Redbooks

ibm.com/redbooks

Modernizing Applications with IBM CICS

Russell Bonner

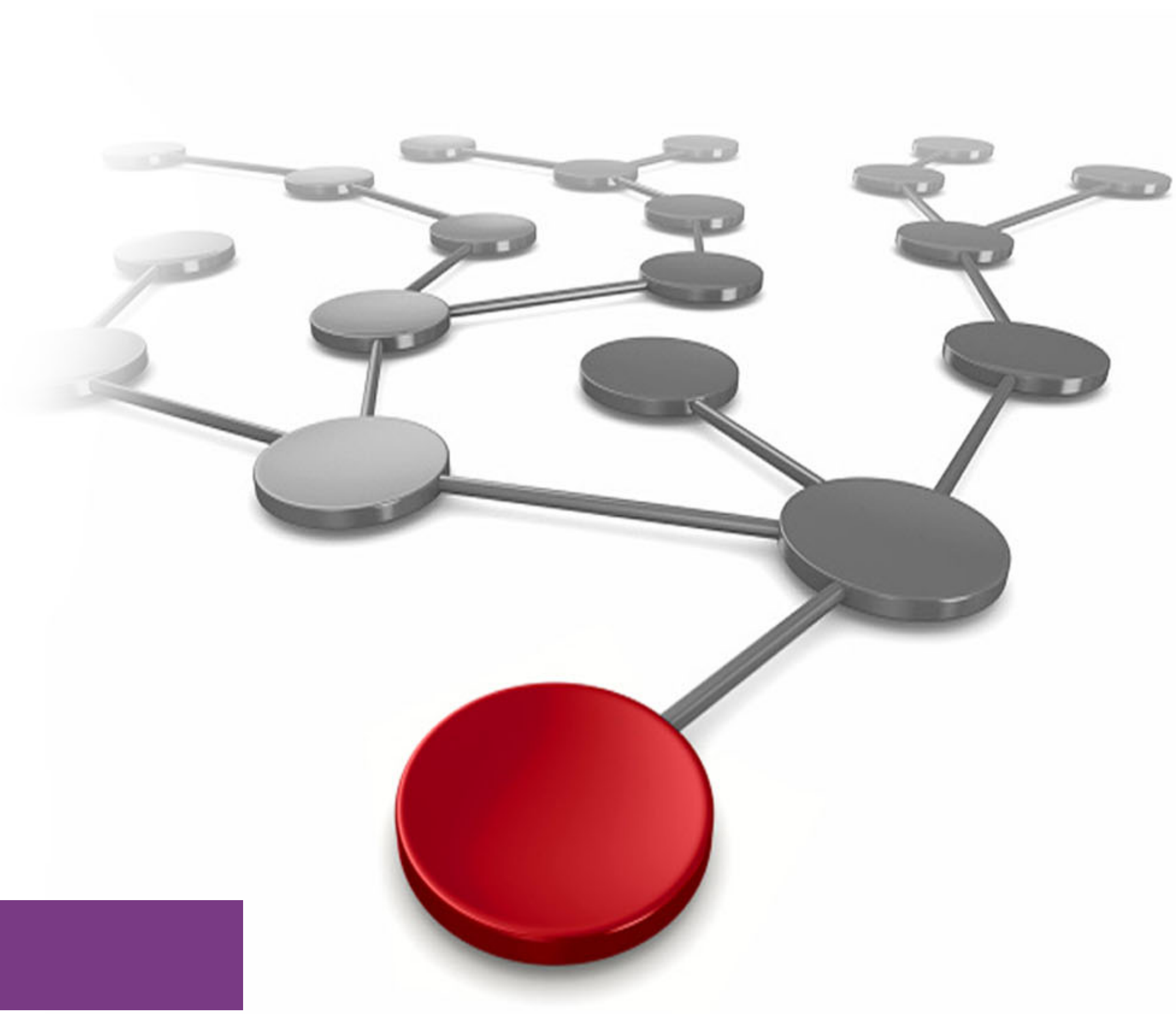
Sophie Green

Ezriel Gross

Jim Harrison

Debra Scharfstein

Will Yates



IBM Z

Redpaper

IBM Redbooks

Modernizing Applications with IBM CICS

December 2020

REDP-5628-00

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

First Edition (December 2020)

Contents

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

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.

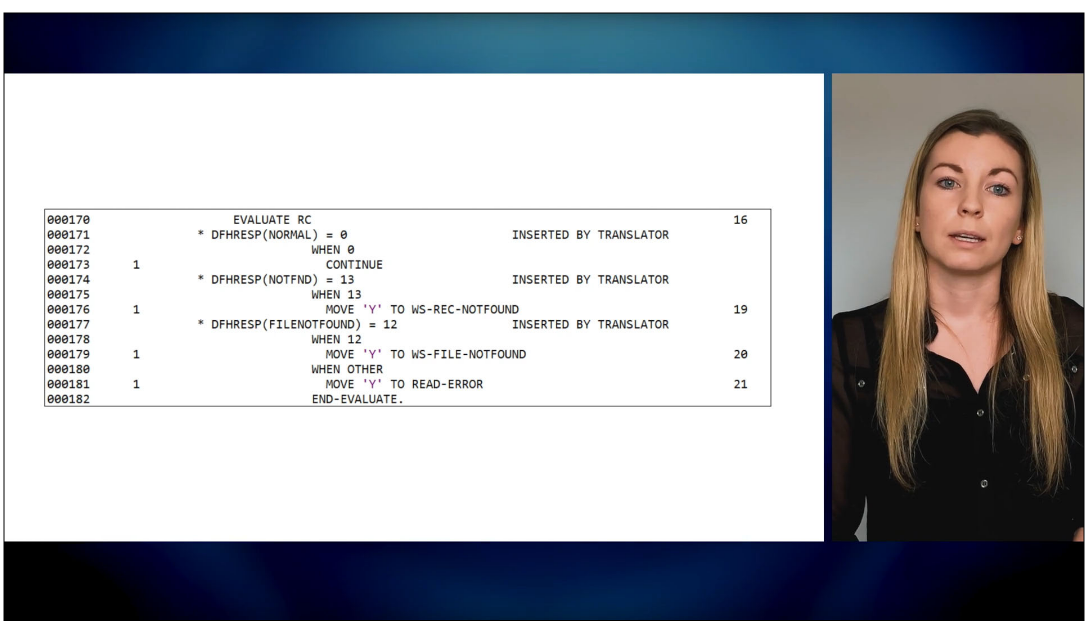


Figure 1 Accompanying education course

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 Srinvasan
- 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

1

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.

2

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 re 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).

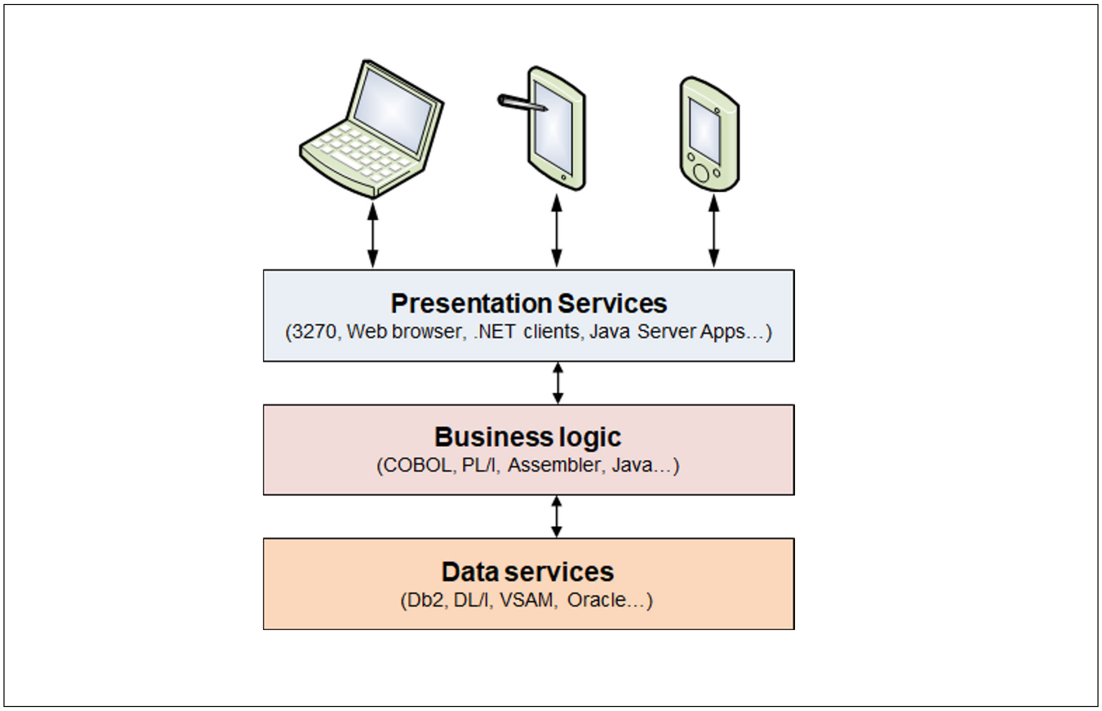


Figure 2-1 Basic application architecture

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).

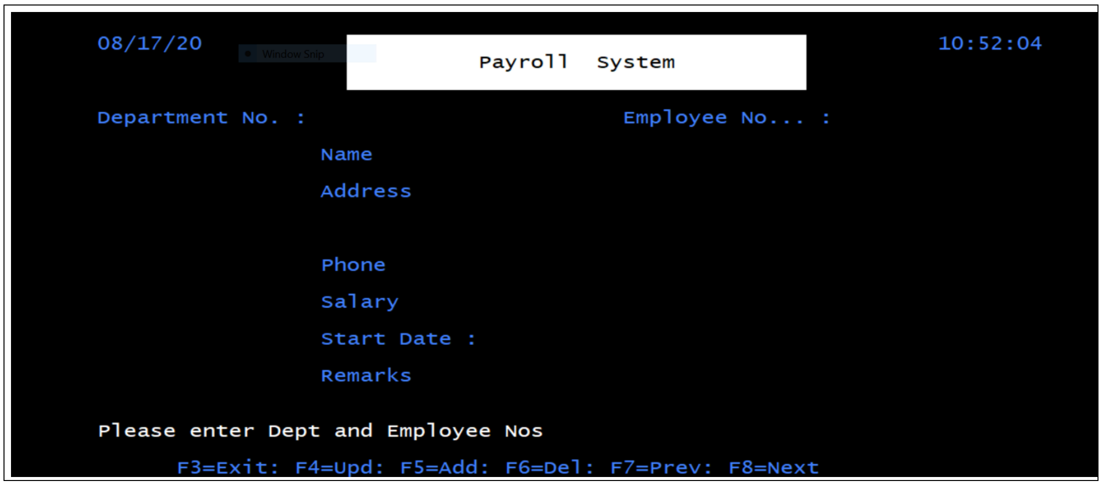


Figure 2-2 Payroll system sample application

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 DFHWBTTA 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).

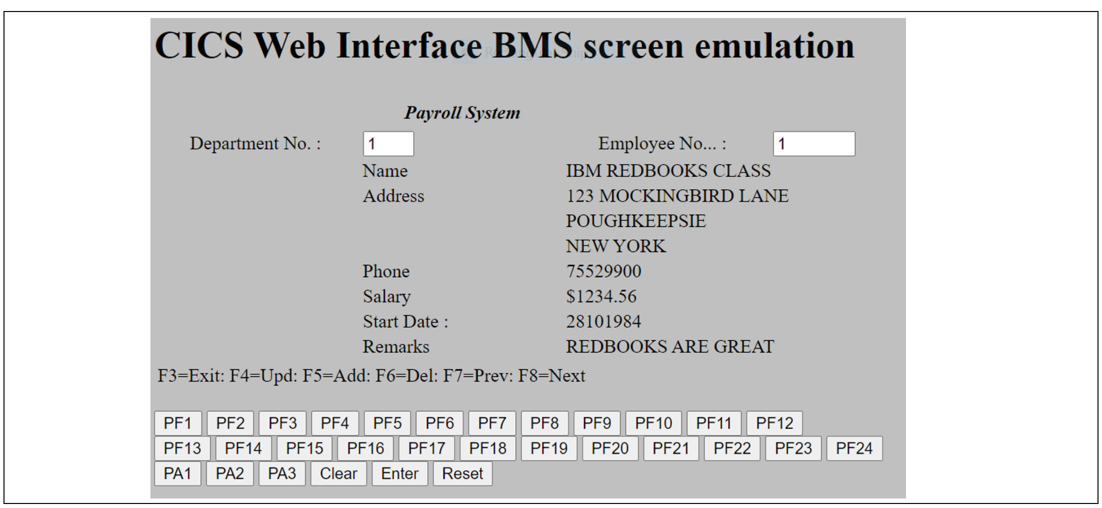


Figure 2-3 CICS web interface

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.

3

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.

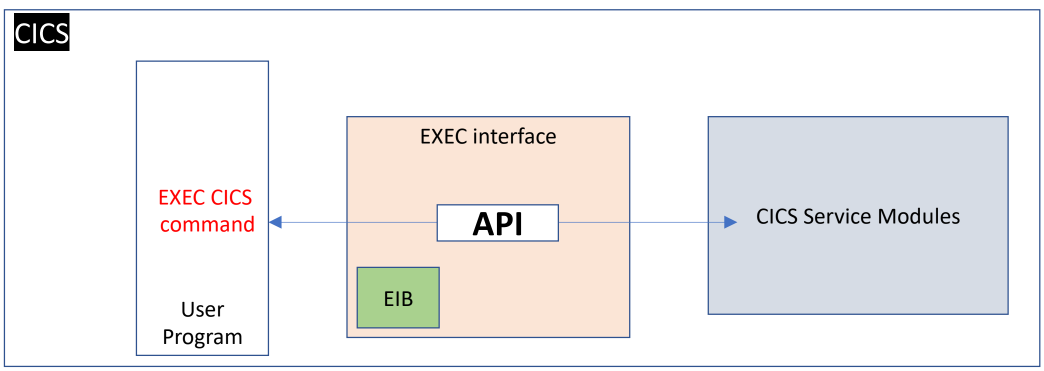


Figure 3-1 CICS application programming interface

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.

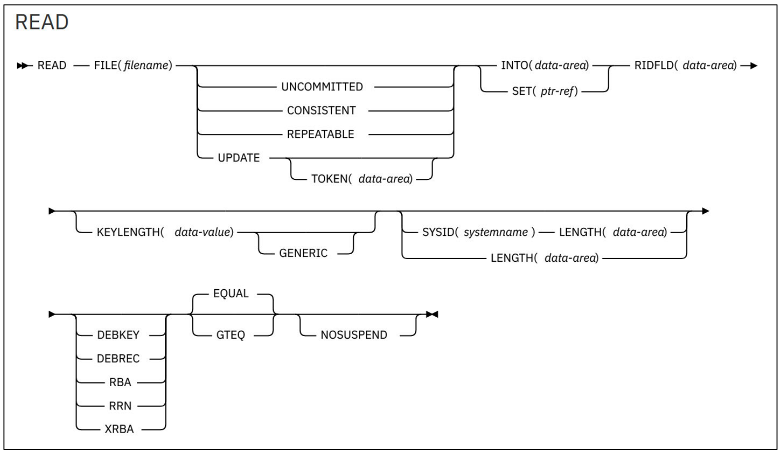


Figure 3-2 Schematic diagram for EXEC CICS READ

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.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. PAYBUS.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 MISC.
    02  ws-key.
        05  ws-department          pic x.
        05  ws-employee-no         pic x(5).
01  payroll-record.
    02  pr-department              pic x.
    02  pr-employee-no             pic x(5).
    02  pr-name                    pic x(20).

PROCEDURE DIVISION.
*
    exec cics read file('PAYROLL')
        ridfld(ws-key)
        into(payroll-record)
    end-exec.
*
    if eibresp not = dfhresp(NORMAL)
       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).

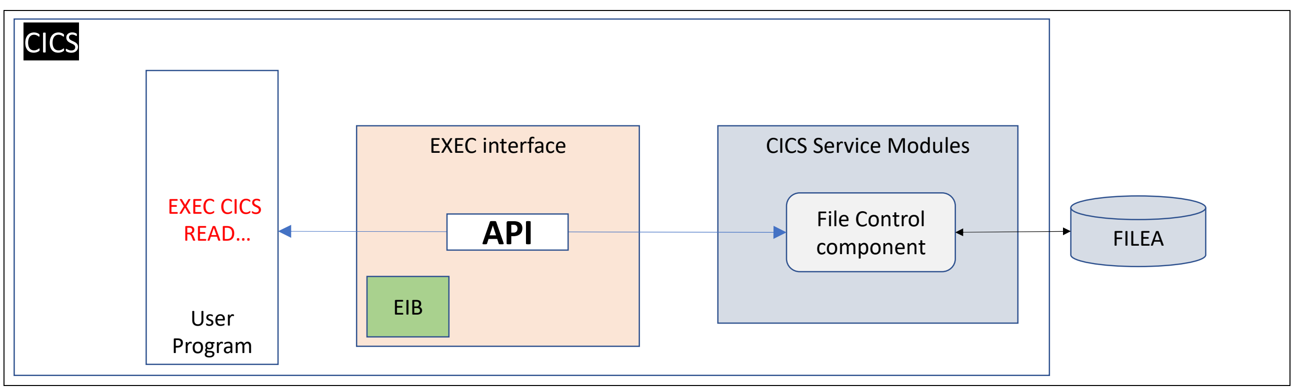


Figure 3-4 API flow

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).

```
*exec cics read file('PAYROLL')
*              ridfld(ws-key)
*              into(payroll-record)
*              nohandle
*end-exec
```

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

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-O`.
- 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).

