Corporation, and might also be trademarks or registered trademarks in other countries.

CICS®	IBM Z®	WebSphere®
CICS Explorer®	Redbooks®	z/OS®
CICSplex®	Redbooks (logo)  ®	
IBM®	UrbanCode®	

The following terms are trademarks of other companies:

Zowe is a trademark of the Linux Foundation.

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

IBM® CICS® is a mixed language application server that runs on IBM Z®. Over the 50 years since CICS was introduced in 1969, enterprises have used the qualities of service (QoSs) that CICS provides to allow them to create high throughput and secure transactional applications that have powered their business. As the IT landscape has evolved, so has CICS to allow these applications to integrate with new platforms and still provide value to the rest of the business. Because of this capability, many businesses still rely on CICS to power their core applications.

This IBM Redpaper publication focuses on modernizing these CICS applications, allowing them to integrate with cloud-native applications. This modernization can be achieved either by constructing application programming interfaces (APIs) that allow new cloud-native applications to connect to your existing assets, rewriting parts of your application in newer languages and hosting them back on CICS, or by using CICS capabilities to extend your applications to provide new capabilities and functions.

The paper takes a traditional example application and shows you how it works. Then, the paper extends the example, rewrites portions of its functions, and enables its APIs. It also explains how CICS applications can use continuous integration (CI) and continuous delivery (CD) to deliver, test, and deploy code into CICS easily and with quality.

Accompanying education course

This paper was developed in tandem with the education course *Modernize applications with IBM CICS*. This course contains a series of video lectures by IBM CICS Developer Sophie Green (see Figure 1) and a set of hands-on lab exercises where you access an IBM Z environment and follow step-by-step instructions to implement various modernization techniques with IBM CICS applications.

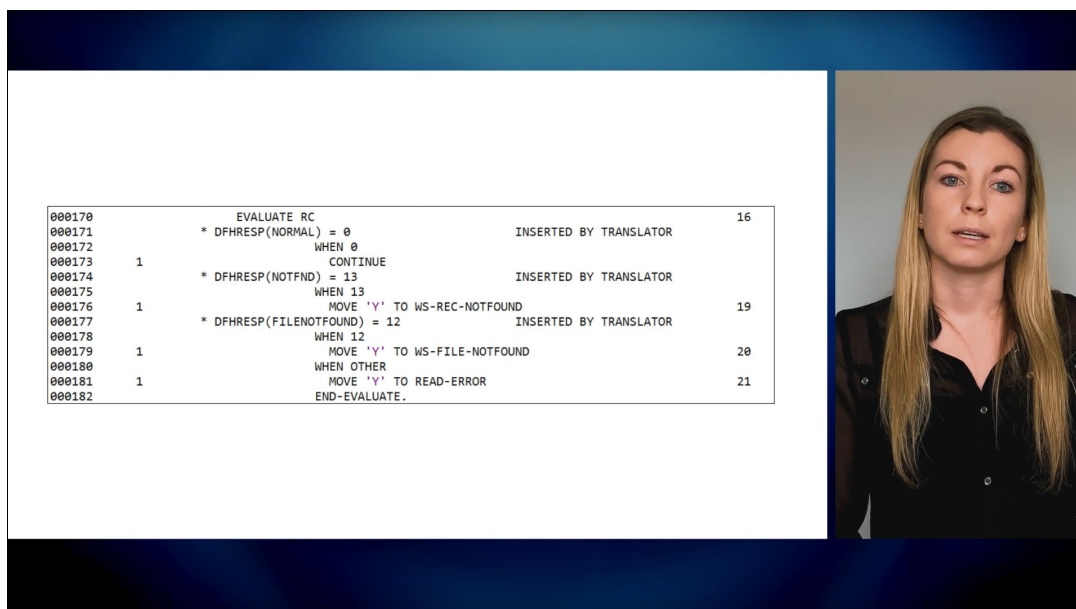


Figure 1 Accompanying education course

To enroll in this course, see [Modernize applications with IBM CICS](#).

Authors

This paper was produced by a team of specialists from around the world:

- ▶ Russell Bonner
- ▶ Sophie Green
- ▶ Ezriel Gross
- ▶ Jim Harrison
- ▶ Debra Scharfstein
- ▶ Will Yates

Thanks to the following people for their invaluable contributions:

- ▶ Kirk Brady
- ▶ Jonathan Gross
- ▶ Sudharsana Srinivasan
- ▶ Martin Keen

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 IBM Redbooks® residency project and help write a book in your area of expertise, while honing your experience by 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.

Find out 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 papers to be as helpful as possible. Send us your comments about this paper 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, IBM Redbooks
Dept. HYTD Mail Station P099
2455 South Road
Poughkeepsie, NY 12601-5400

Stay connected to IBM Redbooks

- ▶ Find 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>



Introduction

In this publication, you learn why IBM CICS has been the trusted core of enterprise applications and transaction processing for over 50 years and why through years of innovation this trust is maintained. This paper provides an overview for writing, updating, and running CICS applications and the new application programming interfaces (APIs), capabilities, and functions that you can use to modernize these applications to run as part of a hybrid cloud application.

1.1 CICS and the hybrid multi-cloud

IBM CICS is a mixed language application server that runs on IBM Z. Over the 50 years since CICS was introduced in 1969, enterprises have used the qualities of service (QoSs) that CICS provides to allow them to create high throughput and secure transactional applications that have powered their business. As the IT landscape has evolved, so has CICS to allow these applications to integrate with new platforms and still provide value to the rest of the business. Because of this capability, many businesses still rely on CICS to power their core applications.

When CICS debuted, the IT landscape was primarily dominated by batch processing. Anytime a record was updated, for example, a bank account or customer details, these changes would be batched together with all the other changes and applied sequentially to the main records. CICS introduced the ability to update that record immediately without the need for a batch window to run. CICS would handle all of the transactionality of the processing without the application programmer having to worry about what would happen if the record that you wanted to update was locked. As the landscape moved to embrace online transactions, CICS introduced capabilities to support different programming styles, such as conversational and pseudo-conversational. These capabilities allowed throughput to increase, and more people could update records at the same time.

Today, the cloud has changed the IT landscape again. Being able to use services, functions, and platforms as a service allows organizations to create digital experiences for their clients. However, at the heart of these experiences is still the core applications that are hosted by CICS. Since 89% of enterprises predict needing to harness multiple public and private or on-premises clouds, we call this landscape the *hybrid multi-cloud*.

1.2 Migrating to the hybrid multi-cloud

Migrating to the hybrid cloud requires clients to look at their existing IT assets and decide how they can migrate them to the cloud. There are three possible approaches:

- ▶ Maintaining the status quo
- ▶ Using cloud-native applications
- ▶ Modernizing existing applications

1.2.1 Maintaining the status quo

The simplest approach is to maintain the status quo and keep the existing applications working. Although this approach does not incur extra cost or introduce new risk, the application might fall behind technologically. This approach might eventually incur problems as the business tries to embed these applications into new digital experiences.

1.2.2 Using cloud-native applications

The second approach is to migrate all applications to the cloud as cloud-native applications. This approach is a high-risk one because there is no guarantee that the migrated application has the same performance, security, or transactional capabilities as the application that it replaces. It is also expensive. The cost of the migration will be high, and you must maintain the existing applications until the migration completes, which incurs further cost.

1.2.3 Modernizing existing applications

The final approach is to modernize the applications that you have and allow them to integrate with cloud-native applications. This modernization can be achieved either by constructing APIs that allow new cloud-native applications to connect to your existing assets, rewriting parts of your application in newer languages and hosting them back on CICS, or by using CICS capabilities to extend your applications to provide new capabilities and functions.

This type is the type of modernization that we focus on in this publication. This paper takes a traditional example application and shows you how it works. Then, the paper extends the example, rewrites portions of its functions, and enables its APIs. It also explains how CICS applications can use continuous integration (CI) and continuous delivery (CD) to deliver, test, and deploy code into CICS easily and with quality.

1.3 CICS Hello World COBOL example

Example 1-1 shows a small COBOL batch program that does nothing except write the phrase Hello World to the terminal.

Example 1-1 COBOL Hello World example

```
IDENTIFICATION DIVISION.  
PROGRAM-ID. HELLO.  
PROCEDURE DIVISION.  
    DISPLAY 'HELLO WORLD'.  
    STOP RUN.
```


CICS provides APIs to programs to allow them to interact with resources. Example 1-2 is the same code but uses CICS APIs to write the same string to the terminal.

Example 1-2 CICS COBOL Hello World example

```
IDENTIFICATION DIVISION.  
PROGRAM-ID. HELLO.  
PROCEDURE DIVISION.  
    EXEC CICS SEND TEXT ('HELLO-WORLD') END-EXEC.  
    EXEC CICS RETURN END-EXEC.
```

In Example 1-2, we look at the identification division of our COBOL program, and here we can see that our program is called “Hello”. The procedure division is where that the interesting stuff happens.

We have two API calls here: One to write the string Hello World to the terminal that started the program, and one to let CICS know that this program completed.



IBM CICS application development

CICS has been around for more than 50 years, and had over 25 different versions since it was first developed. This chapter explores some of the major application enhancements that have occurred over the years and describes the sample application that is used throughout this paper.

2.1 Application development in CICS

When CICS was first introduced in 1969, programming was done in the Assembler language, and CICS resources were invoked by using macro level coding techniques. Today, CICS uses modern programming interfaces while supporting traditional workloads.

2.1.1 Batch processing versus online transaction processing

To understand why CICS was created and it has been so popular over the years, you must understand the change in the programming models that took place in the early 1970s.

With the invention of devices that could interact directly with a user, the programming model had to change to support what was back then only moderately faster machines.

Before the invention of the “green screen”, which today many people describe as old and outdated, there were no screens. Data was processed in a batch manner. Whole files were fed into a program sequentially and the results were only fed back after all records were processed. In fact, sometimes the data was fed in using “punch cards”.

Although batch programming is still used today, the world is now interactive. We have more storage on our smartphones than we had on our mainframes 20 years ago. So, in the 1970s when computers became faster and processing needs became more immediate, IBM built CICS as one of the first transaction processors.

The idea was simple: The user is now a person not a machine or file. Therefore, the interaction is shorter but access to data is random. Because a user is waiting for a response, the processing time must be fast and in most cases subsecond. A good example of this situation is in our everyday life when we pay for something by using a credit card by swiping or placing it into a machine. The longer it takes for a response to come back, the fewer customers that can be served within a period.

2.1.2 Programming paradigm

To process data in a more immediate fashion and interact with a customer by using a 3270-like screen, a new set of programming paradigms needed to be considered. Consider the following types of transaction:

- ▶ Non-conversational transaction
- ▶ Conversational transaction
- ▶ Pseudo-conversational transaction

As you can see, the word *conversation* appears in each of the programming styles because we assume that you are having a conversation. The conversation is usually between the system and the user. The idea is that there is a beginning and an end to these conversations, but what happens in between can be different.

Non-conversational transaction

In the case of a *non-conversational transaction*, there is a start and an end but they run together so that the transaction is complete within one iteration. You can think of this transaction as an application programming interface (API) call where the result is available immediately and there is no need to continue when the result is received. For example, requesting the balance of your checking account requires that you provide the account number and receive the balance back. Any other work that you want to perform is not directly tied to this request.

Conversational transaction

A *conversational transaction* requires you to provide several inputs where the path is decided based on the inputs themselves. This interaction means that the program must respond to an input with a request for more input. Eventually, the transaction produces a final result, but it takes an uncertain path through the program to get there. What makes the programming style conversational is that every time a response for input is sent to the user, the system waits for the user to respond. The program is suspended, and any resources that are acquired by the program remain locked until the outcome is achieved.

A great example of this type of transaction is a menu system. The first screen prompts a user with several options, but only when the user responds will the program determine the next processing leg.

Pseudo-conversational transaction

So why do you need the last programming style? Well, imagine that a user enters a transaction that produces a screen and gets a phone call before they respond. They remain on the phone for an hour. As the program is suspended holding resources until they respond, there cannot be as many users in a system because the inconsistency alone is hard to predict. This programming style has the benefits of non-conversational programming but allows for the requirements of conversational.

Pseudo-conversational programming has the programmer end the task and the program right after it responds to the user with a prompt for input. This way, the latency that is associated with user “think time” is removed from the picture.

How does this style affect the underlying program? The answer is that the program must be able to be restarted where it left off when the user finally responds. To do this task, some state information must be kept in the system that allows the program to know where it left off and keep any data that is required to restart in the middle of a conversation.

This area is known as the communication area (COMMAREA). Although there are newer methods to keep state information between pseudo-conversations, the COMMAREA is still heavily used to hold state data for CICS programs and to pass information between different programs.

2.1.3 Basic architecture of a CICS program

With the adoption of the pseudo-conversational programming style, CICS regions could host many users running the same or different applications. However, in the mid-1970s address spaces were still only 16 MB, so the biggest limitation to adding more users and programs was space.

One of the solutions was to create separate CICS address spaces, which are known as *regions*, that were dedicated to individual services. Regions that hosted terminals were labeled TORs, the ones that housed applications are known as AORs, and data services were moved to FORs or DORs. At the same time, instead of creating a single program for whole applications, development was broken down into categories (see Figure 2-1).

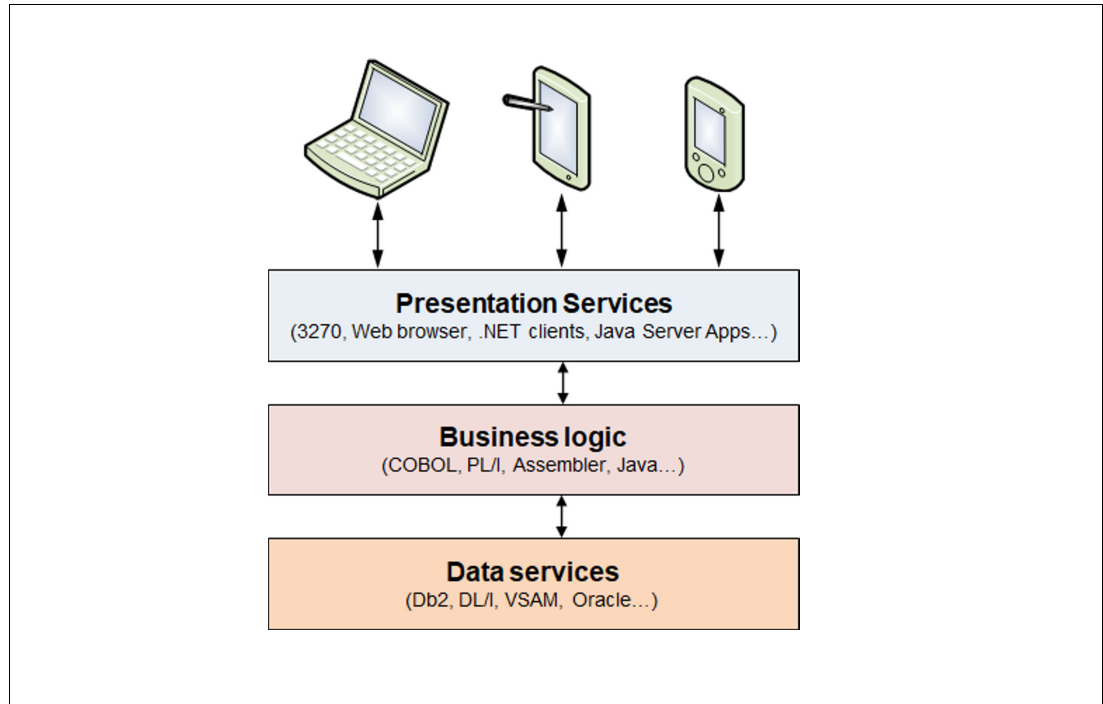


Figure 2-1 Basic application architecture

The categories that were initially created closely matched the region demarcations, but soon the industry adopted a modular programming style to allow for the maximum reuse of common routines. These developments caused applications to be split into many individual modules run over possibly multiple regions, and the modern CICS application was born.

As a result, CICS programming today is divided into three main categories.

Presentation services layer

The *presentation services layer* is used for all communication with the user. The advantage of using a separate presentation layer is the ability to swap out or add new forms of presentation without affecting the back-end business logic of the application itself. Imagine the transition from terminals to tablets to mobile phones for the services we use today. Each can have different requirements for how data is presented to the display, but require only a change in one category to support the new device.

Most presentation services have moved from CICS because other platforms tend to be more suitable for that processing. HTML pages that contain items such as radio buttons or selection lists can be easily processed at the local client, where CICS is simply used as the delivery mechanism to deliver the pages.

Business logic layer

The *business logic layer* is where you perform the work on the customer data, which is still primarily processed by application programs in CICS. Although customers have incorporated new languages such as Java and Node.js into these applications, the core COBOL applications remain because it would take many years to replace them.

CICS with IBM z/OS® can handle massive workloads with security, integrity, and performance that are unmatched by other platforms. Customers have attempted to rewrite whole CICS applications on other platforms and rarely have been able to match the reliability, integrity, and performance. Besides, if you take an application as a whole and rewrite it in another language for running on another platform and matched the performance in CICS, what would you have in the end? A large bill and the same thing with which you started.

Data services layer

The last layer is the *data services layer*. Even though many customers have chosen not to separate data access from business logic, the housing and access of data can change. One minute you are using VSAM files to house your data, the next you must query that data. With a separate data layer, the storage and retrieval of data is left to a module that can be replaced as requirements change.

Some programmers argue that the module in this layer can be used to go to other platforms through web services to retrieve data, whose underlying storage methodology is unknown to the program requesting it.

Over time, this separation of services has allowed CICS applications to be incorporated easily and quickly into newer applications, and in some cases with little or no changes to the underlying programs.

2.1.4 CICS resources

CICS is a resource-driven system that requires most resources be defined to the system before its use. Although this task is not usually the responsibility of an application programmer, it is important to understand because if this resource is missing, an application that might rely on it cannot function correctly.

As you embark on the road of programming in CICS, you learn quickly that there is a partnership between the application developer and the CICS systems programmer.

CICS was designed originally as a table-driven system, which means that if a resource is required in a CICS region, it must first be defined to the resource table for that resource. If it is not defined, the resource is not available in that CICS environment.

The function of defining resources was relegated to the CICS Systems programmer because they design the environment and choose where different pieces of an application run. They are also responsible for setting the attributes of that resource definition so that it best uses what that particular environment has to offer.

An example of a resource is a VSAM file. The file may be in the same region as the application, or the systems programmer can choose to place the file in a different region. The systems programmer can manipulate the definitions in CICS to place it in either region, and it works the same way for the application.

In terms of attributes, the systems programmer might decide that the VSAM file is a critical file and set it to be automatically backed out if a failure during application processing occurs.

Coordination between the applications developer and the systems programmer is required to set all attributes correctly. However, if the programmer is looking only to extend an existing application where the resources are defined and require no more CICS resources, then this task can be accomplished by the application developer without any interaction with the systems programmer.

2.2 CICS sample application

In this paper, we use an example application to describe some features within CICS. Specifically, we use a payroll system example (see Figure 2-2).

```
08/17/20                               10:52:04
                               Payroll System

Department No. :                      Employee No... :

                                Name
                                Address

                                Phone
                                Salary
                                Start Date :
                                Remarks

Please enter Dept and Employee Nos

F3=Exit: F4=Upd: F5=Add: F6=Del: F7=Prev: F8=Next
```

Figure 2-2 Payroll system sample application

A payroll application is a good example of a service that has been hosted in CICS for years by many large organizations. The sample program that we use is reduced to some simple functions so that the complexity of the source code is reduced.

The transaction ID is PAYR. When entered on a blank CICS screen, the initial program that starts is PAYPGM. This program contains the presentation logic for the application and determines its state. For example, the first time the program starts, there is no state data, so the program sends the initial screen, which in CICS is known as a MAP. Maps can be stored in a MAPSET, which can contain more than one map for the application. The MAP that is used in our application is called STUDMAP.

After the first time that the program starts, the program must query the state data to determine whether it is in the middle of a function, such as an update or whether the request is simply a display of a record.

Input to this program is supplied through the 3270 screen. It accepts a 1-byte department number and a 5-byte employee number. Entering these values and pressing the Enter key or a function key reruns the PAYPGM program, which reads the values that are entered and determines the function to run.

The only field validation that is done by this program is for the department and employee fields. They must be physically entered because they are required fields and must also be numeric. All other field validation for things such as date are removed to simplify the logic.

After PAYPGM has validated these fields, it determines the function that the user is requesting. The program builds this function into a COMMAREA and passes it to PAYBUS for processing.

PAYBUS is the business logic program for this application. It determines the function that is requested, such as ADD or DELETE, and invokes the necessary section of the code to process the request. Although this task might seem relatively simplistic and non-conversational, it is a true pseudo-conversational example due to the interaction with the 3270 screen.

Imagine that you want to ADD an employee. You enter a department and employee number, but the rest of the screen appears to be locked. When you enter the data into PAYPGM, the program determines that this request is an ADD request and calls PAYBUS to verify that this person does not exist. If that is the case, then it returns that information to PAYPGM, which responds to the user by sending the screen and unlocking the fields to allow the ADD to proceed.

When the user enters all the data and requests the ADD a second time, the record is added. Therefore, ADD is a two-step process that requires a conversation. The code that is used is pseudo-conversational and a good example of how that processing takes place.

Finally, for simplicity the data is stored on a VSAM Key Sequenced Data Set (KSDS) file so the data portion is not separated into another program.

2.3 CICS modernization

As the world moved from 3270 green screens to browsers, IBM provided several modernization techniques to allow CICS applications to thrive in the new world.

One of the early application modernizations in CICS was called the *3270 web bridge*. This item allowed a programmer to get an HTML representation of their MAP, which was produced when the MAP was assembled. The bridge and a program that was supplied by CICS that was called DFHWTBTA allowed the application to run from a browser with no changes to any of the applications. The screens were primitive, but CICS supplied utilities to make them look more like web pages (see Figure 2-3).

CICS Web Interface BMS screen emulation

Payroll System

Department No. : 1

Employee No... : 1

Name IBM REDBOOKS CLASS

Address 123 MOCKINGBIRD LANE

POUGHKEEPSIE

NEW YORK

Phone 75529900

Salary \$1234.56

Start Date : 28101984

Remarks REDBOOKS ARE GREAT

F3=Exit: F4=Upd: F5=Add: F6=Del: F7=Prev: F8=Next

PF1 PF2 PF3 PF4 PF5 PF6 PF7 PF8 PF9 PF10 PF11 PF12

PF13 PF14 PF15 PF16 PF17 PF18 PF19 PF20 PF21 PF22 PF23 PF24

PA1 PA2 PA3 Clear Enter Reset

Figure 2-3 CICS web interface

As customers embarked on the path of separating presentation logic from business logic, IBM created features such as CICS Web Support and web services, which allowed the programmer to bypass the presentation layer and simply pass data directly to the business logic.

These features support passing the data as name-value pairs for HTML pages and as XML for use by a client web service. With these new features, CICS allows a programmer to turn the back-end business logic into an XML-based service.

Today, there is support to create RESTful services by using JSON with various free and for purchase utilities so that CICS applications can be supported by mobile applications or any other application within the enterprise.

Other modernizations include the ability of CICS applications to be clients of external web services. For example, there is a CICS application that used to pay an external service for address scrubbing, but now calls a free service on the web to provide the same function.

2.4 CICS built-in transactions

To help application development, CICS provides several transactions or built-in functions that are known as the CICS supplied transactions, which provide the ability to build and test CICS application programs.

CICS supplies over 30 transactions that both system and application programmers can use to monitor, control, and build applications in CICS. Here we highlight a few of the ones that are most used by CICS applications programmers while developing applications. By using these transactions, applications programmers can test, debug, and modify data that is associated with an application.

Here are some of the applications:

- ▶ CICS Execution Diagnostic Facility (CEDF)/CEDX: Execution diagnostic facility transactions. They provide interactive debugging.
- ▶ CADP/DTCN: Provides access to the CICS Debug Tool, which is a Source-Level Debugger that is supplied with LE370.
- ▶ CICS Execute Command Interpreter (CECI)/CECS: A command interpreter transaction. Allows **EXEC** CICS statements without coding a program.
- ▶ CEBR: Allows a programmer to browse through CICS Temporary Storage or Transient Data Queues (TDQs).

The two most used transactions are CECI and CEDF.

2.4.1 CICS Execute Command Interpreter

With CECI, you can test a CICS command before coding it in an application program. In addition, it runs the command in most cases so that the changes are permanent in the system. For example, if you delete a record from a file by using CECI, it permanently deletes the record from the file.

CECS works the same way as CECI, but it does not run the command. It does only a syntax check so that a programmer knows exactly what attributes are required to properly code the command in a program.

2.4.2 CICS Execution Diagnostic Facility

Another one of the built-in transactions is the CEDF. CICS provides two transactions that you can use for testing and following the flow of application programs without having to supply special program testing procedures. It has a sister transaction that is called CEDX, which allows the testing of non-terminal-based applications. Using CEDX, you can enter an application transaction as input to test on a 3270 screen even though the underlying transition that is captured is not terminal-based.

Type CEDF on a blank screen and the message that is returned lets you know that CEDF is active. Clear the screen and type in the application transaction, which in our example is PAYR. The following screen is displayed.

CEDX works the same way, but you type CEDX, a space, and then the user transaction. CEDX captures that transaction on the users' current screen if anyone runs it in the environment, whether it is on a terminal or not.

With CEDF, you can intercept your application programs at the program's initiation, at each CICS command, and at the program termination. CEDF helps you to isolate and focus on problems in your application programs.



Coding applications to run in IBM CICS

In this chapter, we look at how to code your application programs to run in CICS.

3.1 Introduction to the EXEC CICS application programming interface

CICS is a mixed language application server that runs on IBM Z. As an application server, CICS hosts and provides services to application programs. In this type of environment, CICS manages resources on behalf of the application so that the application programmer can concentrate on coding the business logic without having to think about specific characteristics of CICS resources. For example, CICS opens and closes files so that the COBOL program does not need any File Descriptor entries in the FILE SECTION of the COBOL program.

How does your program request access to CICS resources? CICS provides an application programming interface (API) that gives the application program access to CICS services.

The API consists of several parts (see Figure 3-1).

- ▶ A command-level programming interface, commonly known as the EXEC CICS interface.
- ▶ An EXEC infrastructure consisting of the EXEC interface module DFHEIP and a set of EXEC processor modules. Also, a control block that is called Execute Interface Block (EIB) is used to store the status of the current CICS request being run by the application.
- ▶ An EXEC stub that is link-edited with the COBOL program.
- ▶ A command-level translator that interprets the command-level commands.

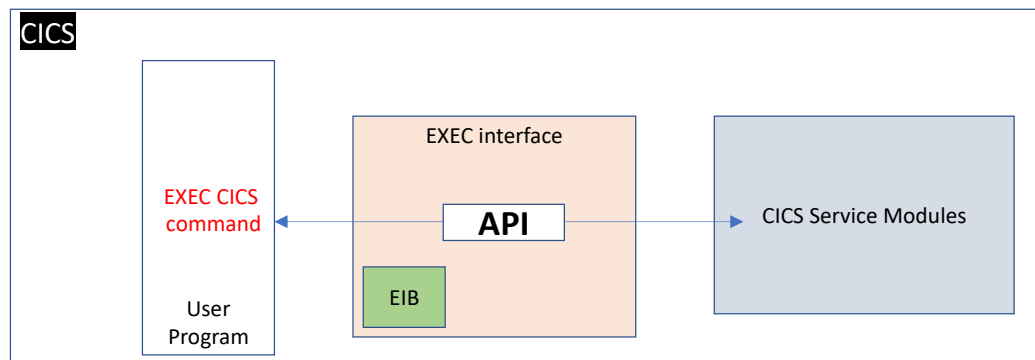


Figure 3-1 CICS application programming interface

The **EXEC CICS** command is the portion that goes in your COBOL program. For each request to access a CICS resource, a command must be coded in the application logic. The command starts with **EXEC CICS**, which is short for EXECUTE CICS. This **EXEC CICS** string tells the command-level translator that an API command was found. Then, there is the command itself, which determines the function being performed on the CICS resource.

There are over 340 commands that are available. The command is followed by one or more options and their arguments. In this case, we finish the command by using the string **END-EXEC**.

