

Visual COBOL®

NEW APPLICATION MODERNIZATION TOOLS
FOR THE JAVA DEVELOPER



Copyright © 2021 by Micro Focus. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, Inc. 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 646-8600. Requests to Micro Focus for permission should be addressed to the Legal Department, Micro Focus, 700 King Farm Blvd, Suite 125, Rockville, MD 20850, (301) 838-5000, fax (301) 838-5034.

Micro Focus, Net Express, Server Express, Visual COBOL, COBOL Server, and Micro Focus Server are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. Other product and company names mentioned herein may be the trademarks of their respective owners.

The example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious. No association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred.

The book expresses the author's views and opinions. The information contained in this book is provided without any express, statutory, or implied warranties. Neither the authors, Micro Focus, Box Twelve Press, nor their resellers or distributors will be held liable for any damages caused or alleged to be caused either directly or indirectly by this book.

This document is provided "as-is". Information and views expressed in this document, including URL and other Internet Web site references, may change without notice. Some examples depicted herein are provided for illustration only and are fictitious. No real association or connection is intended or should be inferred.

ISBN: 978-0-578-79047-3

This book was set in Akzidenz-Grotesk by Box Twelve Press.



Contents at a Glance

CHAPTER 1	
Introduction	1
CHAPTER 2	
Visual COBOL and Eclipse	5
CHAPTER 3	
What is Visual COBOL for JVM?	19
CHAPTER 4	
A Short Guide to Procedural COBOL	29
CHAPTER 5	
An Example Application	41
CHAPTER 6	
A COBOL-Based REST Service	67
CHAPTER 7	
Automated Testing	99
CHAPTER 8	
User Interface Modernization	127
CHAPTER 9	
Containerizing COBOL Applications	155
CHAPTER 10	
COBOL and Microservices	185



Contents

CHAPTER 1

Introduction	1
What Is Visual COBOL for JVM?.....	1
Why Have We Written This Book?.....	2
Who Is This Book For?.....	2
Prerequisites.....	3
Downloading the Examples.....	4
Summary.....	4

CHAPTER 2

Visual COBOL and Eclipse	5
Hello World	5
Creating COBOL JVM Projects in Eclipse.....	7
Using Maven with Visual COBOL.....	10
Summary.....	18

CHAPTER 3

What is Visual COBOL for JVM?	19
COBOL Dialects.....	19
What is Managed Code?	20
COBOL Source Formats	21
The Visual COBOL Object Model.....	23
Summary.....	27

CHAPTER 4

A Short Guide to Procedural COBOL	29
COBOL Applications.....	30
Structure of a COBOL program.....	31
Procedure Division.....	34
Copybooks	38
Summary.....	40

CHAPTER 5

An Example Application **41**

Introducing the Example.....	41
Generating Example Data.....	59
Calling COBOL from Java.....	61
Summary.....	66

CHAPTER 6

A COBOL-Based REST Service **67**

The Application.....	67
The Interoperation Layer	70
The MonthlyInterest Class.....	85
Creating a REST Interface	88
Summary.....	97

CHAPTER 7

Automated Testing **99**

Strategies for Testing.....	99
Introducing MFUnit.....	101
Testing the BusinessRules Layer.....	107
Testing the Interoperation Layer	117
Testing the Application End-to-End.....	120
Summary.....	126

CHAPTER 8

User Interface Modernization **127**

UI Choices	127
The Credit Service UI Application.....	130
Getting Started with React.....	135
Summary.....	154

CHAPTER 9

Containerizing COBOL Applications **155**

Containerizing Applications for the Cloud	155
Changing from ISAM to a Database	161
Running the Revised CreditService Application.....	173
Containerizing the CreditService	175
Summary	183

CHAPTER 10

COBOL and Microservices **185**

Why Do I Need a Platform?	185
Kubernetes	187
Serverless Computing	199

Index**211**

Dedication

This book is dedicated to my Mum for being an inspiration and to my wife and daughter for their patience through all the weekends I missed with them while I wrote this.

Acknowledgements

This book would not have been possible without the expert help and assistance of the following people:

- The Micro Focus COBOL development team that builds and delivers such brilliant software
- Robert Sales, the father of Visual COBOL
- Mark Conway for being an advocate for these books
- Ed Airey for leading the project
- Scot Nielsen for assistance and advice
- The technical reviewers who checked the examples and kept me honest: Stephen Gennard, Guy Sofer, and Sezgin Halibov
- Box Twelve Press for editorial and publishing support

About the Author



Paul Kelly has worked at Micro Focus for more than 20 years. Paul worked on Visual COBOL for 10 years, initially on Visual Studio development, and later on Eclipse, before helping to develop a cloud-based SaaS offering. In 2017, Paul wrote *Visual COBOL: A Developer's Guide to Modern COBOL*. In his spare time, Paul plays the guitar and experiments with different ways of making coffee.

CHAPTER 1



Introduction

This chapter outlines the purpose of this book and its intended audience.

What Is Visual COBOL for JVM?

Visual COBOL for JVM is COBOL compiled to Java bytecode. Putting COBOL on the JVM (Java Virtual Machine) enables object-oriented extensions to COBOL so that you can code classes, objects, and methods, just as you can with Java. But you can also compile existing procedural COBOL code and run it on the JVM, too.

Visual COBOL does not convert COBOL source to Java source; it compiles COBOL source directly to JVM bytecode, so that all the intellectual assets in your COBOL source code base are retained. This is a unique feature of Visual COBOL for JVM—there are translators that will convert your code from procedural COBOL to Java, but machine-translated code is harder to work with in the future than the original human-authored source.

Once COBOL source is compiled for the JVM, you can easily integrate legacy code with newer environments. For example, you can run COBOL code directly on an application server, and use it as the back end to REST services provided by Java. The Java and the COBOL can interoperate directly because they are both running on the same JVM. You can also step seamlessly between one language and the other when you are debugging. COBOL is one of several languages that are able to run on the JVM, although COBOL can also be compiled as .NET code or native code. (When you compile to native code, the Visual COBOL extensions to the language are not available.)

You can use the Visual COBOL syntax to extend your legacy code and provide modern APIs to access it, using a language that produces artifacts easily consumed from Java, while being able to interoperate with the record structures that are found in COBOL business applications.

Chapter 4 explains the platform in more depth.

Why Have We Written This Book?