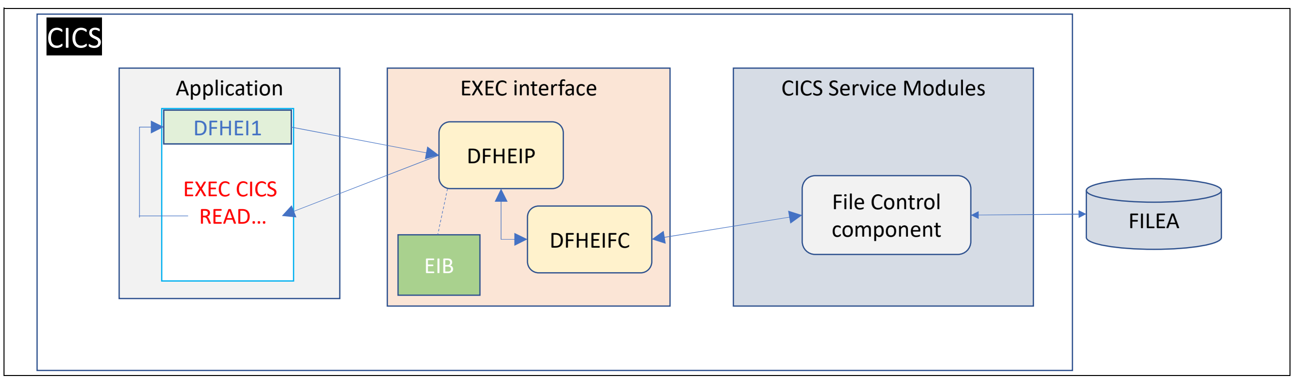


Figure 3-6 Enhanced API flow

- 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 andEIBRESP2`. 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(payroll-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 updated the EIB, but does not handle the condition.

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

4

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).

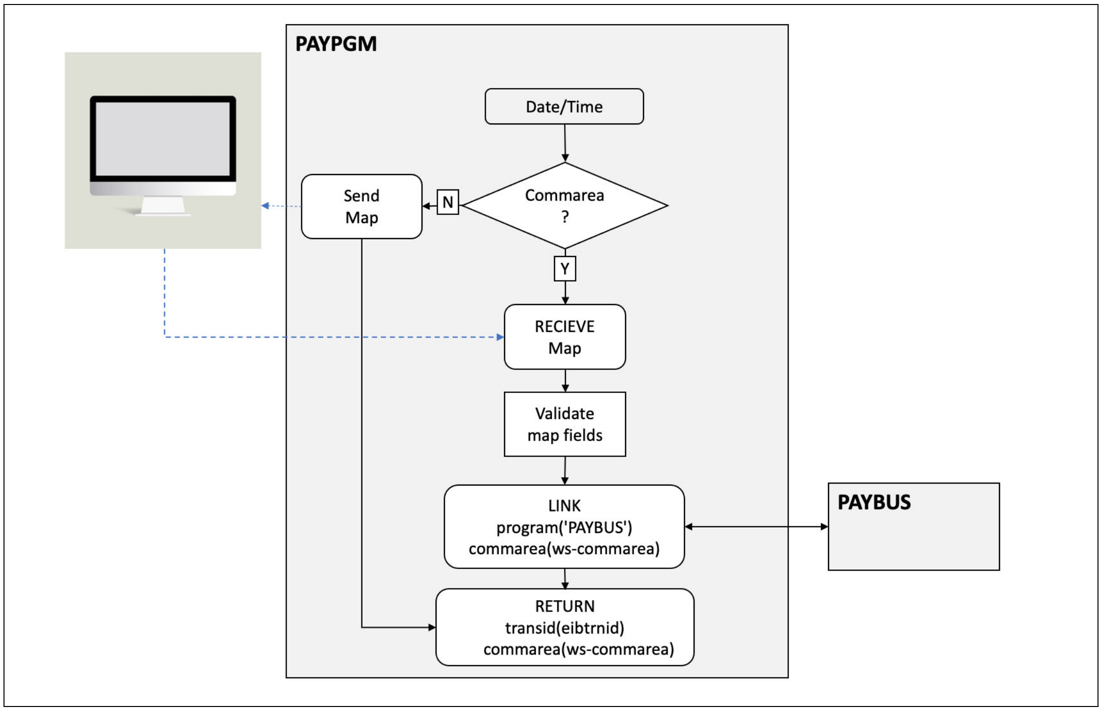


Figure 4-1 PAYPGM program flow

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).

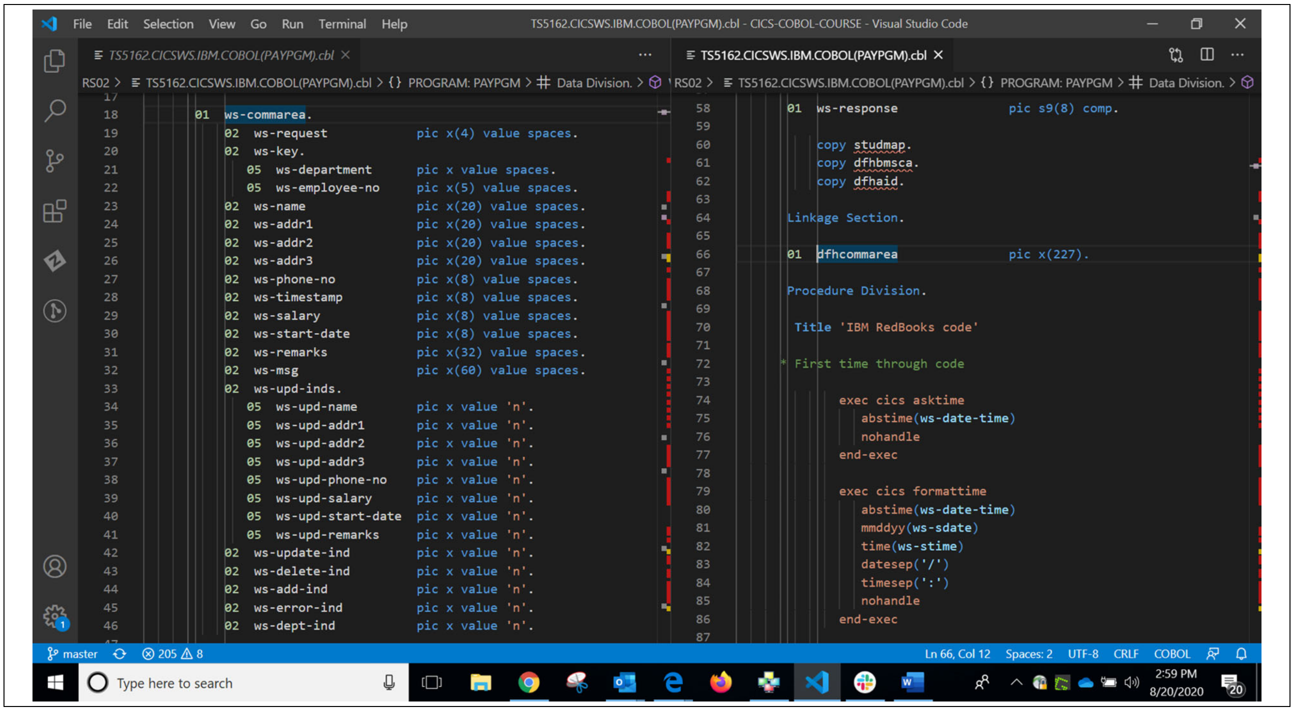


Figure 4-2 COMMAREA definition of PAYPGM

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.

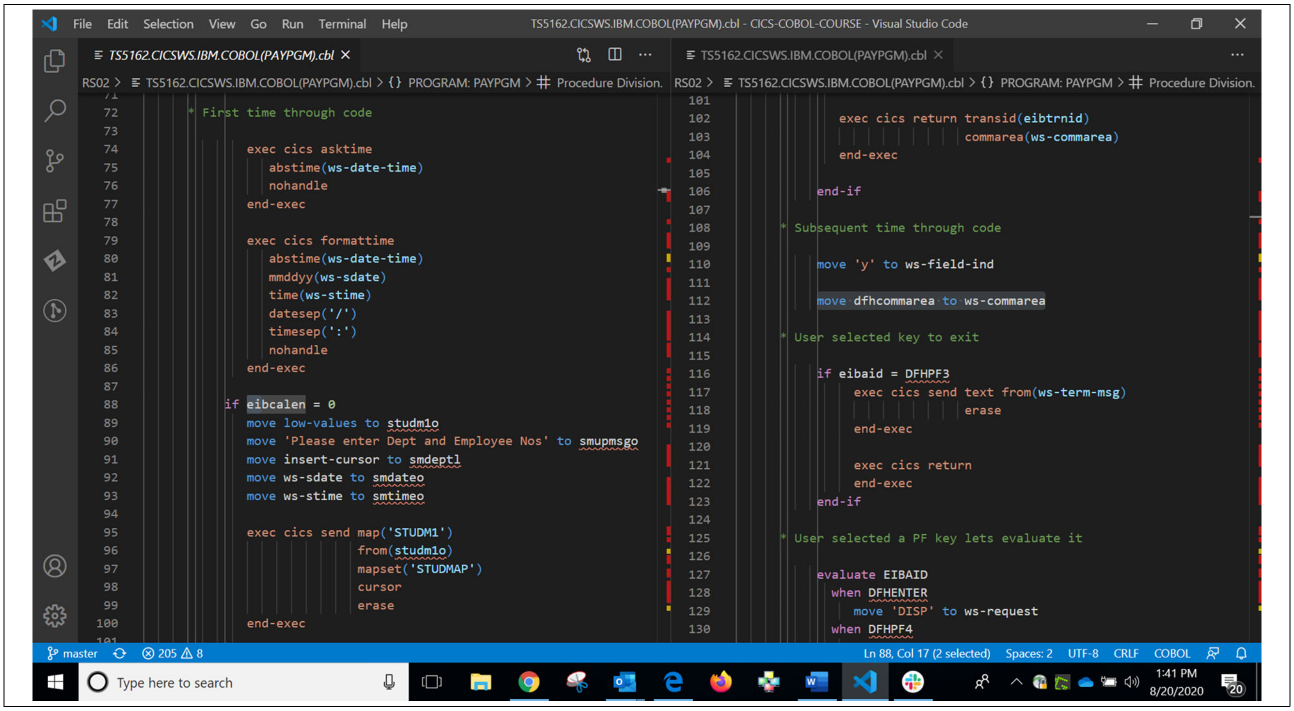


Figure 4-3 First time through code

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 `tranid` 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).

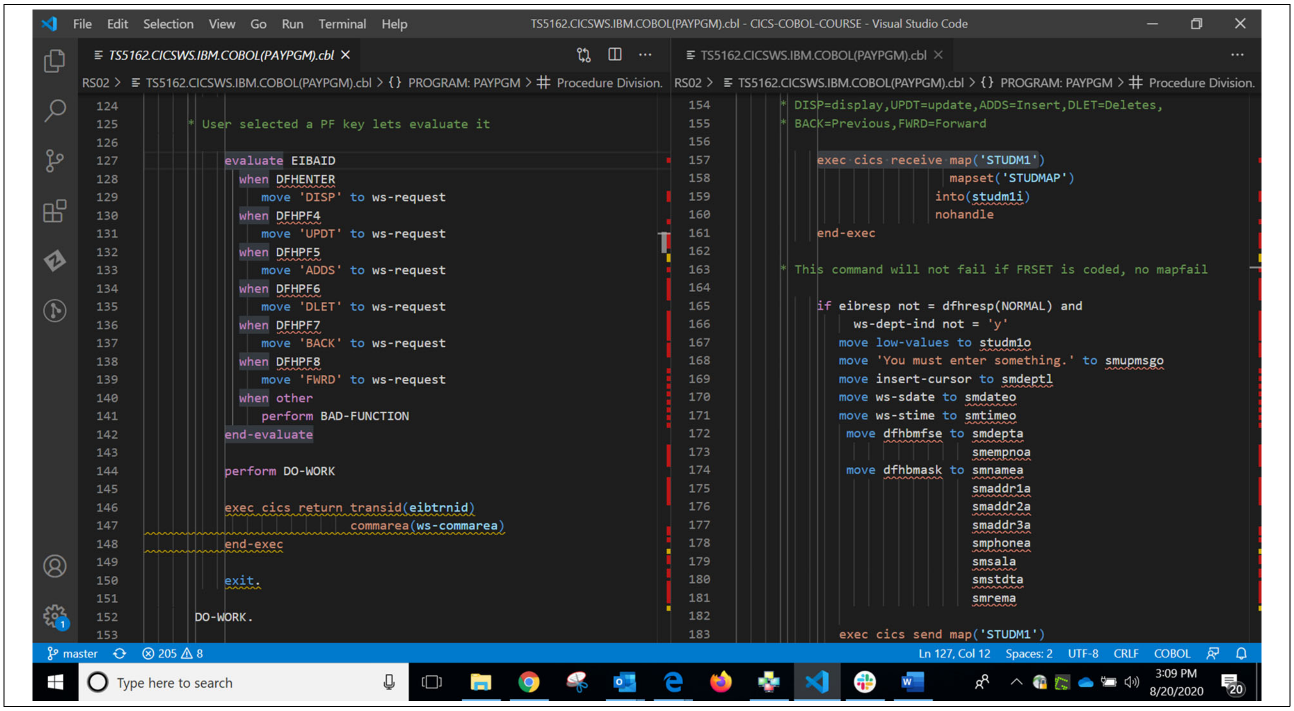


Figure 4-4 Function evaluation

As you can see from the evaluate statement, the key that the user pressed is passed to a CICS program in an EIB field called EIBAID. The program is structured to evaluate the key and set the request based on the key pressed.

Once done, a routine that is called `DO-WORK` is performed, and then you can see a familiar CICS return statement that nominates the successor transaction with the COMMAREA in place.

The `DO-WORK` paragraph starts by getting the data that the user entered on the screen with a **`CICS RECEIVE MAP`** command. In older versions of the code, an error that is called a MAPFAIL would be possible if the user did not enter any data because CICS maps were coded not to return fields that were empty to avoid data transmission of blank fields. Today, the number of bytes of transmission is considered so small that it is more efficient to send all fields, entered or not, than run through the code that is required to test whether the field was entered.

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.

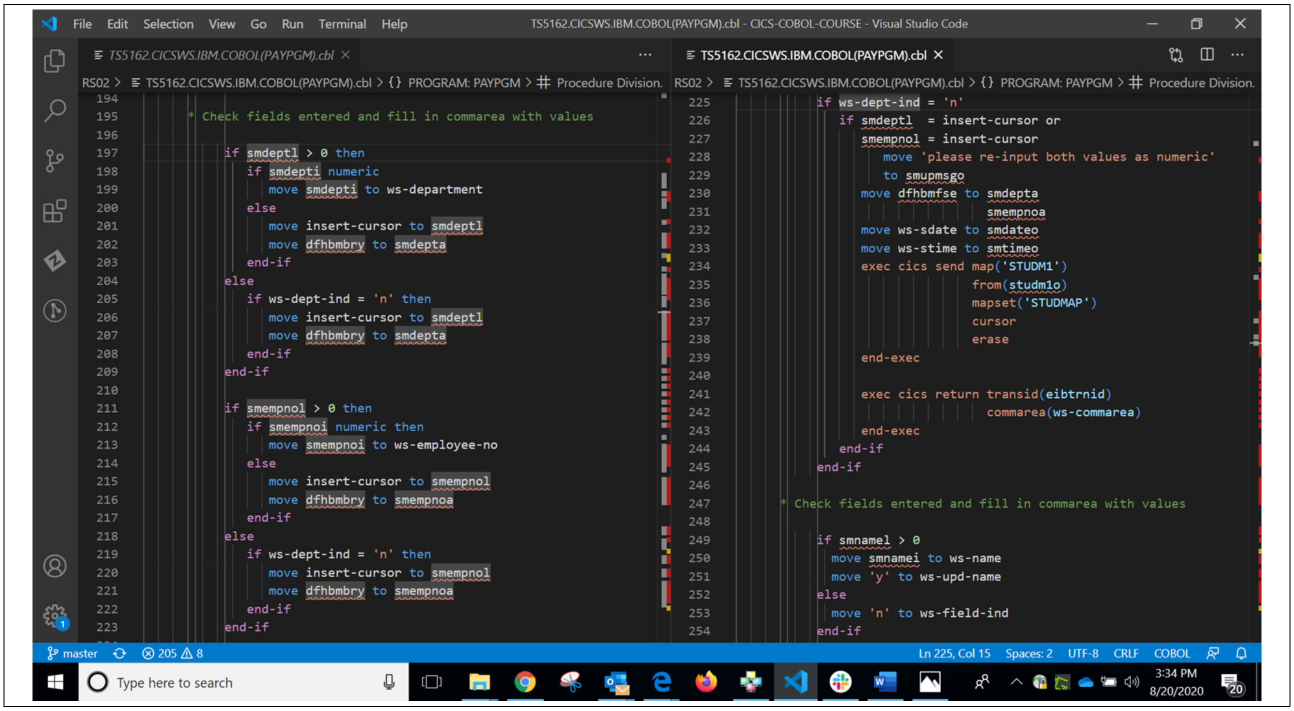


Figure 4-5 Field validation

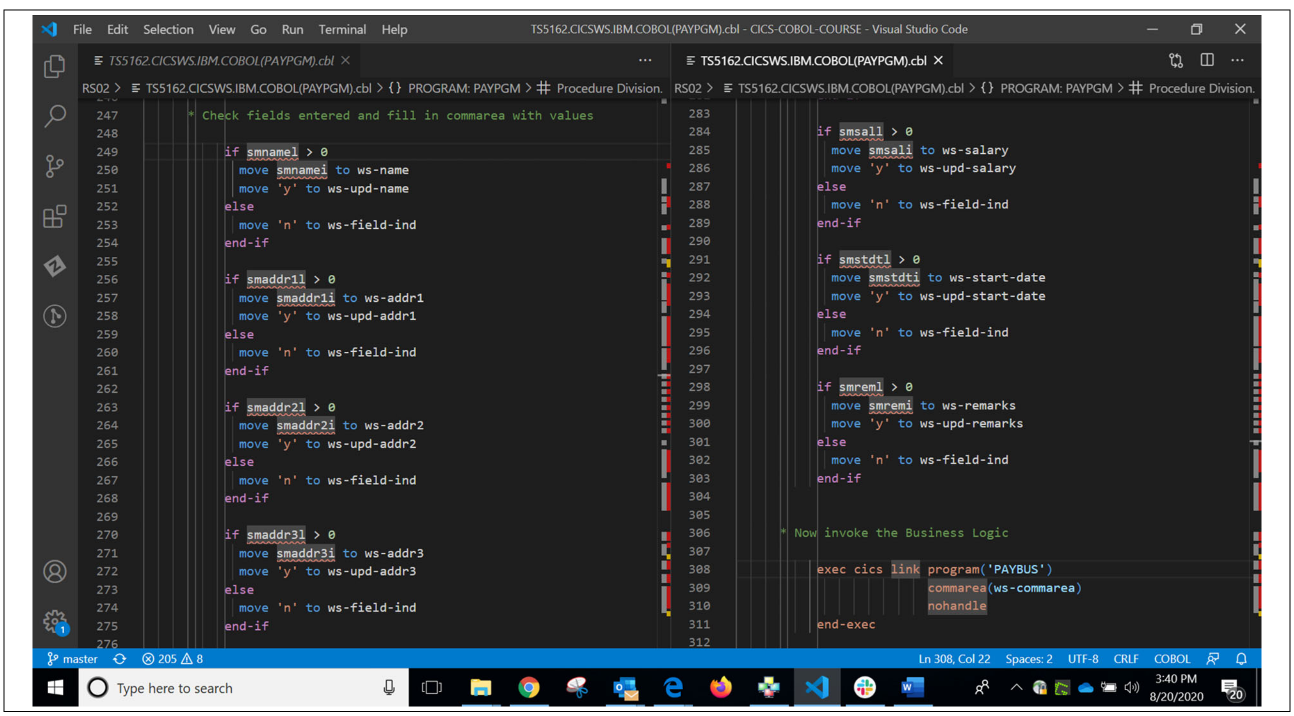


Figure 4-6 Field validation and link to PAYBUS

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.