Here is the template for an **EXEC CICS** command:

```
EXEC CICS command option(arg) .... END-EXEC
```

IBM Knowledge Center has a summary of each command, as shown in Figure 3-2 on page 17:

- ▶ A schematic diagram illustrating the syntax and a description of the command.
- ▶ A list of options and their usage.

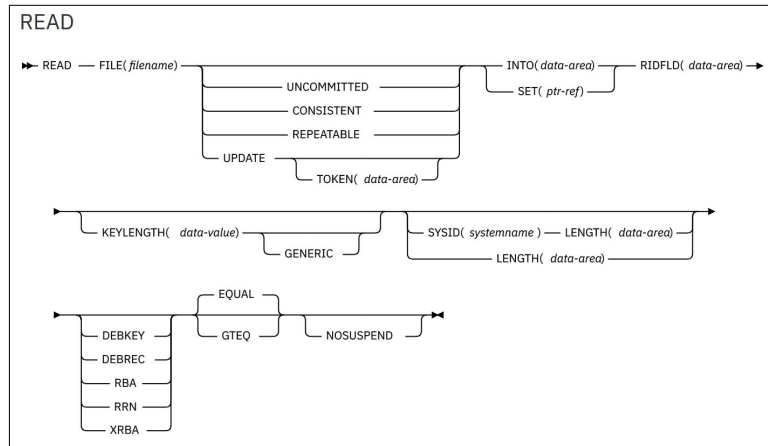


Figure 3-2 Schematic diagram for EXEC CICS READ

The schematic diagrams, or you might hear them referred to as *railroad diagrams*, are essential to the application programmer. They explain exactly what can be specified on an **EXEC CICS** command. Luckily, modern editors can supply context help to aid you when writing your code.

3.2 CICS API example

This section describes simple CICS API example by using the Payroll System application.

Our program PAYBUS must read a record from a file that is called “PAYROLL”, with a key that is stored in the variable *ws-key*. The data is returned to a Working Storage variable that is called *payroll-record*.

Looking at the schematic diagram for a **READ** shows that options **FILE** and **RIDFLD** (the record key) are mandatory. Options **INTO** and **SET** are alternatives, but you must use one of them. The other options are optional. So, based on this diagram, the code in the program looks like the following string:

```
EXEC CICS READ FILE('PAYROLL') RIDFLD(ws-key) INTO(payroll-record) END-EXEC
```

EXEC CICS tells the translator that a CICS command is starting. The command in this case is a **READ**, and you need three options that contain arguments:

- ▶ **FILE**: Use the file name “PAYROLL”.
- ▶ **RIDFLD**: Use the record key *ws-key*.
- ▶ **INTO**: The location in your program, where CICS places the record, which in this case is *payroll-record*.

The command is stopped by the string **END-EXEC**.

The argument for **FILE** is hardcoded with the string ‘PAYROLL’, which is why it is in quotation marks, but the other arguments *ws-key* and *payroll-record*, are variables in the **WORKING-STORAGE** SECTION of the COBOL program.

In the program that is called PAYBUS, the API example looks like Figure 3-3.

```
1 IDENTIFICATION DIVISION.  
2 PROGRAM-ID. PAYBUS.  
3 ENVIRONMENT DIVISION.  
4 DATA DIVISION.  
5 WORKING-STORAGE SECTION.  
6 01 MISC.  
7     02 ws-key.  
8         05 ws-department      pic x.  
9         05 ws-employee-no     pic x(5).  
10    01 payroll-record.  
11        02 pr-department      pic x.  
12        02 pr-employee-no     pic x(5).  
13        02 pr-name            pic x(20).  
14  
15 PROCEDURE DIVISION.  
16 *  
17     exec cics read file('PAYROLL')  
18         ridfld(ws-key)  
19         into(payload-record)  
20     end-exec.  
21 *  
22     if eibresp not = dfhresp(NORMAL)  
23         move 'No such Record' to ws-msg.
```

Figure 3-3 Working storage section of PAYBUS

When this command runs, the API passes the **READ** arguments through the EXEC interface through to CICS. CICS service modules interact with VSAM to access the file and retrieve the record, and then they send the contents back to the program's Working Storage field (Figure 3-4).

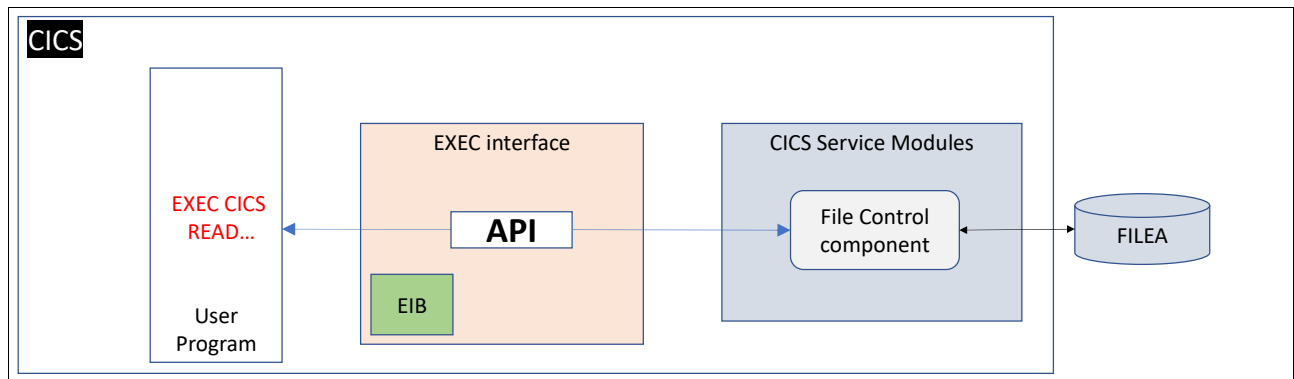


Figure 3-4 API flow

3.3 COBOL translator

These **EXEC CICS** commands are not reserved words in COBOL (or any other language), so how can the compiler understand it?

A command-level translator runs with the COBOL compiler. This translator is either integrated with the COBOL compiler or is a separate pre-compile step. The translator converts the **EXEC CICS** API command into COBOL statements (Figure 3-5 on page 19).

000373		*exec cics read file('PAYROLL')	
000374		* ridfld(ws-key)	
000375		* into(payroll-record)	
000376		* nohandle	
000377		*end-exec	

000378	1	Move 'PAYROLL' to dfhc0080	71
000379	1	Move length of payroll-record to dfhb0020	IMP 22 58
000380	1	Call 'DFHEI1' using by content x'0602f0002700008000f0f0f2f	EXT
000381	1	- '2f2404040' by reference dfhc0080 by reference payroll-record	71 22
000382	1	by reference dfhb0020 by reference ws-key end-call	58 149
000383			

Figure 3-5 Translating the EXEC CICS command

For this **EXEC CICS READ** example:

- ▶ You can see that the **EXEC CICS** command is commented out and replaced by COBOL statements.
- ▶ These COBOL statements call the **EXEC** stub that is named DFHEI1, and passes the COBOL option arguments, 'PAYROLL', ws-key, and payroll-record KEY-I and AREA-0.
- ▶ The translator also copies the EIB copybooks DFHEIBLK and DFHCOMMAREA into the LINKAGE SECTION.
- ▶ The EIB copies the copybooks DFHEIBLK and DFHCOMMAREA onto the PROCEDURE DIVISION header after the "using" phrase.

Based on this knowledge, you can revisit how the API passes control to CICS (Figure 3-6).

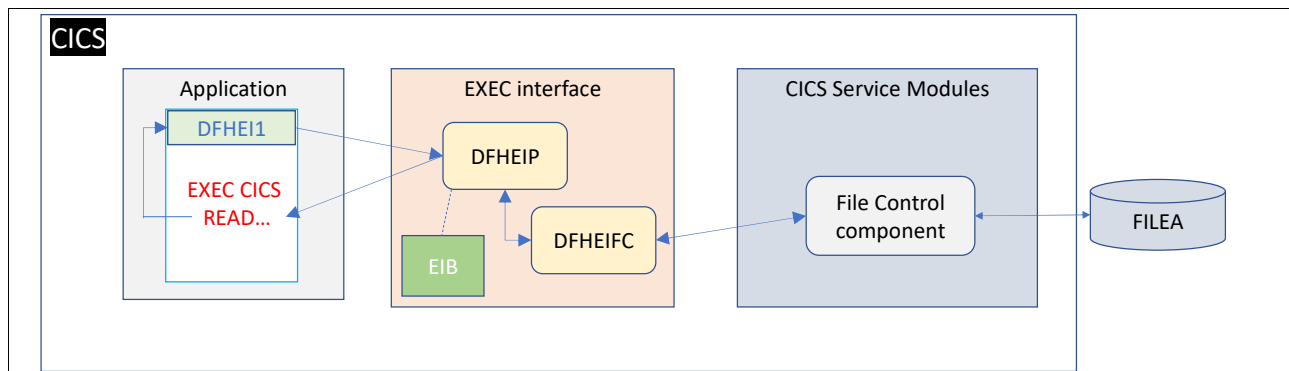


Figure 3-6 Enhanced API flow

- ▶ When the COBOL program runs the **EXEC CICS** command, it links to the **EXEC** stub DFHEI1.
- ▶ DFHEI1 finds the address of the **EXEC** interface module DFHEIP.
- ▶ DFHEIP branches to the relevant processor module for the component that is responsible for handling the request, which in this case is DFHEIFC.
- ▶ The processor module validates the argument list and branches to its CICS service modules.
- ▶ If the arguments were correct, then the record is retrieved from the file and control is passed back to DFHEIP through the processor module.
- ▶ DFHEIP updates the DFHEIB control block with the status of the command.
- ▶ It then returns control back to the application program.

3.4 Response codes

So, how do you know whether the command was successful? When you return to your program, the response from CICS is placed in the EIB fields `EIBRESP` and `EIBRESP2`. The response code in each field consists of a two-digit decimal condition number. These EIB response fields are placed into your program's `LINKAGE SECTION` by the CICS translator at compile time.

If the command was successful, then the `EIBRESP` contains a zero, which means `NORMAL`. However, if the `EIBRESP` contains a nonzero value, then a non-`NORMAL` condition occurred.

Going back to our example, if the **READ** on the file 'PAYROLL' is for a record that does not exist, then the **EIBRESP** would contain 13.

A list of the conditions for each command can be found in the command summary section of IBM Knowledge Center. Looking in `Conditions` under **READ** shows condition number 13, which means `NOT-FOUND`. There might be several reasons for a condition. If so, then `EIBRESP2` is used to further qualify the condition.

So, how do you check for the response code in your program? There are two ways to test the response code of a command:

- Code the **RESP** or **RESP2** option on the command with a Working Storage variable as the argument. CICS places the `EIBRESP` in the working storage variable as the call returns.

```
EXEC CICS READ FILE('PAYROLL') RIDFLD(ws-key) INTO(payload-record) RESP(WC)
RESP2(RC2) END-EXEC
```

The working storage variable can then be tested by using **IF** or **EVALUATE COBOL** statements that test **RESP**.

- Alternatively, code the option **nohandle**. CICS updates the EIB, but does not handle the condition.

```
EXEC CICS READ FILE('PAYROLL') RIDFLD(ws-key) INTO(payload-record) nohandle
END-EXEC
```

The application tests the EIB fields directly.

If the condition is not handled (**nohandle** and **RESP** are not coded), then CICS issues an `abend` condition and stops the program.

In summary, CICS is a powerful mixed-language application server that provides access its resources by using a wealth of interface commands. Over the years, new APIs were added, such as `web`, `asynchronous`, and **SIGNAL EVENT** commands. We cover some of these APIs later in this paper.



Programming an IBM CICS application in COBOL

This chapter explores programming a CICS application in COBOL. It uses the Payroll application as an example and describes the coding that is required for both the presentation logic and business logic portions of the application.

4.1 Presentation logic

The only way to truly understand the concepts that were described in the earlier chapter is to look at the code that makes up the Payroll application. If you recall, there are two programs (PAYPGM and PAYBUS) that make up the presentation and business logic of the application (Figure 4-1).

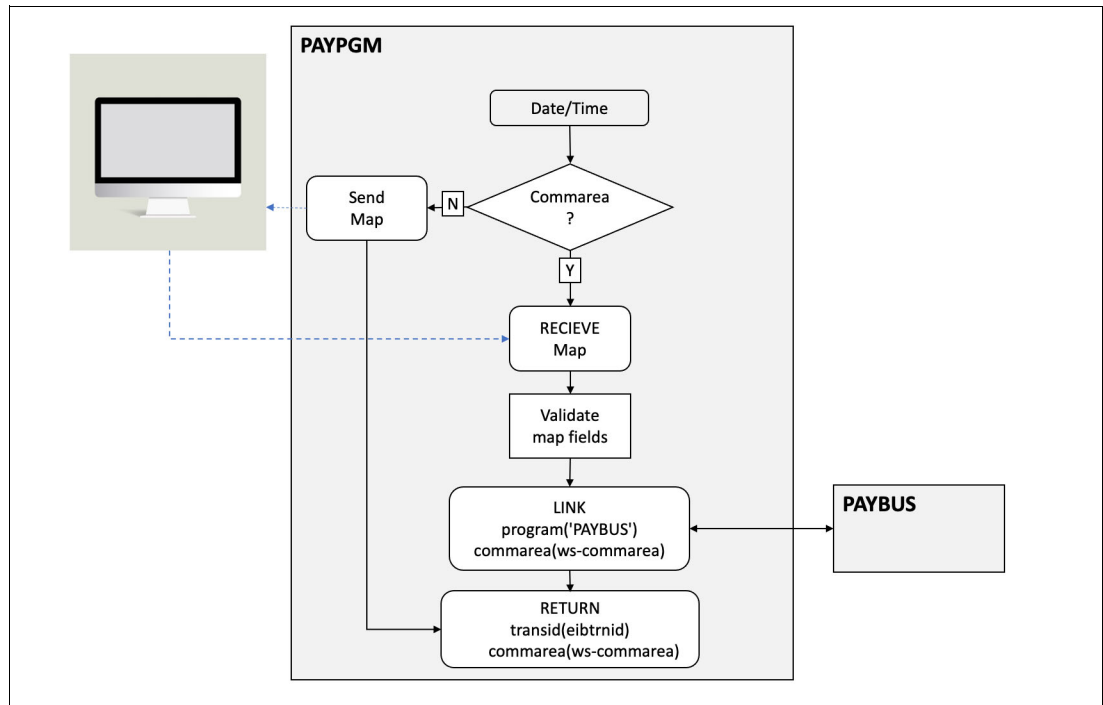


Figure 4-1 PAYPGM program flow

PAYPGM is the presentation logic program that collects information from the screen, which includes the user's wanted function, and passes it to the business logic program for processing by using a **CICS LINK** command.

4.1.1 Communication area

The presentation logic program and business logic program communicate through a communication area (COMMAREA) (Figure 4-2 on page 23).

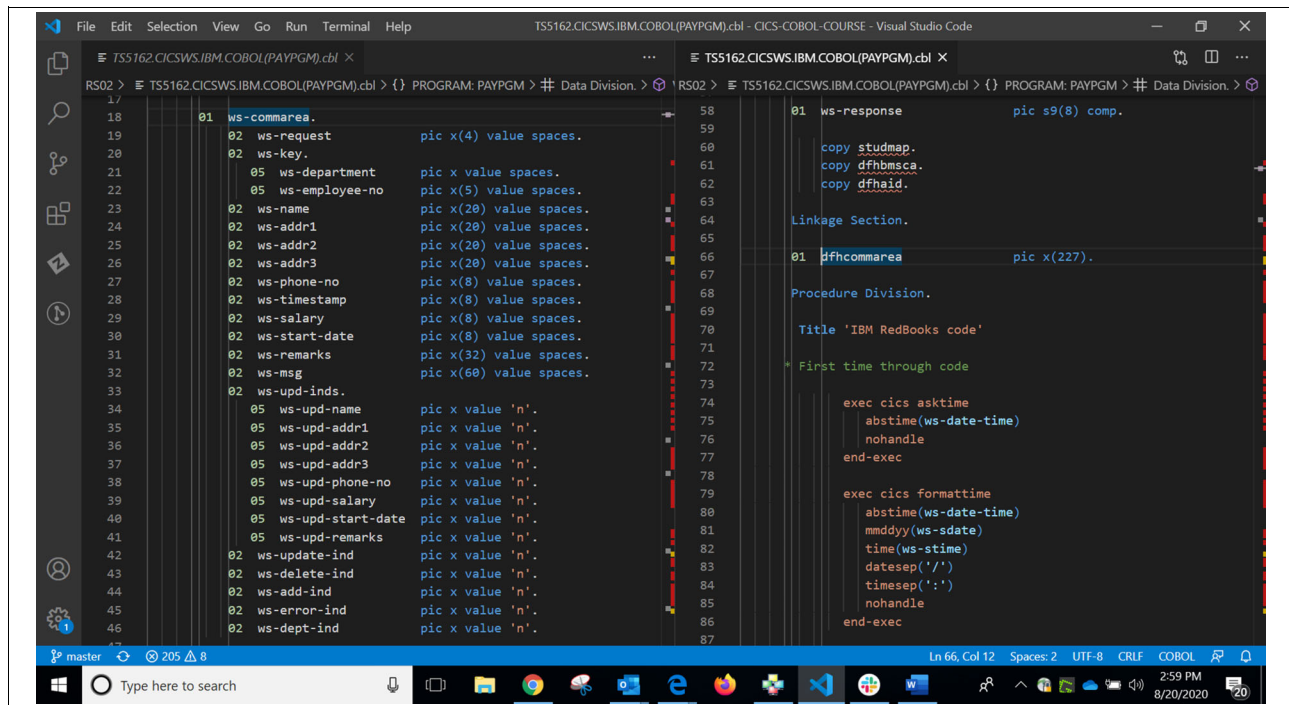


Figure 4-2 COMMAREA definition of PAYPGM

The COMMAREA appears in the Working Storage section of PAYPGM with its field names specified, and in the Linkage section of PAYPGM as a place holder for future invocations. In PAYBUS, the fields appear and are used through the Linkage section.

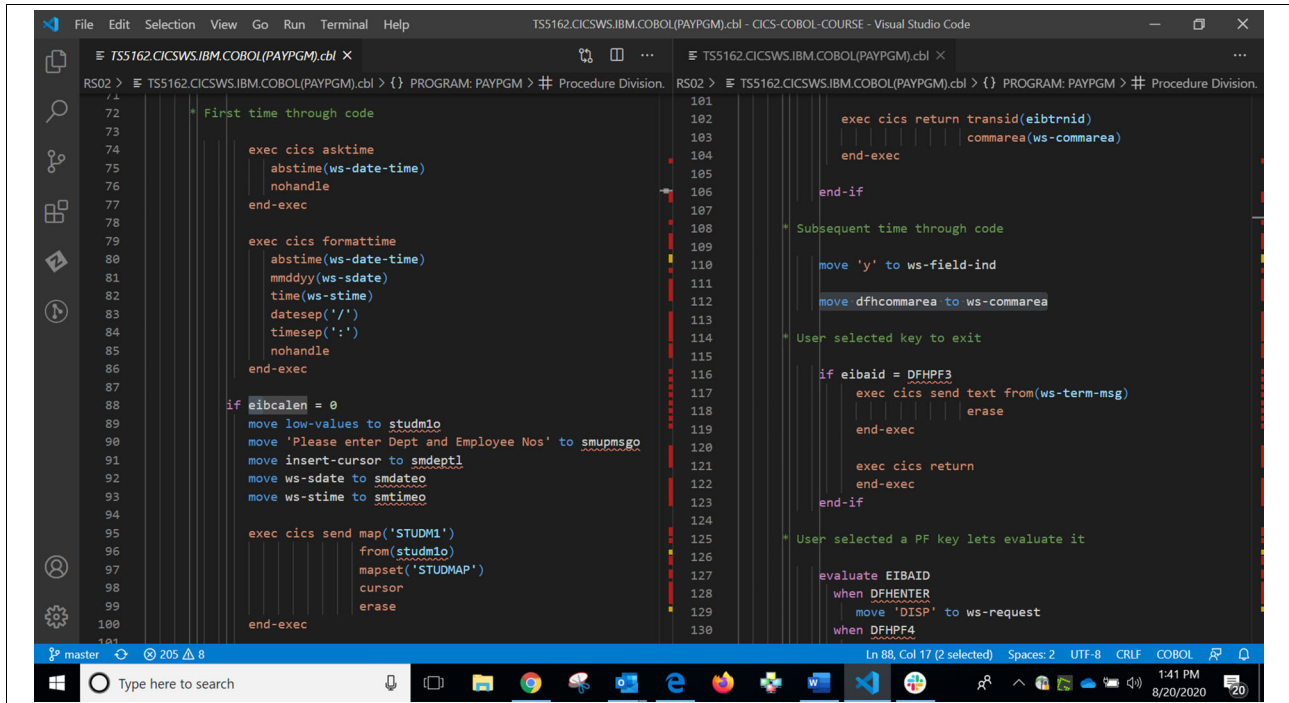
The reason for this approach is that PAYPGM creates the COMMAREA out of working storage the first time with the initial values. On subsequent requests, PAYPGM moves the data from Linkage to Working storage to process any of the field data. This standard is common to many CICS programs.

Looking at the fields in the COMMAREA, the first 10 bytes must be collected and passed to the business logic program PAYBUS, more specifically, the 4-byte request, followed by a 1-byte department number and a 5-byte employee number. These bytes make up the data that must be collected from the user.

The remainder of the COMMAREA is either data from the record being retrieved and displayed on the screen or a set of 1-byte indicators to provide the state information. They are represented as flags, and can either be an 'n' or a 'y'. Initially, they are all set to 'n', and as we progress through the program logic, we see when they change.

4.1.2 First time through processing

Now, look at the first time through processing (Figure 4-3). This processing is different depending on the programmer's preference or whether COMMAREAs are not used, for example, if you use channels and containers, but otherwise it is the most popular method of determining first entry into a program.



```
TS5162.CICSWS.IBM.COBL(PAYPGM).cbl
72 * First time through code
73
74 exec cics asktime
75     abstime(ws-date-time)
76     nohandle
77 end-exec
78
79 exec cics formattime
80     abstime(ws-date-time)
81     mmdyy(ws-sdate)
82     time(ws-stime)
83     datesep('/')
84     timesep(':')
85     nohandle
86 end-exec
87
88 if eibcalen = 0
89     move low-values to studm1o
90     move 'Please enter Dept and Employee Nos' to smupmsgo
91     move insert-cursor to smdeptl
92     move ws-sdate to smdateo
93     move ws-stime to smtimeo
94
95     exec cics send map('STUDM1')
96         from(studm1o)
97         mapset('STUDMAP')
98         cursor
99         erase
100 end-exec
101
102 exec cics return transid(eibtrnid)
103     commarea(ws-commarea)
104 end-exec
105
106 end-if
107
108 * Subsequent time through code
109
110 move 'y' to ws-field-ind
111
112 move dfhcommarea to ws-commarea
113
114 * User selected key to exit
115
116 if eibaid = DFHPF3
117     exec cics send text from(ws-term-msg)
118         erase
119 end-exec
120
121 exec cics return
122 end-exec
123
124 end-if
125
126 * User selected a PF key lets evaluate it
127
128 evaluate EIBRID
129 when DFHENTER
130     move 'DISP' to ws-request
131 when DFHPF4
```

Figure 4-3 First time through code

The first bit of code gets the time and date by using CICS commands, which are displayed on the screen every time the user presses a key, so they appear at the top of the procedure division.

The Execute Interface Block (EIB) is automatically inserted into CICS programs. It has many useful fields on entry. One is EIBCALEN, which contains the COMMAREA length. If the COMMAREA length is zero, there is no COMMAREA, so this is the first time through when using COMMAREA. Otherwise, if the user ends this task with a COMMAREA, which we see, then on subsequent calls it never will be zero.

The first time through this program, the BMS map is prepped with the time, date, and a message to the user to enter data, then the MAP is sent.

It is the **return** command with the **transid** and **COMMAREA** options that is the most interesting. You can see that the return is nominating the next transaction to run by using EIBTRNID, which is the current transaction ID. This situation means that if the user presses Enter or a function key, the same PAYR transaction runs. You also can see that how the COMMAREA to be passed to the next invocation of this program is coded.