Micro Focus decided to follow up to *Visual COBOL—A Developer’s Guide to Modern COBOL* with a deeper dive into some topics. This book focuses solely on Visual COBOL for JVM as we are looking at different ways of deploying applications, something that is platform specific.

You don’t have to read *Visual COBOL—A Developer’s Guide to Modern COBOL* before reading this book—but you might find it useful to have it around as a reference, particularly if you are not familiar with Visual COBOL. There is some overlap between this book and the other one, but whereas the developer’s guide focuses attention on the Visual COBOL language, this book looks more closely at the practical issues of deploying COBOL applications on JVM platforms.

Who Is This Book For?

The primary audience for this book is Java developers who need to work with a legacy COBOL application. We’ve assumed knowledge of Java and the JVM, and familiarity with OO concepts. We’ve provided an introduction to the new syntax in Visual COBOL that enables object-orientation and also to the way data structures are defined in legacy procedural COBOL programs. This should be enough to understand the examples in this book and will help you with other legacy code you have to work with.

But it is not intended as a detailed primer on the COBOL language. If you want to learn more about Visual COBOL, *Visual COBOL—A Developer’s Guide to Modern COBOL* provides much more detail about the syntax that enables you to write classes and interfaces using a simplified and straightforward COBOL dialect. And there are many books available from which you can learn the traditional COBOL syntax used to create many line-of-business applications.

COBOL programmers who need to migrate existing applications to new platforms will also find a lot of helpful information in this book. However, if you don’t have any background with object-oriented concepts or Visual COBOL, I’d recommend that you work through some of the examples in *Visual COBOL—A Developer’s Guide to Modern COBOL* first.

Visual COBOL—A Developer’s Guide to Modern COBOL

Since I’ve mentioned *Visual COBOL—A Developer’s Guide to Modern COBOL* many times in this introduction, you might be wondering what it is and where you can get it. It was written by the same author as this book, and provides tutorial and reference information

on the Visual COBOL syntax, using worked examples. It also covers Visual COBOL for .NET—yes, you can compile exactly the same sources to either .NET or JVM, enabling cross-platform programming if required.

It is available as a free e-book from Micro Focus, or as a printed copy from Amazon (just search for Visual COBOL on the Amazon website). To get the free download from Micro Focus, go to <https://www.microfocus.com/campaign/visual-cobol-book/>.

Prerequisites

To be able to run the examples in this book, you will need Visual COBOL for Eclipse. This includes an Eclipse IDE with plug-ins that enable background parse, syntax assistance, compiling, and debugging of your COBOL code. This product is available for Windows and several Linux distributions, including SUSE and Red Hat. Not all Visual COBOL products include the Eclipse IDE—some are designed as server-only products, and there is also a line of products known as the Visual COBOL Development Hub, which provides command-line tools such as the compiler, but which can also be driven remotely from a Visual COBOL Eclipse on another machine.

Go to <https://www.microfocus.com/products/visual-cobol/> for an introduction to the range of Visual COBOL products. In addition to commercially licensed products from Micro Focus, you can also download Visual COBOL Personal Edition, which is free for noncommercial use (see <https://www.microfocus.com/en-us/products/visual-cobol-personal-edition/overview>).

The examples have been compiled, built, and run using Visual COBOL 4.0. You can use earlier product releases, but be aware that there might be particular syntax that will not always work with earlier versions of the product. The object-oriented features of the language are always evolving to make the language more productive. Backward compatibility is important to Micro Focus, so older code can always be compiled with later product versions, even if later code can't always be compiled with earlier product versions.

You will also need a JDK installed. The JDK version depends on the Visual COBOL product you are using, but at the time of writing, Visual COBOL 4.0 has a prerequisite of Oracle JDK 8—download and install the latest version of JDK 8 on your Visual COBOL machine.

You can run most of the examples on Windows or Linux; we have tested them on Windows and SUSE Enterprise Linux Desktop 12, service pack 2. Visual COBOL itself is tested on a wider number of platforms—at the time of writing, you can see the whole list at <https://www.microfocus.com/products/visual-cobol/tech-specs/>.

Downloading the Examples

All of the examples are available online and supplied with Eclipse project files so that you can import them directly into an Eclipse workspace. Some projects are used in more than one chapter—we would recommend that you use a separate workspace to run the examples from each chapter. To download them, go to <https://github.com/MicroFocus/visual-cobol-for-java-developers-book>.

Summary

This book is a primer to practical techniques for using Visual COBOL for JVM to build applications that extend existing COBOL code using modern frameworks such as Spring. It is aimed primarily at Java developers, but COBOL developers who have an understanding of Java and OO will also find it very useful.

For a comprehensive guide to the Visual COBOL language itself, see the sister book *Visual COBOL—A Developer's Guide to Modern COBOL*. You can get the free e-book from <https://www.microfocus.com/campaign/visual-cobol-book/>. Or you can get a printed copy from Amazon (just search for Visual COBOL on the Amazon website).

CHAPTER 2



Visual COBOL and Eclipse

This chapter is a short “getting started” for readers who haven’t used Visual COBOL before, but it also explains the use of Maven with Visual COBOL (something not yet covered in Micro Focus documentation). In this chapter, you’ll learn about:

- Creating and running “Hello World” in Visual COBOL
- Creating COBOL JVM projects in Eclipse
- Turning COBOL components into Maven dependencies so that they can be consumed more easily from Java

If you haven’t already done it, install a Java Development Kit (JDK) and Visual COBOL for Eclipse product (see the “Prerequisites” section in Chapter 1).

Hello World

The simplest Hello World we can write with Micro Focus COBOL is shown in Listing 2-1.



Before running any of the examples in this book, you must ensure you have a compatible Java Development Kit (JDK) installed as explained in the “Prerequisites” section of Chapter 1. On Linux, the PATH and JAVA_HOME environment variables must point to your Java installation before you run cobsetenv as described in the following steps.

Listing 2-1 *Hello World*

display “Hello World”

To run this program:

1. Create a text file called HelloWorldProcedural.cbl, and enter the code shown in Listing 2-1. Start the text in column 8 or later (different COBOL source formats are explained in the “COBOL Syntax and Source Formats” section in Chapter 3).
2. If you are running Visual COBOL on Windows, open a Visual COBOL command prompt (this is part of the Visual COBOL menu group). If you are running Visual COBOL on Linux, open a command prompt and run
 . cobsetenv
in the Visual COBOL bin directory (usually /opt/microfocus/VisualCOBOL/bin) before proceeding.