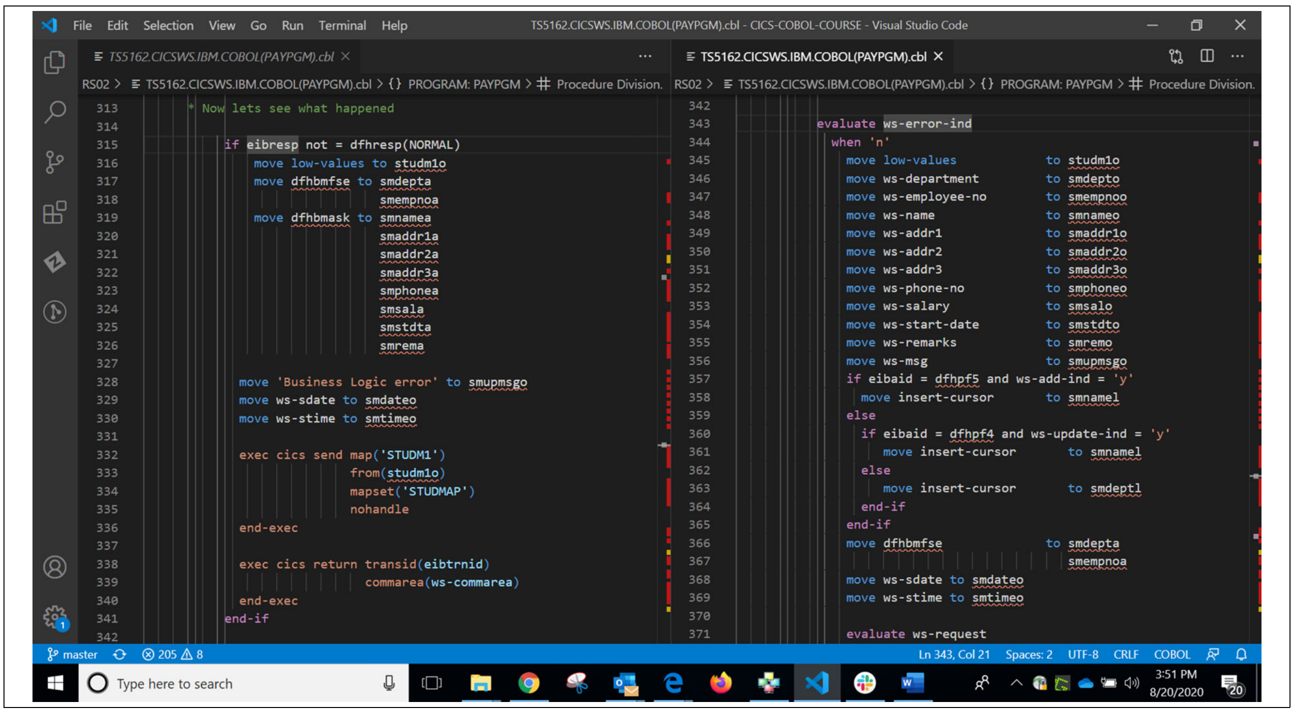


Figure 4-7 Link return code processing

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).

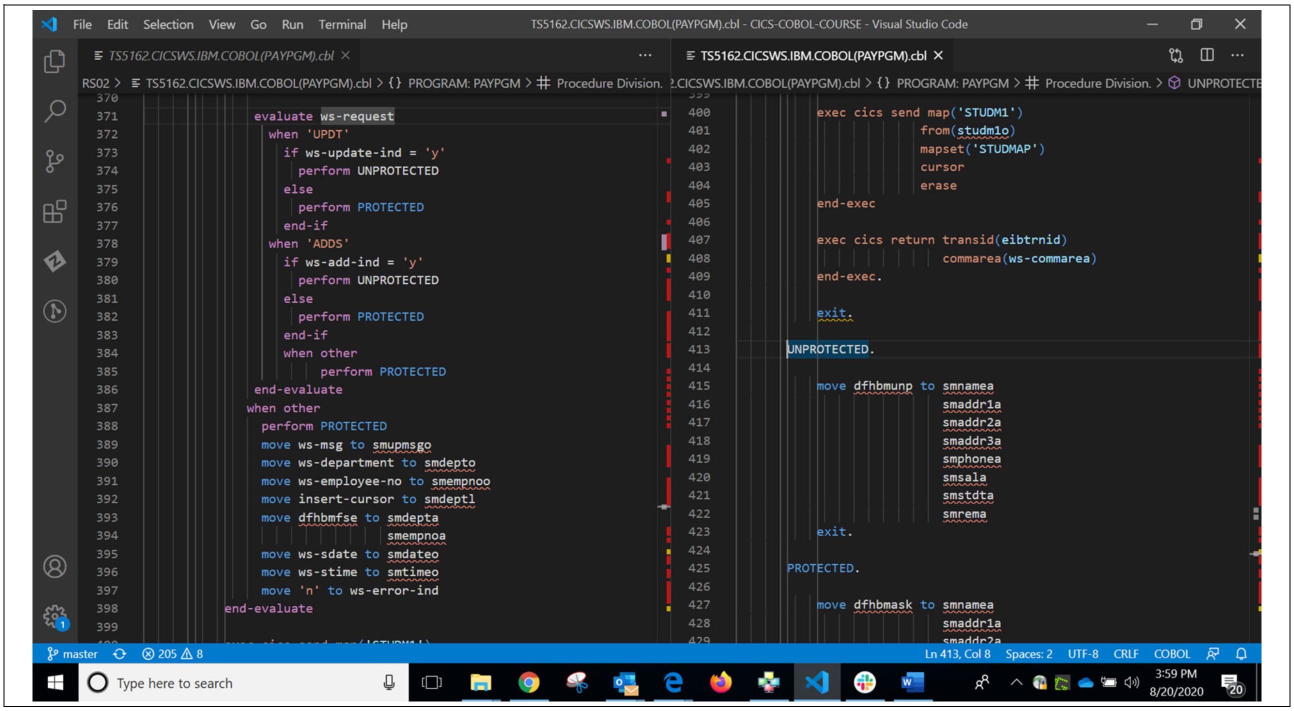


Figure 4-8 Remaining processing

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).

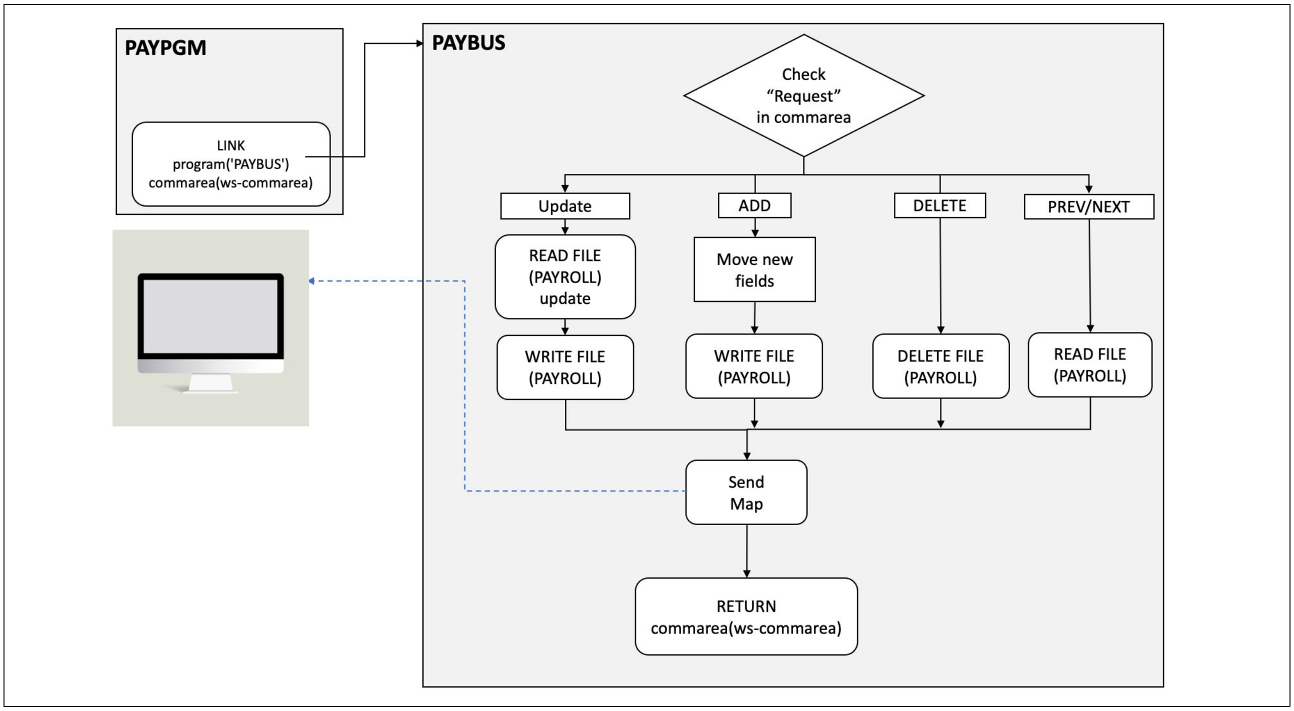


Figure 4-9 PAYBUS program flow

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.

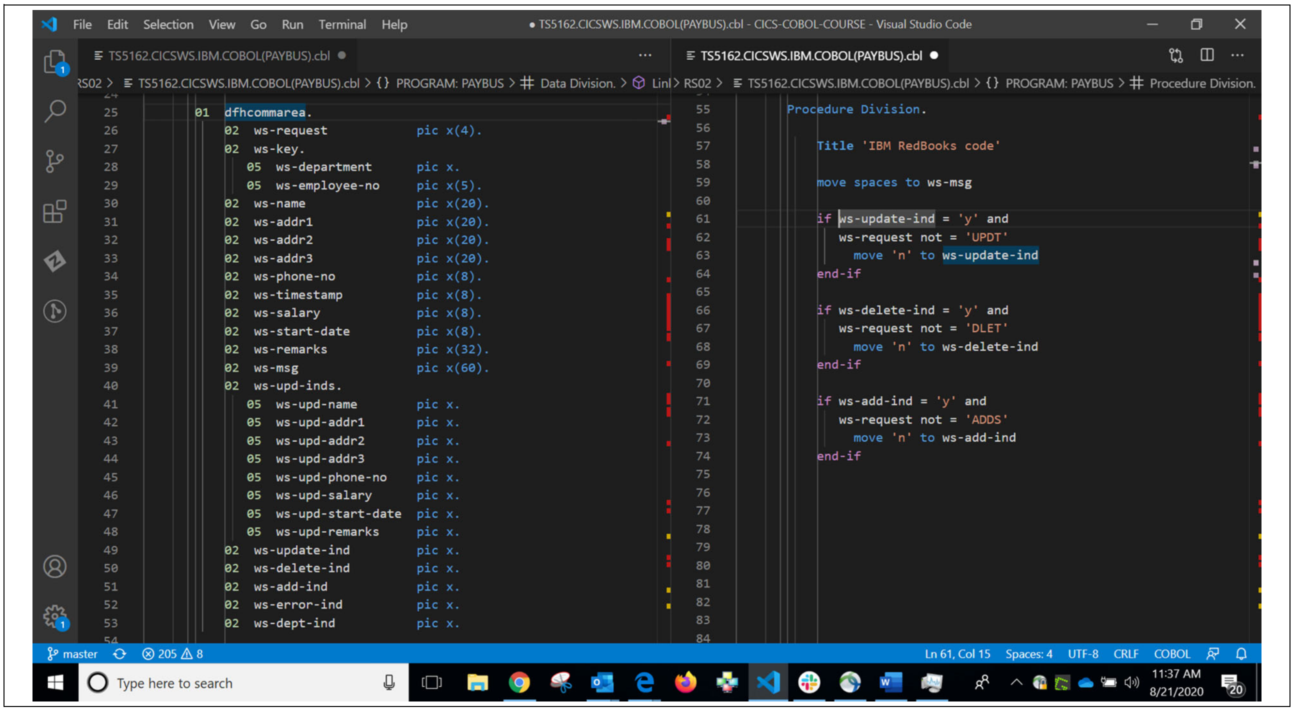


Figure 4-10 COMMAREA and special processing

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.

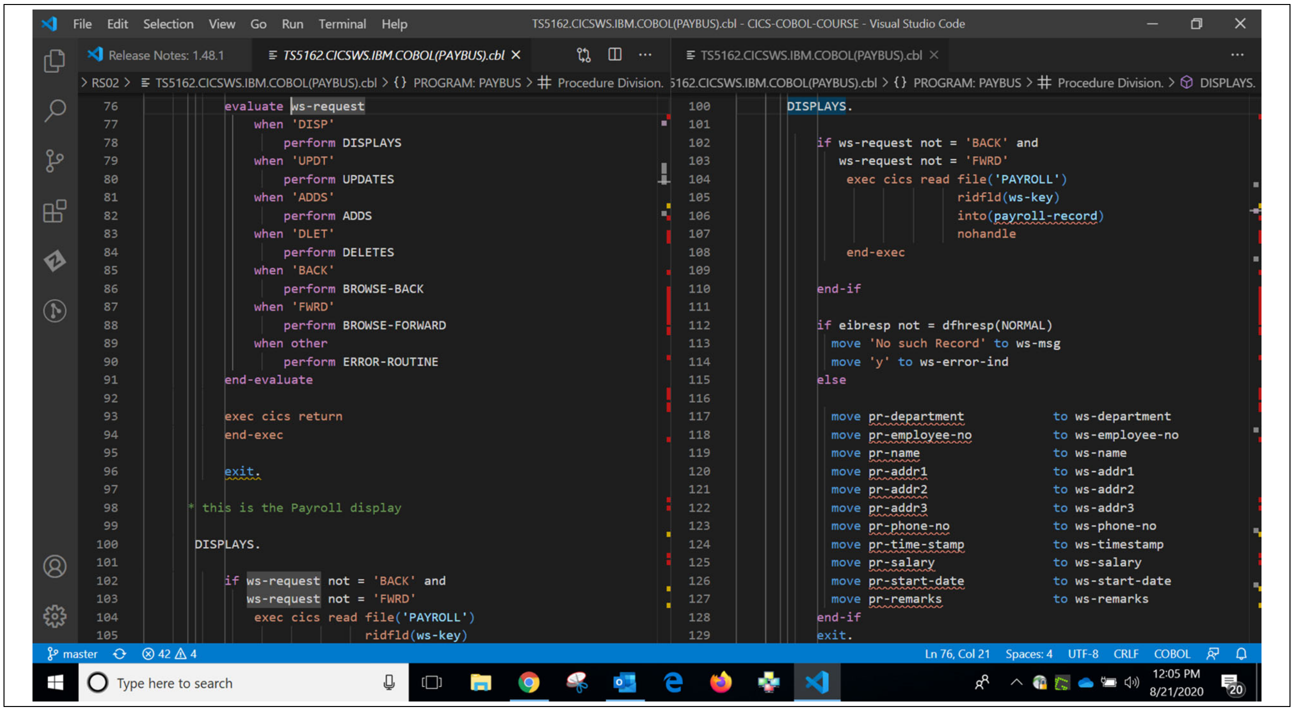


Figure 4-11 Request analysis

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).

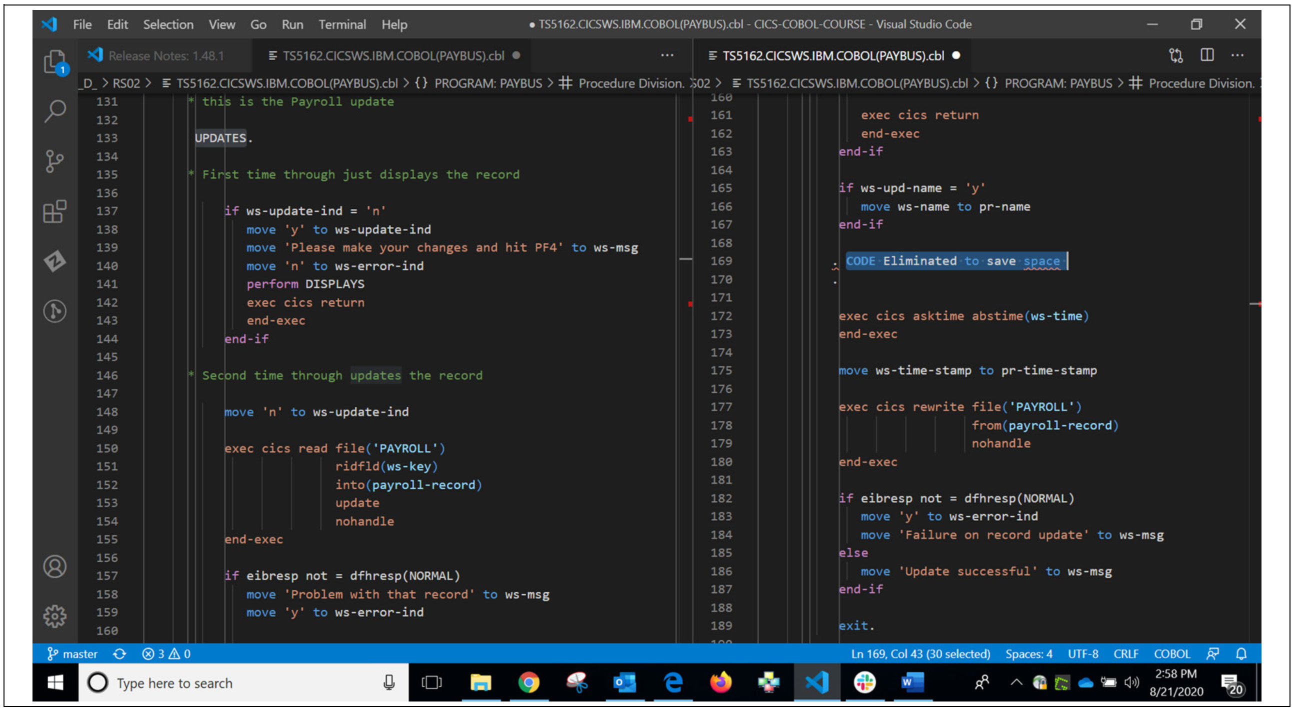


Figure 4-12 Updating a record

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).