4.1.3 Function evaluation

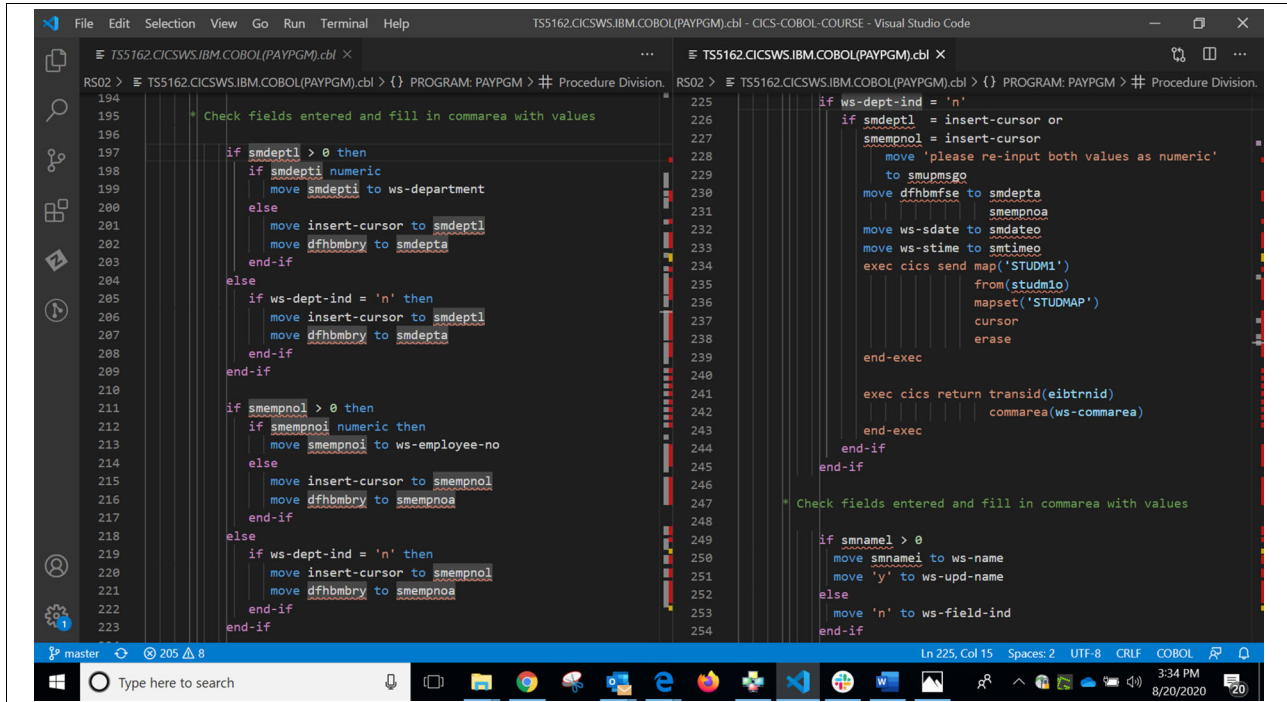
Now, we explore some standard programming that is found in a presentation logic program. We start with an evaluation of the function that is requested and move on to receiving the data from the screen (see Figure 4-4).

```
TS5162.CICSWS.IBM.COBL(PAYPGM).cbl
124
125
126
127 evaluate EIBAID
128   when DFHENTER
129     move 'DISP' to ws-request
130   when DFHPPF4
131     move 'UPDT' to ws-request
132   when DFHPPF5
133     move 'ADDs' to ws-request
134   when DFHPPF6
135     move 'DLET' to ws-request
136   when DFHPPF7
137     move 'BACK' to ws-request
138   when DFHPPF8
139     move 'FWRD' to ws-request
140   when other
141     perform BAD-FUNCTION
142   end-evaluate
143
144   perform DO-WORK
145
146   exec cics return transid(eibtrnid)
147     commarea(ws-commarea)
148   end-exec
149
150   exit.
151
152   DO-WORK.
153

TS5162.CICSWS.IBM.COBL(PAYPGM).cbl
154
155 DISP=display,UPDT=update,ADDs=Insert,DLET=Deletes,
156 BACK=Previous,FWRD=Forward
157
158 exec cics receive map('STUDM1')
159   mapset('STUDMAP')
160   into(studm1)
161   nohandle
162 end-exec
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2590
2591
2592
2593
2594
2595
2596
2597
2598
2599
2600
2601
2602
2603
2604
2605
2606
2607
2608
2609
2610
2611
2612
2613
2614
2615
2616
2617
2618
2619
2620
2621
2622
2623
2624
2625
2626
2627
2628
2629
2630
2631
2632
2633
2634
2635
2636
2637
2638
2639
2640
2641
2642
2643
2644
2645
2646
2647
2648
264
```

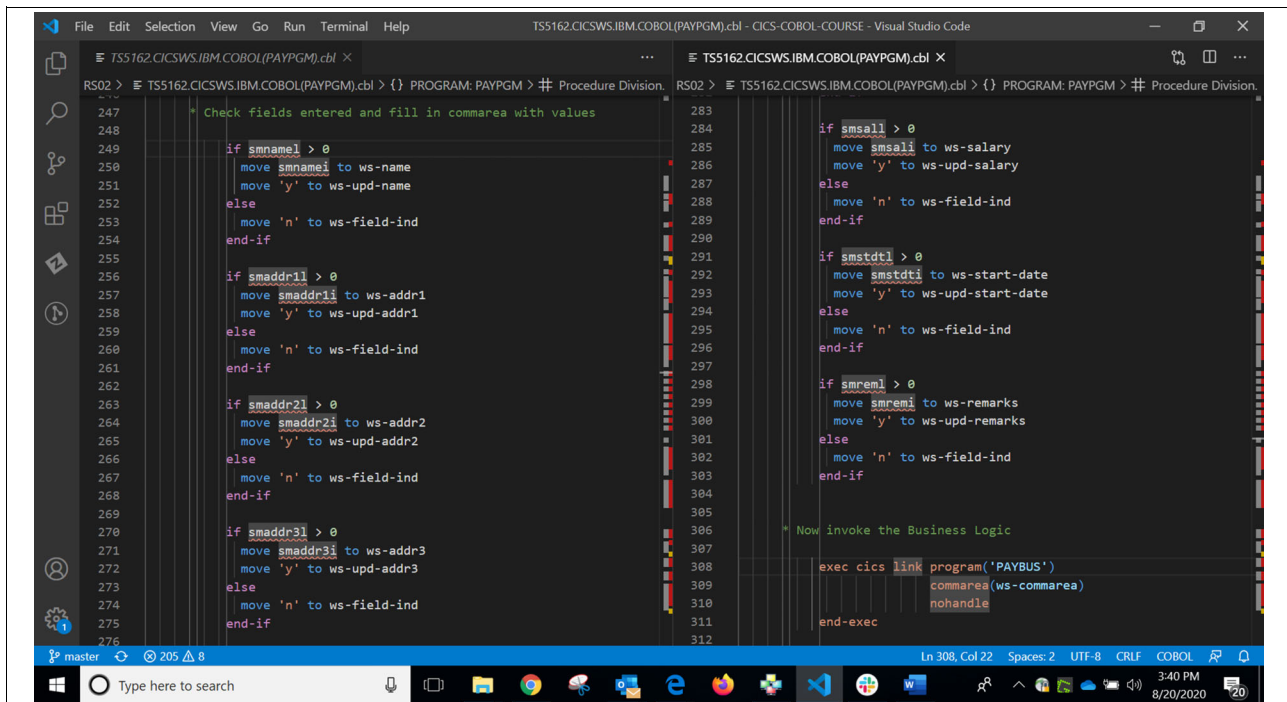
4.1.4 Field validation and link to PAYBUS

The next section of code is about field validation (see Figure 4-5 and Figure 4-6). Again, this program is structured for simplistic field validation, and most real-life examples have various complex routines to validate the data that is entered by the user.



```
TS5162.CICSWS.IBM.COBO(PAYPGM).cbl - CICS-COBOL-COURSE - Visual Studio Code
TS5162.CICSWS.IBM.COBO(PAYPGM).cbl x
TS5162.CICSWS.IBM.COBO(PAYPGM).cbl x
RS02 > TS5162.CICSWS.IBM.COBO(PAYPGM).cbl > {} PROGRAM: PAYPGM > # Procedure Division.
194
195 * Check fields entered and fill in commarea with values
196
197 if smdept1 > 0 then
198   if smdept1 numeric
199     move smdept1 to ws-department
200   else
201     move insert-cursor to smdept1
202     move dfhbmbr1 to smdepta
203   end-if
204 else
205   if ws-dept-ind = 'n' then
206     move insert-cursor to smdept1
207     move dfhbmbr1 to smdepta
208   end-if
209 end-if
210
211 if smempno1 > 0 then
212   if smempno1 numeric then
213     move smempno1 to ws-employee-no
214   else
215     move insert-cursor to smempno1
216     move dfhbmbr1 to smempnoa
217   end-if
218 else
219   if ws-dept-ind = 'n' then
220     move insert-cursor to smempno1
221     move dfhbmbr1 to smempnoa
222   end-if
223 end-if
224
225 if ws-dept-ind = 'n'
226   if smdept1 = insert-cursor or
227     smempno1 = insert-cursor
228     move 'please re-input both values as numeric'
229     to smupmsgo
230     move dfhbmfr1 to smdepta
231     move dfhbmfr1 to smempnoa
232     move ws-sdate to smdateo
233     move ws-stime to smtimeo
234     exec cics send map('STUDM1')
235     from(studm1o)
236     mapset('STUDMAP')
237     cursor
238     erase
239   end-exec
240
241   exec cics return transid(eibtrnid)
242   commarea(ws-commarea)
243 end-exec
244 end-if
245
246 * Check fields entered and fill in commarea with values
247
248 if smname1 > 0
249   move smname1 to ws-name
250   move 'y' to ws-upd-name
251 else
252   move 'n' to ws-field-ind
253 end-if
254
```

Figure 4-5 Field validation



```
TS5162.CICSWS.IBM.COBO(PAYPGM).cbl - CICS-COBOL-COURSE - Visual Studio Code
TS5162.CICSWS.IBM.COBO(PAYPGM).cbl x
TS5162.CICSWS.IBM.COBO(PAYPGM).cbl x
RS02 > TS5162.CICSWS.IBM.COBO(PAYPGM).cbl > {} PROGRAM: PAYPGM > # Procedure Division.
247
248 * Check fields entered and fill in commarea with values
249
250 if smname1 > 0
251   move smname1 to ws-name
252   move 'y' to ws-upd-name
253 else
254   move 'n' to ws-field-ind
255 end-if
256
257 if smaddr11 > 0
258   move smaddr11 to ws-addr1
259   move 'y' to ws-upd-addr1
260 else
261   move 'n' to ws-field-ind
262 end-if
263
264 if smaddr21 > 0
265   move smaddr21 to ws-addr2
266   move 'y' to ws-upd-addr2
267 else
268   move 'n' to ws-field-ind
269 end-if
270
271 if smaddr31 > 0
272   move smaddr31 to ws-addr3
273   move 'y' to ws-upd-addr3
274 else
275   move 'n' to ws-field-ind
276 end-if
277
278
279 if smcall > 0
280   move smcall to ws-salary
281   move 'y' to ws-upd-salary
282 else
283   move 'n' to ws-field-ind
284 end-if
285
286 if smstdt1 > 0
287   move smstdt1 to ws-start-date
288   move 'y' to ws-upd-start-date
289 else
290   move 'n' to ws-field-ind
291 end-if
292
293 if smrem1 > 0
294   move smrem1 to ws-remarks
295   move 'y' to ws-upd-remarks
296 else
297   move 'n' to ws-field-ind
298 end-if
299
300 * Now invoke the Business Logic
301
302 exec cics link program('PAYBUS')
303 commarea(ws-commarea)
304 nohandle
305 end-exec
306
307
308
309
310
311
312
```

Figure 4-6 Field validation and link to PAYBUS

The long list of **IF** statements are checking to see whether a field is complete. The program accomplishes this task by checking the length that is returned from the **RECEIVE** command that was issued earlier. If the length is greater than zero, there is data to update in the **COMMAREA**.

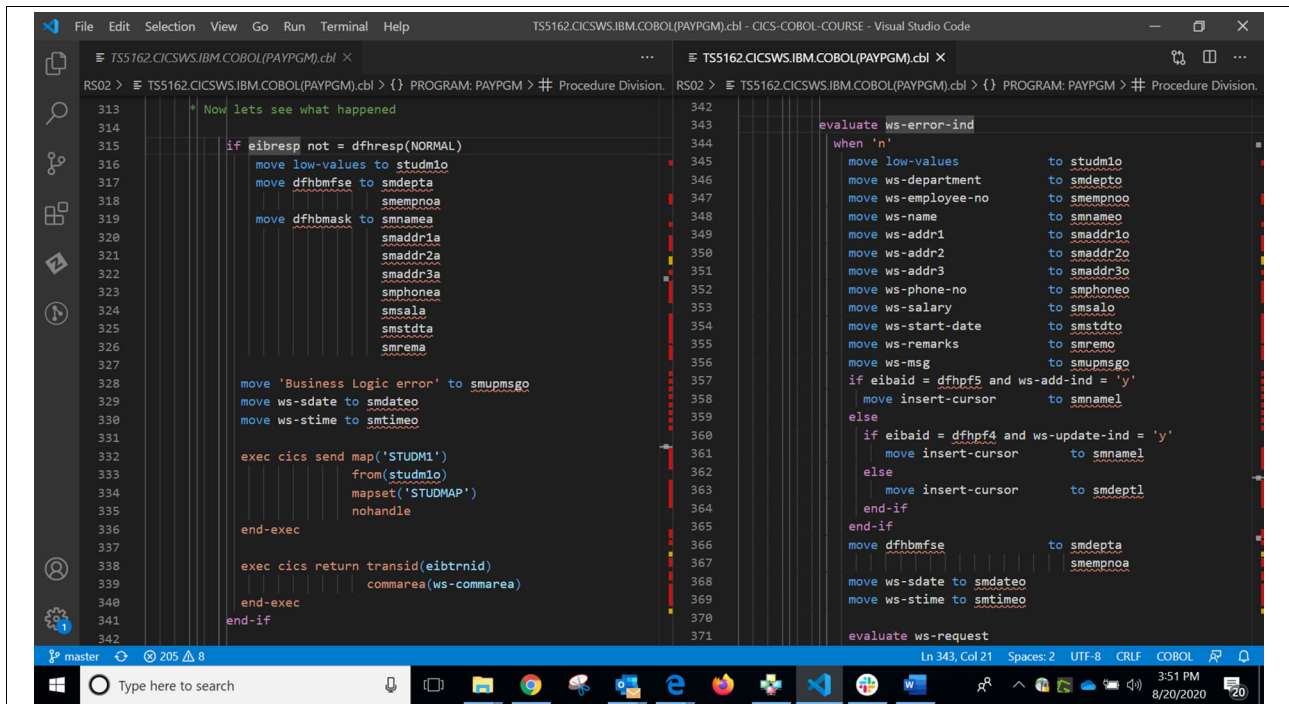
The program also updates the flag that is associated with each new field from **n** to **y**, which notifies the business logic program that the data is new.

The last piece of this code is the call to the business logic (**PAYBUS**) to process the request. It is a **CICS LINK** command, where the data being passed is the **COMMAREA**.

Rather than explore the business logic now, let us see what the presentation logic program does after the business logic returns to it. After the **LINK** command completes, we still need to check whether that command ran properly.

4.1.5 Checking the return code from the link

Figure 4-7 starts by checking the return code from the **LINK** command to see whether it ran successfully. If not, you see some common logic that returns the error to the user screen.



```
TS5162.CICSWS.IBM.COBL(PAYPGM).cbl
313  * Now lets see what happened
314
315  if eibresp not = dfhresp(NORMAL)
316      move low-values to studm1o
317      move dfhbmfs to smdepta
318      move dfhbmfs to smempnoa
319      move dfhbmfs to smnamea
320      move dfhbmfs to smaddr1a
321      move dfhbmfs to smaddr2a
322      move dfhbmfs to smaddr3a
323      move dfhbmfs to smphonea
324      move dfhbmfs to smsala
325      move dfhbmfs to smstdta
326      move dfhbmfs to smrema
327
328      move 'Business Logic error' to smupmsg
329      move ws-sdate to smdateo
330      move ws-stime to smtimeo
331
332      exec cics send map('STUDM1')
333          from(studm1o)
334          mapset('STUDMAP')
335          nohandle
336      end-exec
337
338      exec cics return transid(eibtrnid)
339          commarea(ws-commarea)
340      end-exec
341  end-if
342
TS5162.CICSWS.IBM.COBL(PAYPGM).cbl
342  evaluate ws-error-ind
343
344  when 'n'
345      move low-values to studm1o
346      move ws-department to smdepta
347      move ws-employee-no to smempnoa
348      move ws-name to smnameo
349      move ws-addr1 to smaddr1o
350      move ws-addr2 to smaddr2o
351      move ws-addr3 to smaddr3o
352      move ws-phone-no to smphoneo
353      move ws-salary to smsalo
354      move ws-start-date to smstdto
355      move ws-remarks to smremo
356      move ws-msg to smupmsg
357      if eibaid = dfhpf5 and ws-add-ind = 'y'
358          move insert-cursor to smname1
359      else
360          if eibaid = dfhpf4 and ws-update-ind = 'y'
361              move insert-cursor to smname1
362          else
363              move insert-cursor to smdept1
364          end-if
365      end-if
366      move dfhbmfs to smdepta
367      move dfhbmfs to smempnoa
368      move ws-sdate to smdateo
369      move ws-stime to smtimeo
370
371  evaluate ws-request
```

Figure 4-7 Link return code processing

If the command worked, we still must check whether the business logic program returned an error by checking an error indicator flag. If so, we return the business-specific error message that was provided by the business logic program.

4.1.6 Remaining presentation logic processing

Assuming everything worked to this point, we have some presentation house keeping to do before returning the results of the request back to the user (see Figure 4-8).

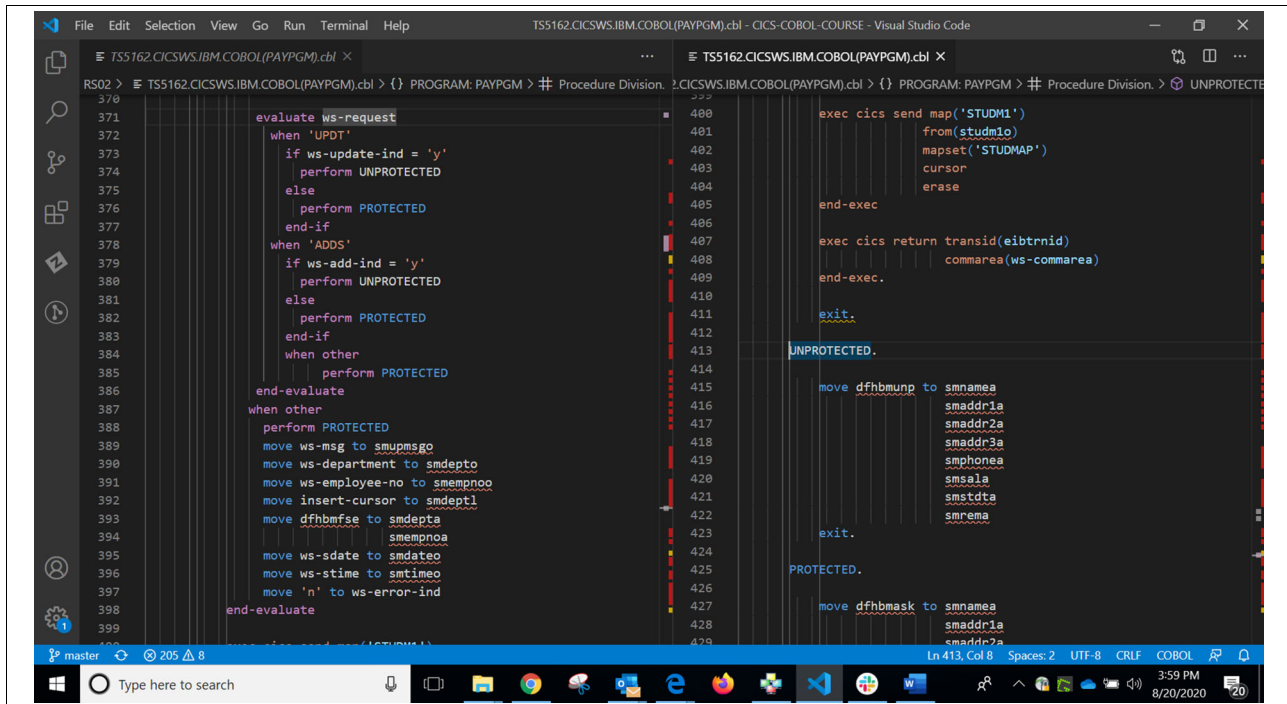
The image is a screenshot of a Visual Studio Code editor window. The title bar at the top reads 'TS5162.CICSWS.IBM.COBOL(PAYPGM).cbl - CICS-COBOL-COURSE - Visual Studio Code'. The editor has two tabs open, both named 'TS5162.CICSWS.IBM.COBOL(PAYPGM).cbl'. The left tab is active and shows COBOL code from line 370 to 399. The code includes an 'evaluate ws-request' block with 'when' clauses for 'UPDT' and 'ADDS', each containing 'if' statements and 'perform' calls for 'UNPROTECTED' and 'PROTECTED'. It also includes 'move' statements for various data fields like 'ws-msg', 'ws-department', 'ws-employee-no', 'insert-cursor', 'dfhbmfs', 'smdateo', 'smtimeo', and 'ws-error-ind'. The right tab shows code from line 400 to 429, including 'exec cics send map' and 'exec cics return transid' statements, followed by 'UNPROTECTED.' and 'PROTECTED.' sections with 'move' statements for 'dfhbmnp' and 'dfhbmnp' to 'smnamea'. The status bar at the bottom indicates 'Ln 413, Col 8', 'Spaces: 2', 'UTF-8', 'CRLF', 'COBOL', and the system clock shows '3:59 PM 8/20/2020'.

Figure 4-8 Remaining processing

Most of the fields on the screen are protected until the user must supply input. In our application, this action is required only if the user updates an employee payroll record or adds an employee. In those specific cases, we must modify the screen when responding to unprotect the fields.

The code here calls a routine to do protection and unprotection of input fields as required by the processing, and then it sends the results to the user.

4.2 Business logic

The business logic program is the core piece of a CICS application. It performs the functions that are required by the application regardless of the user interface, and in most cases, the data store (Figure 4-9).

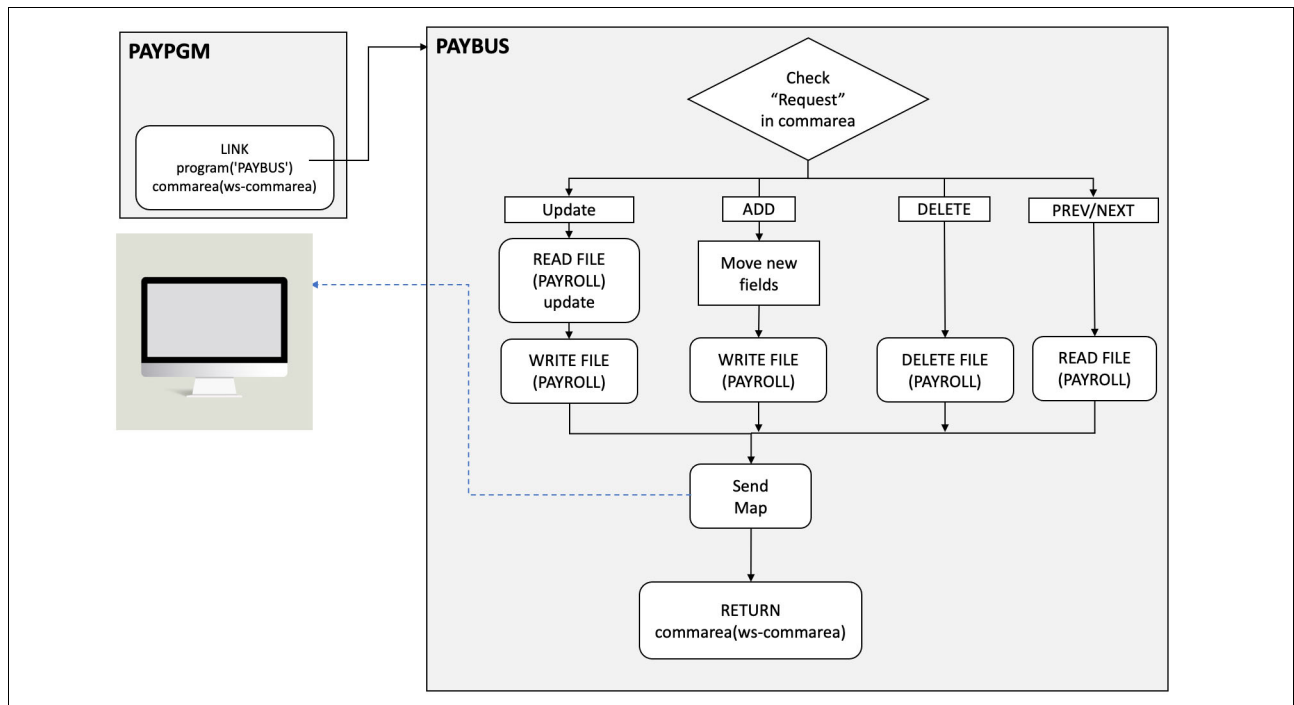
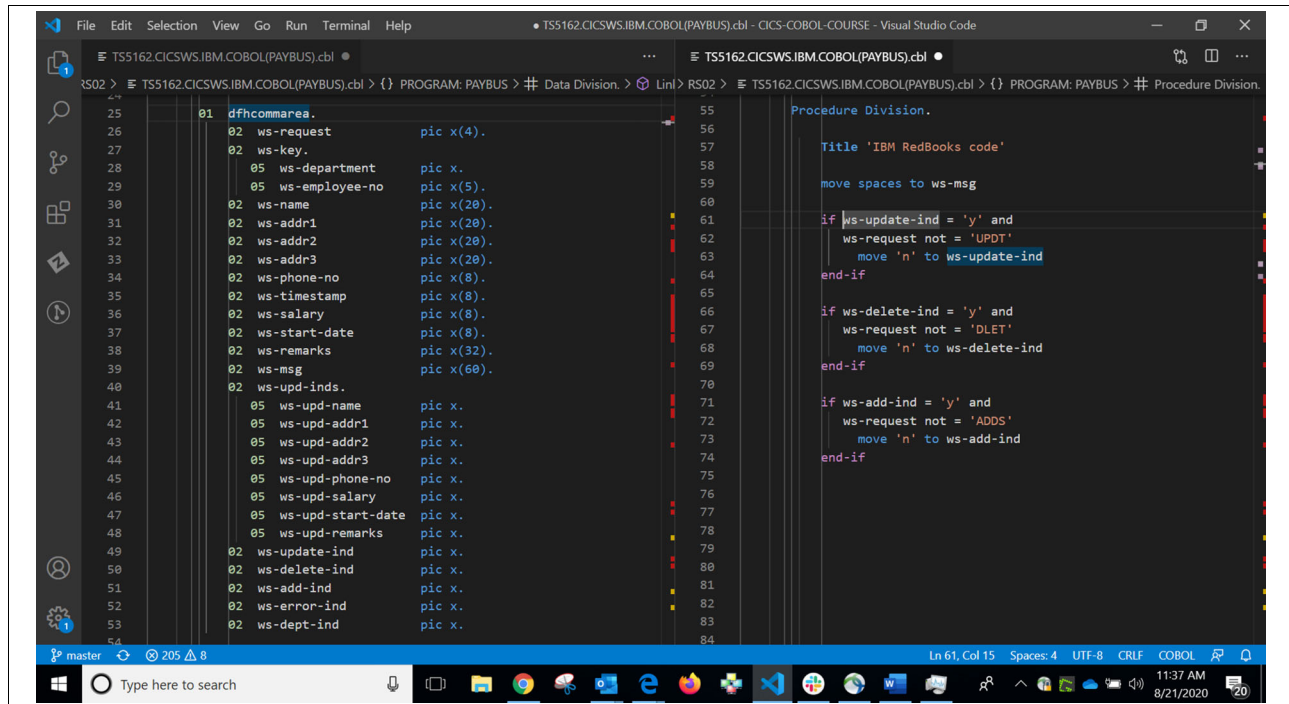


Figure 4-9 PAYBUS program flow

The business logic normally consists of a driver program that coordinates the running of the business logic by invoking subsequent programs for different functions. In our case, this program does all these functions. This program contains the entire business logic and the data store.

4.2.1 COMMAREA and special processing

PAYBUS receives data in its Linkage section (see Figure 4-10). This section is passed by its caller as a COMMAREA, which in this case is by an **EXEC CICS LINK** command.



```
TS5162.CICSWS.IBM.COBO(PAYBUS).cbl > {} PROGRAM: PAYBUS > # Data Division. > Lin1> RS02 > TS5162.CICSWS.IBM.COBO(PAYBUS).cbl > {} PROGRAM: PAYBUS > # Procedure Division.

25 01 dfhcommarea.
26 02 ws-request      pic x(4).
27 02 ws-key.
28 05 ws-department  pic x.
29 05 ws-employee-no pic x(5).
30 02 ws-name        pic x(20).
31 02 ws-addr1       pic x(20).
32 02 ws-addr2       pic x(20).
33 02 ws-addr3       pic x(20).
34 02 ws-phone-no    pic x(8).
35 02 ws-timestamp   pic x(8).
36 02 ws-salary      pic x(8).
37 02 ws-start-date  pic x(8).
38 02 ws-remarks     pic x(32).
39 02 ws-msg         pic x(60).
40 02 ws-upd-inds.
41 05 ws-upd-name    pic x.
42 05 ws-upd-addr1  pic x.
43 05 ws-upd-addr2  pic x.
44 05 ws-upd-addr3  pic x.
45 05 ws-upd-phone-no pic x.
46 05 ws-upd-salary pic x.
47 05 ws-upd-start-date pic x.
48 05 ws-upd-remarks pic x.
49 02 ws-update-ind  pic x.
50 02 ws-delete-ind  pic x.
51 02 ws-add-ind     pic x.
52 02 ws-error-ind  pic x.
53 02 ws-dept-ind    pic x.
54

55 Procedure Division.
56
57 Title 'IBM RedBooks code'
58
59 move spaces to ws-msg
60
61 if ws-update-ind = 'y' and
62   ws-request not = 'UPDT'
63   move 'n' to ws-update-ind
64 end-if
65
66 if ws-delete-ind = 'y' and
67   ws-request not = 'DELET'
68   move 'n' to ws-delete-ind
69 end-if
70
71 if ws-add-ind = 'y' and
72   ws-request not = 'ADDS'
73   move 'n' to ws-add-ind
74 end-if
75
76
77
78
79
80
81
82
83
84
```

Figure 4-10 COMMAREA and special processing

The first section of the code in this program is special processing for the 3270 presentation logic program. The program begins by resetting flags if a user interrupts a function that is normally done in two steps by using a 3270 screen. These functions are **ADD**, **UPDATE**, and **DELETE**. For example, if a user starts an update by pressing PF4 on the 3270 screen, but then does not confirm the update by pressing PF4 again, the flags in the COMMAREA must be reset to indicate that we are no longer in middle of an update.

4.2.2 Request analysis

After any special processing completes, the business logic program analyzes the request and gives control to the routine that processes it (Figure 4-11 on page 31). Again, if this had been a more complex application, there might be calls to other programs to process these requests.