Change directory to the location where you created the file, and enter the compile command for Windows or Linux:

Windows:

```
cobol HelloWorldProcedural.cbl jvngen;
```

Linux:

```
cob -j HelloWorldProcedural.cbl
```

The compiler creates the file HelloWorldProcedural.class in the same directory as the source file.

3. Run it on Windows or Linux:

Windows:

```
java HelloWorldProcedural
```

Linux:

```
cobjrun HelloWorldProcedural
```

 On Linux platforms, you normally run COBOL programs (or Java programs that call COBOL) using the cobjrun command. This command sets up the COBOL run-time environment before invoking Java to run the bytecode; it also ensures UNIX signal handling will work correctly with the COBOL run-time. See the Micro Focus Visual COBOL documentation for more information about cobjrun.

On Windows, opening a COBOL command prompt (or running createnv.bat in your Visual COBOL installation directory) puts COBOL run-time files on the CLASSPATH—so you can run the .class file with the java command.

Writing Hello World as a Class

In the previous section, we wrote a Hello World program. It's very short compared with the Java equivalent. However, our program was just procedural COBOL, and we could have compiled it to native code and it would have executed the same.

Visual COBOL is able to compile procedural COBOL code so that it can compile existing legacy COBOL code. It does this by actually creating the bytecode for a class, so that to the JVM, it looks like a Java class, and running it creates a single instance of the synthesized class and then executes the program code. We'll look at some of the implications of this later in the book when we use COBOL RunUnits to run our code in environments such as application servers.

That's what the compiler does, but we can also choose to write "Hello World" as a class with a static method (see Listing 2-2)—which is the way "Hello World" is usually written in Java. The syntax looks different to Java, but there's a class identifier, a method identifier, and the method is marked as static. The parameters to the main method are an array of strings (the equivalent to `String[] args` in Java). However, "string" is a reserved word in Visual COBOL, but it is equivalent to declaring a `java.lang.String` in Java. There are a few classes that are represented as "native" types in Visual COBOL, which makes it much easier to write cross-platform code that runs on .NET or Java. See Chapter 14 in *Visual COBOL—A Developer's Guide* for more information.

Listing 2-2 *Hello World written as a class*

```
class-id HelloWorld.  
    method-id main(args as string occurs any) static public.  
        display "Hello World"  
    end method.  
end class.
```

You can compile and run this program the same way you did `HelloWorldProcedural` in the previous section.

Creating COBOL JVM Projects in Eclipse

We are going to assume that you have some familiarity with Eclipse already, and just introduce the plug-ins used for working with COBOL. As you will learn later in the book, Visual COBOL supports compilation to either native code or Java bytecode. You can create projects for either type of executable in Eclipse, but in this book, we are only really concerned with compiling to JVM.

Understanding the COBOL Perspective in Eclipse

On Windows, you can start Eclipse from the Start menu. The Micro Focus menu group for Visual COBOL might be named Micro Focus Enterprise Developer, Micro Focus Team Developer, or Micro Focus Visual COBOL depending on which product you have installed. The menu group will have an icon to start Eclipse.

On Linux systems, there may be a Micro Focus COBOL icon on the desktop to start Eclipse, or you can open a command prompt and use the cobsetenv command to set up a COBOL environment (see the “Hello World” section earlier in this chapter). Then, you can enter the `eclipse` command to start Eclipse for Visual COBOL.

Once Eclipse is running and you have selected or created a workspace, you can open the COBOL Perspective. You can find it from the Eclipse menu: **Window > Perspective**

> Open Perspective > Other. Once you've opened the COBOL Perspective once, there is also a toolbar icon (the blue-bordered “CBL” box) to switch back to it after you've been using a different perspective.

The COBOL Perspective is very useful for working with COBOL projects. It opens the COBOL Explorer window. This looks a little like the Project Explorer, but has a COBOL-centric perspective; it only shows COBOL projects.

If you right-click on a COBOL project and select Properties, you'll see a Micro Focus group on the left side of the Properties window (see Figure 2-1). This has all the settings for a COBOL project. You can see these properties from the COBOL Explorer or the Project Explorer, but you can't see them from the Package Explorer.

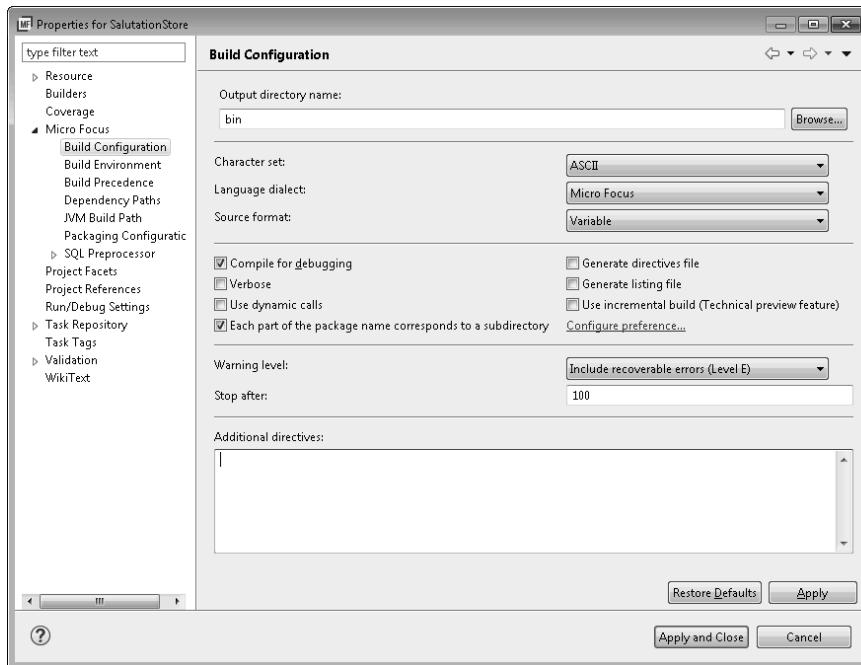


Figure 2-1 Micro Focus group in the Build Properties window

Creating and Running JVM Hello World in Eclipse

To create a COBOL JVM project for Eclipse:

1. Open the COBOL Perspective as explained in the previous section.
2. Click **File > New > COBOL JVM Project**. (If you didn't select the COBOL Perspective, **COBOL JVM Project** doesn't appear on the menu after you click **New**, but you can still find it by clicking **Other** and filtering the Wizards list by COBOL.)
3. Enter **HelloWorld** as the project name. By default, Eclipse will create your project under your workspace directory, but you have the option of deselecting **Use default location** and putting the project somewhere else.
4. Click **Finish**.