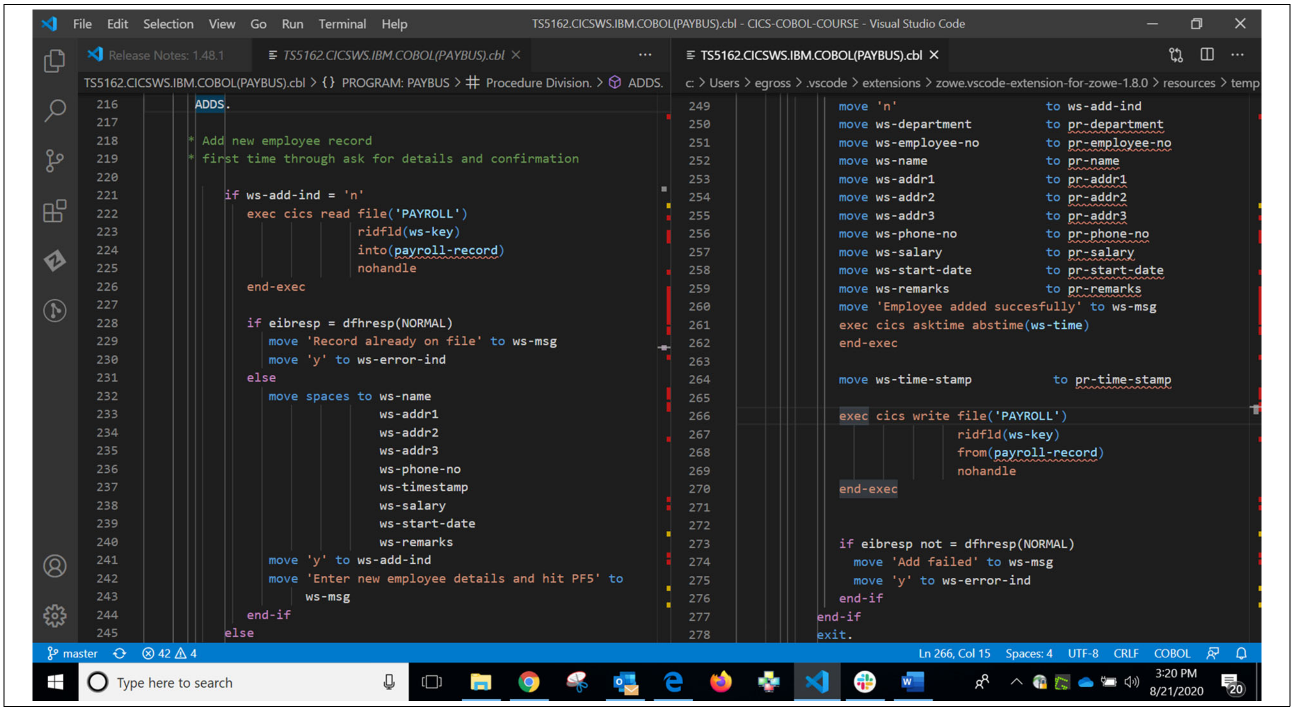


Figure 4-13 Adding a record

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).

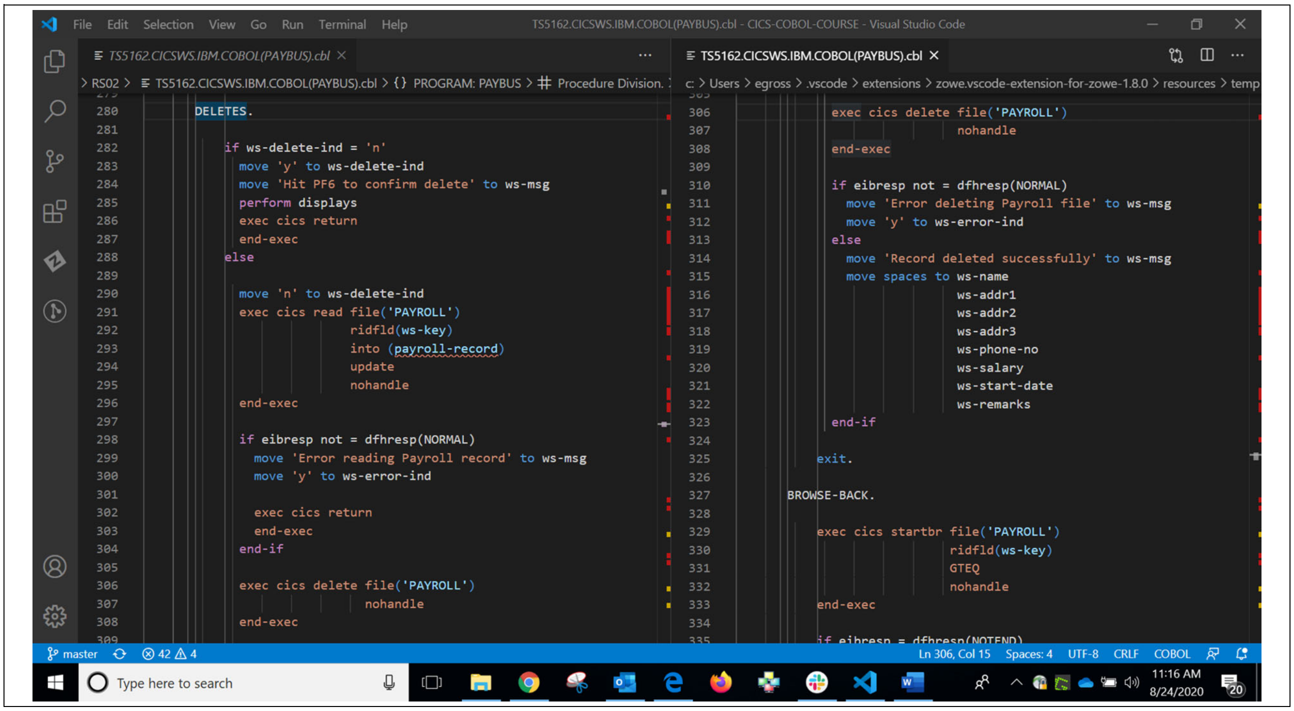


Figure 4-14 Deleting a request

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.

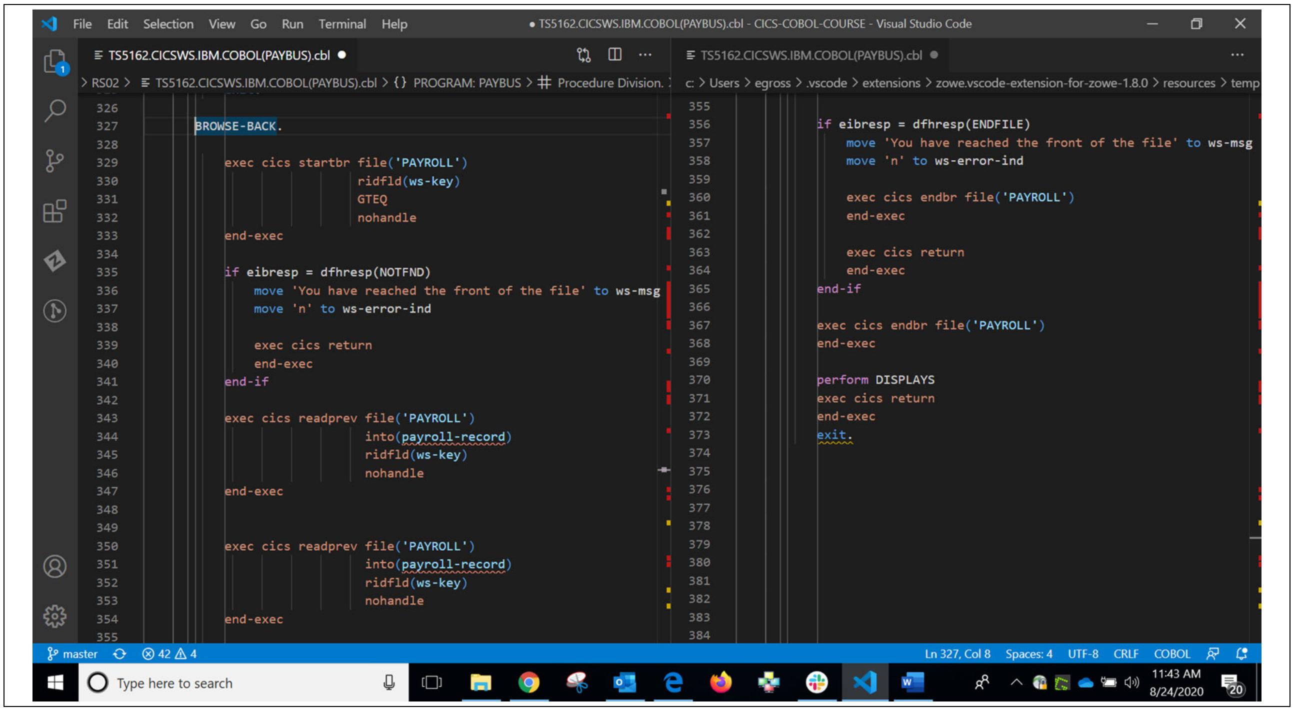


Figure 4-15 Browsing forward and backward

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.

5

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.

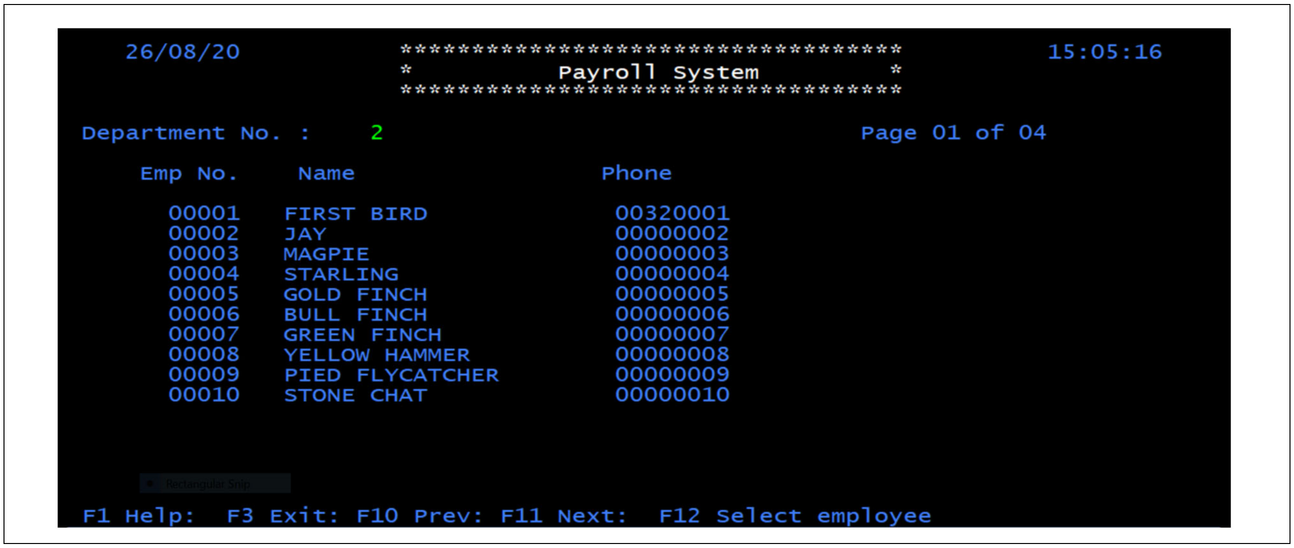


Figure 5-1 Department browse screen

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).

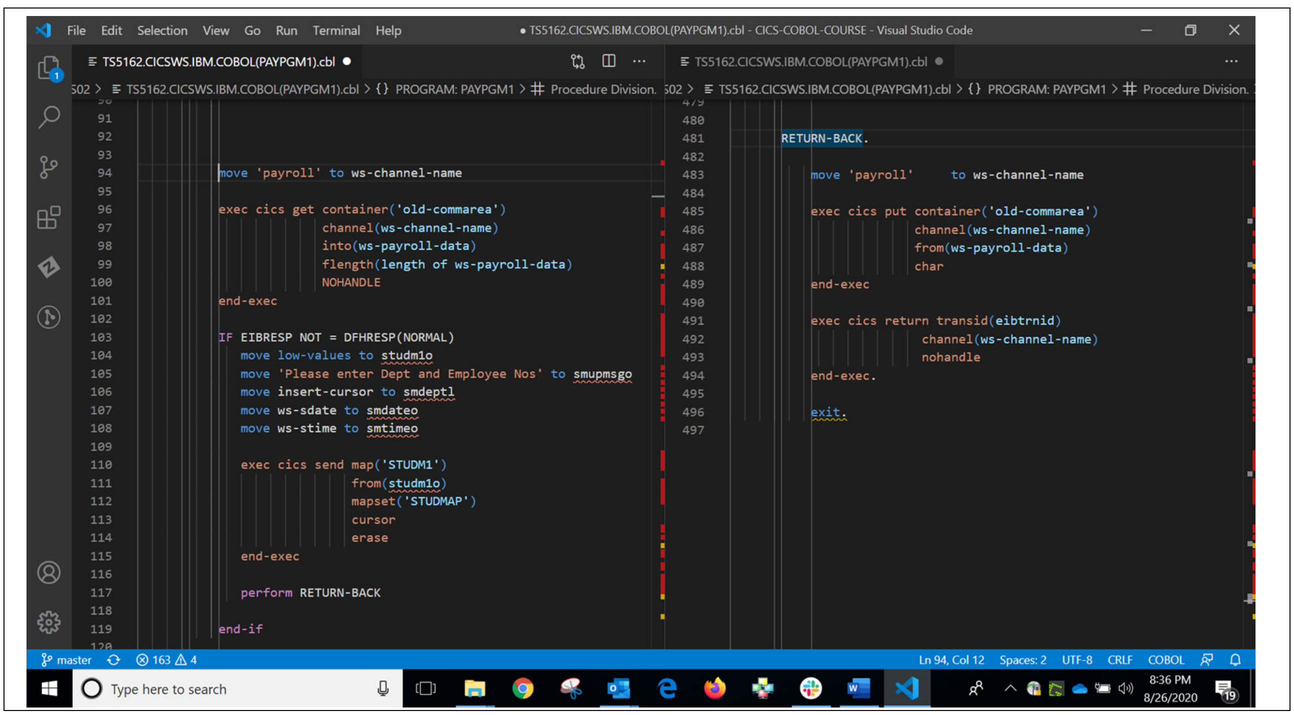


Figure 5-2 EXEC CICS GET and PUT CONTAINER commands

We may have a separate input and output container, and we may have a separate container for each structure. Each separate area requires an extra named container that when passed to a receiving program can choose which containers it wants to view and or replace.

For simplicity, we choose to use one container to hold the old COMMAREA contents.

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).

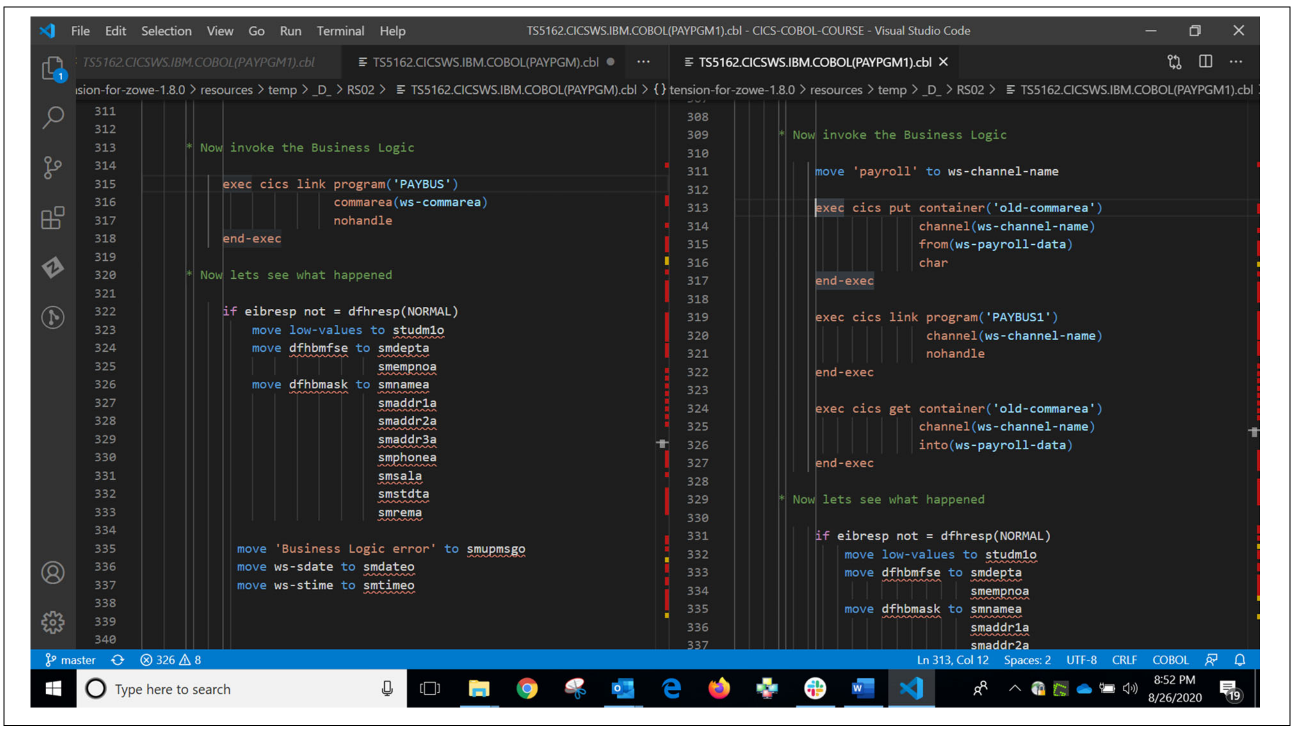


Figure 5-3 Starting the business logic

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).