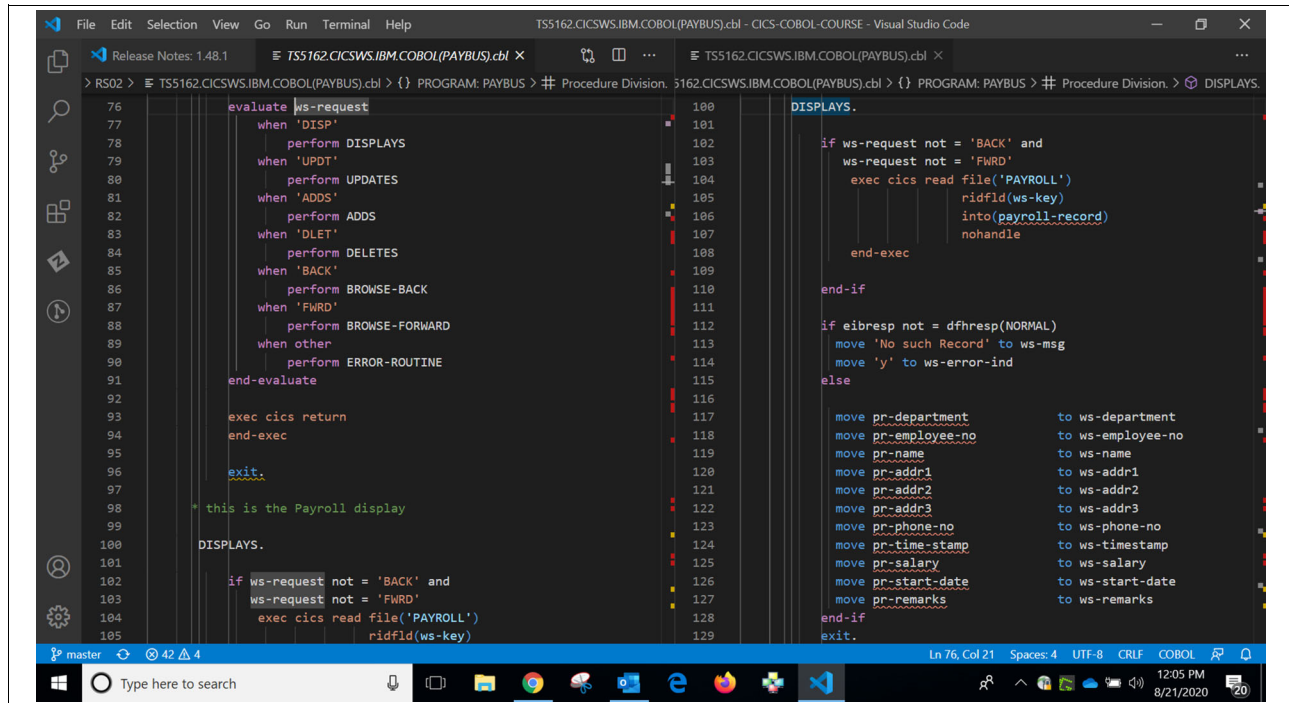


Figure 4-11 Request analysis

The first step is an evaluation of the request, which is supplied in a COMMAREA field that is passed. Based on the request type, the function is performed.

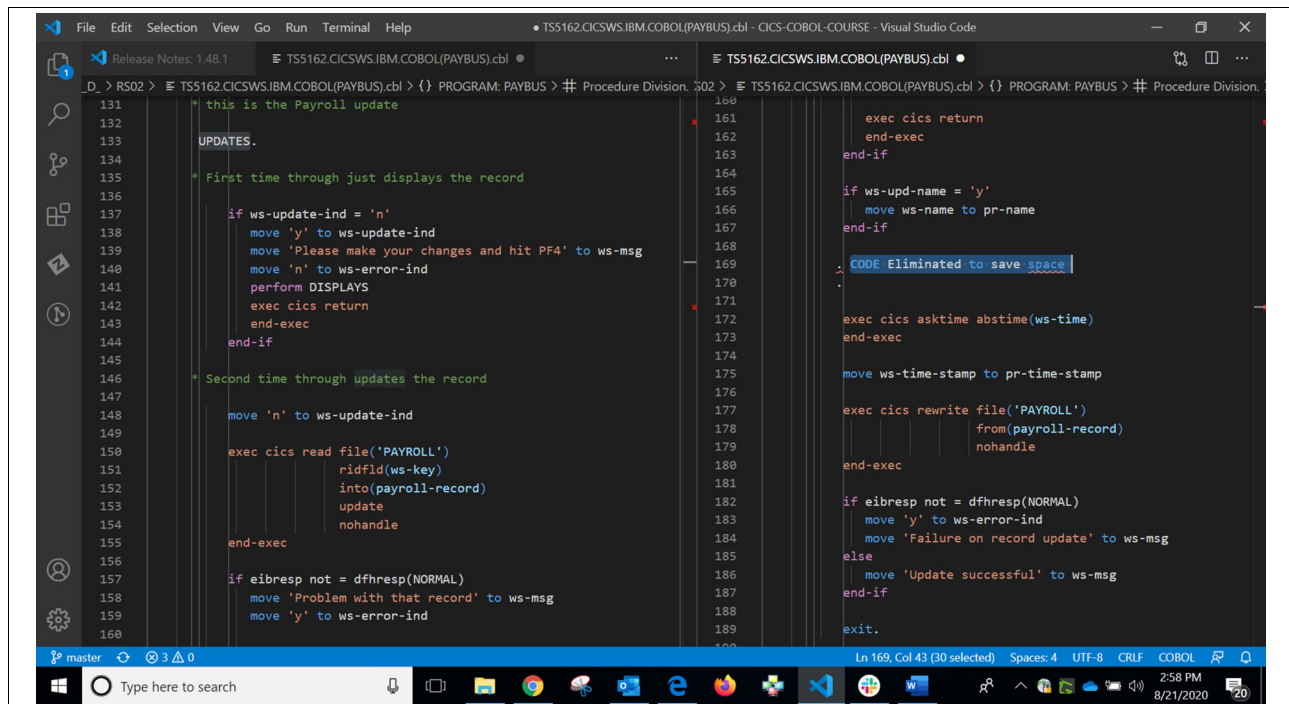
The DISPLAY routine is the first routine that is highlighted here. The DISPLAY routine is normally the routine that reads a record and returns the contents to the caller. However, the **IF** statement at the top implies that it is also called from the FORWARD and BACKWARD routines without first reading a record.

Therefore, this DISPLAY routine is a generic one that displays the record and is used by other routines to move data that is read from the VSAM file into the COMMAREA to return to the caller.

An error from the read command assumes that the record was not found. There are other errors to contend with, such as “file not open”. We simplified the code here for brevity, but it is a best practice to consider all possible errors and send a proper response.

4.2.3 Updating a record

Here is an example of a more complex routine (see Figure 4-12).



The screenshot shows the Visual Studio Code editor with two open COBOL files. The left file, `TS5162.CICSWS.IBM.COBL(PAYBUS).cbl`, contains the main logic for the payroll update routine. It starts with a comment "this is the Payroll update" and a section labeled "UPDATES.". The logic is divided into two main paths: "First time through just displays the record" and "Second time through updates the record". The first path checks if `ws-update-ind = 'n'` and displays a message. The second path checks if `ws-update-ind = 'y'` and performs a `READ UPDATE` operation on the `PAYROLL` file. It also includes error handling for `dfhresp(NORMAL)` and `dfhresp(UPDATE)`. The right file, `TS5162.CICSWS.IBM.COBL(PAYBUS).cbl`, contains the `EXEC CICS` commands for the routine, including `EXEC CICS RETURN`, `EXEC CICS ASKTIME`, `EXEC CICS REWRITE`, and `EXEC CICS EXIT`. A comment "CODE Eliminated to save space" is visible in the right file.

Figure 4-12 Updating a record

The first complication with this routine is that it is invoked twice. The first time verifies that the record exists by calling the `DISPLAY` routine and presenting the record back to the caller. This action has two purposes:

- ▶ To verify that the record exists.
- ▶ To unlock the fields on the 3270 display so the user can update only those fields that they want to update. An example is raising a person's salary. The remainder of the record would remain unchanged but only that field would be updated.

With CICS, you can update a record by adding the attribute **UPDATE** to a regular **READ** command. When the **UPDATE** is coded, CICS expects the **REWRITE** command to be issued next. A key does not have to be passed because CICS assumes that you are updating the record from a previous **READ** with the update attribute.

The remainder of the code between the **READ UPDATE** and **REWRITE** is about updating only the fields that the user changed and leaving the remaining fields as they are. This task is accomplished by using and testing flags that are sent by the caller.

There is also an extra piece of logic that collects a write a timestamp into the record. Although this logic not used in this piece of code, the idea is that if two people are trying to update the same record concurrently, one of the user's updates might be lost. In a production copy of this code, there is an extra test to make sure that the timestamp of the record that is read and presented to the user matches the timestamp of the record being updated. If it changed, the user is warned that the record changed since they last saw it, and the update would fail.

What happens if the transaction abends or fails in middle of the update? Would the new data or the old data be displayed on a lookup? The answer is that it depends. The systems programmer controls whether a VSAM file is recoverable. If it is recoverable, any changes to a record are backed out if a failure occurs. If this function is not set, then the update takes place. Talk to your systems programmer about an application's needs when dealing with issues such as recovery.

4.2.4 Adding a record

The next function is adding a record. It is a two-step process like an update (see Figure 4-13).

```

216 ADDSD.
217
218 * Add new employee record
219 * first time through ask for details and confirmation
220
221 if ws-add-ind = 'n'
222     exec cics read file('PAYROLL')
223         ridfld(ws-key)
224         into(payroll-record)
225         nohandle
226     end-exec
227
228 if eibresp = dfhresp(NORMAL)
229     move 'Record already on file' to ws-msg
230     move 'y' to ws-error-ind
231 else
232     move spaces to ws-name
233     move ws-addr1
234     move ws-addr2
235     move ws-addr3
236     move ws-phone-no
237     move ws-timestamp
238     move ws-salary
239     move ws-start-date
240     move ws-remarks
241     move 'y' to ws-add-ind
242     move 'Enter new employee details and hit PF5' to
243     ws-msg
244 end-if
245
249 move 'n' to ws-add-ind
250 move ws-department to pr-department
251 move ws-employee-no to pr-employee-no
252 move ws-name to pr-name
253 move ws-addr1 to pr-addr1
254 move ws-addr2 to pr-addr2
255 move ws-addr3 to pr-addr3
256 move ws-phone-no to pr-phone-no
257 move ws-salary to pr-salary
258 move ws-start-date to pr-start-date
259 move ws-remarks to pr-remarks
260 move 'Employee added successfully' to ws-msg
261 exec cics asktime abstime(ws-time)
262 end-exec
263
264 move ws-time-stamp to pr-time-stamp
265
266 exec cics write file('PAYROLL')
267     ridfld(ws-key)
268     from(payroll-record)
269     nohandle
270 end-exec
271
272 if eibresp not = dfhresp(NORMAL)
273     move 'Add failed' to ws-msg
274     move 'y' to ws-error-ind
275 end-if
276
277 exit.
  
```

Figure 4-13 Adding a record

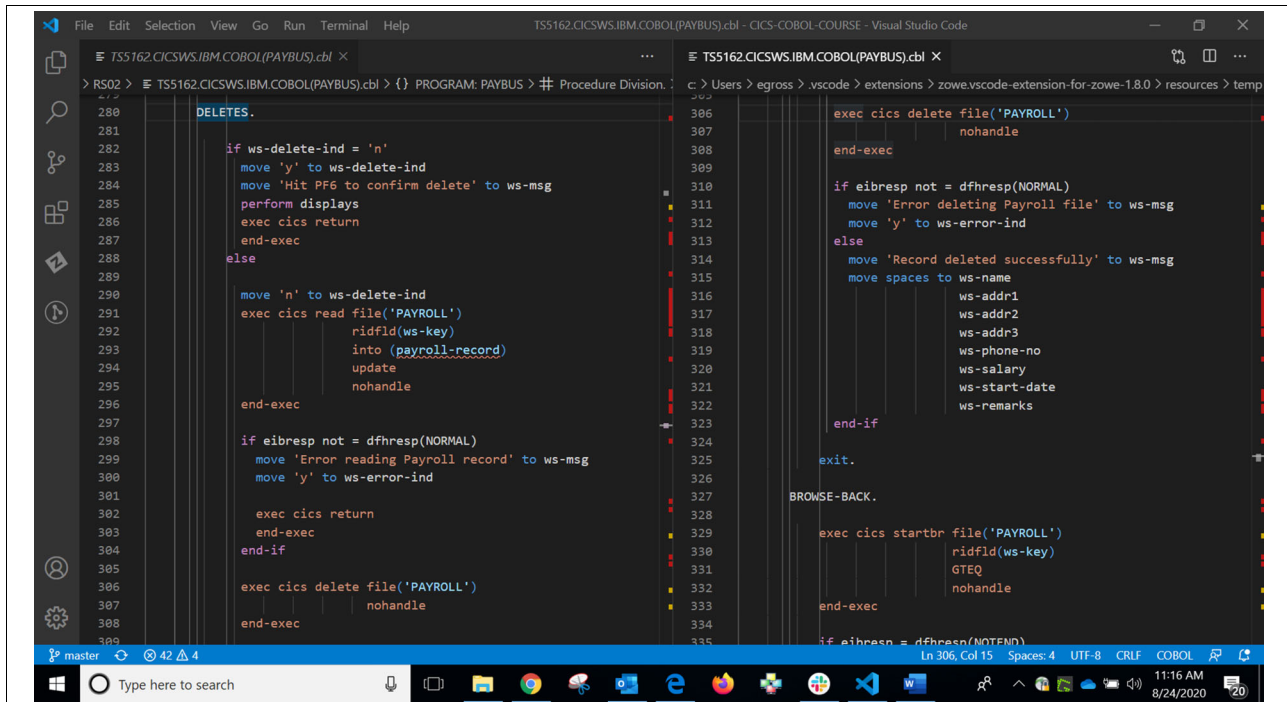
The first time through the task, the ADD function checks whether the record exists. If so, it sends back the record with a message that it is on the file.

If the record does not exist, this program clears all the fields that are required for adding a record and returns to the caller. In the case of the 3270 presentation logic program, this situation gives it the chance to unlock the fields so that the user can enter the new fields. On the second request, it adds the record that is presented by the user. It also places a timestamp in the record to use with the **UPDATE** command.

As with an **UPDATE** command, if a failure occurs during the run of this transaction, the record might or might not be added depending on the recoverability of the file.

4.2.5 Deleting a request

The next function is a delete request (see Figure 4-14).



```
280      DELETES.
281
282      if ws-delete-ind = 'n'
283          move 'y' to ws-delete-ind
284          move 'Hit PF6 to confirm delete' to ws-msg
285          perform displays
286          exec cics return
287          end-exec
288      else
289          move 'n' to ws-delete-ind
290          exec cics read file('PAYROLL')
291          ridfld(ws-key)
292          into (payroll-record)
293          update
294          nohandle
295          end-exec
296
297      if eibresp not = dfhresp(NORMAL)
298          move 'Error reading Payroll record' to ws-msg
299          move 'y' to ws-error-ind
300
301          exec cics return
302          end-exec
303      end-if
304
305      exec cics delete file('PAYROLL')
306      nohandle
307      end-exec
308
309      if eibresp not = dfhresp(NORMAL)
310          move 'Error deleting Payroll file' to ws-msg
311          move 'y' to ws-error-ind
312      else
313          move 'Record deleted successfully' to ws-msg
314          move spaces to ws-name
315          move ws-addr1
316          ws-addr2
317          ws-addr3
318          ws-phone-no
319          ws-salary
320          ws-start-date
321          ws-remarks
322      end-if
323
324      exit.
325
326      BROWSE-BACK.
327
328      exec cics startbr file('PAYROLL')
329      ridfld(ws-key)
330      GTEQ
331      nohandle
332      end-exec
333
334      if eibresp = dfhresp(NOTEND)
```

Figure 4-14 Deleting a request

The **DELETE** request is also a two-step process, where a user presses PF6 to invoke a delete when entering a Department and Employee. The first time through this task, this program calls the **DISPLAY** routine to return the record back to the caller that they are about to delete. Pressing PF6 a second time invokes the **DELETE** command.

The first thing this program does as part of step two is a **READ** with the update flag on the record about to be deleted. This action ensures that when we lock the record that we verify that no other customer is in middle of an update. After **READ** runs, we locked the record so that we can delete it later.

4.2.6 Browsing forward and backward

The last bit of code that we look at in this program is browsing forward and backward (see Figure 4-15 on page 35). Because the code is practically identical between the two, they are described together.

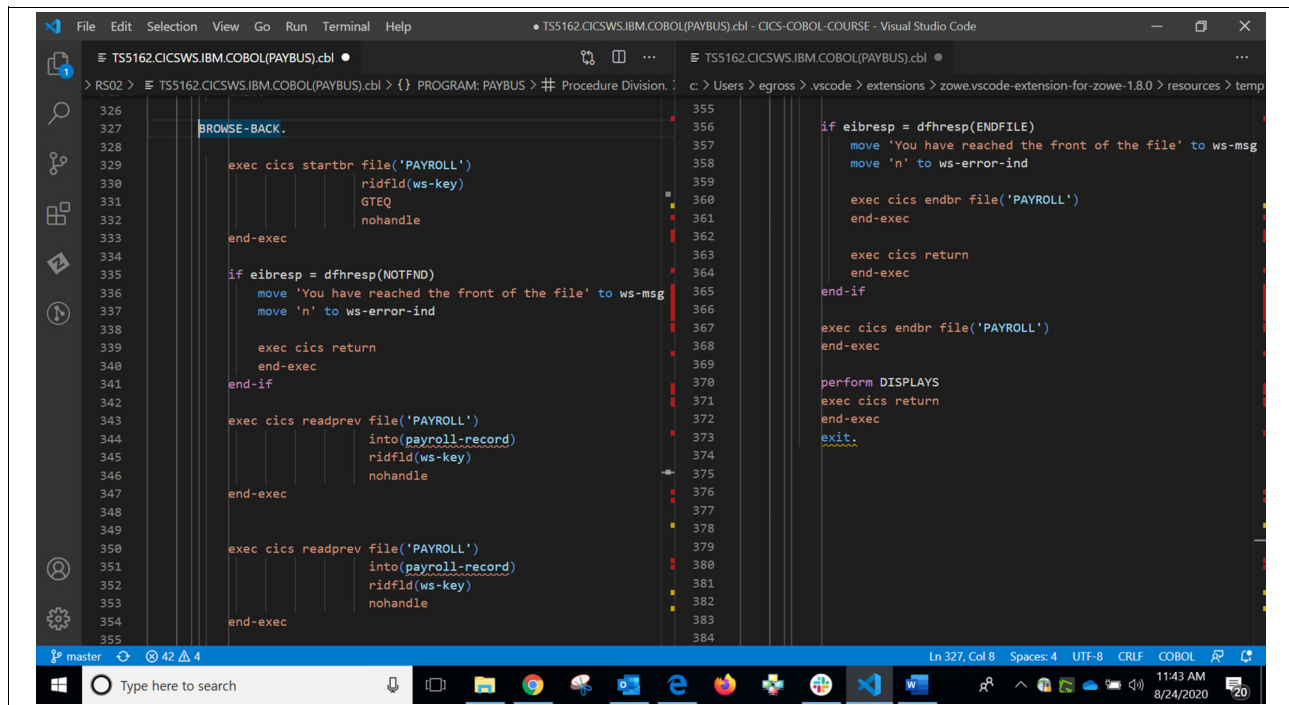


Figure 4-15 Browsing forward and backward

The concept of browsing through a keyed file is not new, but one that is used by many other products when accessing a keyed file is new.

In the case of CICS, you see that we start with a **start browse** command. This command is presented with the key in which to start browsing with the attribute **GTEQ**. **GTEQ** means greater than or equal to, and you use it to position the pointer to the record that is requested or the next one if this record does not exist.

So, if a customer wanted department one employee one and the record does not exist, the system automatically positions the pointer to the next available record for the browse forward and the previous record for the browse backward. Should a not found condition be returned, it means that we are at the front or back of the file, so we notify the user that there are no more records before or after that key.

The next set of commands appears to be a bug. Why would the programmer code two of the exact same commands one right after the other? In this example, they are **readprev** commands to browse backward. However, you can check that if we were browsing forward, we still see two commands that are called **readnext**.

The answer in the case of **readprev** is that we are pointing to the NEXT record to read. The first **readprev** positions us at the current record, and the next one backs up to the previous record. Had we not used two of these commands, we would end up reading the same record over and over.

As you can imagine, **readnext** works the same way. The first **readnext** reads the current record because that is where we are positioned. The next **readnext** reads the record that follows the current record.

Lastly, you can see that an **ENDFILE** condition means we are at the start or end of the file depending on whether we are going forward or backward, so we receive that error.

Also, because **STARTBROWSE** holds a pointer to the file, it is good coding etiquette to end the browsing as soon as there are no more plans to browse the records. Ending the task works too, but if someone added code to this program and did not notice that they were still in a browse, it might cause problems in the future. Therefore, this program ends the browse when it has no need to browse any more records.



Modernization by using channels and containers

This chapter describes the modernization the Payroll application to use channels and containers.

5.1 Examining the existing functions

When considering the modernization of an application, the first pass should focus on the functions that are still valid versus older functions that can be ignored because they are no longer applicable in the new environment.

Figure 5-1 shows a copy of a Departmental Browse screen that was available in the original application. It was started by entering a department and employee on the main screen and then pressing PF9, which displayed the screen.

Emp No.	Name	Phone
00001	FIRST BIRD	00320001
00002	JAY	00000002
00003	MAGPIE	00000003
00004	STARLING	00000004
00005	GOLD FINCH	00000005
00006	BULL FINCH	00000006
00007	GREEN FINCH	00000007
00008	YELLOW HAMMER	00000008
00009	PIED FLYCATCHER	00000009
00010	STONE CHAT	00000010

F1 Help: F3 Exit: F10 Prev: F11 Next: F12 Select employee

Figure 5-1 Department browse screen

Based on the fact that the screen size allowed only 10 employee records, the corresponding code browsed an entire department by placing each record in a temporary storage queue in CICS, and then the code rendered a screen with the first 10 employee records.

The user could page through the contents of the department by using this screen and pressing PF11 to move forward and PF10 to move backward through the records.

When the presentation logic was moved from a 3270 screen to a browser, which could handle all records with a scroll bar, the code was removed to reduce the size and complexity of both the presentation and business logic modules.

Now, a client can call the business logic program several times to retrieve all the records in a department and display them as they prefer.

This example is only one example of modernization. Another example is to convert the communication area from a COMMAREA to channels and containers.

5.2 Introducing channels and containers

The channel and containers interface was added in 2005 to address a growing problem in the world of CICS. The size of data that needed to be stored between pseudo-conversations or passed among programs was getting larger. The maximum size of a COMMAREA could never exceed 32 K because of the field that was used to hold and specify the length of the COMMAREA.

Another issue was that customers were overloading the COMMAREA with data from various different sources even if the program that had control did not directly use the COMMAREA. This situation ensures that some program in the link chain eventually would have access to the data, and the easiest way to pass it was to keep the data in the COMMAREA.

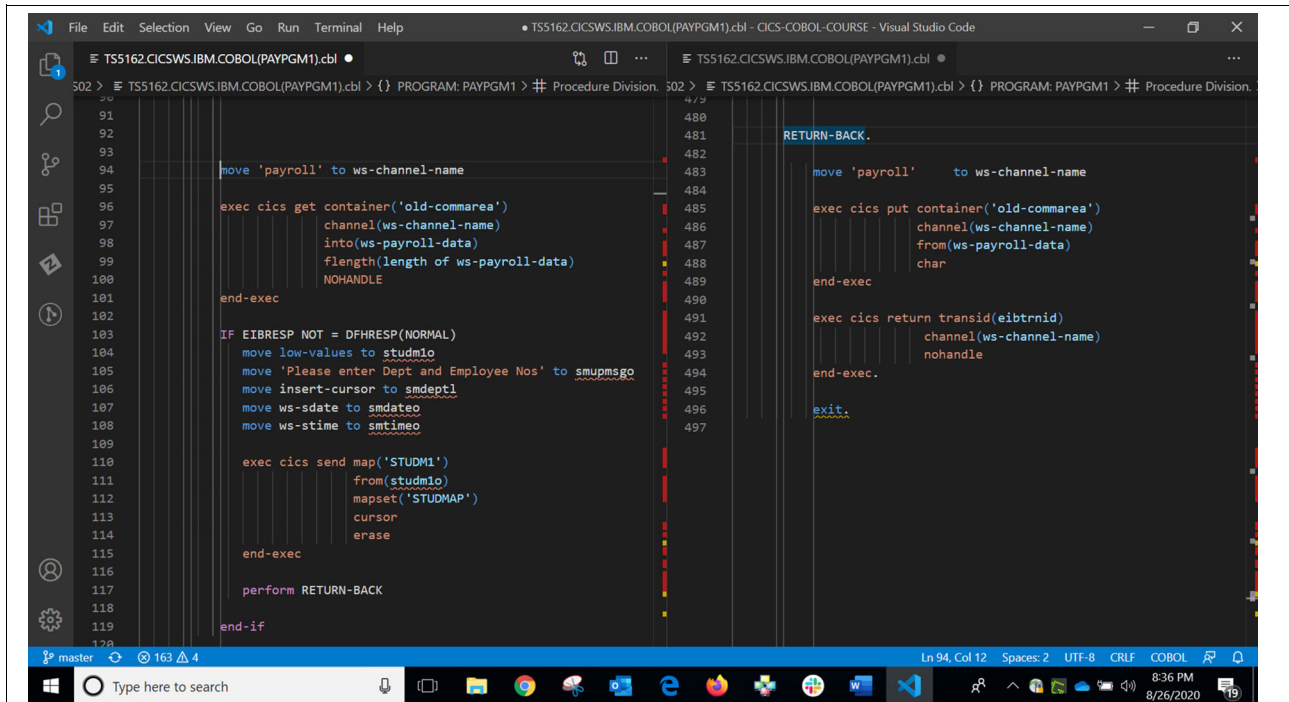
To solve these problems, channels and containers were introduced. The channels and containers interface uses a 16 character channel name as an anchor and 16 character container names to hold the data. In theory, you can create an unlimited number of containers in a channel that can be unlimited in size.

With this new function, CICS can resolve both the 32 KB COMMAREA limitation the overloaded copybook issue that results when using a COMMAREA.

The presentation and business logic programs that we use in our example do not have the COMMAREA size issue, but the newer channel and containers interface can make it easier for external systems to call the business logic program with this interface. So, let us explore the changes that are required to modernize this application to use channels and containers.

5.3 From COMMAREA to channels and containers

When converting to using channels and containers, the Linkage section is no longer required. Because the data must be physically placed and retrieved from a container in a channel, we provide the copybooks in working storage (Figure 5-2).



```
TS5162.CICSWS.IBM.COBOL(PAYPGM1).cbl
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2590
2591
2592
2593
2594
2595
2596
2597
2598
2599
2600
2601
2602
2603
2604
2605
2606
2607
2608
2609
2610
2611
2612
2613
2614
2615
2616
2617
261
```


In the code, you can see that the major difference is that the data must be read from a container before its use, which is accomplished by running a **GET** command. Because we also rely on the COMMAREA length field that is named EIBCALEN. During the first time through of the logic, the code on the left replaces both the receiving of our data the first time through logic.

Now, we get the data from the channel called payroll out of a container called old-commarea. If this command fails, then this situation is the first time through.

On the right side, you can see that we replaced the return with a COMMAREA by building a routine that puts (through the **PUT** command) the data into the container before the **return** command, which now references a channel rather than the COMMAREA. We converted all return *with COMMAREA* commands throughout this program to perform this routine before exiting the program.

The only other change in the presentation logic program is how it prepares for the call to the business logic and receives its results (see Figure 5-3).