You now have an empty project. If you use the COBOL Explorer to look at it, you will see a **src** directory as well as folders for the COBOL JVM Runtime System and the JRE. The project structure is similar to the structure of an Eclipse Java project.

Next, to add some code to the project:

1. Click **File > New > COBOL JVM Class**. This opens the **New COBOL JVM Class** dialog box. It looks very similar to the dialog box for creating a new Java class.
2. Enter **HelloWorld** as the name and click **Finish**.
3. Expand the **src** folder in the COBOL Explorer and you can see the default package and below it **HelloWorld.cbl**. The wizard has created some default template code inside **HelloWorld.cbl**, but you can delete all of it and replace it with the code in Listing 2-2, and then save your changes.

The Eclipse default is to build projects automatically each time a source code change is saved, so the console window should show (among other things) the message "BUILD SUCCESSFUL".

Finally, to run Hello World:

1. Right-click on the **HelloWorld** project in the COBOL Explorer.
2. Click **Run As > COBOL JVM Application**.
3. A Select COBOL JVM Application dialog box opens. Select **HelloWorld** from the list and click **OK**.

You should see **Hello World** in the Eclipse console. If you click **Run > Run Configurations**, there is now an Eclipse run configuration for **HelloWorld**.

Using Maven with Visual COBOL

Maven has become a de facto standard for representing the dependencies of Java projects. Other popular build engines such as Gradle will import Maven POM files and will consume dependencies stored in Maven repositories. The easiest way to consume COBOL dependencies in any sizable Java project is to make its artifacts available through Maven.

Visual COBOL doesn't yet have native Maven support, but we will show you that it is relatively easy to make a COBOL project consumable from a Java Maven project.

A Quick Introduction to Maven

This section is a 30-second introduction for any readers who haven't used Maven. A Maven project is defined by a file called pom.xml in the project's root directory. This file is the Project Object Model (POM). The POM file defines the artifact to be built and provides an identifier and version number.

The identifier consists of a group ID and an artifact ID. Group IDs are constructed rather like Java package names and are unique to an organization defining them. For example, many Apache libraries are defined under the group ID "org.apache.commons." The group ID and artifact ID between them provide a global unique identifier for a library. The version number enables a library dependency to be fixed to a particular version.

The POM defines all the dependencies required to build the project using the group and artifact IDs and, optionally, the version number. When you build a Maven project, it fetches all the dependencies from a repository.

Where is the repository? Maven actually relies on a hierarchy of repositories. It first searches the local repository on the machine where Maven is running (this is in a directory called .m2 in the user's home directory). If it can't find the dependency there, it fetches it from a remote repository. By default, this is at <http://repo.maven.apache.org>, but you can configure other repositories. Organizations often configure internal repositories to store artifacts on their own servers.

Maven copies artifacts fetched from remote repositories to your own local repository the first time they are requested, which makes subsequent builds much faster. Figure 2-2 shows the relationship between Maven, the POM file, and repositories. Maven reads the POM file to get the build instructions and dependency list. It fetches dependencies from the local repository if they are cached, or from one or more remote repositories, depending on how your Maven system is configured. It then orchestrates the build process, producing a final build artifact (which is also cached to your local repository).

POM file dependencies are transitive; each dependency fetched from a repository will have a POM file that lists its own dependencies, and these are also fetched as needed.

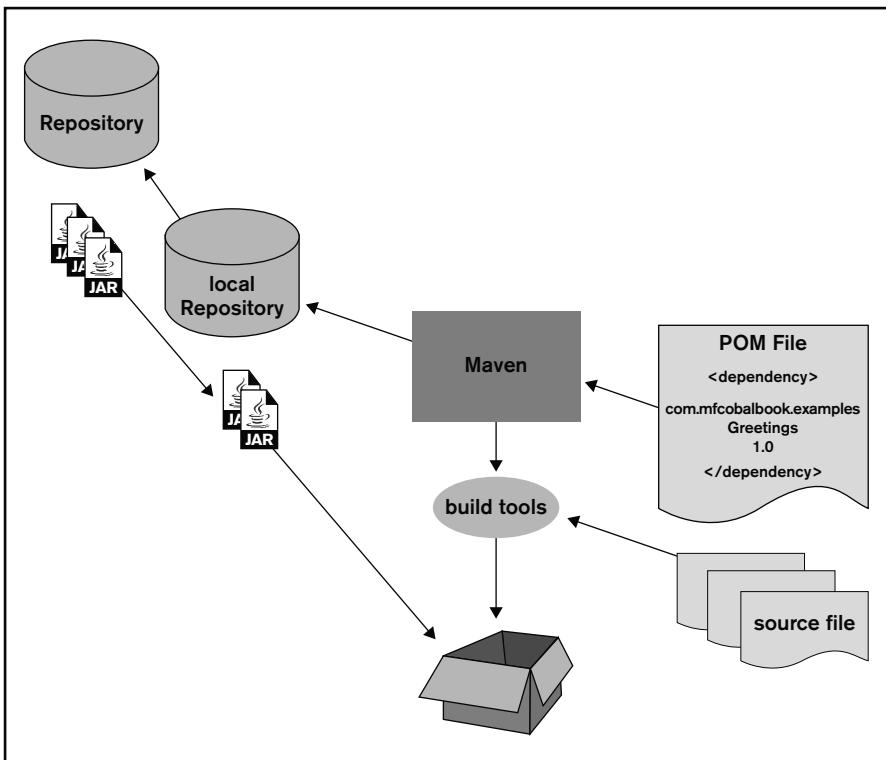


Figure 2-2 Maven build

Installing Maven

The copy of Eclipse installed with Visual COBOL already has support for Maven projects built in, but you will need to install your own copy of Maven to work with the examples in this book.

The home of Maven at the time of writing is <https://maven.apache.org>. You can find instructions here to install it on Windows or on Linux. Many Linux distributions will also make Maven available through their own package managers. If you are running SUSE Enterprise or OpenSUSE, you can find Maven packages for installation through <https://software.opensuse.org/package>.

Once you have Maven installed, make sure it is working by typing:

```
mvn -version
```

You should see a version label for Apache Maven, as well as some other information listing JRE, locale, and operating system.

Creating a Maven Java Project with a COBOL Dependency

We will now set up a Java project with Maven and have it rely on a COBOL dependency.