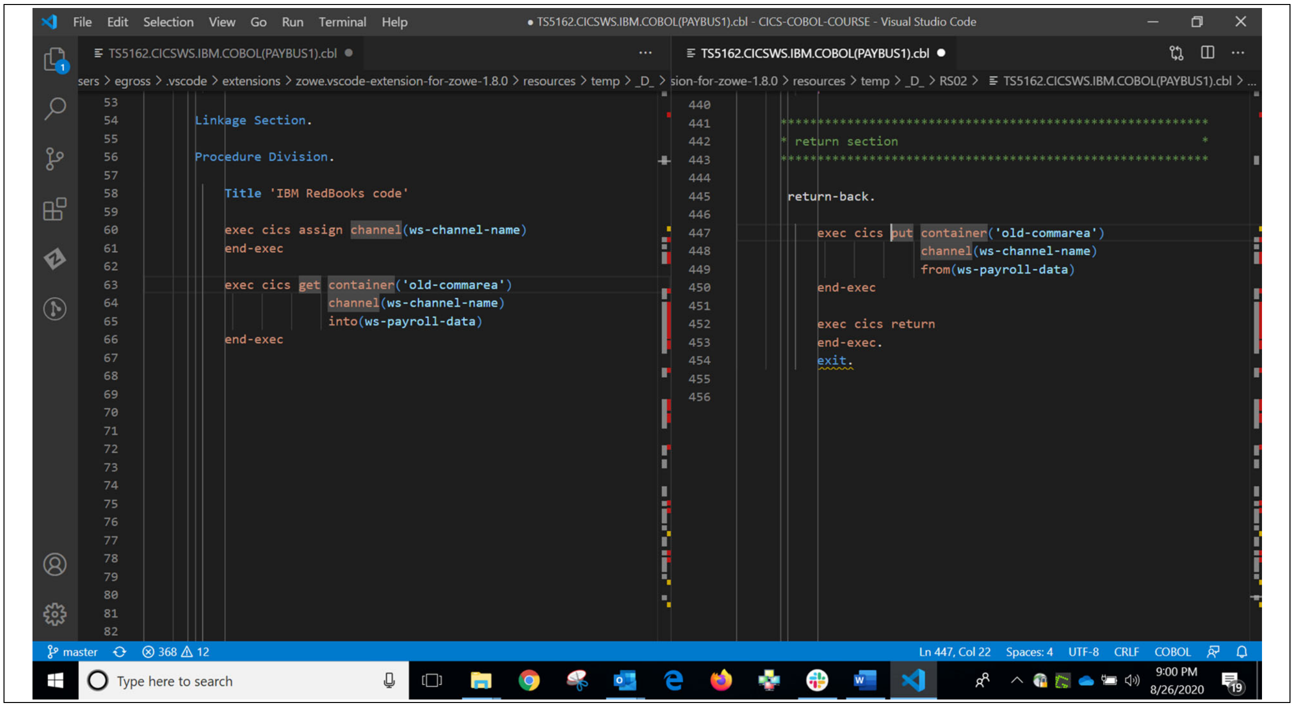


Figure 5-4 Changes to PAYBUS

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).

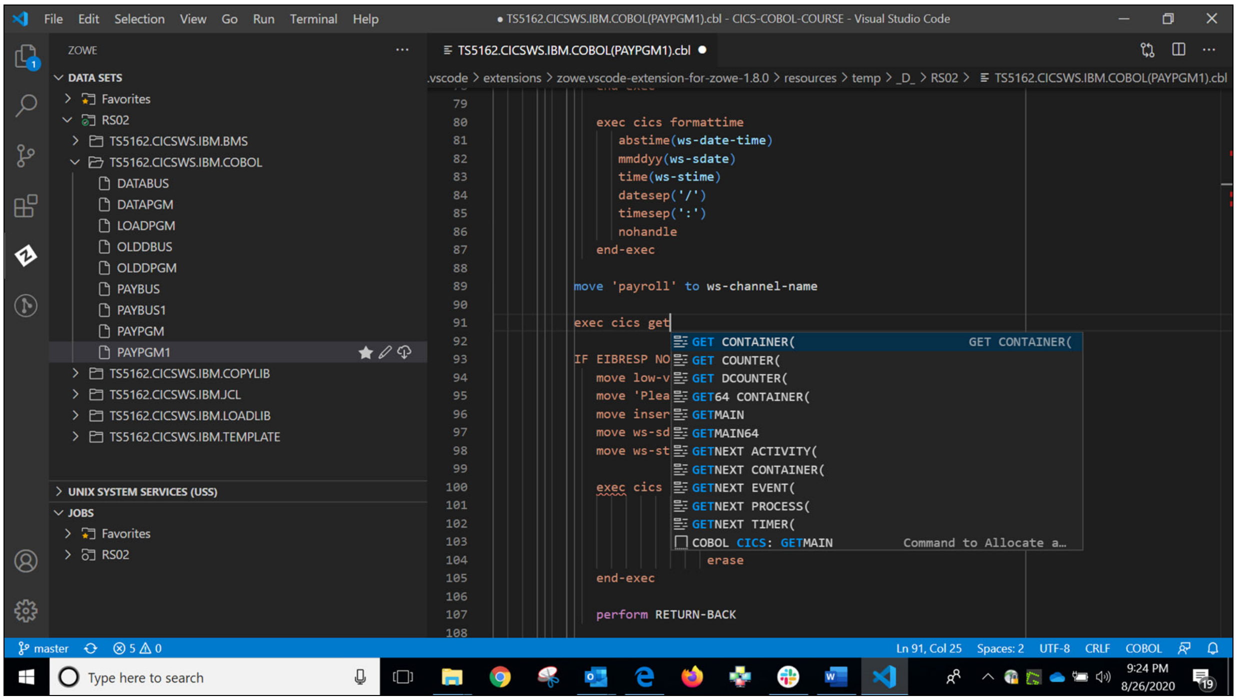


Figure 5-5 Auto-complete CICS commands

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 re 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).

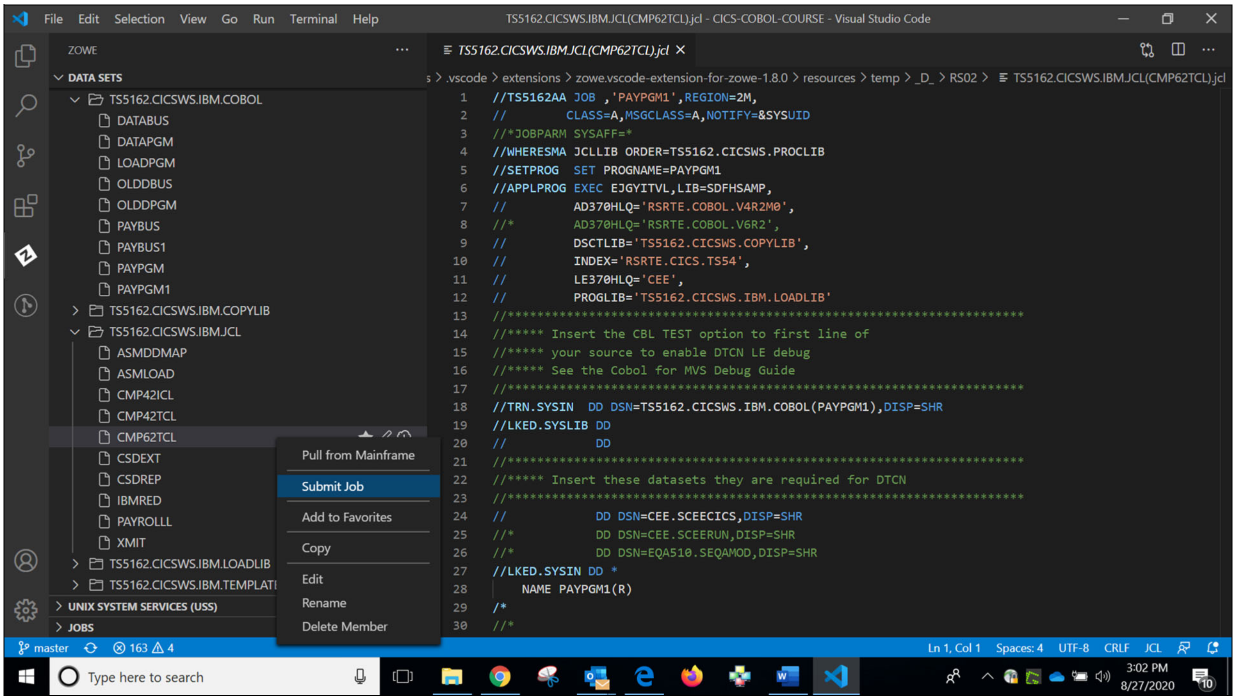


Figure 5-6 Submit job

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).

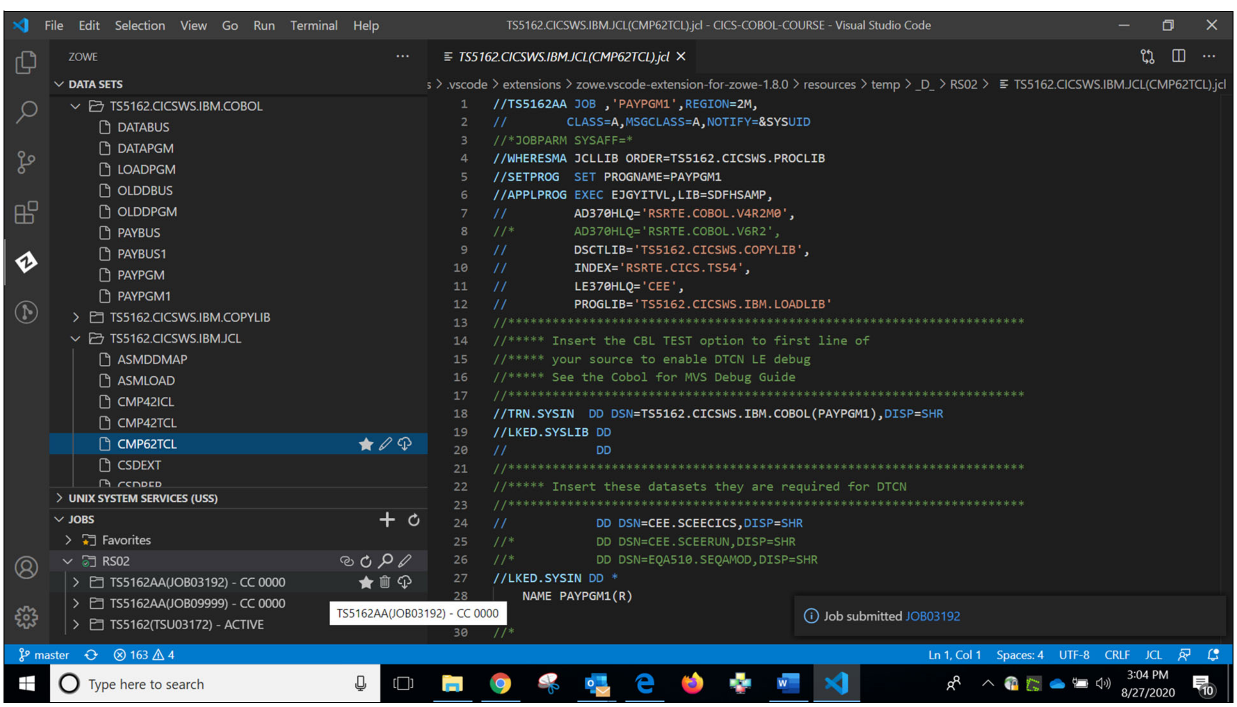


Figure 5-7 Job number

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).