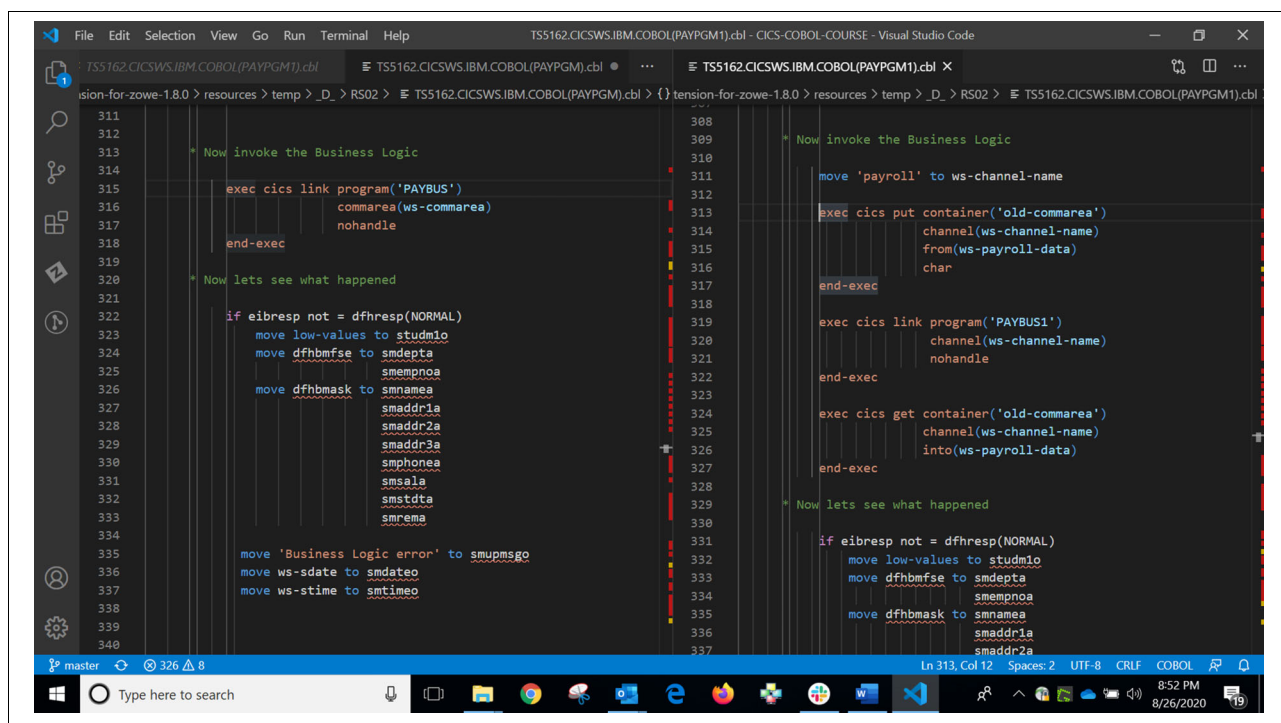


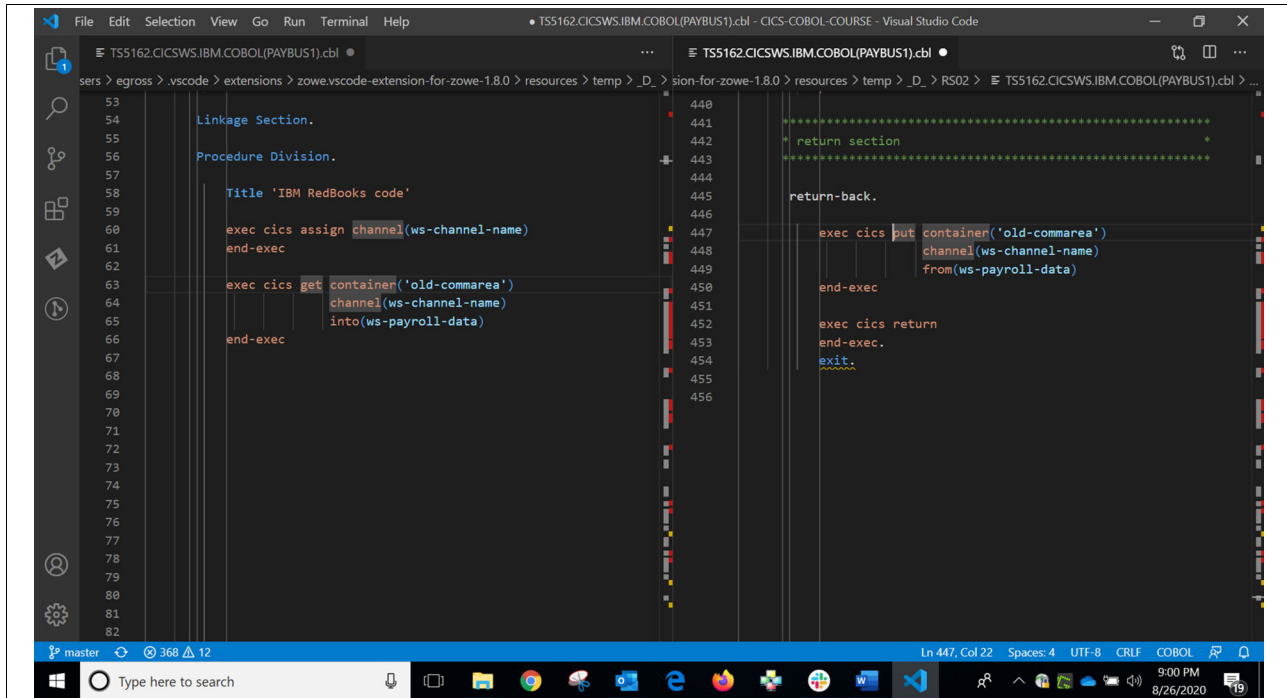
Figure 5-3 Starting the business logic

On the left side, you can see that in the older version of this program that it used a COMMAREA. We run **EXEC CICS LINK** with a COMMAREA to start the business logic because we are passing a pointer to where the data was.

On the right side, you see the new code that is required to pass a channel with containers to the business logic program. We must first place the data in the container by running **PUT** command, and then we issue the link with the channel attribute.

When we return, we must run a **GET** command to retrieve the results that are passed back from the business logic. The remainder of the presentation logic program is the same as it was before.

The changes to the business logic program PAYBUS are smaller because this program does not link to any other program. If it required a link to another program, we would follow the same logic of **PUT**, **LINK**, and **GET** (Figure 5-4).



```
53  
54 Linkage Section.  
55  
56 Procedure Division.  
57  
58 Title 'IBM RedBooks code'  
59  
60 exec cics assign channel(ws-channel-name)  
61 end-exec  
62  
63 exec cics get container('old-commarea')  
64 channel(ws-channel-name)  
65 into(ws-payroll-data)  
66 end-exec  
67  
68  
69  
70  
71  
72  
73  
74  
75  
76  
77  
78  
79  
80  
81  
82  
440  
441 *****  
442 " return section  
443 *****  
444  
445 return-back.  
446  
447 exec cics put container('old-commarea')  
448 channel(ws-channel-name)  
449 from(ws-payroll-data)  
450 end-exec  
451  
452 exec cics return  
453 end-exec.  
454 exit.  
455  
456
```

Figure 5-4 Changes to PAYBUS

Because we chose to use a single container, we must put that container into the copybook for reference in the program. This copybook used to be in the Linkage second, but was moved to working storage as part of the change to use channels and containers.

On the left, you can see the code that is used when opening this new version of PAYBUS. We run an **ASSIGN CHANNEL** command so we do not have to rely on the name of the channel being preset because we find it out. The container name must be known because there can be several in a channel.

As with the code seen in the end of PAYPGM, the presentation logic program (see Figure 5-2 on page 39), we must make sure to put (through the **PUT** command) the data back into the channel before returning to the calling program.

If we split the data across multiple containers in a channel, we would need a separate **GET** container command to retrieve the data in each container.

5.4 Working with CICS programs in Visual Studio Code

Here you look at how you change a CICS program by using Visual Studio Code, and what is required to prepare a program change for testing in CICS (see Figure 5-5).

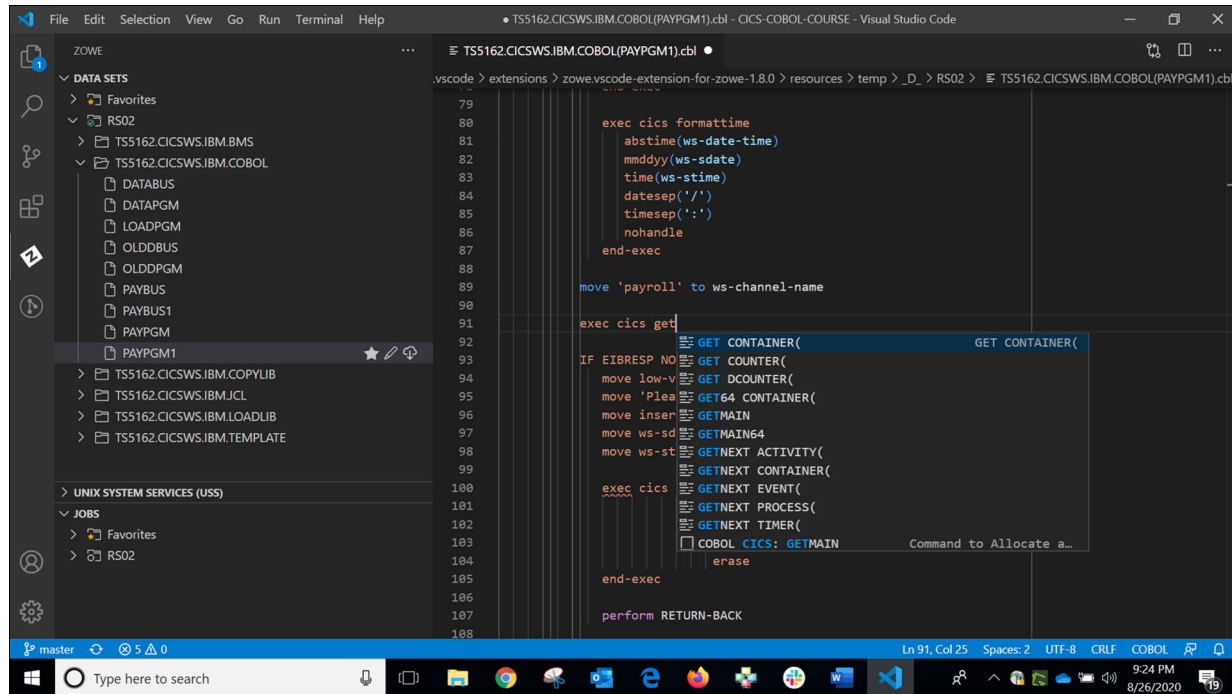


Figure 5-5 Auto-complete CICS commands

If you install the IBM Z Open Editor extension in Visual Studio Code, it can auto-complete COBOL instructions and CICS commands. Start typing the CICS command, and you are automatically prompted with the commands for CICS. After you press Enter for the needed command, you can always press Ctrl + Spacebar to have the prompt return for any other attributes that are associated with that command.

By using this feature, you ensure that the command has the right syntax and that it is more likely that the compile completes without any syntax errors.

After you complete your changes, you must recompile the program or programs that changed. This task is relatively easy to do in Visual Studio Code if you have a compile Job Control Language (JCL) setup (see Figure 5-6 on page 43).

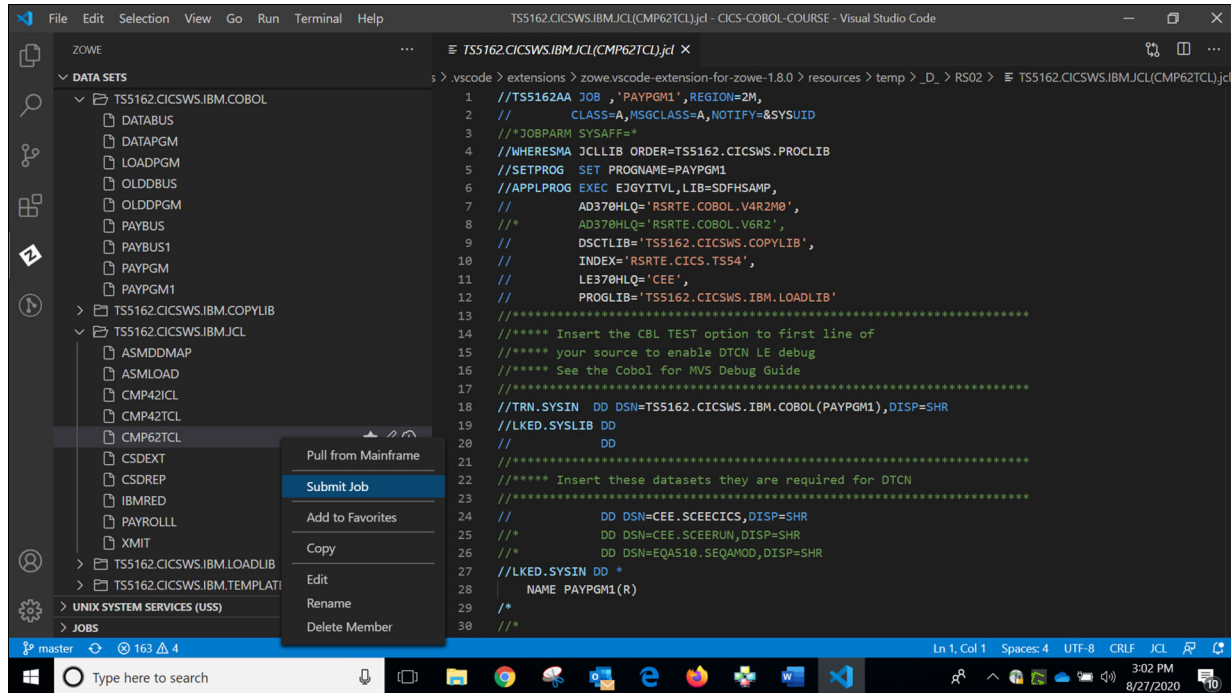


Figure 5-6 Submit job

If you go to your JCL library and find the job that does a compile, you can edit it by double-clicking it. To submit the job, select it, right click, and select **Submit Job** to submit the job.

After the job is submitted, you receive a message in the lower right of the screen that contains the job number of the JCL that was submitted (see Figure 5-7).

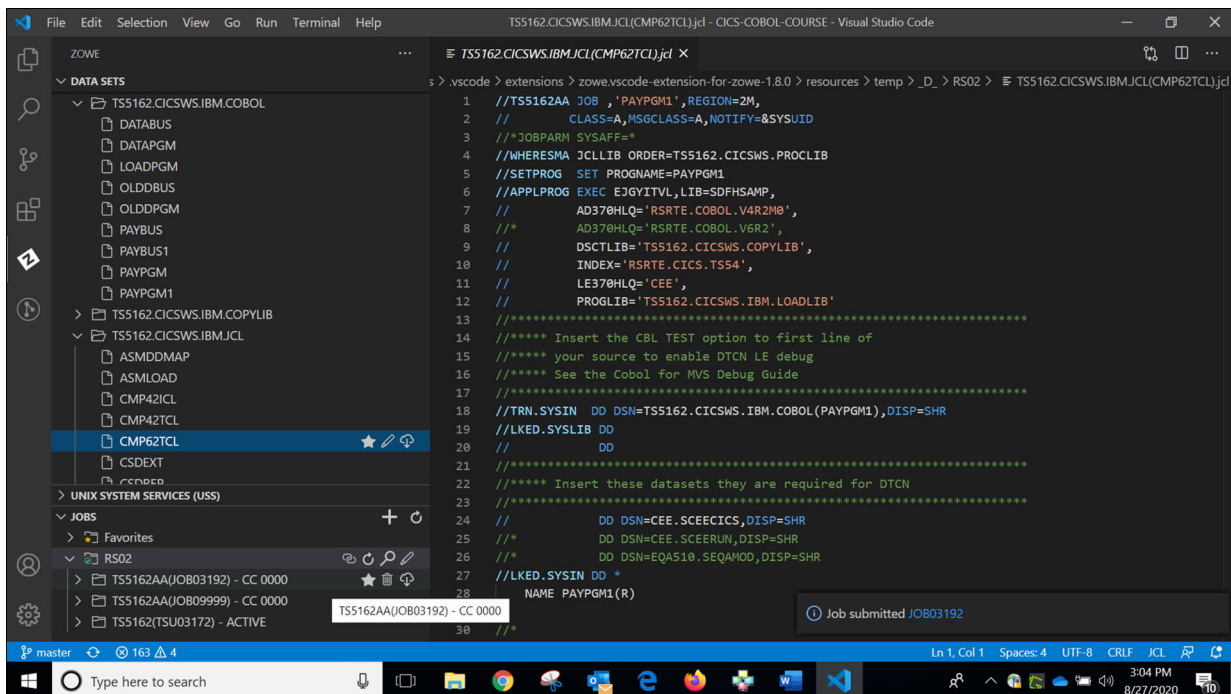


Figure 5-7 Job number

Using the left side of the Visual Studio Code screen, go to your jobs folder to find the matching job number. In our case, the condition codes were all zero, so everything worked. If not, the highest condition code is displayed.

If you must see what went wrong, you can open the job as a file and review each of the steps (see Figure 5-8).

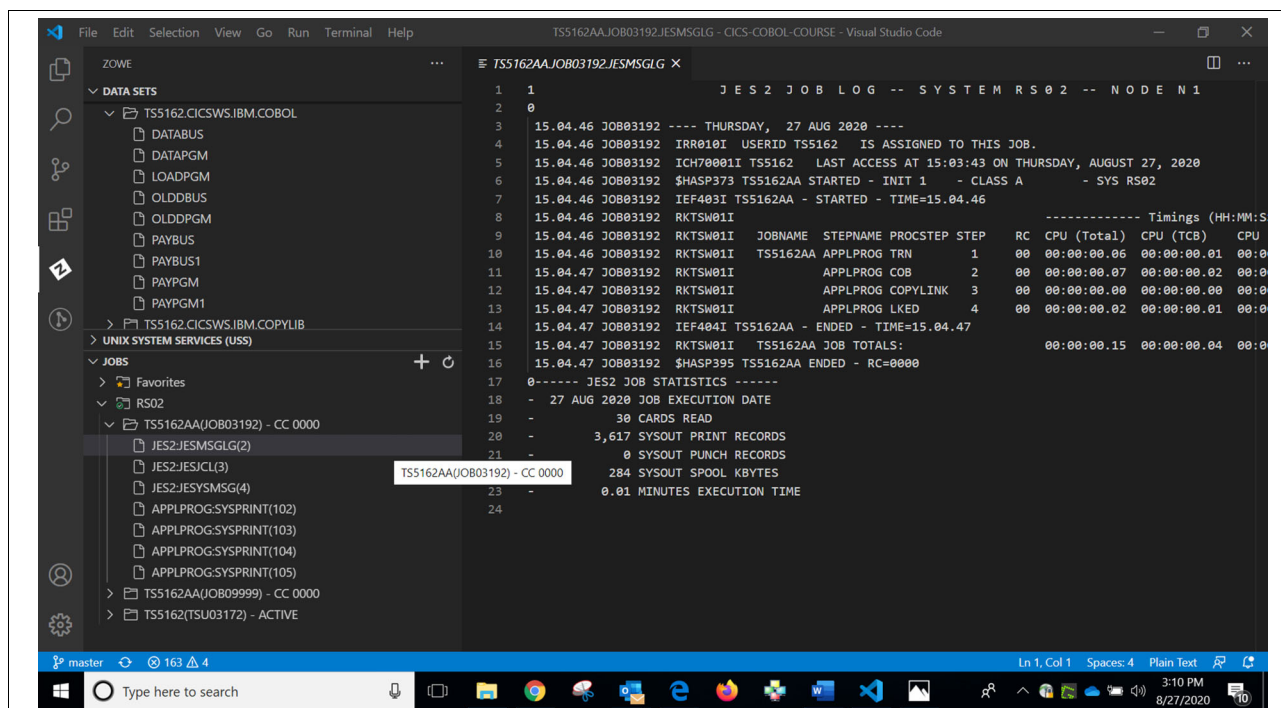


Figure 5-8 Job log

Here, we open the JES Message Log to review the return codes from each step. If we found a nonzero return code, we click the SYSPRINT file below the JES Message Log to see what went wrong in that step.

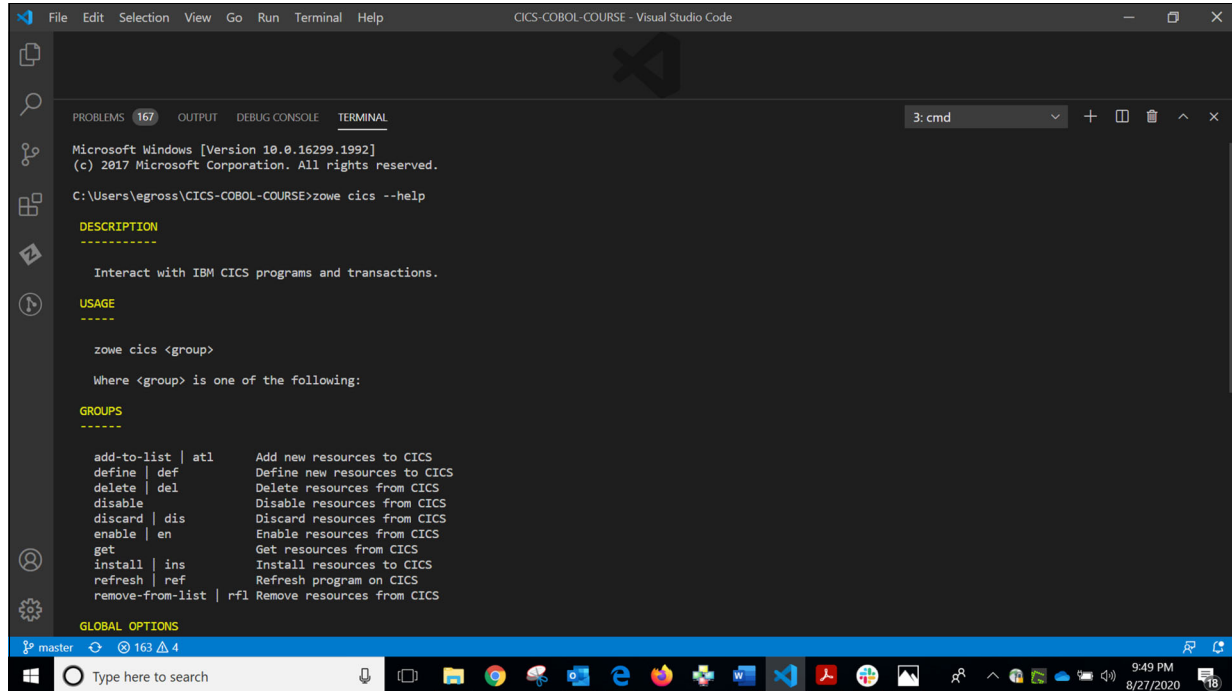
5.5 CICS and Zowe

Although Visual Studio Code may be installed with the IBM Z Open Editor extension, you cannot use it to work directly with a CICS region. To create a connection to CICS to perform a task, you must install the Zowe Command-Line Interface (Zowe CLI). After the CLI is successfully installed, you must install the Zowe CLI plug-in for IBM CICS.

For more information about installing both the plug-in and the CICS extension, see [Zowe Docs](#).

After the plug-in and extension for CICS are installed, you can read the documentation for the available CICS commands by opening a terminal in Visual Studio Code and running the following command (see Figure 5-9 on page 45):

```
zowe cics --help
```



```
Microsoft Windows [Version 10.0.16299.1992]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\egross\CICS-COBOL-COURSE>zowe cics --help

DESCRIPTION
-----
Interact with IBM CICS programs and transactions.

USAGE
-----
zowe cics <group>

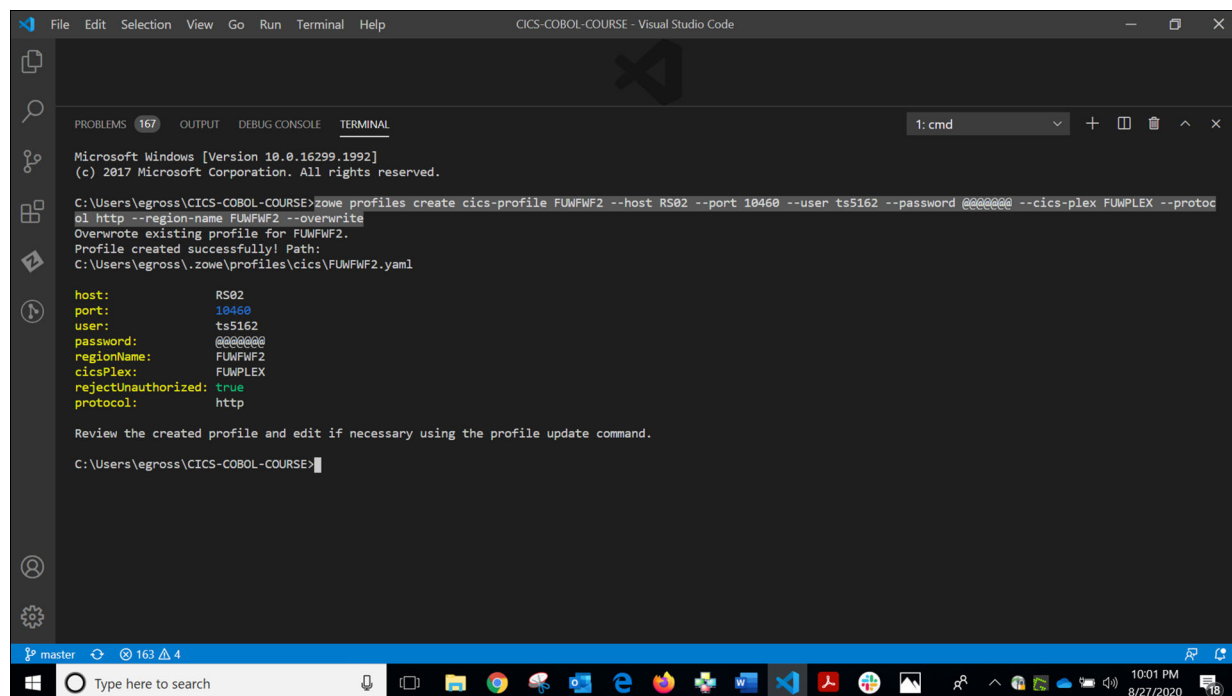
Where <group> is one of the following:

GROUPS
-----
add-to-list | atl      Add new resources to CICS
define | def          Define new resources to CICS
delete | del          Delete resources from CICS
disable | dis         Disable resources from CICS
discard | dis         Discard resources from CICS
enable | en           Enable resources from CICS
get | get             Get resources from CICS
install | ins         Install resources to CICS
refresh | ref         Refresh program on CICS
remove-from-list | rfl Remove resources from CICS

GLOBAL OPTIONS
```

Figure 5-9 Zowe CICS help

The commands that are most useful when working with a CICS program are the **get** and **refresh** program commands. However, to run these commands, we must first create a profile to communicate with our CICS region (see Figure 5-10).



```
C:\Users\egross\CICS-COBOL-COURSE>zowe profiles create cics-profile FUNFWF2 --host RS02 --port 10460 --user ts5162 --password @@@@@@ --cics-plex FUNPLEX --protocol http --region-name FUNFWF2 --overwrite
Overwrote existing profile for FUNFWF2.
Profile created successfully! Path:
C:\Users\egross\.zowe\profiles\cics\FUNFWF2.yaml

host:      RS02
port:      10460
user:      ts5162
password:  @@@@@@
regionName: FUNFWF2
cicsPlex:  FUNPLEX
rejectUnauthorized: true
protocol:  http

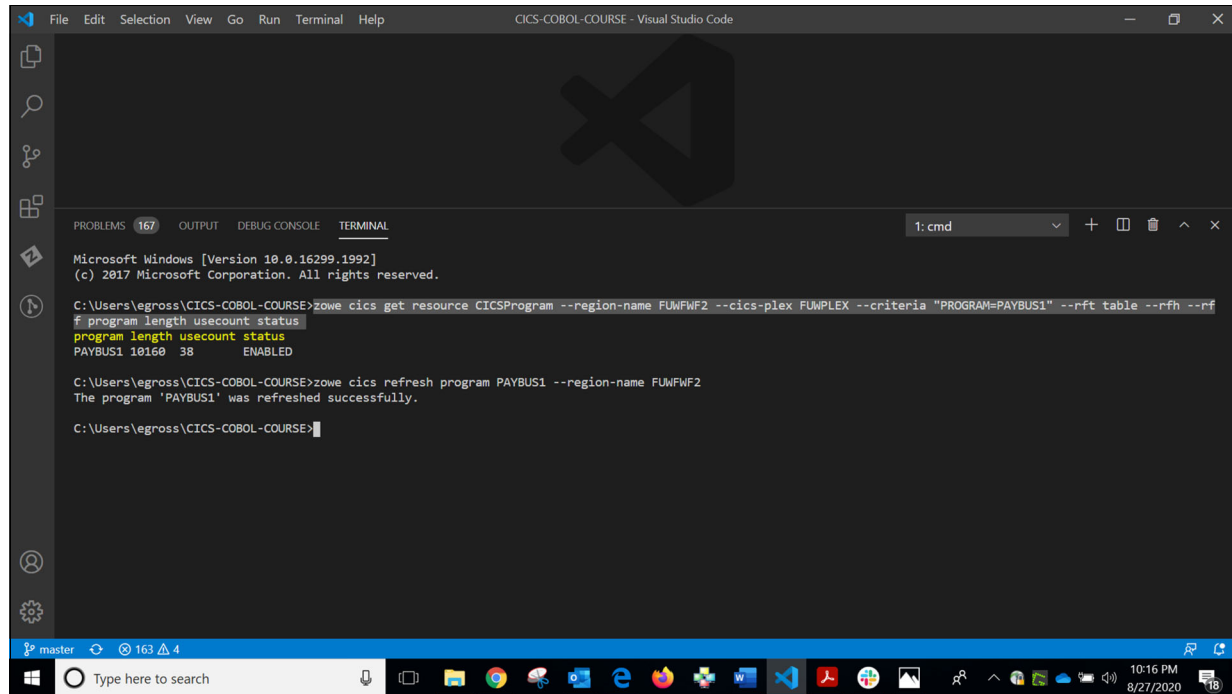
Review the created profile and edit if necessary using the profile update command.

C:\Users\egross\CICS-COBOL-COURSE>
```

Figure 5-10 Zowe profile

Setting up a profile requires entering much information. At a minimum, you need the protocol (HTTP OR HTTPS), the host and port number to which the CICS region will be listening, a user ID and password to which to connect, and the region name. In Figure 5-10 on page 45, we also used the IBM CICSplex® name in case the program that we are working with runs in multiple regions.