The application is a multilingual “Hello World” that fetches strings from a COBOL program. First, we’ll create the COBOL component and ensure that it adds itself to the local repository every time it is built.

Then, we’ll create a Java project and include the COBOL component as a dependency. Finally, we’ll build and run it to show that everything is working. This is a technique we are going to use throughout the book.

The Project Structure

We are going to have a separate COBOL and Java project that together will make our *HelloWorld_II* application. Before you start, create a folder called *HelloWorld_II* where you will put the two other projects. Wherever you see *HelloWorld_II* in the following, put in the full path to this folder.

Adding a COBOL Component to the Repository

To create a COBOL JVM project in Eclipse and package it as a .jar file:

1. Open the COBOL Perspective and click **File > New > COBOL JVM Project**.
2. Enter **SalutationStore** as the project name.
3. Deselect the **Use default location** check box and enter a location of *HelloWorld_II/*SalutationStore. Click **Finish**.
4. Right-click on the **SalutationStore** project and click Properties.
5. Expand the Micro Focus group and click **Packaging Configuration**.
6. Check these two check boxes:
 - **Create packaging target in the build script**
 - **Create JAR file after build**
7. Click **Apply and Close**.

Every time this project builds, it creates a .jar file in the dist subdirectory. Now, we are going to add a custom build step to the project to store it in the repository every time. But first, we are going to create a POM file to describe the artifact we are storing. We don’t have to do this, but it makes it easy to manage the COBOL JVM run-time system as another dependency. This means we also need to add the COBOL JVM run-time system to our repository.

There are a number of .jar files in the COBOL JVM run-time system; however, we add two of them for this example. They need to be identified to Maven with a group ID, artifact ID, and version number. The two .jar files to add are below (Maven identification in parentheses):

- mfcobol.jar (com.microfocus.cobol.rts, mfcobol, 4.0.0)
- mfcobolrts.jar (com.microfocus.cobol.rts, mfcobol, 4.0.0)

I've picked 4.0.0 as the version number because that's the product version used while writing this book—you should use the version number of the product you have installed.

Open a command prompt or terminal, and navigate to C:\Program Files (x86)\Micro Focus\Visual COBOL\bin if you are on Windows, or /opt/microfocus/VisualCOBOL/lib if you are on Linux. The exact path depends on the product you have installed—it's Visual COBOL if you've installed Visual COBOL, or Enterprise Developer if you've installed Enterprise Developer.

Then, run these two commands:

```
mvn install:install-file -Dfile=mfcobolrts.jar  
-DgroupId=com.microfocus.cobol.rts -DartifactId=mfcobolrts  
-Dversion=4.0.0 -Dpackaging=jar  
  
mvn install:install-file -Dfile=mfcobol.jar  
-DgroupId=com.microfocus.cobol.rts -DartifactId=mfcobol  
-Dversion=4.0.0 -Dpackaging=jar
```

These two files are now in your local repository, and can be included as dependencies in any build. If we have a number of COBOL components in an application, they will all reference the COBOL RTS, but at build time, Maven recognizes that it is the same dependency in a number of components and includes it once in the final build output.

Adding the Code

To add the code to the COBOL project:

1. Right-click on the src directory in your SalutationStore project and add a new COBOL class to your project, with the package name **com.mfcobolbook.examples** and class name **SalutationStore**.
2. Replace the template code with the code in Listing 2-3.

Listing 2-3 Salutation store code

```
class-id com.mfcobolbook.examples.SalutationStore public.
01 greetingsDictionary          dictionary[string, string].
method-id new.
    invoke initData()
end method.
method-id initData.
    create greetingsDictionary
    write greetingsDictionary from "Hello World" key "en"
    write greetingsDictionary from "Bonjour le monde" key "fr"
    write greetingsDictionary from "Hallo Welt" key "de"
end method.

method-id fetchGreeting(language as string)

    returning result as string.
set language to language::toLowerCase()
read greetingsDictionary into result key language
    invalid key
        set result to "I don't speak this language"
    end-read
end method.
end class.
```

Adding a Custom Builder

We'll add a custom builder to the SalutationStore project to ensure that the SalutationStore is added to the repository each time it is built. First, we need a POM file to describe the artifact and its dependencies. Add a file called pom.xml (see Listing 2-4) to the root of the SalutationStore project.

Listing 2-4 POM file for SalutationStore

```
<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>

  <groupId>com.mfcobolbook.cobol</groupId>
  <artifactId>SalutationStore</artifactId>
```

```
<version>0.0.1-SNAPSHOT</version>

<packaging>jar</packaging>

<properties>
    <cobolVersion>4.0.0</cobolVersion>
</properties>

<dependencies>
    <dependency>
        <groupId>com.microfocus.cobol.rts</groupId>
        <artifactId>mfcobol</artifactId>
        <version>${cobolVersion}</version>
    </dependency>
    <dependency>
        <groupId>com.microfocus.cobol.rts</groupId>
        <artifactId>mfcobolrts</artifactId>
        <version>${cobolVersion}</version>
    </dependency>
</dependencies>
</project>
```

To add the custom builder:

1. Right-click on the **SalutationStore** project, click **Properties**, and then click **Builders** (it's near the top of the Properties window).
2. Click **New**, and then from the **Choose configuration type** dialog box, select **Program** and click **OK**.
3. A dialog box opens to edit a launch configuration. Give it the name `MvnInstall_SalutationStore`.
4. You now need to set the **Location** and **Working Directory** fields. These are different on Windows and Linux, and will also depend on where Maven is installed on your system. We'll refer to this as MAVEN_HOME.

Windows:

Set the Location to MAVEN_HOME\bin\mvn.cmd

Set the Working Directory to MAVEN_HOME

Linux:

Set the Location to /usr/bin/mvn.

5. Now, set the arguments. This is the same on all platforms:

```
install:install-file -Dfile=${build_project}/dist/SalutationStore.jar
-DpomFile=${build_project}/pom.xml
```

6. Finally, set when the builder should be run. Click the **Build Options** tab, and under Run the builder, select only the two check boxes for **During manual builds** and **During auto builds**.

Now, every time the Eclipse builds our project, it creates a .jar file (we added the instructions to this to the project in the previous section). Then, the custom builder we created above will put the .jar file in the local repository, storing it with the artifact ID, group ID, and version number we put into pom.xml.