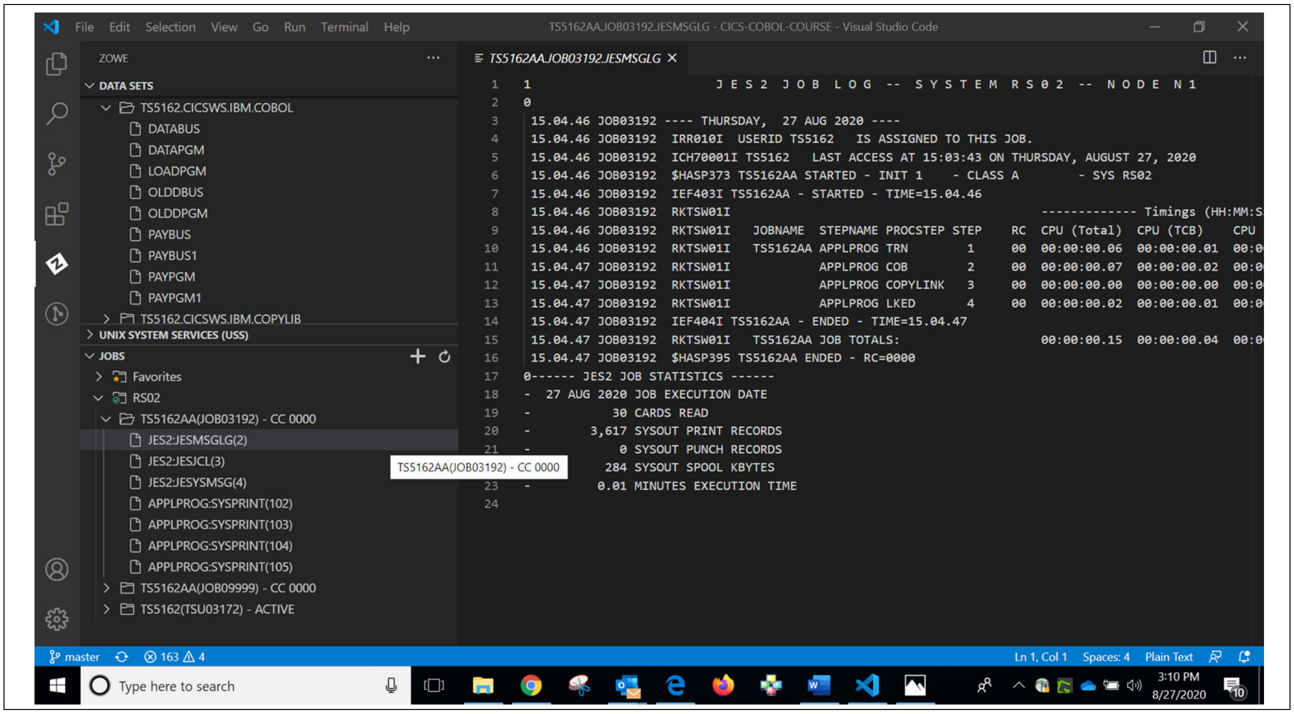


Figure 5-8 Job log

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
```

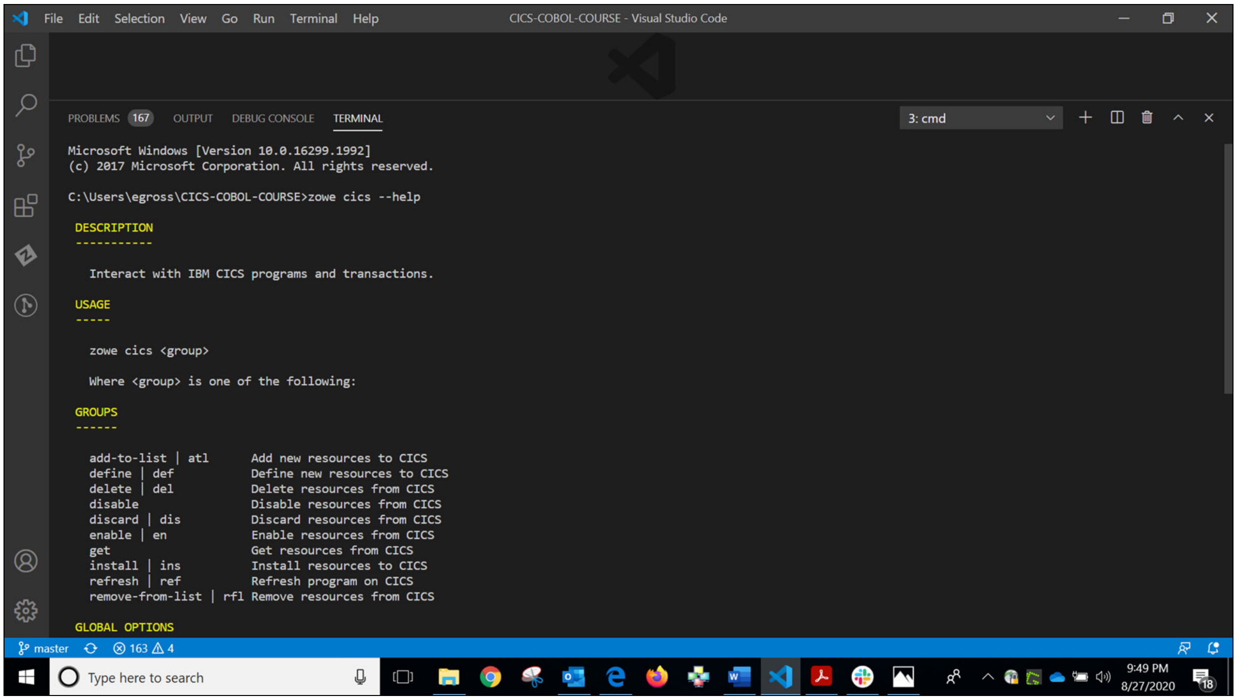


Figure 5-9 Zowe CICS help

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).

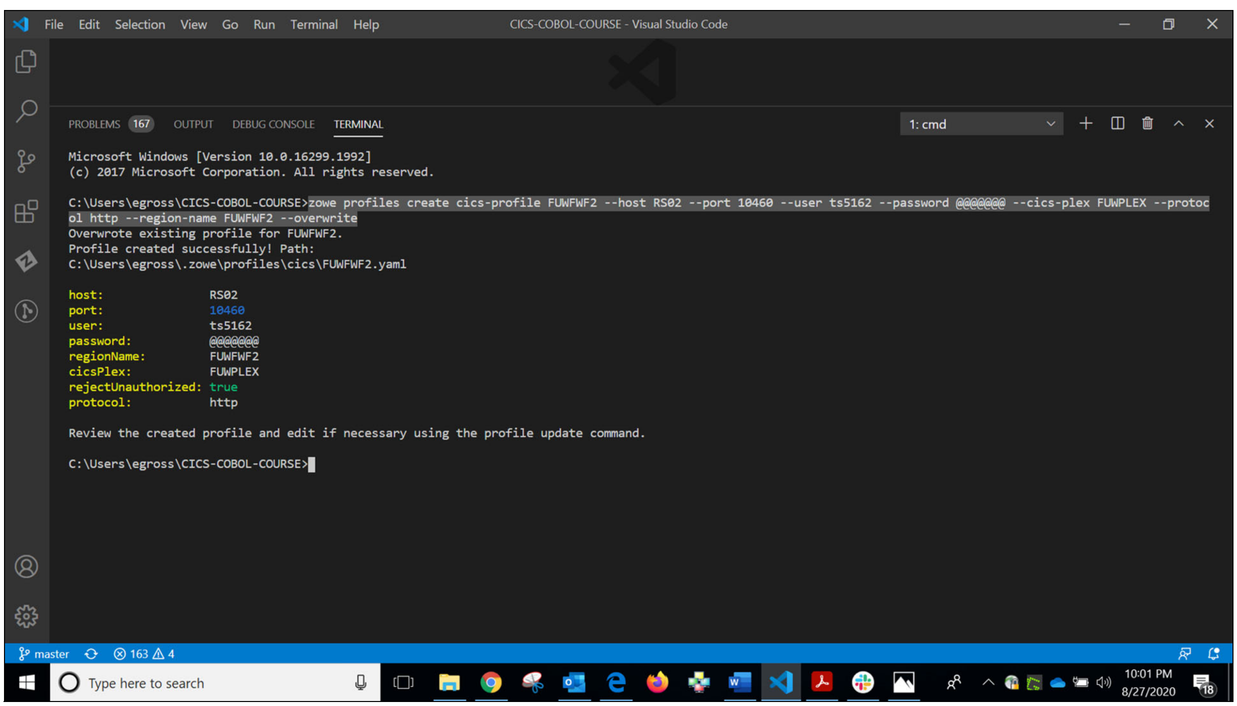


Figure 5-10 Zowe profile

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).

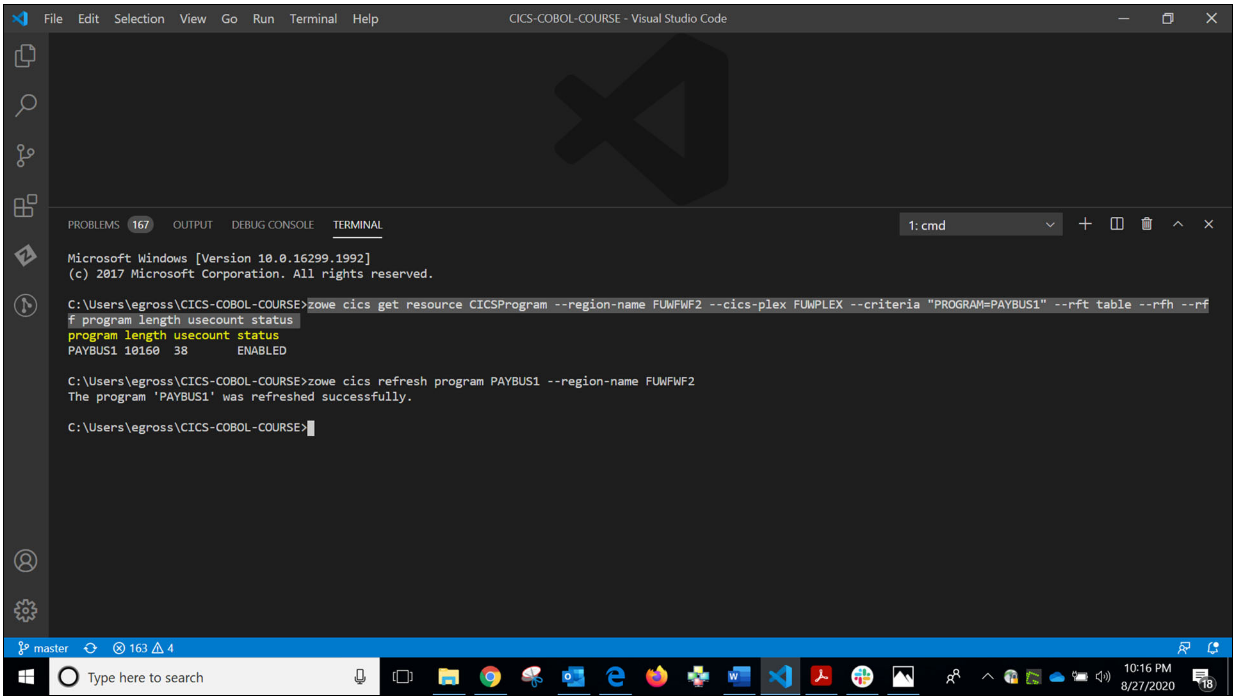


Figure 5-11 The get command

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.

6

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 genera-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).

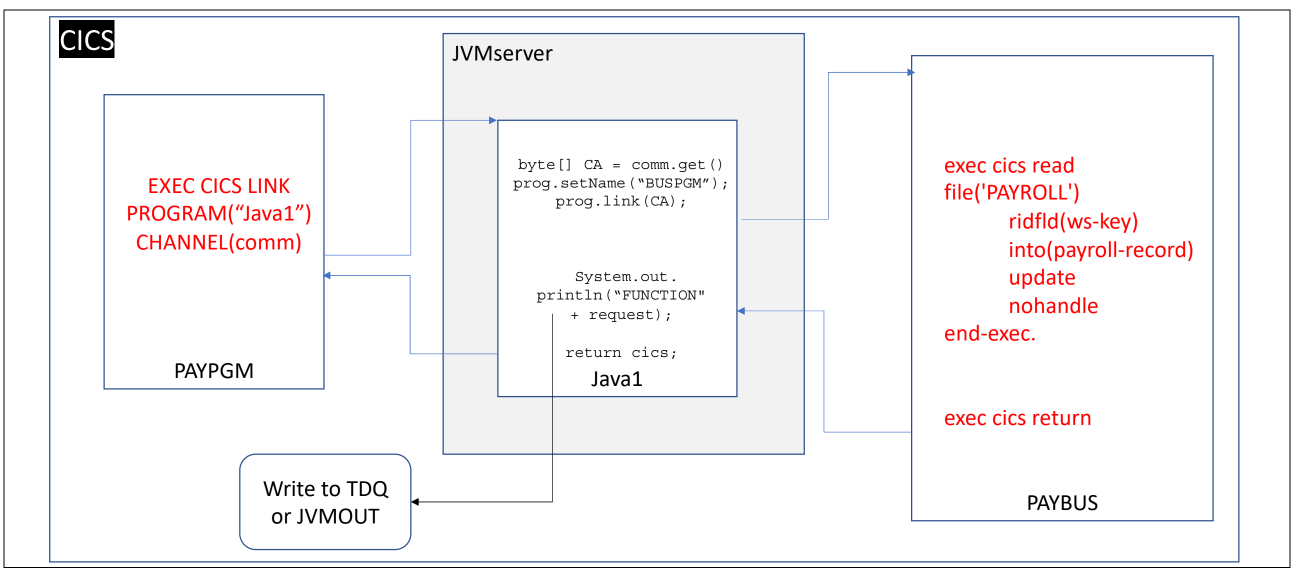


Figure 6-1 Audit logging application

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.

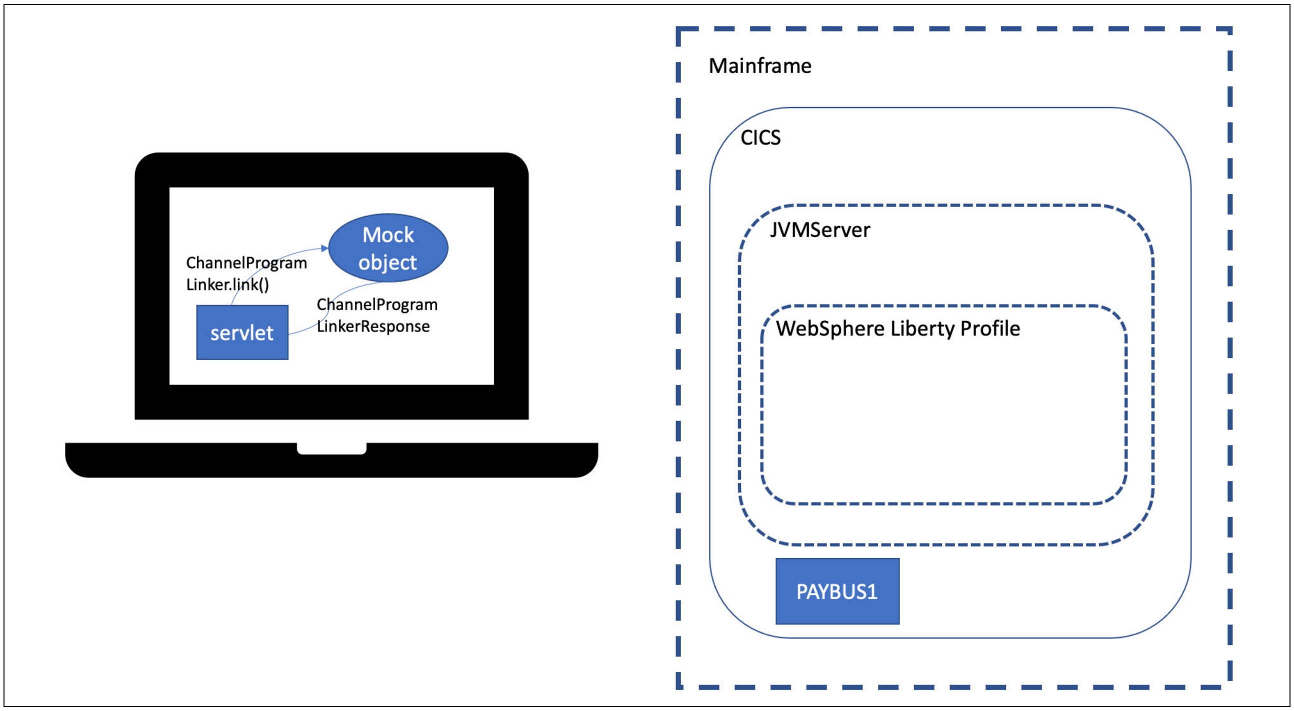


Figure 6-2 Servlet running on a local laptop

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

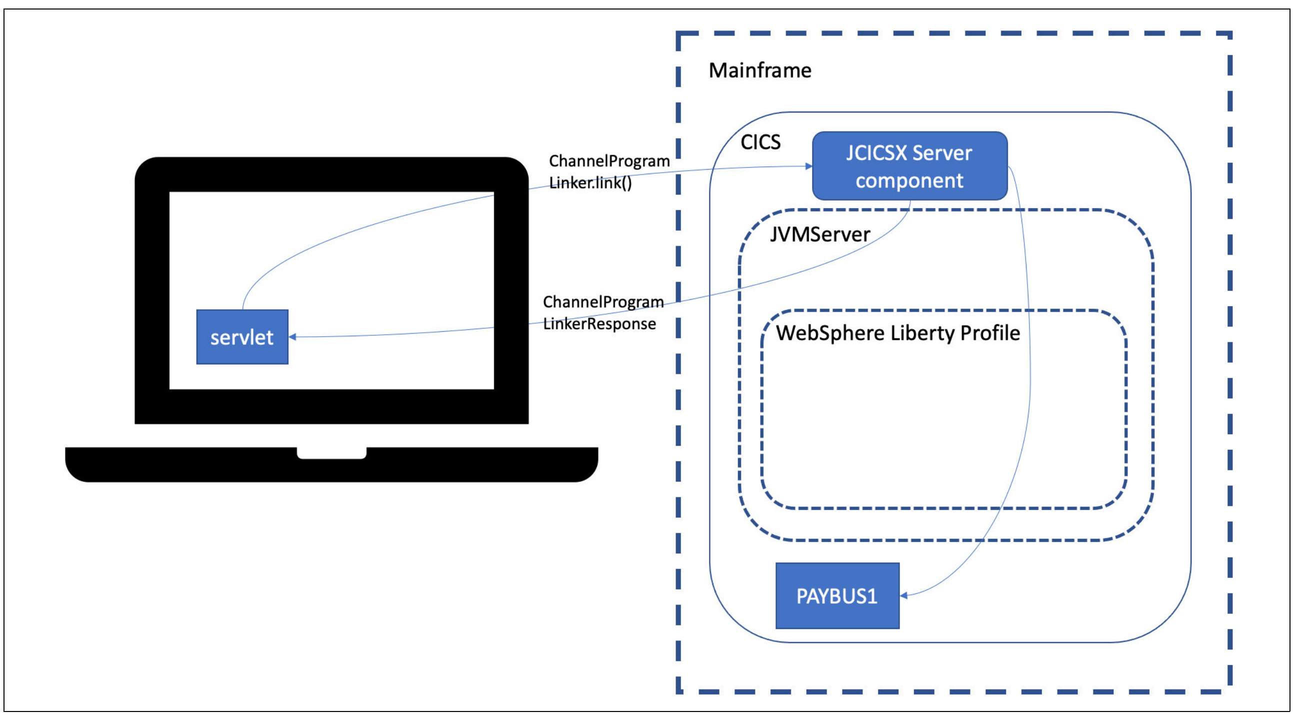


Figure 6-3 Remoting capability

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).

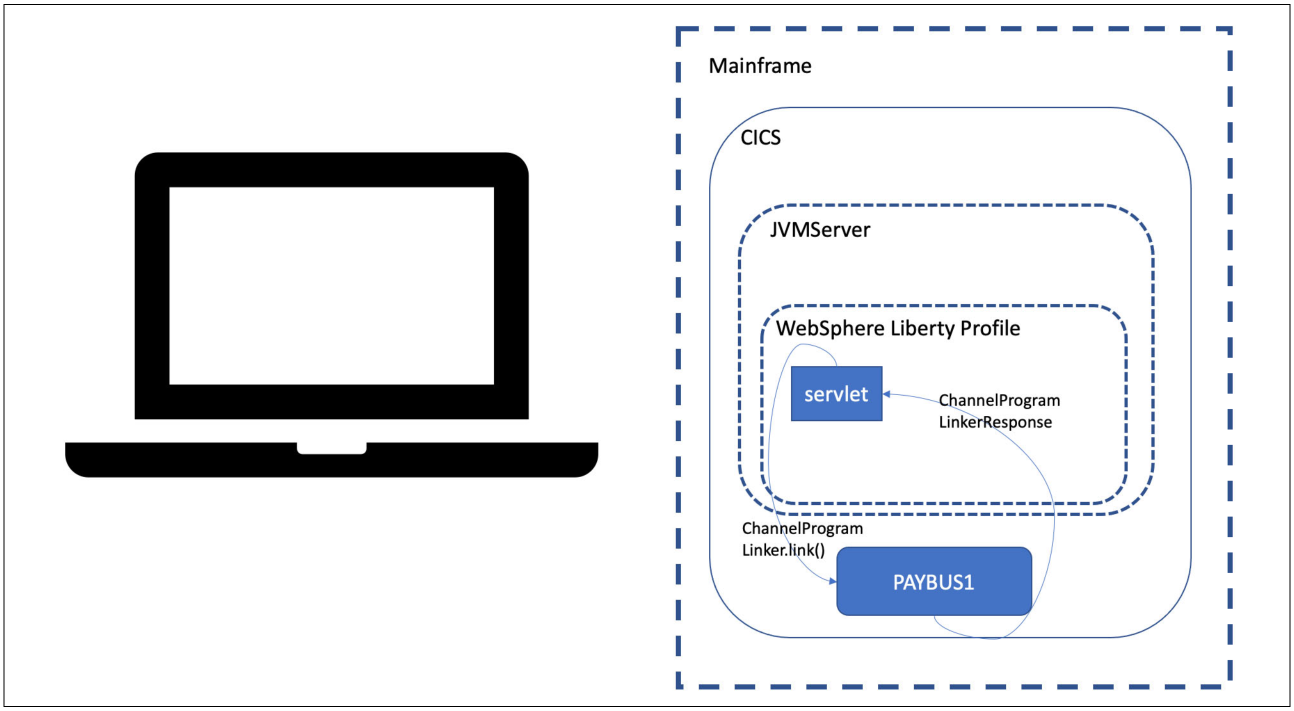


Figure 6-4 Application running in WebSphere Liberty Profile in CICS

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.

7

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.

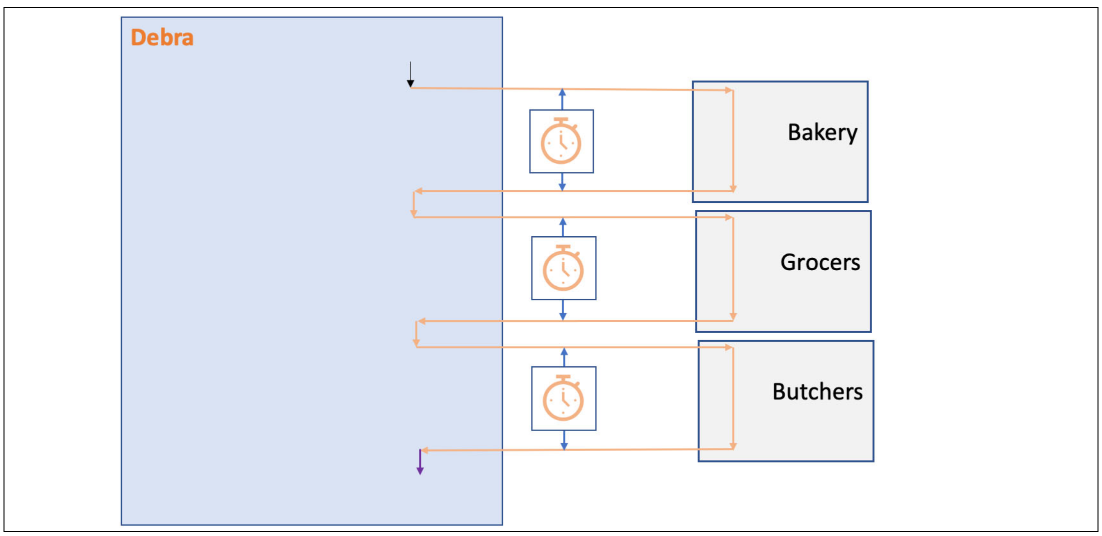


Figure 7-1 Shopping in a serial fashion

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.

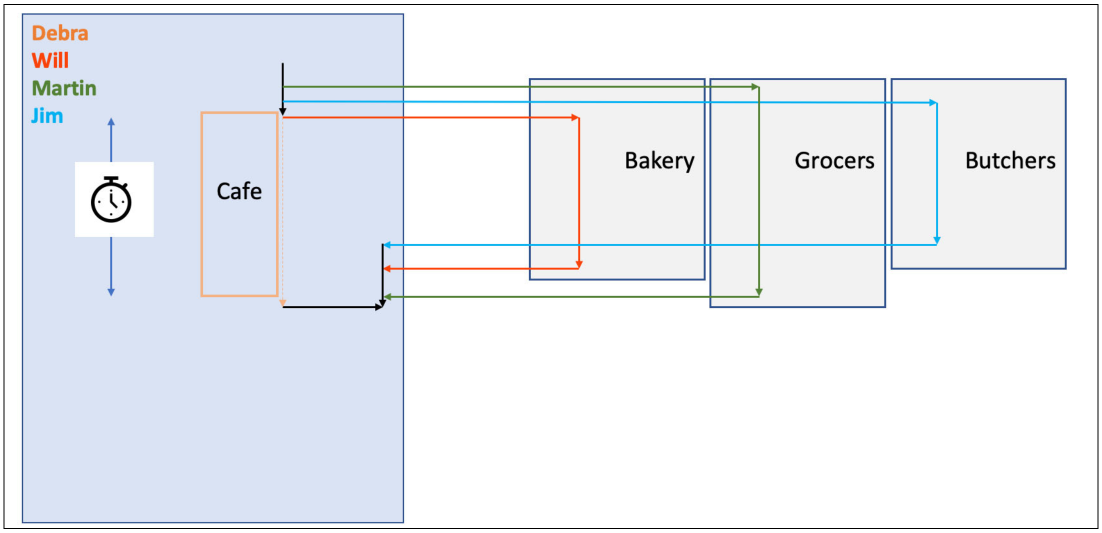


Figure 7-2 Asynchronous shopping

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).

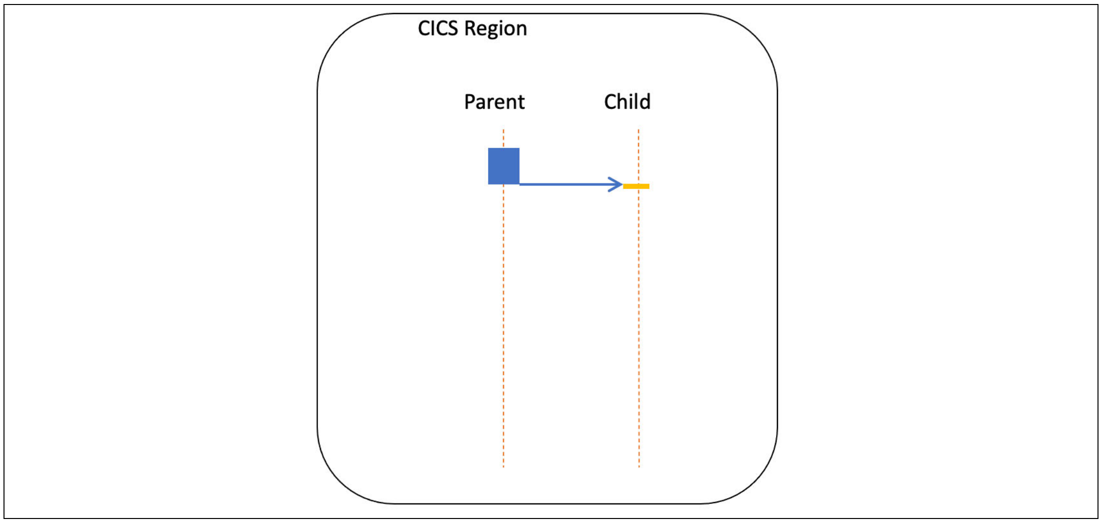


Figure 7-3 Parent-child 1

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).

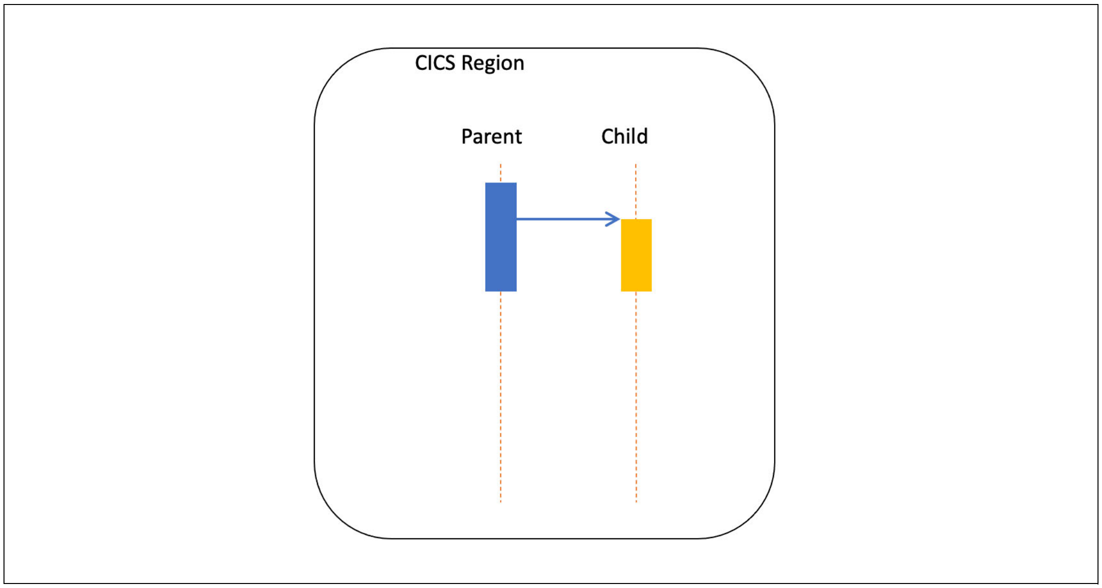


Figure 7-4 Parent-child 2