To verify that the program is defined in the region that is specified in the profile, run the **get** command (see Figure 5-11).

The image is a screenshot of a Visual Studio Code window titled 'CICS-COBOL-COURSE - Visual Studio Code'. The interface shows a dark-themed editor with a large 'X' watermark. Below the editor is a panel with tabs for 'PROBLEMS', 'OUTPUT', 'DEBUG CONSOLE', and 'TERMINAL'. The 'TERMINAL' tab is active, showing a Windows command prompt. The prompt displays the output of the 'cics get resource' command, which returns a table of program attributes. The table has columns for 'program', 'length', 'usecount', and 'status'. The first row shows 'PAYBUS1' with a length of 10160, a usecount of 38, and a status of 'ENABLED'. Below the table, the prompt shows the output of the 'cics refresh program' command, which successfully refreshes the program 'PAYBUS1'. The terminal window is titled '1: cmd' and has a dropdown menu for selecting the terminal type. The Windows taskbar is visible at the bottom, showing the search bar and various application icons. The system clock in the bottom right corner indicates the time is 10:16 PM on 8/27/2020.

```
Microsoft Windows [Version 10.0.16299.1992]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\egross\CICS-COBOL-COURSE>zowe cics get resource CICSProgram --region-name FUMFWF2 --cics-plex FUMPLEX --criteria "PROGRAM=PAYBUS1" --rft table --rft --rf
f program length usecount status
program length usecount status
PAYBUS1 10160 38 ENABLED

C:\Users\egross\CICS-COBOL-COURSE>zowe cics refresh program PAYBUS1 --region-name FUMFWF2
The program 'PAYBUS1' was refreshed successfully.

C:\Users\egross\CICS-COBOL-COURSE>
```

Figure 5-11 The **get** command

The **get** command that we used is more complicated than the standard **get** command. In addition to specifying the region name and CICS program name, we asked for the results to be returned in a table. As part of the table, we asked for the attributes of the program name, length, use count, and status. These attributes can be useful because we can watch them change if the program is run or if a change increased or decreased the size.

One of the aspects of CICS that makes it efficient is that the first time a program is run, it is loaded into the core. Subsequent calls to run the same program do not go out and load a new version, but instead use the version that was loaded the first time that it ran. This advantage saves processing time, but to get CICS to load a new version of the program, we must force the issue.

The **refresh** command has many names depending on where it runs. When it runs in CICS from a 3270 screen, it is known as a **newcopy** or **phasein**. When the command runs from Visual Studio Code by using the Zowe CLI extension for CICS, it is known as a *refresh*.

After **refresh** runs, we can test our new version of the program. If we changed the length of the program, we can run another **get** command because the program length shows the size of the new version of the program, which provides some assurance that we changed the program.



Modernizing applications with Java

So far, we have focused on COBOL as a language with which to write IBM CICS applications. However, COBOL is not the only language that CICS supports. An application can be created from a set of programs that is written in different languages. This polyglot capability is paramount in allowing developers to take applications that are written in COBOL and create Java components, or even rewrite elements in Java and then host them in CICS. The new Java component can still access the CICS application programming interface (API) and call existing COBOL programs by using a native interface.

6.1 Why use Java with CICS

Breaking the presentation from the business logic of an existing application provides the key to modernizing an application. This situation is also true when you use Java as part of your application modernization. If the core of your application has a defined interface, then it is easy to consider writing new applications that can use that application or rewriting it in Java. If the interface uses a structured channels and container interface, then this situation is even better.

But why replace or extend applications with Java? What benefits can there be for your organization?

It is probably easier to find Java application programmers than COBOL programmers, which means that there is a larger skill pool that you can draw from to access new developers. There is also a wider range of both protocol and framework support in Java than there is for other languages like C or COBOL. You can interact with JSON objects or service an HTTP GET request in COBOL, but it is probably easier in Java.

Java runs well on the mainframe. If you have a specialty engine like a zIIP processor in your IBM Z hardware, then most your Java code can run on that engine rather than on your general-purpose engine, which also means that running the code does not affect your monthly license charge for the specialty engine.

Finally, the best reason for using Java to write CICS applications is that there is much support in CICS for Java.

6.2 Writing CICS Java applications

If you are writing in Java, then you have an integrated development environment (IDE), such as Eclipse, IntelliJ, or Visual Studio. You also might use Maven or Gradle as a build toolkit for your Java projects, so CICS provides its Java API as a Maven or Gradle dependency, which makes it easy to use your existing IDE to write Java in CICS. If you cannot use Maven or Gradle, you can use the IBM CICS SDK for Java to access the JCICS API, but in our example we focus on the Maven and Gradle options.

So what types of applications might you write? If you are writing a general-purpose audit logging component or creating an API that is based in Java to extend your existing applications, there is support in CICS to help you, from supporting plain old Java objects (POJOs) to running full Java Platform, Enterprise Edition applications or Spring Boot applications. CICS embeds both a Java virtual machine (JVM) server and an IBM WebSphere® Liberty application processor to run your Java applications. Because this JVM runs within the CICS application server, CICS ensures that your Java application runs with the same transactional, security, and performance concerns as the rest of the application running in CICS. CICS also supplies the APIs to the Java application to call both native CICS APIs and link to other programs.

6.2.1 Hello World code sample

This section looks at the Hello World program again and rewrites it to use Java running in CICS (see Example 6-1).

Example 6-1 Hello World code sample

```
public class App
{
    public static void main( String[] args )
    {
        Task.getTask().out.println("Hello World");
    }
}
```

This sample code is not much more complicated than a standard Hello World example in Java. We need use only the Task object to integrate with the CICS terminal. Although this is a simple bit of code, it is not of much use. Who wants to be writing to a terminal in Java? Let us move to something a bit more interesting.

6.2.2 Moving the Payroll application to Java

In previous chapters, we looked at a basic application that allows basic updates to be made against records in a file. Currently, this application uses a 3270 interface. We can take that application and modernize it to use Java.

Here we expose the PAYBUS1 application as a simple **GET API** call within Java. We also extend the application by writing a simple audit logging application to log the requests that we received (see Figure 6-1).

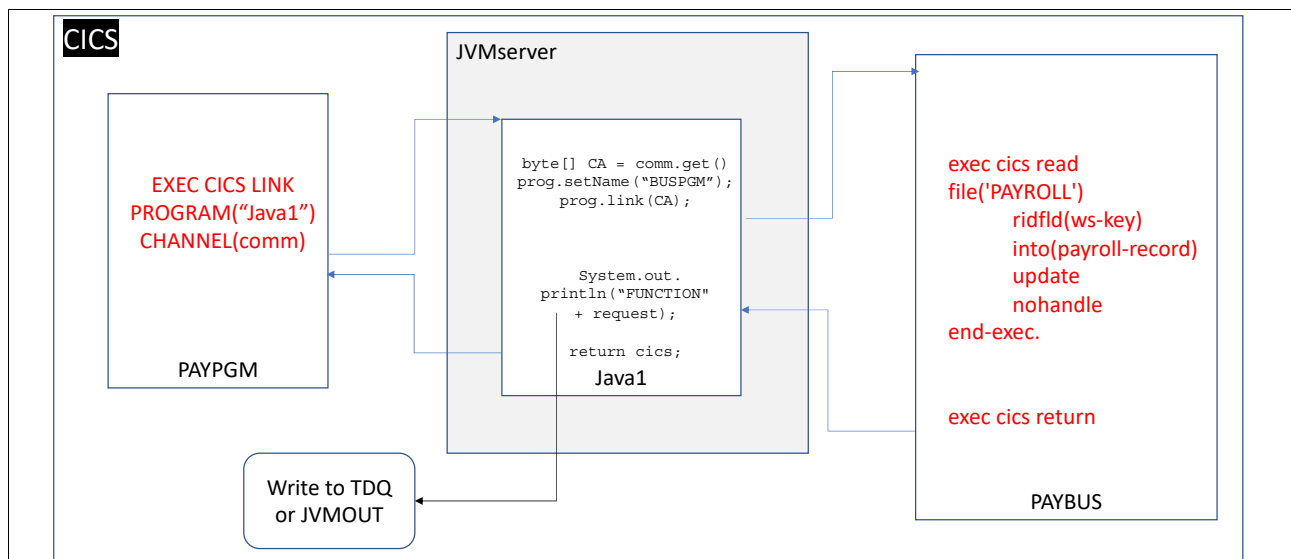


Figure 6-1 Audit logging application

Within CICS, we already have the PAYBUS program and a FILE DEFINITION for the file PAYROLL. Now, we also include a JVM server definition, which creates a JVM inside our CICS region. Within the JVM, we define an instance of the WebSphere Liberty Profile that we run our Java code inside. Within the WebSphere Liberty JVM, we create a simple JAX-RS servlet that responds to **HTTP GET** calls and responds with JSON.

After we receive a request, we use the channel and container CICS API to deliver a request to PAYBUS1 and ask CICS to run that program. After the program completes, we again use the channel and container API to retrieve the response from PAYBUS1, convert it to JSON, and send the response back to the user.

Then, we extend the application more and create a method within our servlet to perform some logging against a transient data queue (TDQ) in CICS. Each time a request is made, we write the data to this auditable log.

PAYBUS was modernized to use channels and containers earlier in this paper. Now, PAYBUS needs a single container that contains the function that we want to use, which in this case is DISP followed by the employee number to retrieve the employee details. To achieve this task, we must use the CICS API for Java that is called JCICSX to perform a **CICS LINK** to the program PAYBUS1 by passing it a channel within which our container was created.

Example 6-2 shows the code for this task.

Example 6-2 CICS LINK

```
response = task.createProgramLinkerWithChannel(programName, channelName)
            .setStringInput(containerName, inputData)
            .link()
            .getOutputCHARContainer(containerName)
            .get();
```

We direct the CICS task to create an object to link to a CICS program with a channel by using the method `createProgramLinkerWithChannel` and pass it the name of the program and the channel that we want to use. Then, we call `setStringInput` to create a container and put our line of string data into that container. Now, we have set up all the input data in the correct containers for PAYBUS1.

We call the method `link`, which requests that CICS runs the PAYBUS1 program by passing it the channel and container. This action is a *blocking method call*, which does not return until PAYBUS finishes processing. After control returns to our Java program, we retrieve our response from PAYBUS1, which is in the same container as the input data, so we call the **get output char** container to retrieve the container from the channel. Finally, a simple call of the **get** method retrieves our response data as a string.

This line of code is long, but it is a valuable and powerful line of code. We run a COBOL program from within a Java program, which is an impressive feat. We did not worry much about any data manipulation, byte alignment, COBOL runtime environments, or other such items. The CICS API protected us from that complexity so that we could easily achieve our goal. But, we are not done because we must interact with a TDQ to log data.

Instead of calling an existing CICS program to write the data for us, we use the CICS API to write the data directly to a TDQ so that it can be offloaded to a file for archiving.

Example 6-3 shows the code that we use.

Example 6-3 Writing to the transient data queue

```
TDQ queue = new TDQ();
queue.setName("AUDIT");
queue.writeString(message);
```

Example 6-3 on page 50 is more straightforward than Example 6-2 on page 50. We create an instance of the TDQ object, set the name of the TDQ that we want to interact with, and then call the `writeString` method by passing it the string that we want written. CICS handles the code page conversion that is necessary, finds the TDQ, and performs the write for us. The TDQ even can be on a different CICS region and CICS still would have handled it.

If you look at other Java and CICS examples, you might notice that they do not use the same objects and methods as this example because they are probably using the old CICS API that is called JCICS, and this example uses JCICSX. Both APIs can be used within CICS and are available as artifacts within Maven Central.

6.3 Unit testing Java applications

If you are considering writing Java in CICS, you probably have some experience writing Java on other platforms. You might have written a Java component for a web application or a basic Java application to run on your laptop. You probably wrote a few lines of code, tested it, probably found a bug, fixed it, and repeated the cycle. You probably want to do the same tasks with the Java code example in this paper.

6.3.1 Writing a basic unit test

First, write some unit tests for the sample Java code. *Unit tests* are automated tests that are stored with the programs that they test. A good unit test should test only the target program, and it should not rely on any other programs or external services.

Using our servlet from the previous section as an example, we write a unit test that calls this module with a customer number as a string and returns an instance of the `EmployeeData` object that contained the correct data.

Example 6-4 shows the unit test.

Example 6-4 Unit test example

```
@Test
    public void testServlet(){
        EmployeeData empData = p.getEmployee("12345");
        assertEquals("Invalid employee name", "William Leslie Yates",
empData.getName());
        assertEquals("Invalid employee address", "IBM Hursley Park,
Winchester", empData.getAddress());
        assertEquals("Invalid employee phone", "12346789",
empData.getPhoneNumber());
    }
```

When we run this test, it fails with a `RuntimeException` when it tries to call `getEmployee`. When our servlet attempted to use the CICS API, it failed because there is not a CICS system to run those API calls.

We can resolve this error by using two new technologies, which are known as *mocking* and *remoting*. They are both key features of the new JCICSX API. Both of these technologies run and test your Java code locally on your workstation. For our unit test, we use mocking.

6.3.2 Mocking with CICS applications

With mocking, we programmatically replace the objects in the CICS API with mock objects that return only the data we prepared. We create these mock objects in the setup method of the test by annotating them with `@Before`, as shown in Example 6-5.

Example 6-5 Mocking example

```
@Before
    public void setUp() throws Exception {
        String containerData = "DISP 12345" +
                                "William Leslie Yates      " +
                                "IBM Hursley Park, Winchester " +
                                "                               " +
                                "12346789";

        // mock the task object, as we don't want the unit tests to actually call
        CICS
            task = Mockito.mock(CICSContext.class);
            cpl = Mockito.mock(ChannelProgramLinker.class);
            cplr = Mockito.mock(ChannelProgramLinkerResponse.class);
            rcc = Mockito.mock(ReadableCHARContainer.class);
            Mockito.when(task.createProgramLinkerWithChannel("PAYBUS1",
"PAYROLL")).thenReturn(cpl);
            Mockito.when(cpl.setStringInput("old-commarea", "DISP
12345")).thenReturn(cpl);
            Mockito.when(cpl.link()).thenReturn(cplr);
            Mockito.when(cplr.getOutputCHARContainer("old-commarea")).thenReturn(rcc);
            Mockito.when(rcc.get()).thenReturn(containerData);

            p = new Payroll();
            p.setContext(task);
    }
```

Mockito is a Java based framework for creating mock objects to use in a JUnit test. Here, we create mock objects of all the CICS API classes that our Payroll servlet uses, and define the data that should be returned when methods on those classes are called. When our test runs, the mock objects are used instead of the real objects in the JCICSX API. Our unit test exercises only the Java code we wrote, and stubs out the rest of the CICS application. We can develop new methods within our Java class and get fast feedback from our JUnit tests each time we build our project.

In this example, we mocked out a simple call to a CICS program with a channel and container interface, but any class within the JCICSX API can be mocked.

Mocking and unit tests are a good way of ensuring that the Java code that we write works as planned. However, if you look at that setup method again, you see that the data that we used when mocking the `get` method of the readable char container was hardcoded into the test itself. How do we get that data so that we can use it in our test class?

6.3.3 JCICSX remoting

The answer lies in the JCICSX remoting capability. To explain this concept, we use the servlet that we wrote in the last module and for the JUnit tests. In Figure 6-2 on page 53, the servlet runs in a Java Platform, Enterprise Edition application server on a local laptop. The servlet uses the JCICSX API to link to the program PAYBUS1 with a channel.

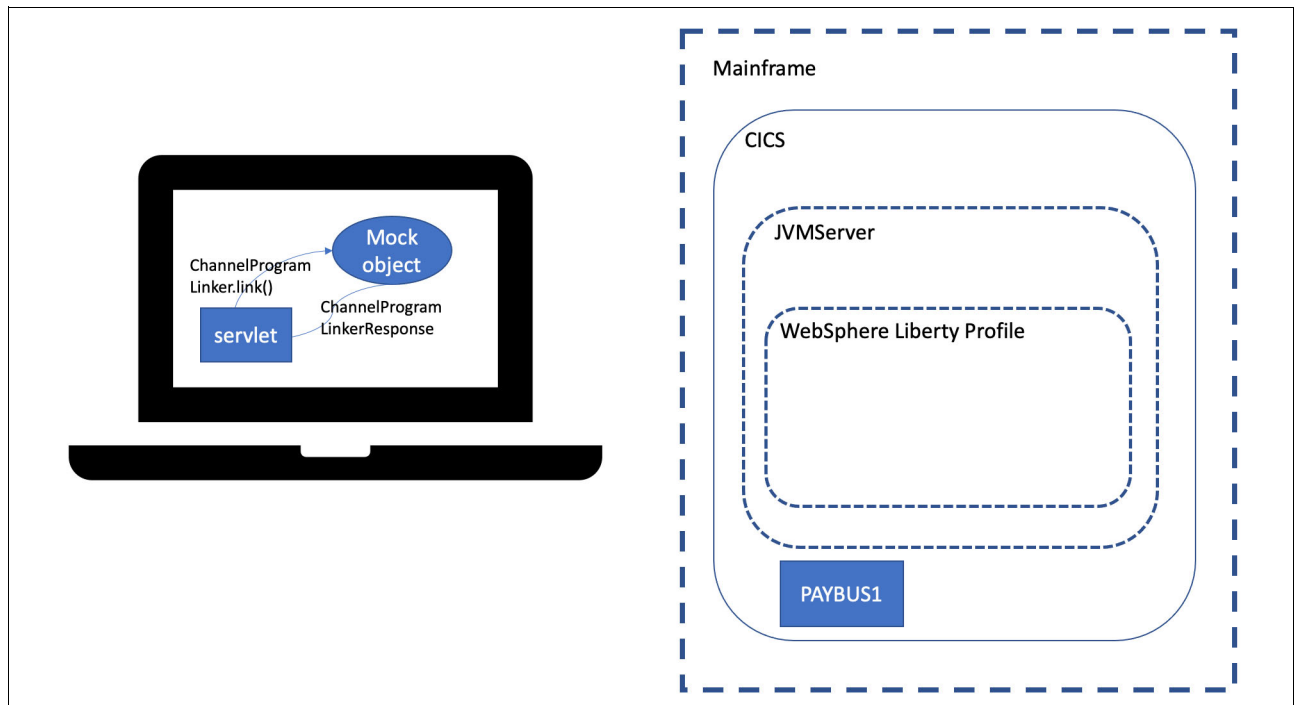


Figure 6-2 Servlet running on a local laptop

We can configure the Java Platform, Enterprise Edition application server on the local machine to use a remoting capability (see Figure 6-3).

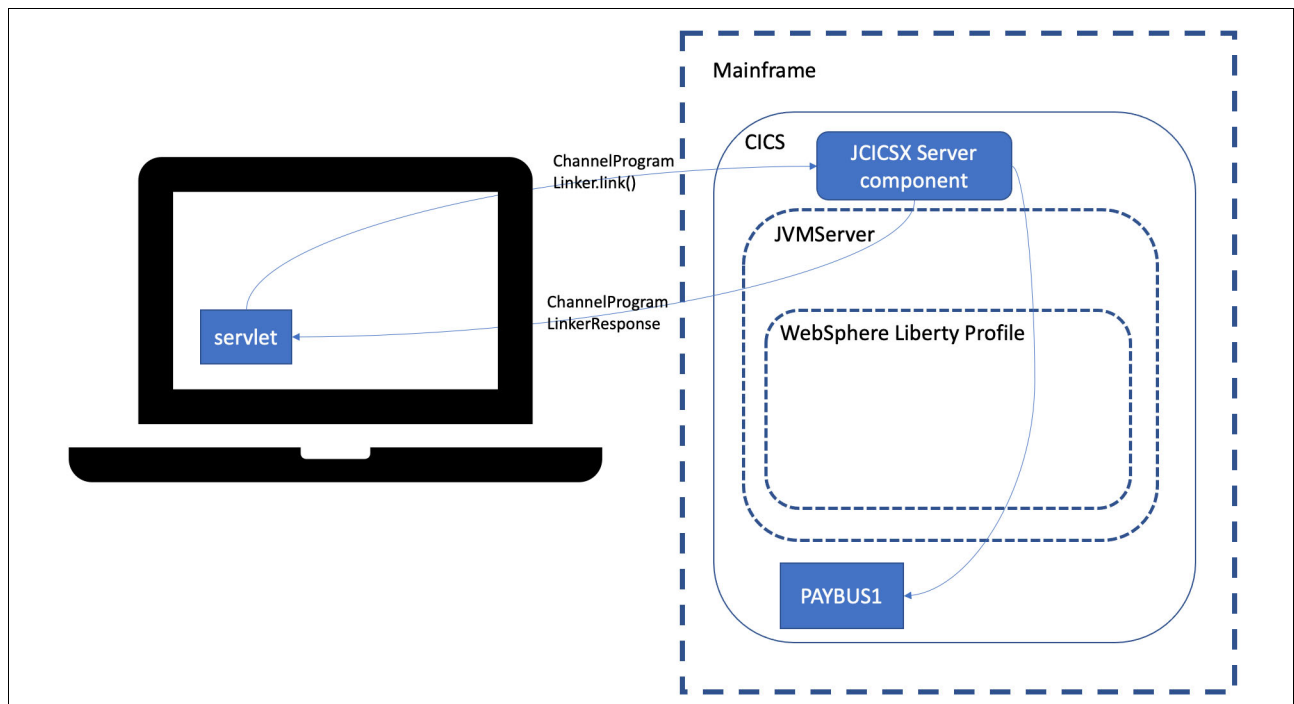


Figure 6-3 Remoting capability

When the servlet accesses the JCICSX API, the API transparently turns those calls into HTTP calls, which the API sends to a real CICS system. Within CICS, a JCICSX server acts as an endpoint for the API and runs the CICS request within the local machine (see Figure 6-4).

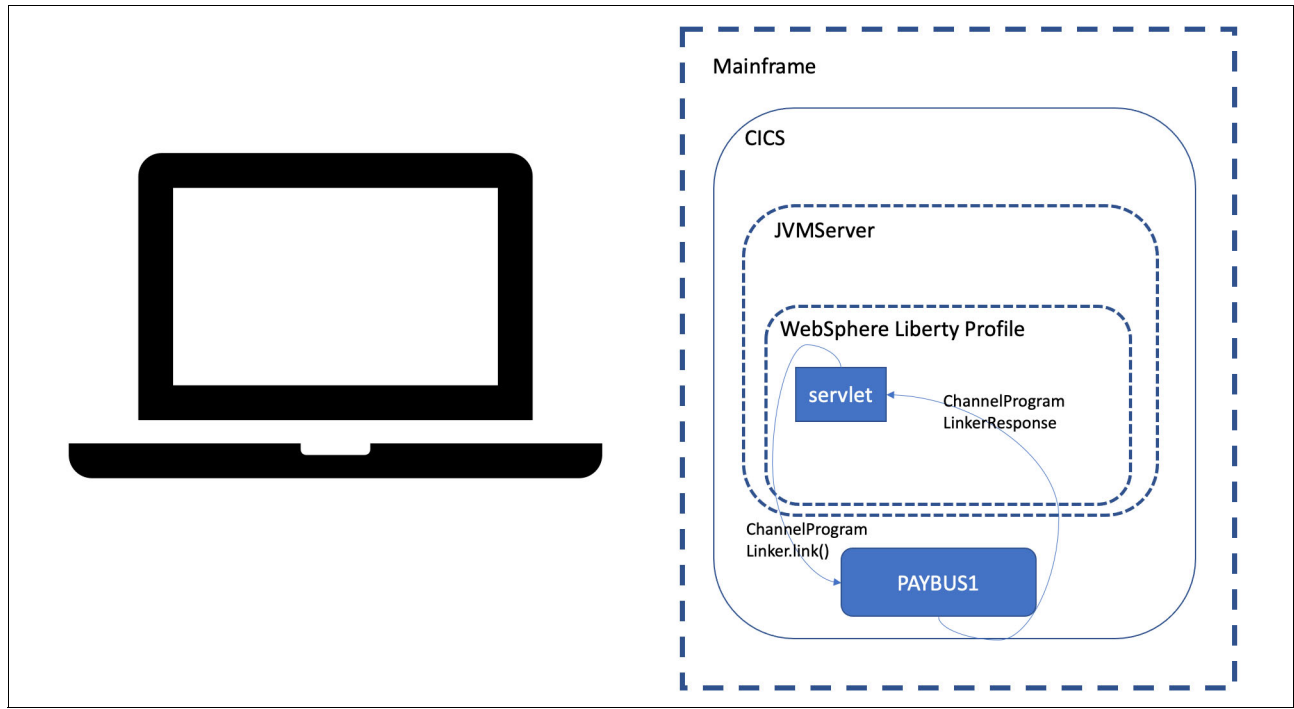


Figure 6-4 Application running in WebSphere Liberty Profile in CICS

The remoting capability is only for test and development because it is not designed as a generic connector for Java applications to connect to CICS.

When we are ready to run our application for real, we deploy the servlet from our local application server into CICS. When our servlet runs, the JCICSX API is aware that we are running inside of CICS. When we run the servlet inside of CICS by using the JCICSX API instead of using the remote connection, the API switches out the remote connection and replaces it with the standard cross-memory calls directly into CICS.

This section showed how writing a Java application for CICS is like writing a Java application for anywhere. You can build it locally, test it locally, and connect it to CICS to test the integration of your Java code with the rest of your CICS services.



Modern IBM CICS application programming features

Previous chapters in this paper described COBOL running inside CICS and the application programming interface (API). You saw how easy it is to include command-level **EXEC CICS** commands in a COBOL program to request CICS services.

As CICS evolves and new features are announced with every release, the API is enhanced to support these new features, for example, introducing web commands to support web protocols, and commands to support channels and containers.

This chapter describes three new features:

- ▶ Asynchronous programming
- ▶ Event processing
- ▶ Link to WebSphere Liberty

7.1 Asynchronous programming

CICS TS V5.4 introduced a set of APIs to simplify asynchronous programming for application programmers.

7.1.1 Asynchronous programming analogy

To explain what we mean by asynchronous programming, consider the following example.