You can check that this is working correctly by going to the Maven repository directory under your user directory (C:\users\username\.m2\repository on Windows or /home/username/.m2/repository on Linux), and then going down the folder path to com, mfcobolbook, examples till you find the directory for the version of the artifact specified in the POM file.

You can also look down com, microfocus, cobol, rts, to ensure the COBOL run-time .jar files we added earlier are present. If not, check the procedure in the “Adding a COBOL Component to the Repository” section, and watch for error messages when running the Maven commands.

Consuming the Component from Java

To create a new Maven Java project in Eclipse and add our COBOL artifact as a dependency:

1. From Eclipse, open the Package Explorer view (Java projects aren't visible in the COBOL Explorer view).
2. Click **New > Other** and then type **maven** into the Wizards field.
3. Select **Maven Project** and click **Next**.
4. Select the **Create a simple project** check box and click **Next**.
5. Deselect the **Use default Workspace location** check box and enter a location of *HelloWorld_///MultiLingualHelloWorld*.
6. Click **Next**.
7. Fill in the fields on the New Maven project dialog box as follows:
Group ID: com.mfcobolbook.examples.java
Artifact ID: MultilingualHelloWorld
Version: .0.0.1-SNAPSHOT
Packaging: jar

Leave the other fields blank and click **Finish**.

8. Eclipse creates the MultilingualHelloWorld project, which contains src and target folders, and a pom.xml file in the project root.

9. Open pom.xml to add the COBOL dependency to the pom.xml file. Eclipse opens POM files in a multitabbed editor, with an Overview pane opened by default. I find it easier to work directly with the XML—so open this by clicking the right-hand tab at the bottom of the editor pane.
10. Add a <properties></properties> element to the file before the closing </project> tag, to set the Java version to 1.8 (this is the version for Java 8). Otherwise, Maven defaults to Java 1.5, and this will be set as the JRE on the Eclipse build path for the project.
11. Add a <dependencies></dependencies> element to the file before the closing </project> tag, and add a dependency for SalutationStore. The complete file is shown in Listing 2-5.
12. Add a Java class with the name MultiLingualHelloWorld and package com.mfcobolbook.java. Add the code in Listing 2-6.
13. You need to update your Eclipse project with the changes you made to pom.xml before the Java project will build cleanly. Right-click **MultiLingualHelloWorld** and click **Maven > Update project**.

***Listing 2-5** POM file for Java component*

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>
    <groupId>com.mfcobolbook.examples.java</groupId>
    <artifactId>MultiLingualHelloWorld</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <properties>
        <maven.compiler.source>1.8</maven.compiler.source>
        <maven.compiler.target>1.8</maven.compiler.target>
    </properties>
    <dependencies>
        <dependency>
            <artifactId>SalutationStore</artifactId>
            <groupId>com.mfcobolbook.cobol</groupId>
            <version>0.0.1-SNAPSHOT</version>
        </dependency>
    </dependencies>

</project>
```

Listing 2-6 code for MultiLingualHelloWorld class

```
package com.mfcobolbook.java;
import com.mfcobolbook.examples.SalutationStore;
public class MultiLingualHelloWorld
{
    public static void main(String[] args)
    {
        SalutationStore store = new SalutationStore();
        System.out.println(store.fetchGreeting("en"));
        System.out.println(store.fetchGreeting("fr"));
        System.out.println(store.fetchGreeting("de"));
        System.out.println(store.fetchGreeting("it"));
    }
}
```

Running the Application

We can now run our Java application. To run it from Eclipse, right-click the MultiLingualHelloWorld project and click **Run as Java Application**. Select MultiLingualHelloWorld from the dialog box and click OK. If everything is working, you'll see the output:

```
Hello World
Bonjour le monde
Hallo Welt
I don't speak this language
```

Summary

In this chapter, we started by building and running the simplest possible COBOL Hello World from the command line. We finished with an application based on a Java project that calls a COBOL project, with all the dependencies managed through Maven. This is a technique we'll be using throughout the book—COBOL business logic that is called from Java. Having Maven manage the dependencies is a little more work when we are setting projects up, but simplifies things greatly in the long run. The other important build system for Java, Gradle, will also consume Maven dependencies, so if your build system is Gradle based, you can use these techniques here as well.

We'll be using this method to turn COBOL components into Maven artifacts throughout this book, for example when we create a REST service using Spring Boot.

CHAPTER 3



What is Visual COBOL for JVM?

This chapter explains how Visual COBOL and the JVM fit together. It should be useful to anyone who hasn't yet had much exposure to Visual COBOL, whether they are a Java or a COBOL programmer. It will help COBOL programmers to understand this chapter if they have some basic Java knowledge already.

In this chapter, you'll learn about:

- COBOL dialects
- The Micro Focus compiler and managed code
- COBOL source formats
- The Visual COBOL object model

COBOL Dialects

With more than 60 years of history, it's no surprise that there is more than one dialect of COBOL in use. Visual COBOL is relatively new as Micro Focus started experimenting with it in about 2002, as a way of using the .NET Common Language Runtime (CLR) to provide object-oriented features.

It has matured rapidly, and is available for the Java Virtual Machine (JVM) as well as the .NET framework and .NET core. However, the Visual COBOL compiler can compile any dialect supported by Micro Focus to run in these environments, providing an easy route to modernizing COBOL applications.



For the rest of this book, all references to .NET and the .NET CLR apply to the .NET framework (Windows only) and .NET Core (cross-platform). Visual COBOL 5.0 supports .NET Core as well as the .NET Framework.

The Micro Focus compiler supports most COBOL dialects; as well as standards ANSI 74 and ANSI 85, it can also compile source code written for most of the major mainframe vendors over the last few decades. Compiler directives enable you to set a particular dialect. A full discussion of dialects and the differences between them is beyond the scope of this book. All the procedural code in this book has been written using the Micro Focus dialect, which enables structured code and does not require all the divisions demanded by some COBOL dialects.

This chapter serves as a lightning guide to the Visual COBOL syntax extensions, but is not a comprehensive guide or tutorial. For that, you should read *Visual COBOL – A Developer’s Guide to Modern COBOL*, which is available as a free e-book from Micro Focus or as a printed copy from Amazon (just search for Visual COBOL on the Amazon website). For the free download from Micro Focus, go to <https://www.microfocus.com/campaign/visual-cobol-book/>.

What is Managed Code?

The Java Virtual Machine (JVM) is a processor that is defined in software rather than hardware. The .NET Common Language Runtime (CLR) is another such virtual machine. When you compile a Java program, the output is Java byte code that is executed by the JVM rather than directly on the CPU of a computer.

We often refer to byte code as “managed code;” the term refers to code for either the CLR or the JVM. Although the JVM and CLR have different specifications and implementations, there are a number of things they have in common. Managed code executable files contain metadata as well as executable code and the environments they run on provide automatic memory management, freeing up memory that is no longer in use (garbage collection).

The metadata describes the executable code. For example, a .class file not only contains all the executable code for the class, it also contains data that identifies the class, all the methods and their signatures, and the fields in the class. By contrast, a native code file consists only of executable code and callable entry points, with no metadata to provide extra information about how the code should be called.

The Micro Focus COBOL compiler can compile COBOL code to either native or managed code. But the object-oriented features of Visual COBOL are only available when you compile to managed code – because they rely on the run-time environment of either the JVM or the .NET CLR. Figure 3-1 shows the relationship between older COBOL dialects, Visual COBOL, managed and native code.

The subject of this book is COBOL compiled to Java byte code and running on the JVM. When COBOL code is compiled to Java byte code, it can run seamlessly inside applications with Java code (or other languages compiled to JVM, like Scala or Kotlin). Visual COBOL provides an excellent way of interfacing between older legacy code and modern object-oriented code; it enables you to target modern cloud platforms as your run-time environment.

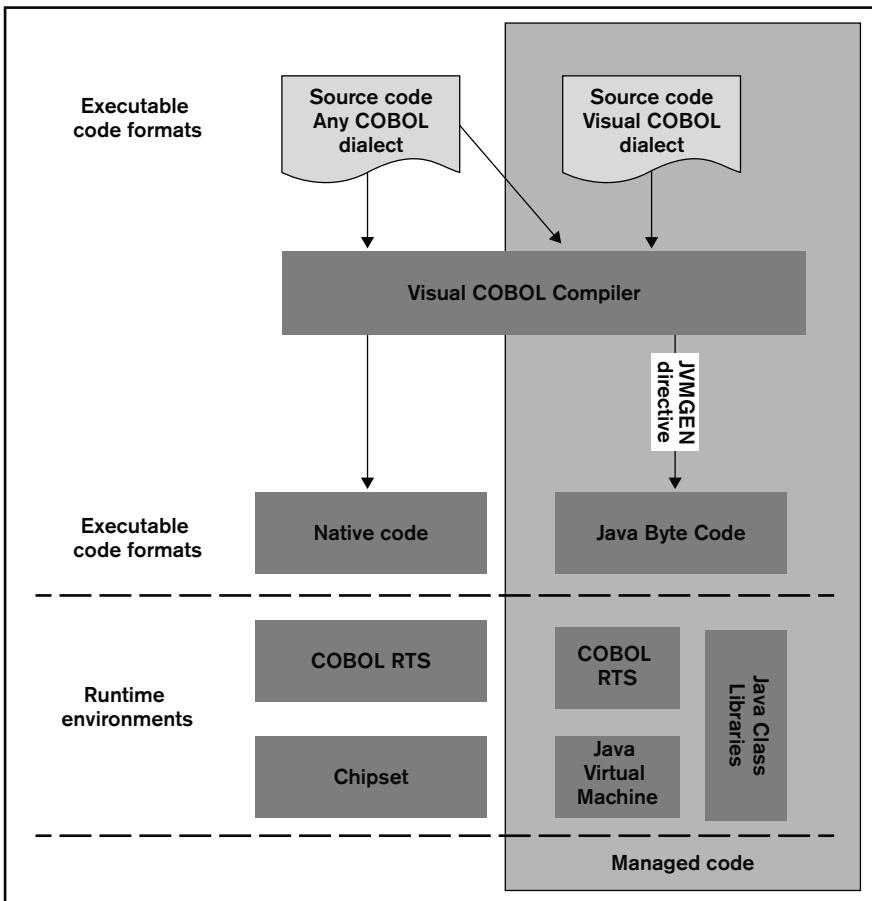


Figure 3-1 Source code dialects and run-time environments

COBOL Source Formats

COBOL dates back to the era of punch cards, which is why traditionally COBOL programs had source lines that went from character position 8 to position 72. The first 6 characters allowed line numbers, which were useful if someone dropped a deck of punched cards or got them out of order. The ANSI 85 COBOL standard allows alphanumeric characters in columns 1-6. Column 7 is an “indicator” column, used to denote comments or compiler directives.

In Micro Focus COBOL this is known as *source format fixed*. The Micro Focus compiler also allows *source format variable* and *source format free*. Source format variable allows source lines up to 255 bytes long, but character positions 1 to 7 have the same meanings as in source format fixed. Source format free does not assign any special meaning to columns 1 to 7, although some characters in column 1 will be treated as the column 7 indicator characters in fixed and variable format. In all COBOL formats statements can span several lines. There is no end of statement marker like the semicolon in Java, although some verbs have an optional explicit end token (ANSI 85 COBOL and later). For example:

```
READ  
...  
END-READ
```

A read statement can include a number of clauses, so END-READ makes the code easier for people to understand.

Source format variable is the format used for most examples shipped with Visual COBOL and is the one we've adopted throughout this book. However, to use the space available for code more efficiently, columns 1 to 6 aren't shown in the listings in the book. As we aren't using line numbers in the examples, these columns are always white space.

The sourceformat directive sets the source format for the compiler. In Visual COBOL it is usually set at the project level through the Build Properties dialog, but you can override it for a particular source file by including the sourceformat directive. For example:

```
$set sourceformat(free)
```

The \$ must be in the column 7 for source formats fixed and any column preceded only by spaces for variable. The sourceformat directive takes effect on the first line following it. You can change source format more than once in a source file, which isn't generally recommended unless you have to deal with copybooks in different source formats. We will discuss copybooks in more detail in the next chapter.

 The American National Standards Institute (ANSI) ratified three standard versions of COBOL, ANSI, ANSI 74, and ANSI 85. There have been many vendor-specific dialects over the years. The Micro Focus COBOL compiler can compile most COBOL source code into JVM byte code.

Comments

In source format fixed and source format variable, an asterisk in column 7 indicates that the rest of the line is a comment. In source format variable *> anywhere that the rest of the line is a comment. Listing 4-1 shows some COBOL code with two comments.

Listing 4-1 Comments in a COBOL program