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).

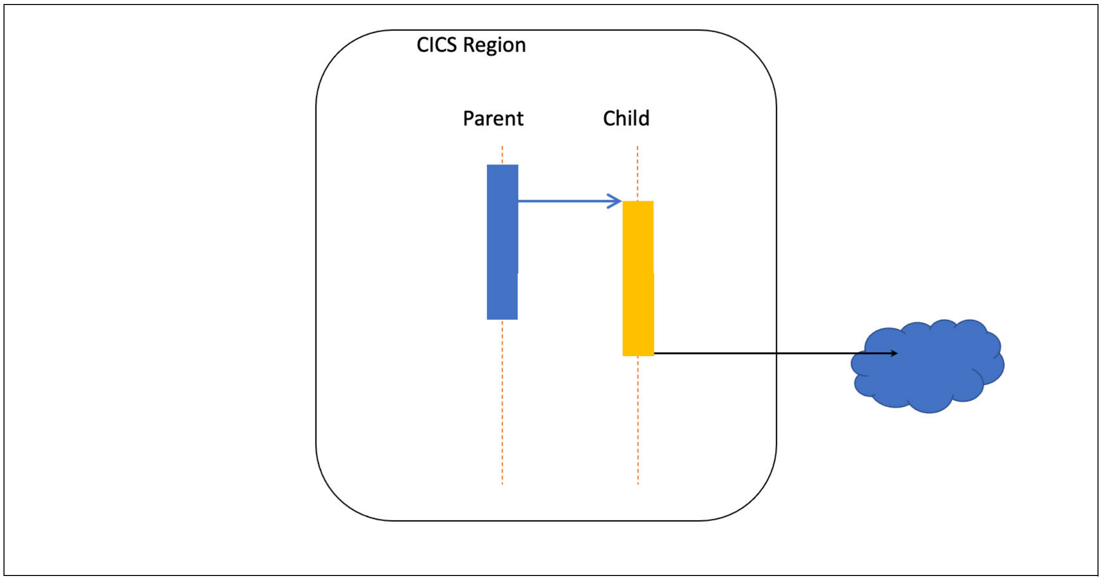


Figure 7-5   Parent-child 3

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).

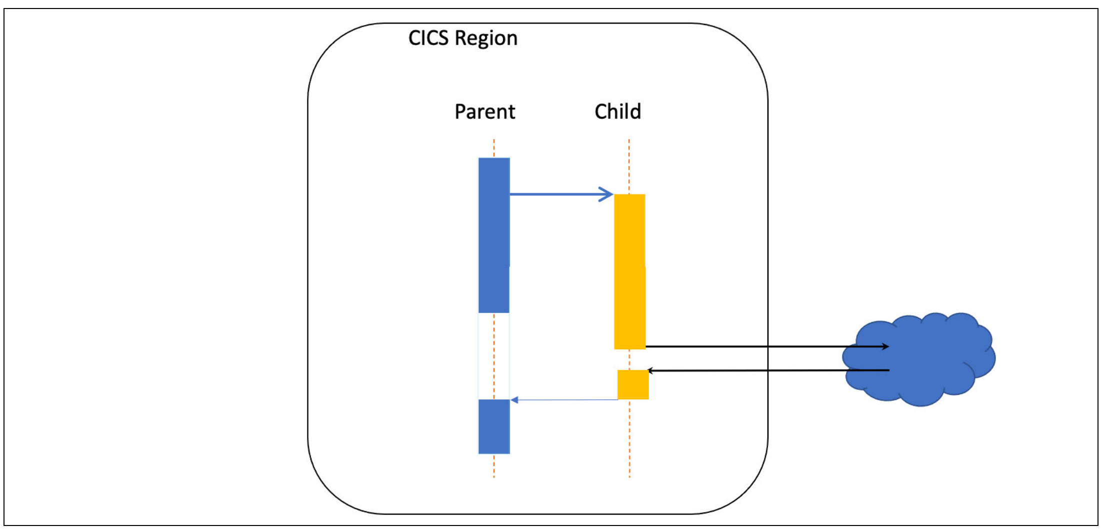


Figure 7-6   Parent-child 4

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.

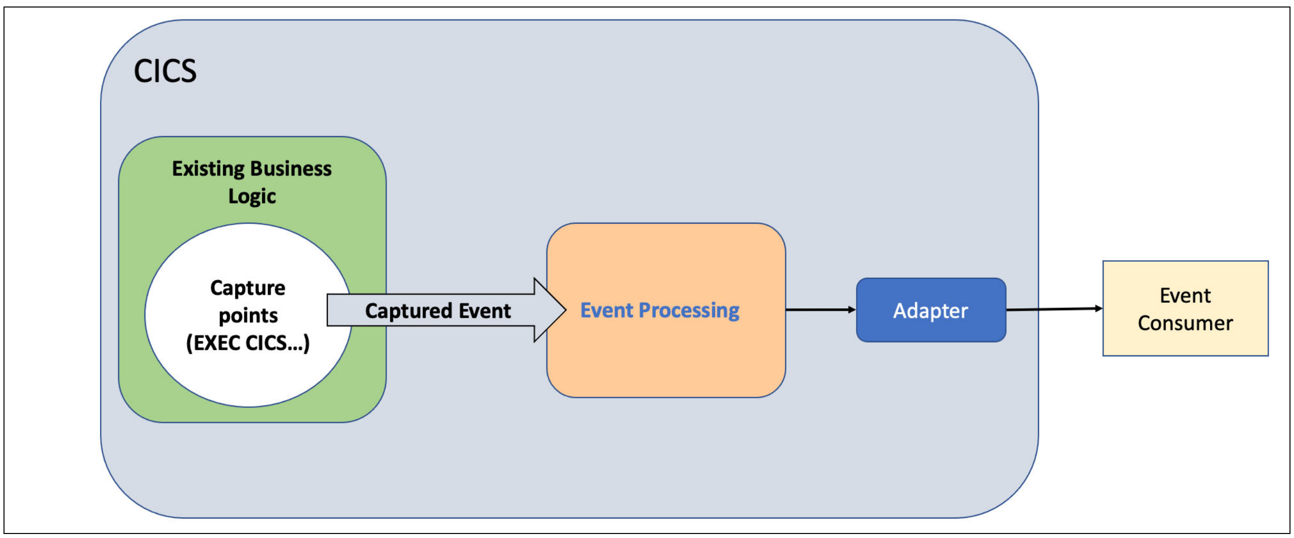


Figure 7-7 Basic event processing flow

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).

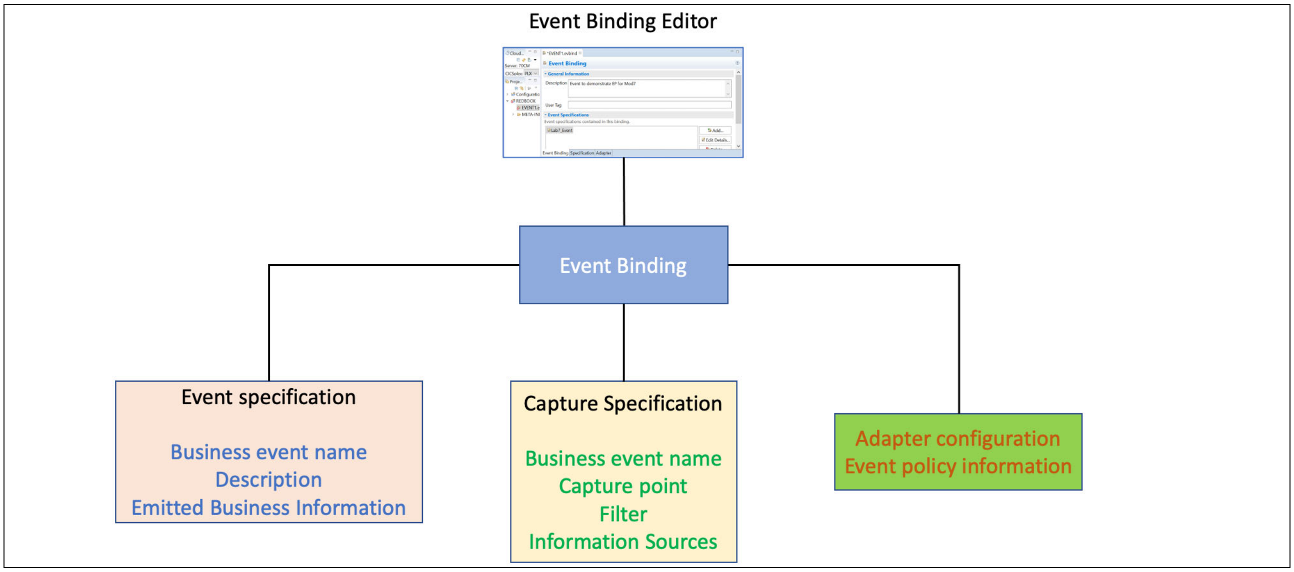


Figure 7-8 Event binding

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).

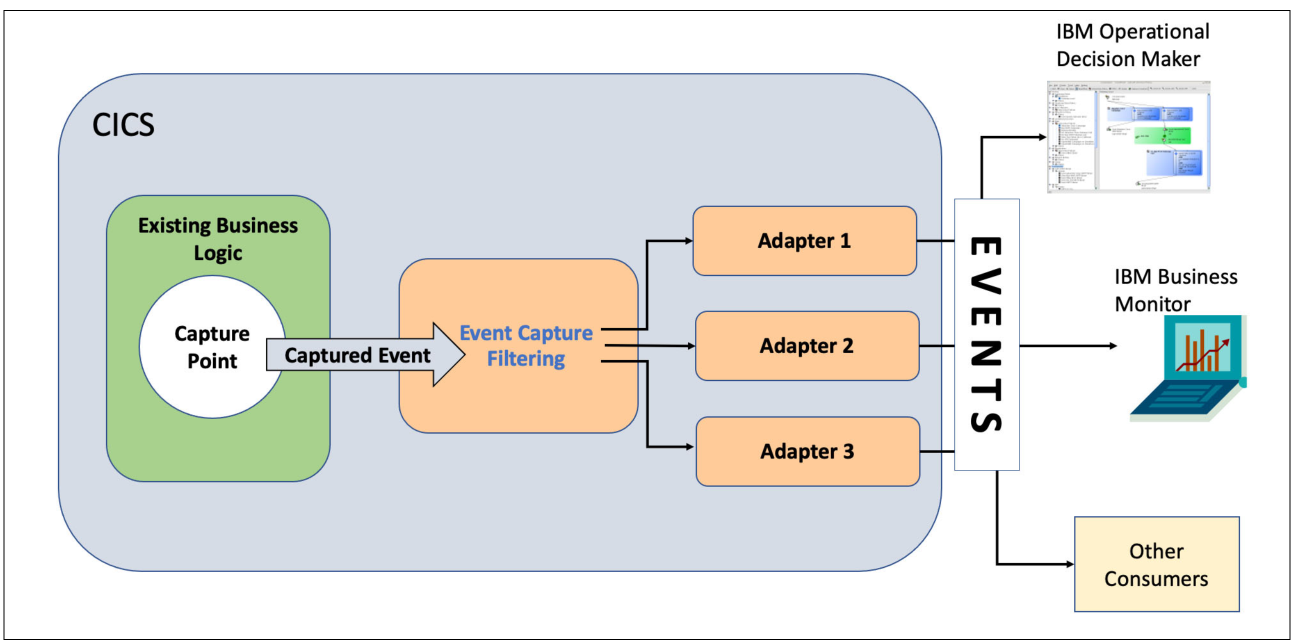


Figure 7-9 Runtime flow

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).

Event specification

Event name: Bet_OverTenThousand_Received
Event description: Whenever a bet is placed over 10 thousand
Emitted business information: Customer name, Betting shop location

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).

Event specification

Event name: Bet_OverTenThousand_Received
Event description: Whenever a bet is placed over 10 thousand
Emitted business information: Customer name, Betting shop location

Capture Specification:
Before EXEC CICS LINK command
to PROGRAM(PLACEBET)
from current_program = BETTING
where *BetVal > 10K

** BetVal is in the Commarea or channel passed on the LINK*

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).

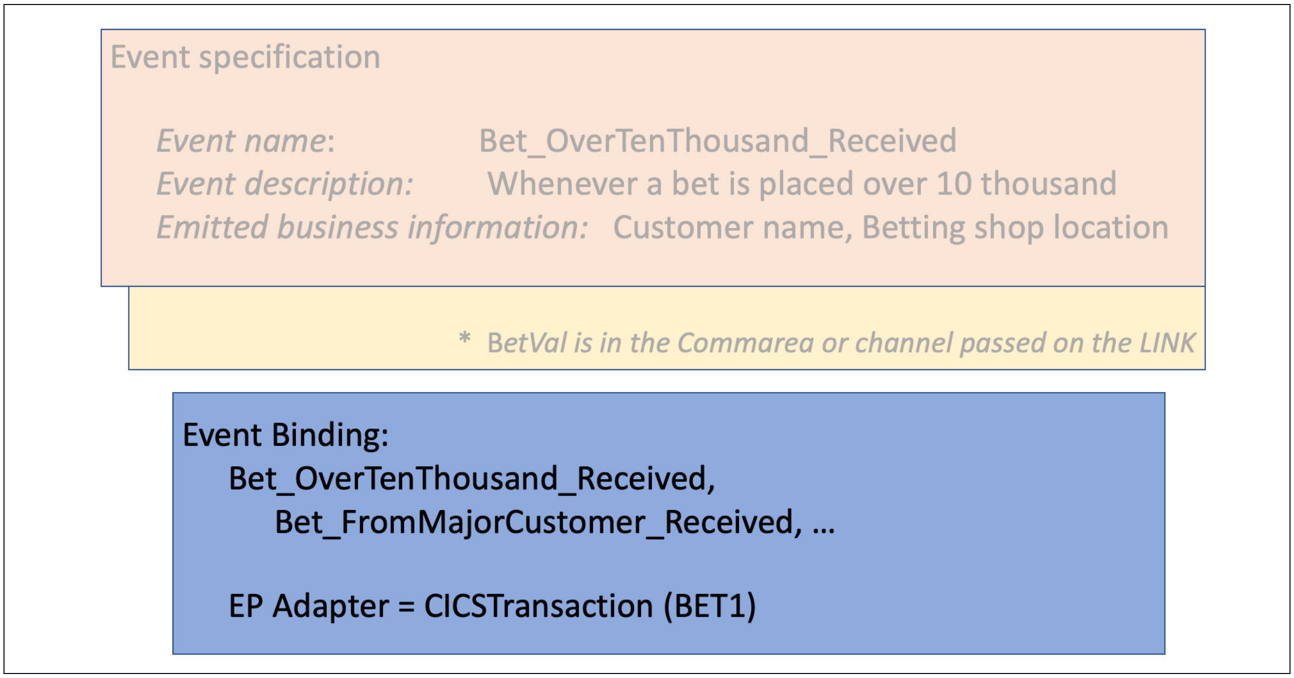


Figure 7-12 Event binding

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).

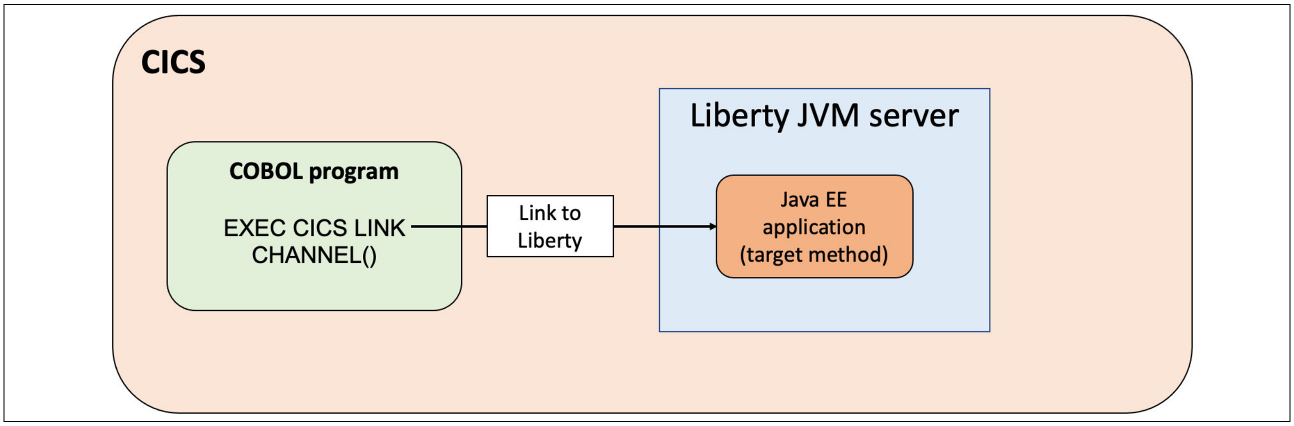


Figure 7-13 Relationship between applications

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).

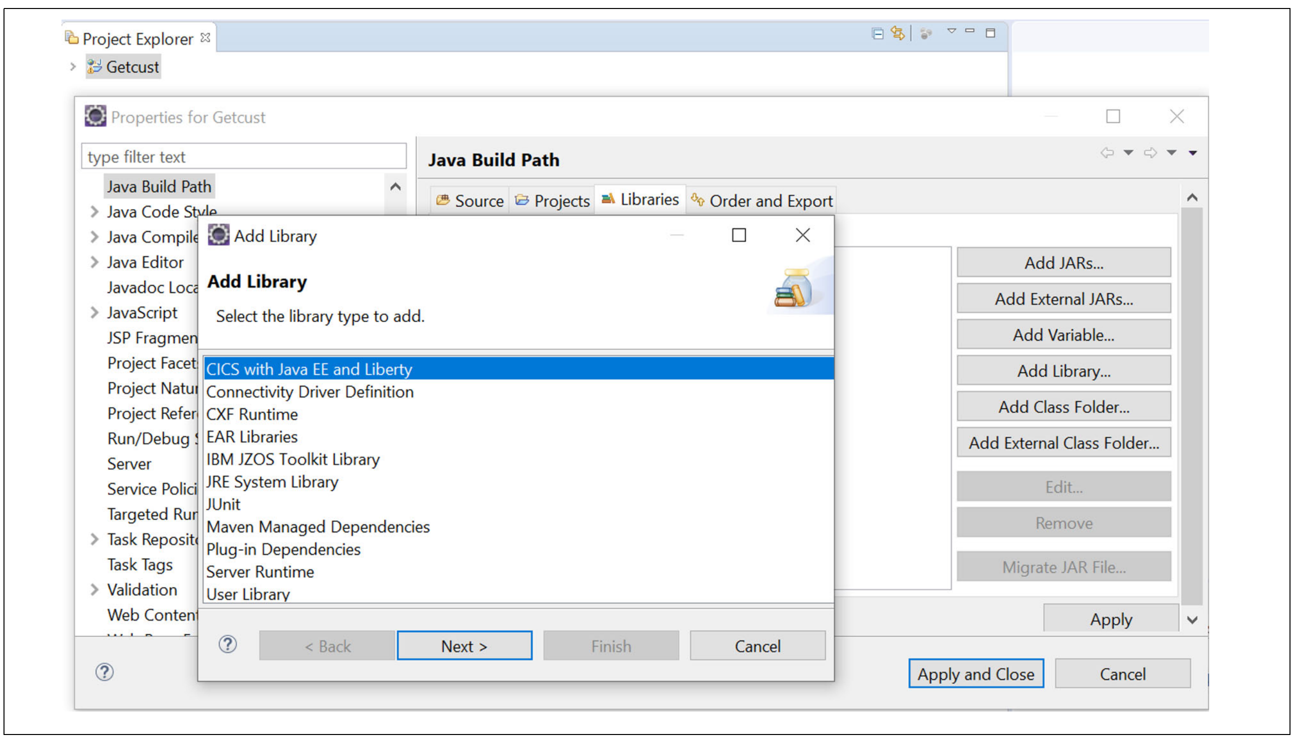


Figure 7-14 Adding a library

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).