Debra goes to the market on Saturday morning. She always parks in the short duration car park, so she has a limited amount of time to complete her shopping. She must buy bread and cakes from the baker, eggs from the butcher, and fruit and vegetables from the grocers. She goes to the shops in sequence: baker, grocer, and butcher. Unfortunately, at the baker shop, she must wait for bread to come out of the oven. She takes too long and gets a parking fine, as shown in Figure 7-1.

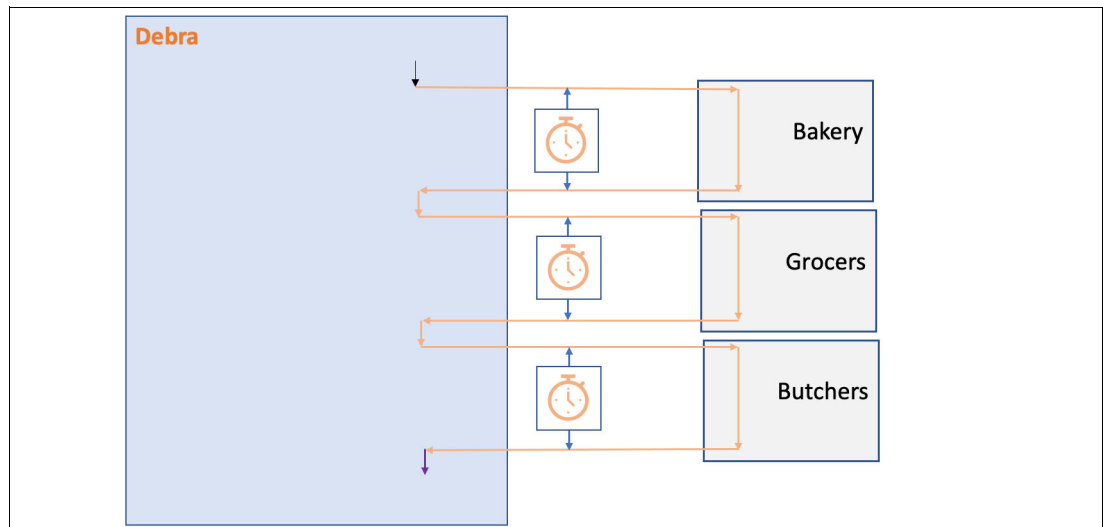


Figure 7-1 Shopping in a serial fashion

The following Saturday, she takes her three sons, Will, Martin, and Jim, with her. She sends Will to the bakers to buy bread and cakes, Martin to buy fruit and vegetables, and Jim to the butchers to buy the eggs.

She waits in her favorite café until they complete their tasks. She asks all three boys to text her when they finish shopping so she can arrange a place to collect them. When they all respond, Debra finishes her coffee, jumps in her car, and goes off to collect them. Debra completed the task asynchronously, as shown in Figure 7-2 on page 57.

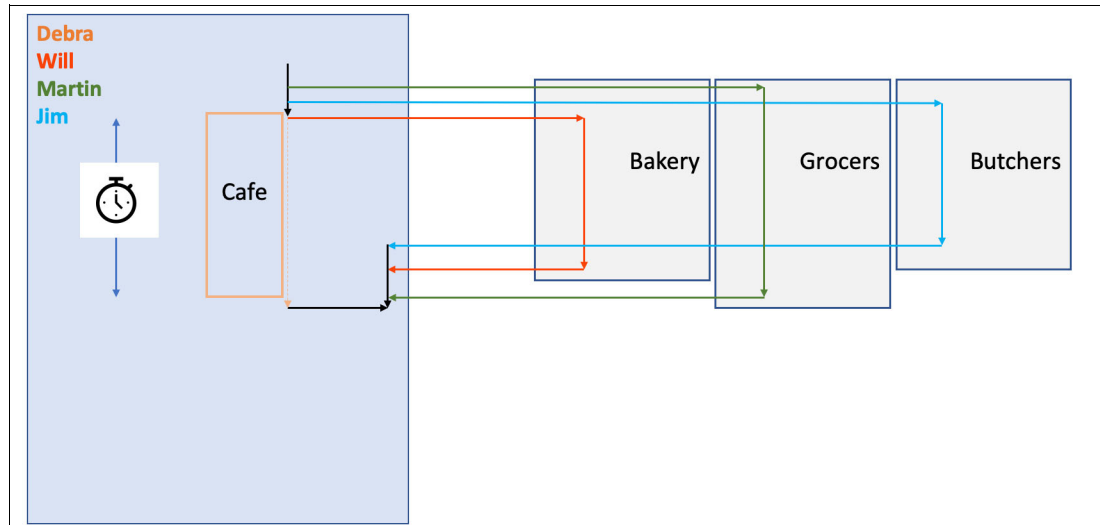


Figure 7-2 Asynchronous shopping

7.1.2 Asynchronous programming principles

The shopping example demonstrates the main principles of asynchronous programming.

- ▶ Split work into separate tasks that can be run independently (sending off her boys to shop).
- ▶ Trace the completion of the asynchronous work (the boys sent their mum a text when finished their shopping).
- ▶ Although the last principle was not demonstrated, it is passing data safely between the main and asynchronous tasks.

Asynchronous programming has been around for a while as a concept, and application programmers have used various programming techniques, such as **EXEC CICS START** and **EXEC CICS DELAY** with some kind of “polling” to test whether the called program completed, or they used an **EXEC CICS WAIT /POST** combination. However, these techniques are non-trivial and notoriously difficult to implement for several reasons, such as timing windows.

The aim of the CICS asynchronous APIs is to simplify this process for the application programmer by providing the function natively within CICS. The CICS asynchronous API is built around a parent to child programming model.

CICS APIs address the three main aspects by using four API commands:

- ▶ **EXEC CICS RUN TRANSID** initiates a CHILD transaction that runs asynchronously.
- ▶ **EXEC CICS FETCH CHILD** inquires on the status of a “specific” CHILD task.
- ▶ **EXEC CICS FETCH ANY** inquires on the status of “any” completed child task that has not yet been fetched.
- ▶ **EXEC CICS FREE CHILD** forgets a child that has not responded.

Additionally, existing **EXEC CICS PUT CONTAINER** and **EXEC CICS GET CONTAINER** commands pass data between asynchronously running processes.

Consider a parent task running some business logic. At some point, a child task runs (**EXEC CICS RUN TRANSID**) and the parent passes data to the child in a **PUT** container. The child runs as a separate task in the same region as the parent (see Figure 7-3).

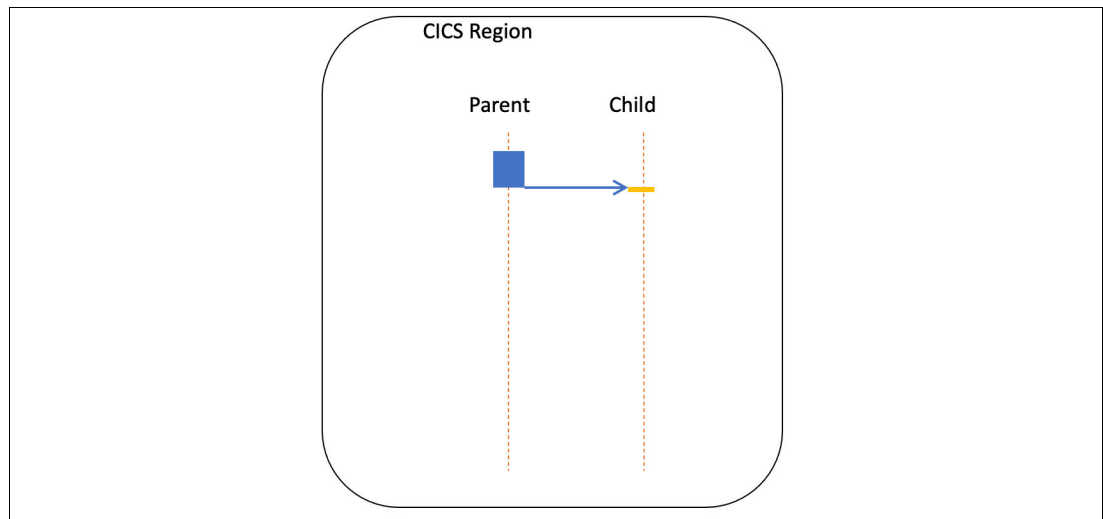


Figure 7-3 Parent-child 1

The parent can continue processing independently of its child it is not blocked by the child, which happens if the parent is linked to the child (see Figure 7-4).

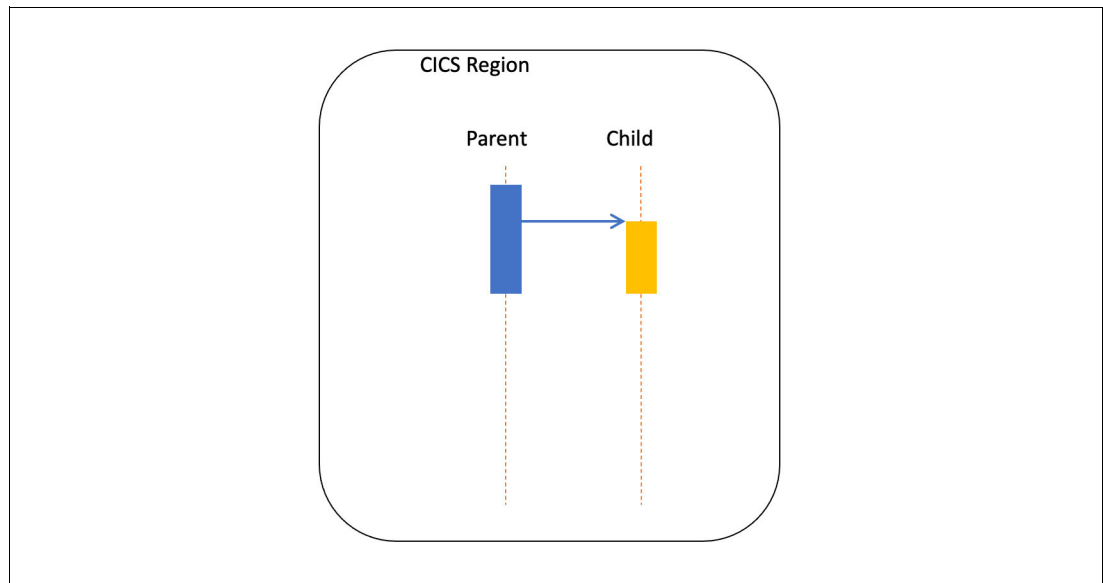


Figure 7-4 Parent-child 2

When the parent is ready for the result from the child, it runs **FETCH CHILD** and waits. It does not have to continuously poll for the result. The child can issue the CICS API and link to a remote region or start a web service (see Figure 7-5 on page 59).

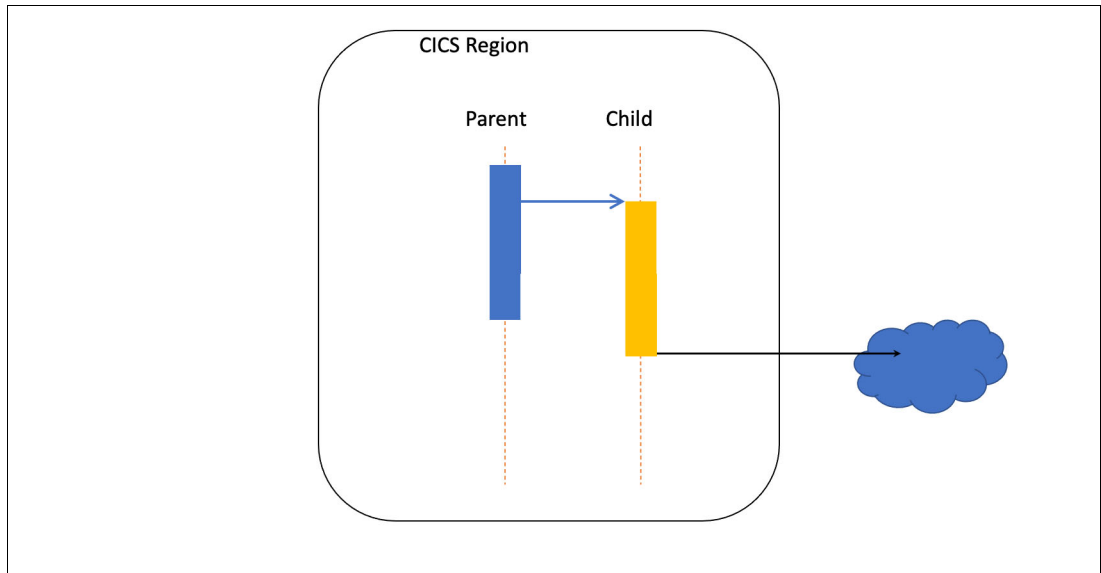


Figure 7-5 Parent-child 3

When the children task completes, the parent resumes its task. The child passes data back to its parent in a container (see Figure 7-6).

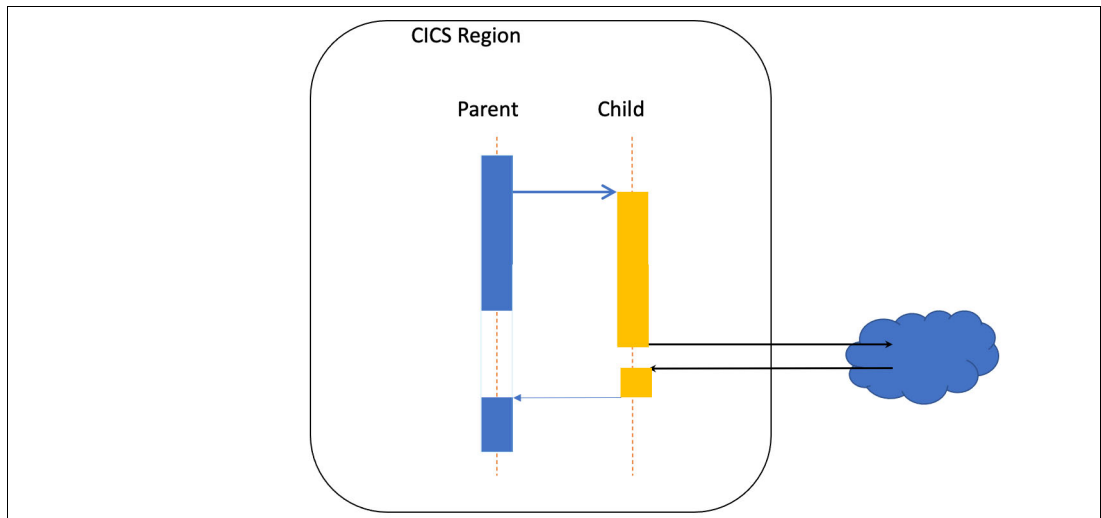


Figure 7-6 Parent-child 4

In summary, the new API provides easy to implement asynchronous programming from your application. You do not need to worry about how it works, just that you can split off independent units of work from the main program and still track their progress.

7.2 Event processing

CICS applications have been the bedrock of mission-critical applications for decades. But for many reasons, the business logic in some legacy programs is locked into the code and can be difficult to enhance or extend.

Event processing is a CICS feature that can detect and respond to events occurring within an application. The ability to detect events and emit them for consumption can provide insight into the application's performance and open many new opportunities.

7.2.1 Event processing in CICS

By using event processing, certain points of interest are registered as capture points. These capture points are based on a subset of existing **CICS API – EXEC CICS** commands plus a new Event Processing API that is named **EXEC CICS SIGNAL EVENT**.

Figure 7-7 shows an overview of event processing.

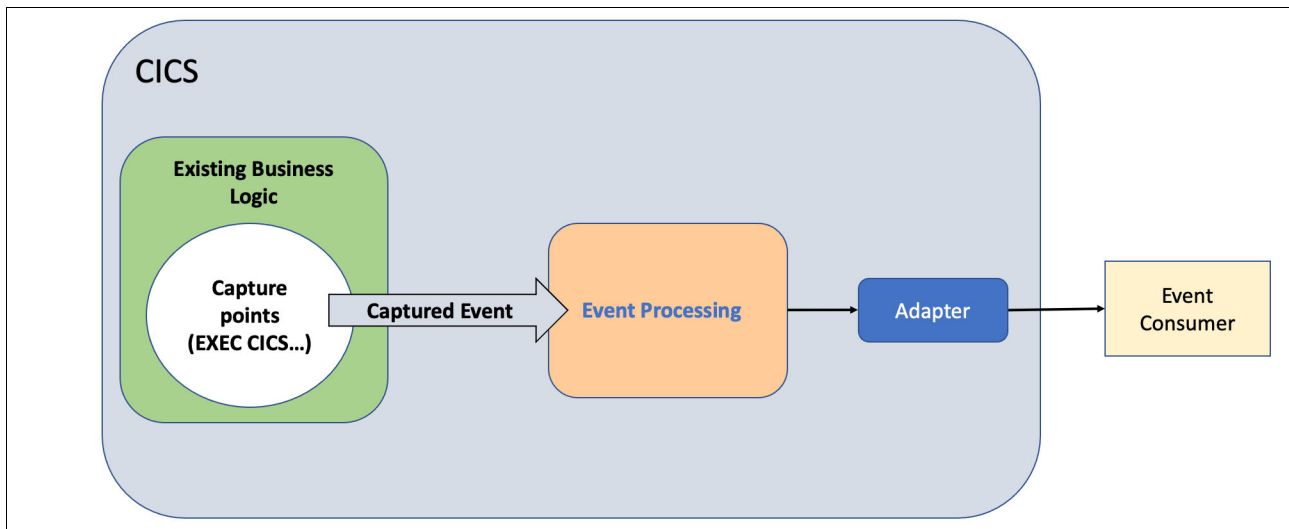


Figure 7-7 Basic event processing flow

By coding a **SIGNAL EVENT** in existing applications, events can be picked up by event processing and sent to an external consumer for evaluation. But, in cases where it is not possible to include the new **SIGNAL EVENT** API into the program (for example, where the COBOL source is lost), it is possible for CICS to generate a business event without amending the application.

By using IBM CICS Explorer®, an event binding is created to define capture specification and link them to an event specification (see Figure 7-8 on page 61).

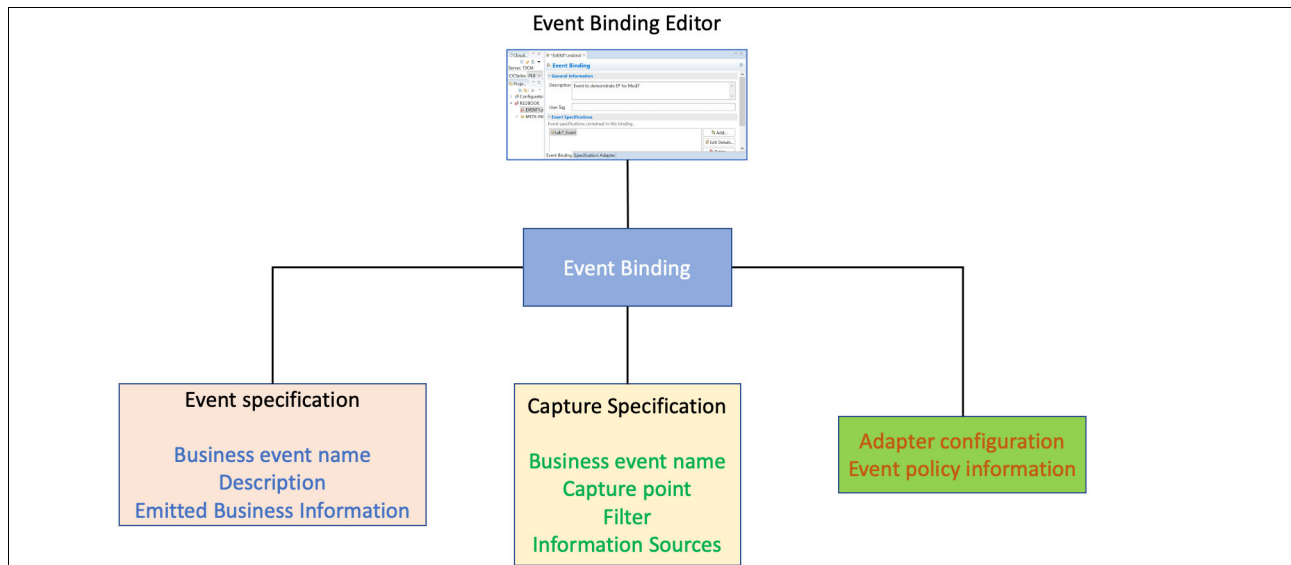


Figure 7-8 Event binding

The event specification defines the business event and what application data is passed to the consumer. The capture specification-defined capture points are **EXEC CICS** commands that can be used to trigger an event. These capture points can be filtered by command options and source information, such as the name of the calling program. Adapters are programs that format and then emit events from a CICS system.

At run time, if the user program issues an **EXEC CICS** command that matches the capture point filter criteria that is defined by the event binding, then event data is passed to the relevant adapter for formatting and then routed to the appropriate external event consumer. One such consumer is IBM Operational Decision Manager (see Figure 7-9).

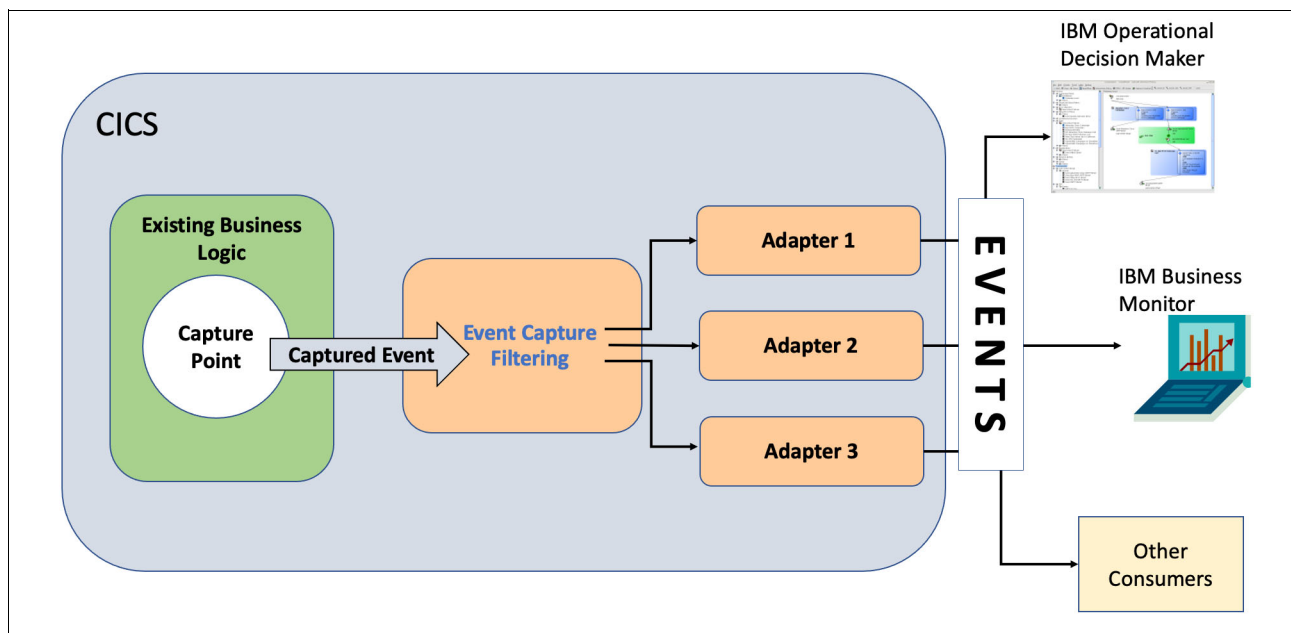


Figure 7-9 Runtime flow

7.2.2 Event processing example

A betting company is interested in bets that are placed over \$10,000. The program that manages bets is BETTING, and the point in which we are interested is when BETTING links to program PLACEBET (see Figure 7-10).

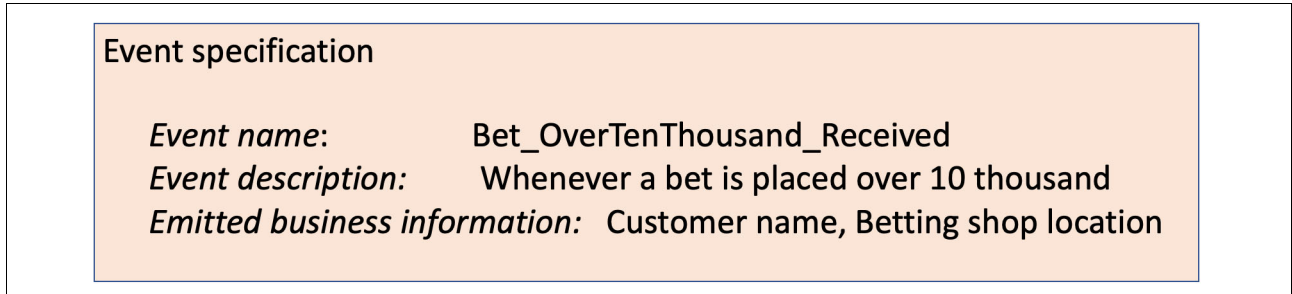


Figure 7-10 Event specification

By using the Event Binding Editor, an event specification is created that names the event and outlines the data that is passed to the event consumer. In this case, the data that is passed is the customer name and betting shop location (see Figure 7-11).

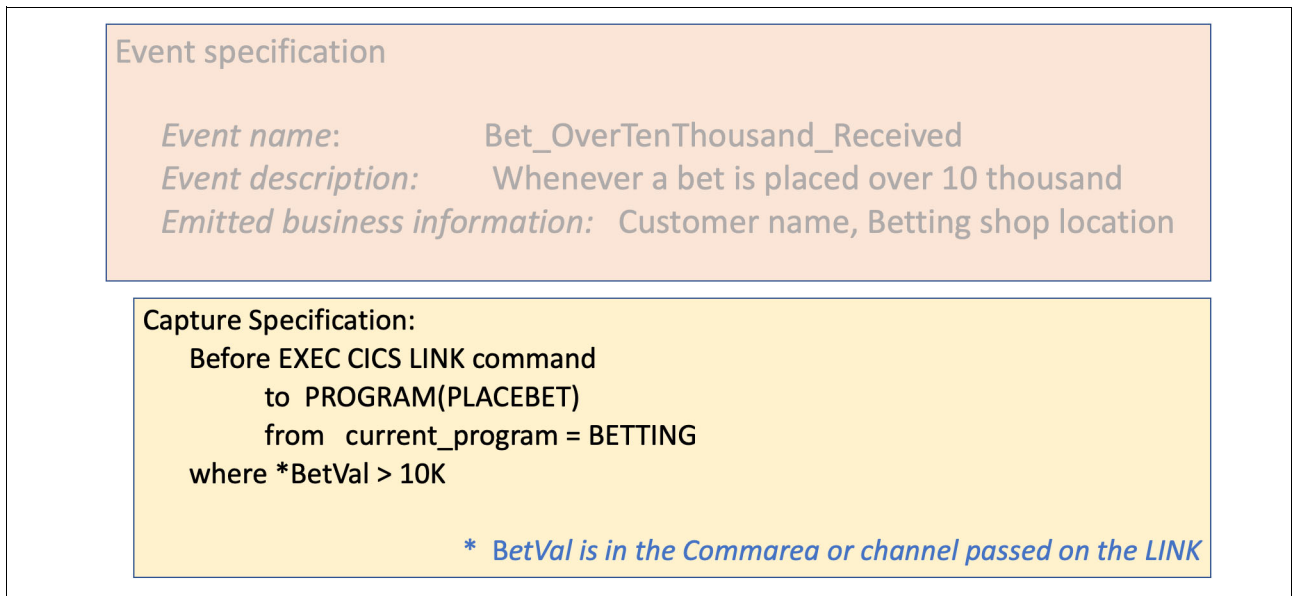


Figure 7-11 Event capture specification

The capture specification specifies the capture point as an **EXEC CICS LINK PROGRAM** to **PLACEBET** in **BETTING** (see Figure 7-12).

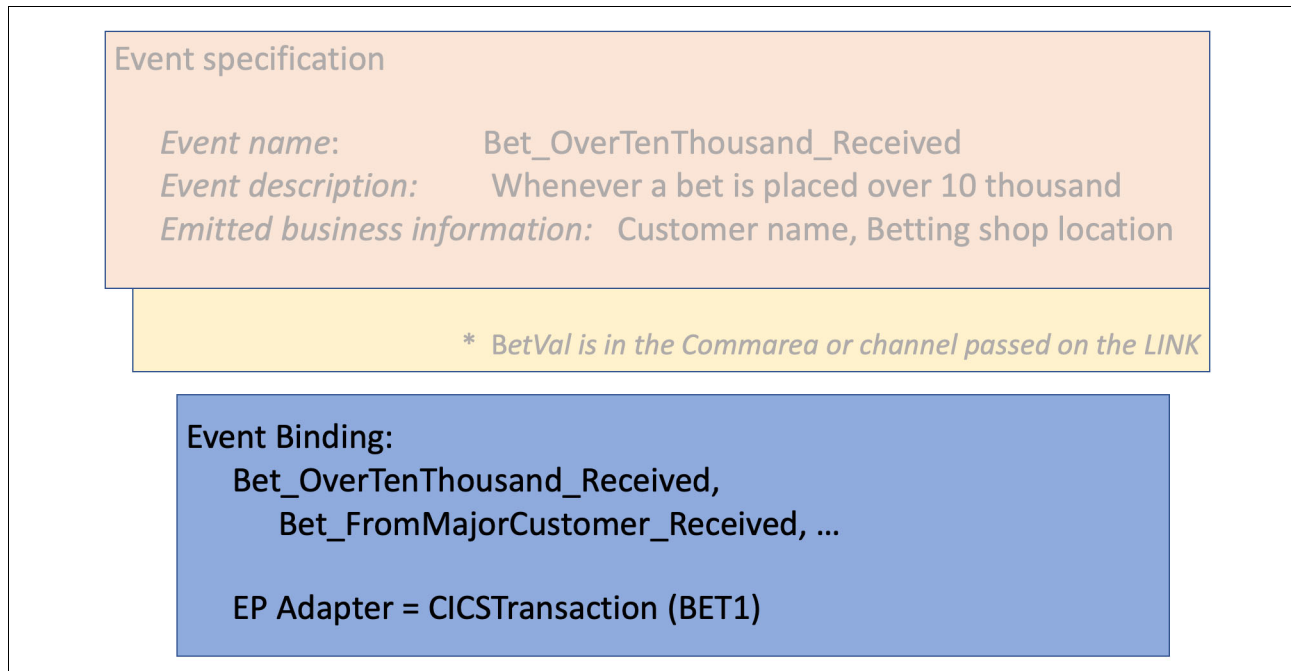


Figure 7-12 Event binding

Finally, the adapter starts a new CICS transaction that is named **BET1**. The exact location of the **LINK** in the program is not specified, and probably is not even known.

In summary, By using event processing, events are defined and controlled independently of the business logic, which extends a business application without modification. This setup provides the business with real-time metrics about the performance of the application, which might unlock the secrets of legacy applications.

7.3 Link to WebSphere Liberty

Chapter 6, “Modernizing applications with Java” on page 47 explained how to extend applications by using Java. A further enhancement to Java support is a Link to WebSphere Liberty feature that was introduced in CICS TS V5.3. The Link to WebSphere Liberty feature allows a non-Java CICS program to start a Java Platform, Enterprise Edition application in a WebSphere Liberty Java virtual machine (JVM) server in CICS.

A Java method is identified as a *target* for **EXEC CICS LINK** (or **START**) in a non Java program. This method acts as an entry point into the Java application that is running in the WebSphere Liberty JVM server (see Figure 7-13).

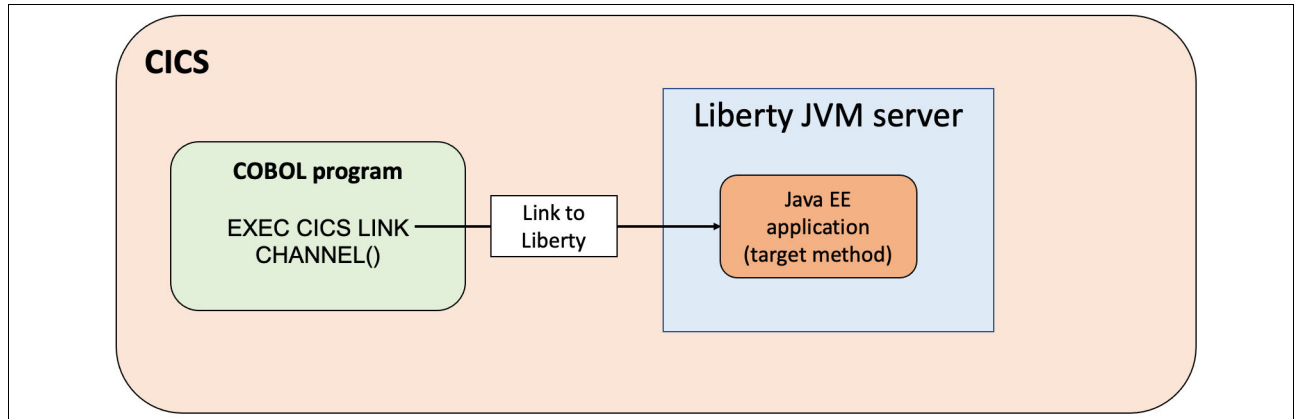


Figure 7-13 Relationship between applications

The following section describes how to set up and enable a non Java program to start a Java application that is named CUSTGET. We illustrate this task by using CICS Explorer, but other tools such as Gradle are available.

Prepare the Java EE application by creating a Dynamic Web project by using CICS Explorer. Add the annotation class to the project class path (see Figure 7-14).

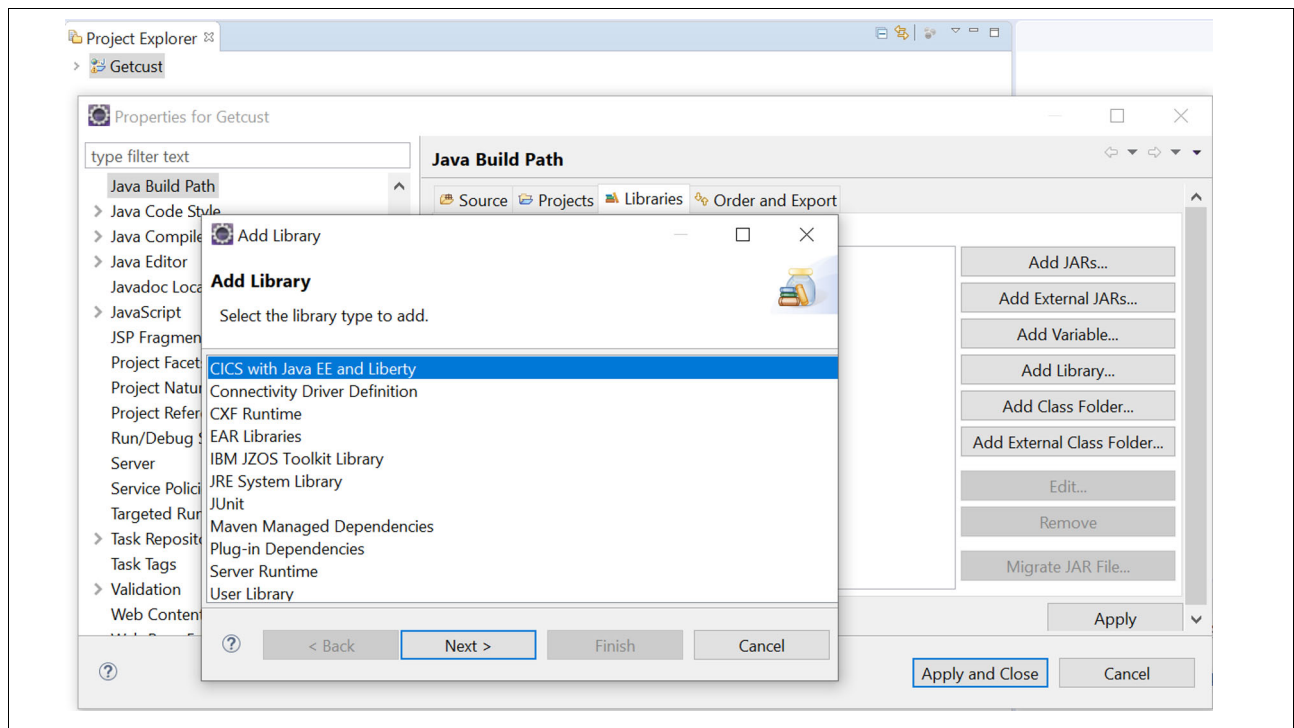


Figure 7-14 Adding a library

In a suitable class, add the method that acts as the target method for the non Java program to use. Then, create the method, including the @CICSProgram annotation, giving it a parameter of PROGRAM. In this case, the program name is CUSTGET (Figure 7-15).

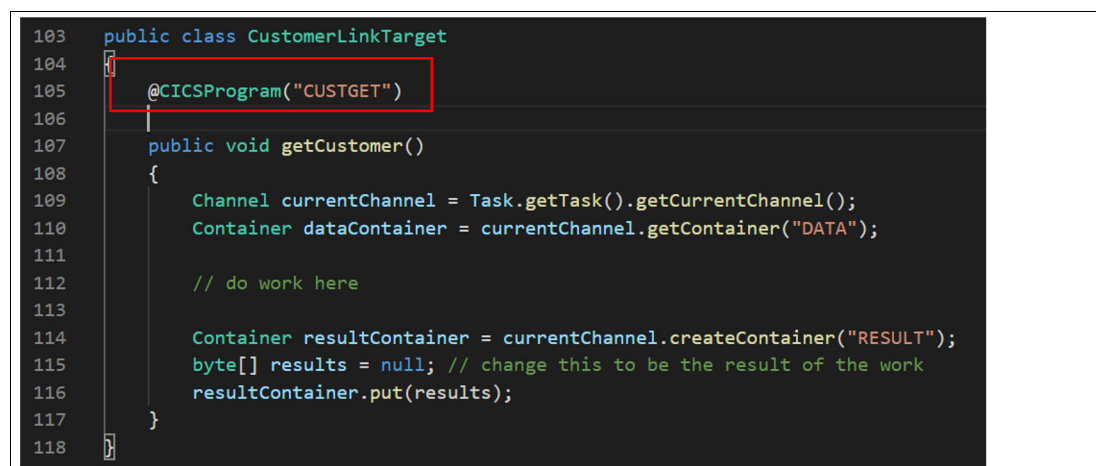


Figure 7-15 @CICSProgram annotation

For @CICSProgram to work with the compiler, **Annotation Processing** must be Enabled for the Web Project so that the compiler can handle the @GETProgram syntax (see Figure 7-16).

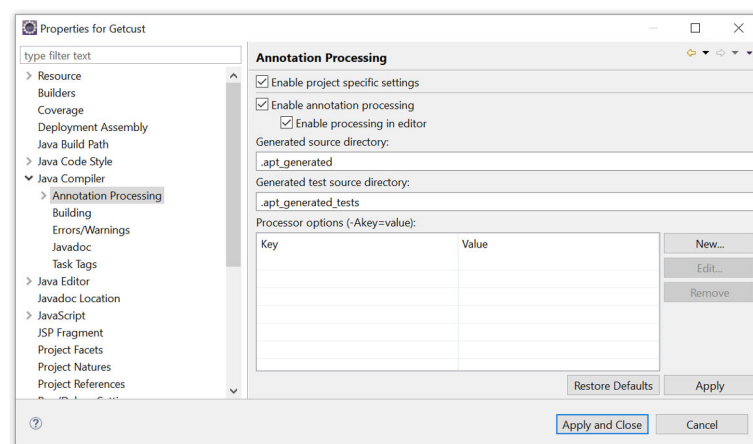


Figure 7-16 Annotation Processing

Code your Java application by using the Target method as the entry point. Now, build the application and export it to the z/OS platform as a WAR file.

During the Java compile, the annotation processor validates the contents and location of the @GETProgram annotation, and generates the code and artifacts that are required by CICS to start the application. The target method becomes available as a linkable program when installed into the WebSphere Liberty JVM.

The application and its artifacts are exported to the z/OS platform as a WAR file, and the application is deployed into a WebSphere Liberty JVM server. When the application is deployed, either as part of a CICS Bundle, or directly from server.xml or from a file by using an <application> element, CICS creates a CICS Program resource dynamically. The name of the RDO program resource is the program name on the @GETProgram annotation in the Java method.

Non-Java programs can use the CICS API to start the Java application by using the target method (Figure 7-17).

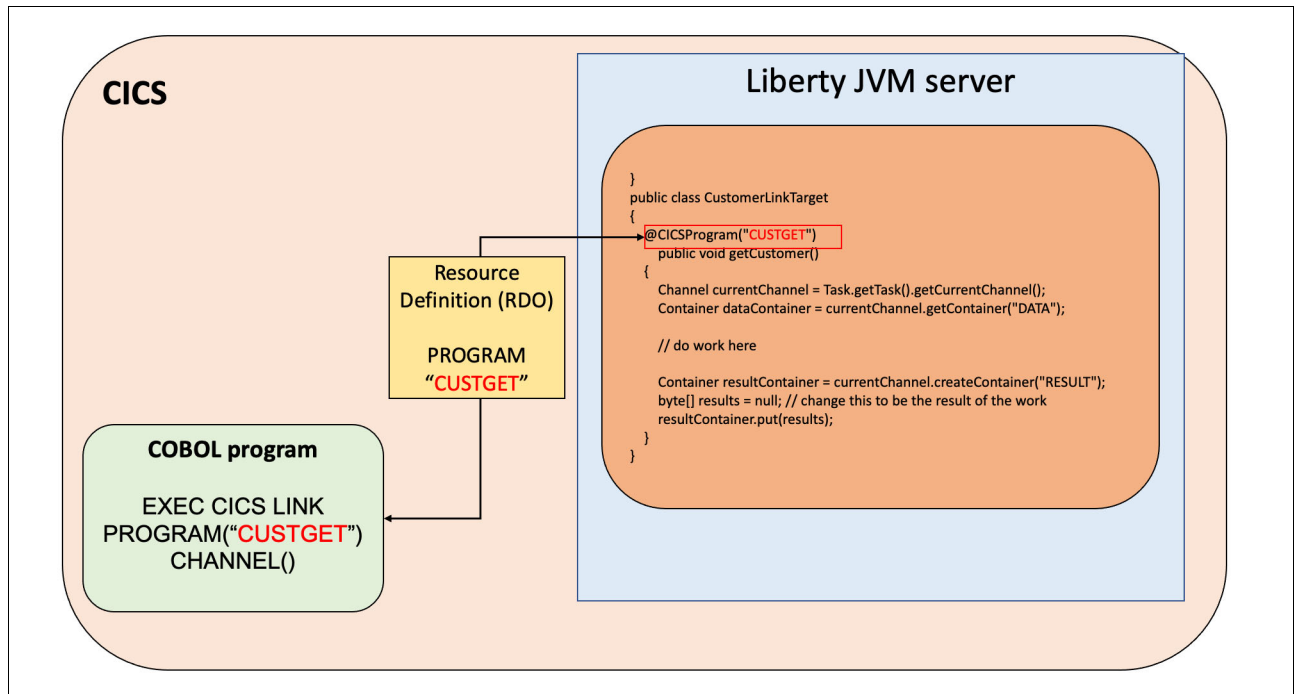


Figure 7-17 Starting the Java application



DevOps and IBM CICS

This chapter looks at how DevOps and CICS applications can form an enterprise-based end-to-end strategy.

8.1 Introduction to DevOps

DevOps is about bringing the principles of lean and agile together as part of a continuous improvement process. It is an end-to-end process that is iterative and never fully completed. It is about removing waste from the system and continuing to improve.

Combining agile and DevOps practices makes it easier to build open and hybrid applications. DevOps is the union of people, processes, and tools to enable continuous integration (CI) and continuous delivery (CD) (CI/CD). However, even with the best tools, DevOps is only a buzzword if you do not have the right culture. The primary characteristic of DevOps culture is increased collaboration between the roles of development and operations.

8.2 DevOps on IBM Z with CICS applications

What does DevOps mean for IBM Z and CICS applications?

The platform or environment that you are building for is irrelevant for DevOps, and you do not get the improvement that you need by having multiple different ways of working with artificial boundaries and silos in between. You need an enterprise-based end-to-end strategy.

A key point is that there is no technical reason why traditional z/OS development, including CICS application development and delivery, cannot be part of an open distributed pipeline. You can transform your z/OS and CICS environments to a true, modern DevOps practice by bridging the gap between mainframe and distributed approaches by letting developers work in the mainframe and distributed worlds in the same way.

IBM wants to provide this pipeline as the automation framework for the DevOps transformation. The IBM approach to a cross-platform delivery pipeline solution is to integrate various open source and third-party tools with enterprise tools so that you can choose a pipeline that is correct for your organization that is based on your business needs.

Some of the open source and third-party integrations IBM offers include Git, Jira, Jenkins, and SonarQube (see Figure 8-1 on page 69).

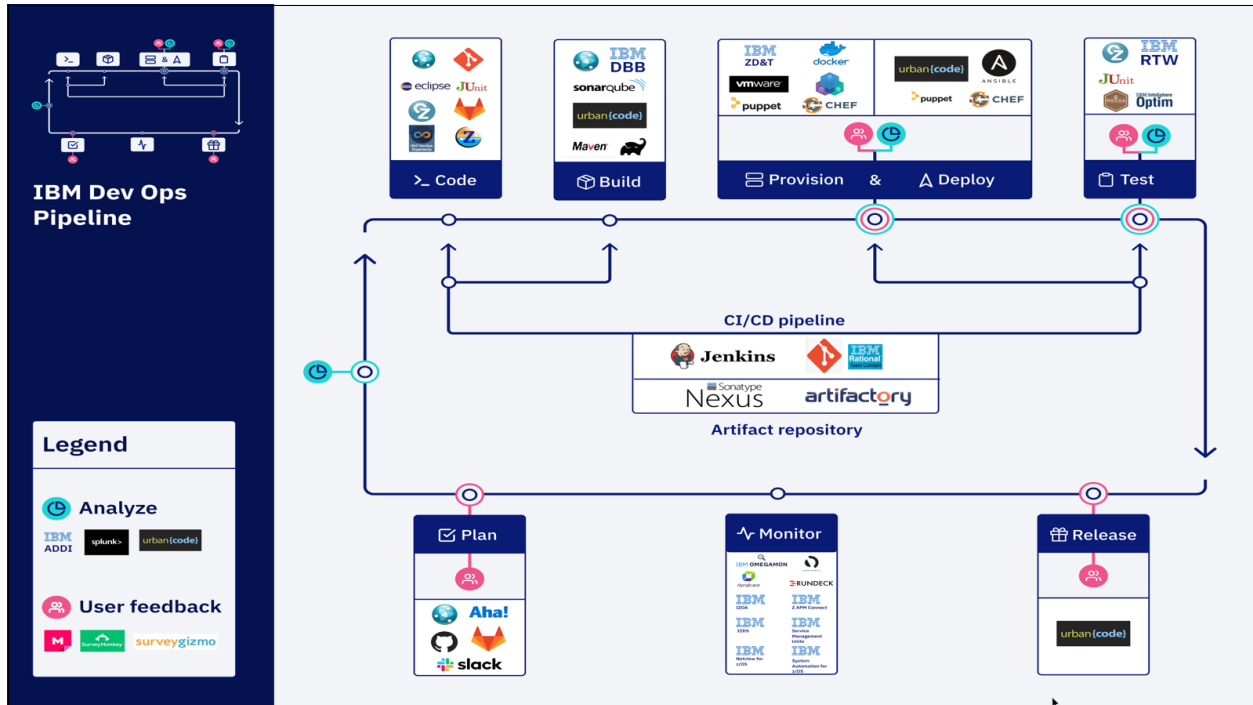


Figure 8-1 IBM DevOps pipeline

All parts of the pipeline are important. You cannot pick one area and say that you are using DevOps. You must consider the entire pipeline and maybe focus on different areas at different times, but the entire pipeline is key to setting the scope for each part.

However, it is equally true that you cannot do everything at once, so starting with deployment might be what many customers do because it lets them set up new environments, and it makes testing easier.

The pipeline is not in one direction: It is a circle with continual analysis and feedback at various points along the way.

The pipeline includes many solution options in many boxes that overlap with each other, which demonstrates that you have choices in how you determine the DevOps pipeline that is correct for you.

There are a few key aspects that are common to all DevOps pipelines. The following sections describe these aspects and how they specifically relate to CICS applications in more detail.

8.2.1 The integrated development environment and debugging

There are many integrated development environments (IDEs) that can be used for mainframe application development, including the IBM Z Open Editor and Zowe. All of them are based on developer choice. Along with these leading-edge IDEs, productive application debugging technologies for CICS, such as IBM Debug for z/OS, are also available to provide a “round-trip” development experience.

8.2.2 Source code management

Modern IDEs integrate with Git. Outside of mainframe development, Git is the de facto standard for source control in all areas of application development.

These integration capabilities enable developers to get the most out of modern source code management (SCM) like Git, and help standardize true parallel development across the enterprise.

Ideally, there should be one SCM so that the resources can be shared. It is a single source of truth and a single place to integrate audit reports and automation. You could use RTC or Git for your SCM. Customers usually choose only one SCM, but CICS applications regardless of language can fully participate in Git ecosystems.

8.2.3 Build solutions

There are many different build solutions, which generally are based on language, for example, Maven or Gradle for CICS Java applications, or IBM Dependency Based Build for traditional CICS applications, for example, in COBOL or PL/I.

IBM Dependency Based Build, which includes a build toolkit, allows you to build your CICS applications and analyze the dependences that are required for building. IBM Dependency Based Build also includes Groovy for automation, which helps bring your CICS applications and mainframe processes into the world of open source so that you can connect your mainframe processes to CI/DC to open source tools like Jenkins.

The CICS team introduced powerful new support for CICS Java developers in CICS Transaction Server V5.6, which includes support for Maven and Gradle as two frameworks for building Java applications and the bundles that they are deployed through, and adding libraries to Maven Central to make those DevOps CI/CD pipeline build chains much easier to create and manage.

With this new release of CICS, the development team developed a way that Java developers could run their CICS applications directly in their IDEs without having to deploy them. Importantly, these applications should still be able to link to programs running in CICS. Crucially, after the developers finished developing them, these applications could be deployed unchanged to real CICS regions.

To address this task, CICS introduced an extension to the existing JCICS application programming interface (API) that is named JCICSX, which is a subset of the JCICS API in terms of functioning. With JCICSX, Java developers can now run their CICS Java applications locally on their laptops while they are still developing them.

8.2.4 Pipeline automation

One of the open source integrations is Jenkins, which is a CI/CD coordinator that provides a single point for audit and integrations.

Jenkins is used as common tool to embrace DevOps principles by enabling the management of CI/CD workflows that can encompass CICS applications. This Jenkins master might be the same one that is running your distributed workloads.

8.2.5 Unit testing

Automated testing is critical to making the DevOps pipeline possible because without automated testing you speed up the deployment but still must wait on the overall process.

Continuous testing is a key part of DevOps, and it means testing earlier in the lifecycle, which results in reduced costs, shortened testing cycles, and continuous feedback on quality. This process of shift-left testing stresses integrating development and testing activities to ensure that quality is built in as early in the lifecycle as possible.

The first step of the testing is done in a clean and automated way and early on in the lifecycle so that you are feeding in tested code into the rest of the lifecycle. This process of shift-left testing ensures that quality is built in as early in the lifecycle as possible.

IBM solutions such as IBM Z Open Unit Test provide advanced and flexible tools for writing and running automated unit tests of batch and CICS programs on IBM Z. Stubbing capabilities for CICS programs are available, meaning developers do not need to deploy to CICS during unit testing, which enables environment independence. Automated data capture and recording for batch and CICS programs enables test scenarios to be integrated into any CI/CD process.

8.2.6 Integration testing with Galasa

Galasa is a test automation framework that is available as an open source project. Galasa originated when the CICS development team was discussing how they did their DevOps pipeline within the CICS organization, and how they approach testing, especially the automated parts of testing in CICS.

With Galasa, you write tests as you want, whether it is by using 3270 scripting or preparing a batch job for run time, or it could be by using web tools, such as Selenium, integrated within the same test, which provides deep integration into IBM Z.

Tests can be run locally on a workstation, but they still connect to the mainframe without having to change the test code.

Using Jenkins, a pipeline can be configured and request that Galasa runs a set of tests. You can specify running a specific test, or you can ask Galasa to run a set of tests based on information about the change set that was delivered.

8.2.7 Deployment

IBM UrbanCode® Deploy is a tool for automating application deployments through your environments from test to production. It is designed to facilitate rapid feedback and CD in agile development while providing the audit trails, versioning, and approvals that are needed in production.

You can use UrbanCode Deploy and the specific CICS TS plug-in to automate the deployment and undeployment of CICS resources. Used with other CICS tools, UrbanCode Deploy can improve workflow efficiency and contribute to a CD environment.

The plug-in includes steps that can automate actions such as the installation of CICS resources, pipeline scanning, the opening and closing of resources, and many more operations.

8.2.8 Analysis

Using discovery tools such as IBM Application Discovery, you can visualize application dependencies, automate the discovery effort with mainframe connectors and SCM integration, identify the most essential test cases and redundant test cases, and gain an understanding of code coverage.

Furthermore, a combination of IBM Application Discovery integrated with other CICS analysis tools such as CICS Interdependency Analyzer allows for 360-degree view of both static applications of CICS application code and a full runtime analysis picture (Figure 8-2).

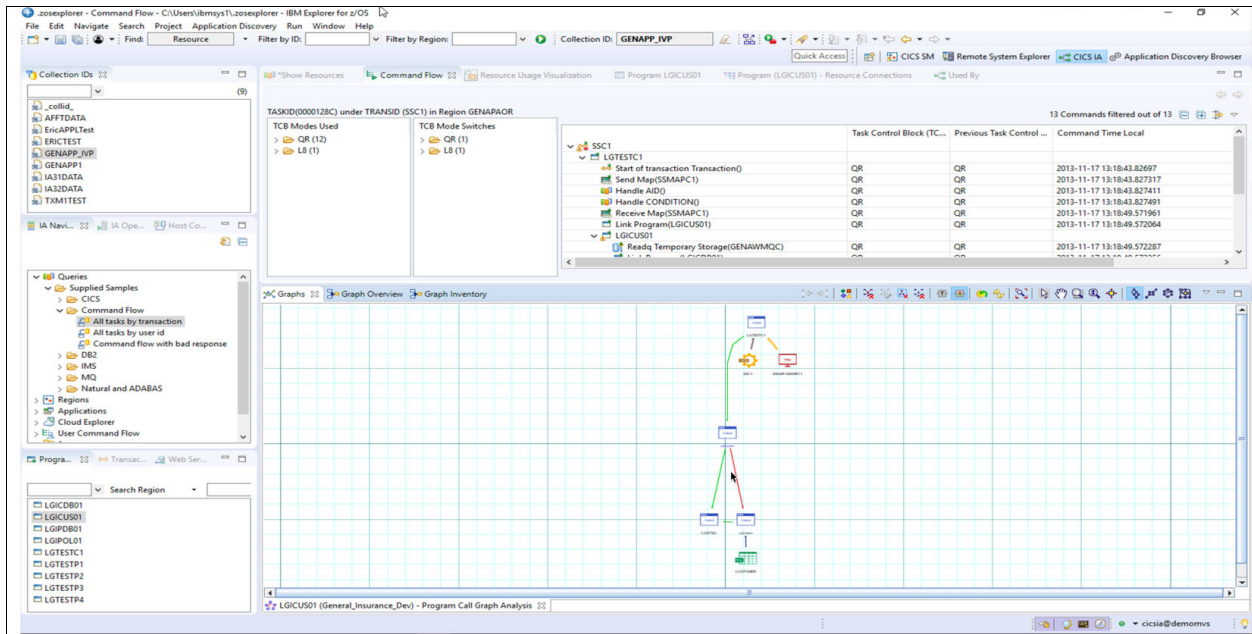


Figure 8-2 Command flow



REDP-5628-00

ISBN 0738459291

Printed in U.S.A.

Get connected