```
public class CustomerLinkTarget
{
    @CICSProgram("CUSTGET")

    public void getCustomer()
    {
        Channel currentChannel = Task.getTask().getCurrentChannel();
        Container dataContainer = currentChannel.getContainer("DATA");

        // do work here

        Container resultContainer = currentChannel.createContainer("RESULT");
        byte[] results = null; // change this to be the result of the work
        resultContainer.put(results);
    }
}
```

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).

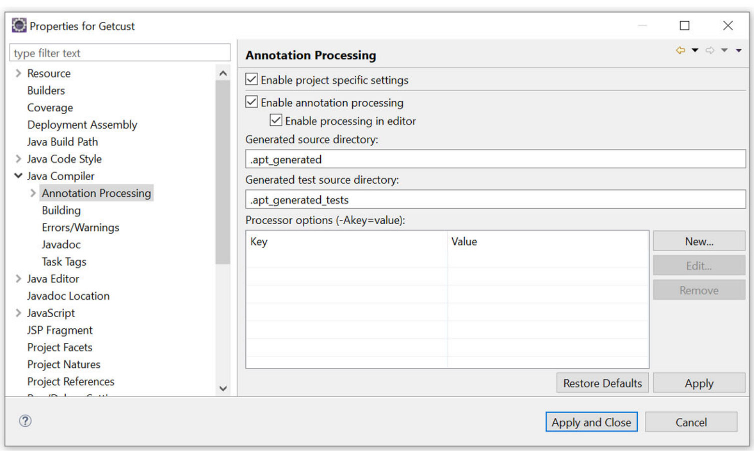


Figure 7-16 Annotation Processing

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).

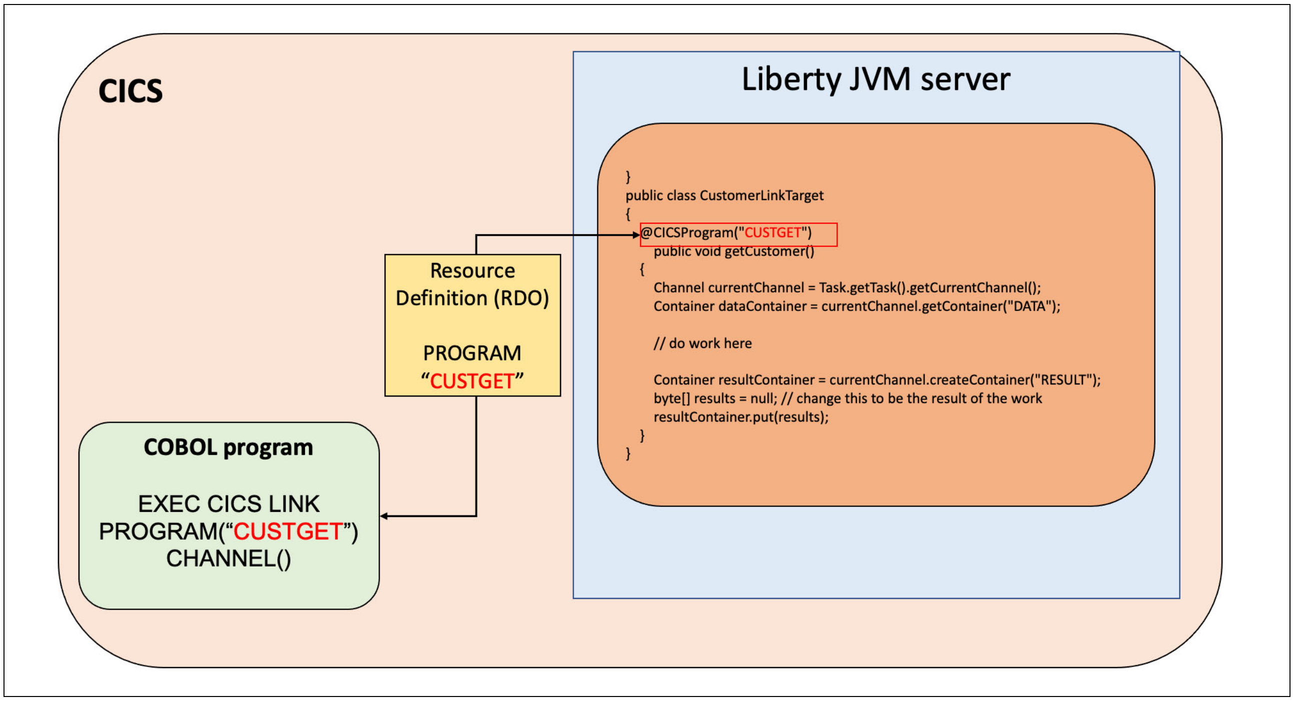


Figure 7-17 Starting the Java application

8

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).

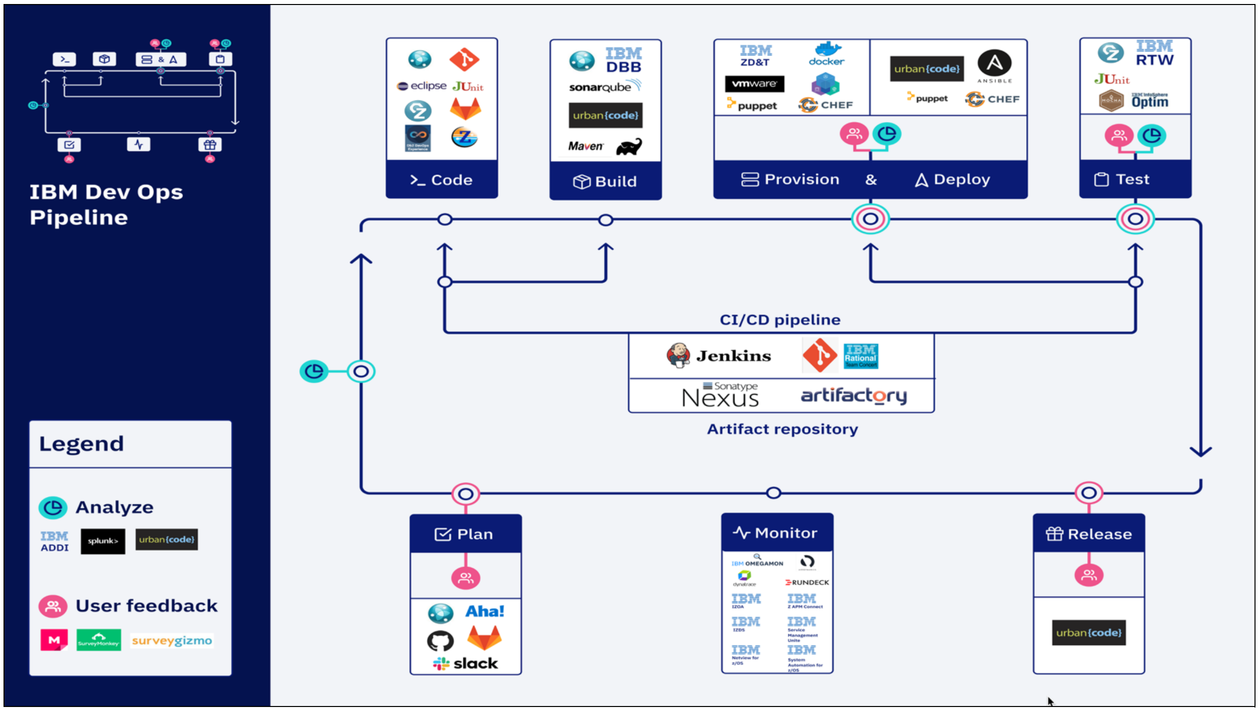


Figure 8-1 IBM DevOps pipeline

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).

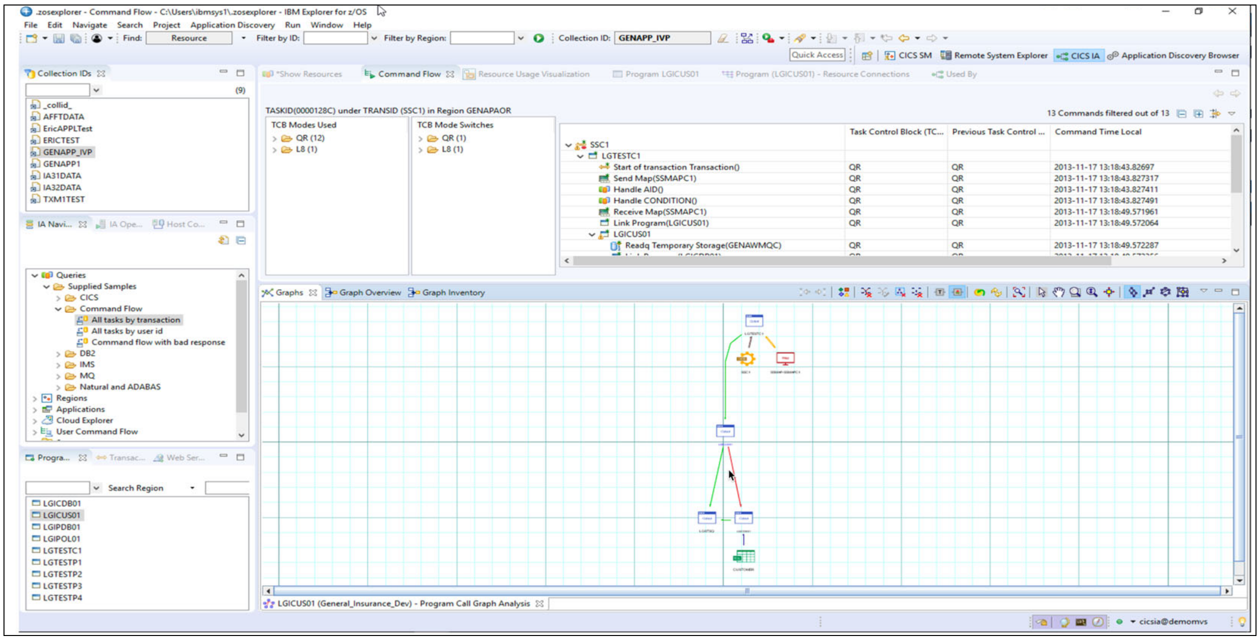


Figure 8-2 Command flow

REDP-5628-00

ISBN 0738459291

Printed in U.S.A.