```
* This is a "hello world" program  
display "Hello World" *> the display statement prints to console
```

Literals

A string literal is a sequence of characters between double quotes or single quotes. You can concatenate two strings using the & operator.

Case Sensitivity

Micro Focus COBOL is not generally case-sensitive; keywords can be written in uppercase or lowercase. Variable names are also not case-sensitive. Entry point and program names are not case-sensitive on Windows, but they are case-sensitive on Linux/Unix. The case/nocase directive enables you to force a particular behavior for code portability.

However, Visual COBOL has to work with the JVM, which *is* case-sensitive, so method names, class names, and properties are treated in a case-sensitive way. Field names are not case-sensitive when referenced directly within the class or program in which they are declared. But if you reference a field using the self identifier, it is treated as case-sensitive.

The Visual COBOL Object Model

The Visual COBOL object model is the same as Java. You can define class types, interfaces and enumerations. A class inherits from a single parent, and all classes are ultimately descended from the root class, `java.lang.Object`. Classes can implement any number of interfaces. The Visual COBOL compiler follows the same rules for type matching as Java.

Visual COBOL also enables you to define a value-type. This is included for compatibility with .NET code. In .NET, numerics and Booleans are all value-types. When you declare a data item that is a value-type the storage is allocated where the item is declared – you only use the new operator when using a constructor with parameters and you work with the data directly rather than with a reference to it.

In Java, numerics and Booleans are primitives (in effect, the same as value types), but there is no facility to define additional primitives either in the Java language or the JVM. COBOL duplicates the assignment and copy semantics of value types, but in reality when you define a value type in Visual COBOL for JVM, the compiler defines a class in your byte code.

Visual COBOL also has methods and fields as Java does, and you control access to them with the `public`, `private` and `internal` modifiers (`internal` is the equivalent of no access modifier in Java). However, Visual COBOL treats methods as public by default and fields as private by default.

Visual COBOL also enables you to define properties, which is another borrowing from .NET. Java only has properties by convention (methods that start with `set` or `get` are treated as properties by most Java tools). You can define and access properties in Visual COBOL as properties. A COBOL property will appear to a Java program as separate get and set methods.

Outline of a Visual COBOL Class

Listing 3-2 shows a simple class that represents a circle. The `Circle` class demonstrates a number of features of Visual COBOL syntax. The class header defines the package and the class name, and is followed by the fields.

Listing 3-2 Simple outline class

```
*> Class header identifies namespace and class name
class-id com.mfcobolbook.examples.Circle.

*> fields. All are "private" unless marked otherwise
    78 PI                      value 3.1415926.
*> SHARED-DATA is a constant available inside and outside
*> the circle class
    01 SHARED-DATA            float-short value PI
                               static public initialize only.
*> Field exposed as a read-only property. The field is private
*> but the read-only property is public by default.
    01 radius                  float-short property with no set.

*> Constructor. Public where there is no access modifier.
method-id new (radius as float-short).
    set self::radius to radius
end method.

property-id Circumference float-short.
getter.
    set property-value to radius * SHARED-DATA * 2
end property.

method-id calculateArea() returning result as float-short.
    set result to radius * radius * SHARED-DATA
end method.
```

```
method-id main(args as string occurs any) static public.  
    declare aCircle = new Circle(5)  
    display aCircle::Circumference  
end method.  
  
end class.
```

The convention of each field being declared with a number at the start will look odd to Java programmers but will make more sense when we look at group-items in Chapter 4. A data item preceded by 78 is a constant and can be any type of numeric or string literal.

Visual COBOL contains data types that match the primitives available in the JVM. For example, float-short is equivalent to Java's float. Our Circle class also makes use of properties and shows two different ways of declaring them. The first is to simply append the property clause to the end of the declaration – radius is both a field and a property. This particular property clause includes the with no set phrase, which means the property can be read but not modified from outside the class.

We've also defined a property the other way, using a property-id header. This enables you to provide a property that executes code. Again, this property only has a getter. If you wanted to add a setter as well, you would include the setter heading before the end-property.

We've also included a method – calculateArea(). This does not take any arguments but returns a value. This could have been a property, too, but was written this way to show a method that returns a value.

The constructor is the new method at the beginning of the class. The constructor is always called new. Just as in Java, you can have overloaded constructors that have different method signatures.

The main() method at the end of the class is a static public method that takes an array of strings as its argument. This makes the class runnable with the java command. The first statement in this method declares local variable aCircle. The declare verb can use type inference to work out the type of the variable from the right side of the assignment – in this case, the construction of a new Circle object. You can also explicitly set the type. For example:

```
declare anotherCircle as type Circle
```

This declares an uninitialized variable anotherCircle, which is of type Circle. The display statement prints the Circumference property to the console.

Packages

Visual COBOL has packages or namespaces like Java. A class declares its package name in the `class-id` header along with the class name. When you want to use a class from another package you can either:

- Fully qualify the class name with the package name, as you would in Java
- Import the package name with the `ilusing` compiler directive

You can set `ilusing` either for the project or at the start of an individual source file. When you set `ilusing` on a source file, it applies only to that source file. If you set it on a project, it applies to all files throughout the project. By default, all projects are set to import the `java.lang` package – but if you are compiling short examples like the ones in this chapter from the command line, include the following at the top of the sourcefile (don't forget the `$` must go in column 7):

```
$set ilusing(java.lang)
```

Java only allows you to define one class per source file and requires that source files are laid out in a directory structure that matches the package name. The Visual COBOL compiler does not require this by default, but the `jvmsourcebase` directive introduced in Visual COBOL 4.0 enforces source file path and name requirements that are the same as Java.

New projects created in Eclipse have `jvmsourcebase` set by default, although you can turn it off by unchecking **Package to subdirectory matching** in the build configuration.

Regardless of the setting of `jvmsourcebase`, the Visual COBOL compiler will always generate `.class` files in a directory structure that matches the package name, just like the Java compiler.

Exceptions

Visual COBOL has `try...` `catch...` `finally` blocks and can throw exceptions using the `raise` verb. Listing 3-3 shows a short program that throws an exception, catches it, and re-raises it. The `finally` block is always executed after code in the `try` block, regardless of whether there was an exception.

Listing 3-3 Code that raises and catches an exception

```
class-id com.mfcobolbook.examples.RaiseClass public.

method-id Main (args as string occurs any) static public.
    try
        raise new Exception()
    catch e as type Exception
        *->      Log the exception
                  display e::getMessage()
```

```
*>      Rethrow this exception
       raise
finally
      display "We always execute this"
end-try
end method.

end class.
```

Visual COBOL does not provide exception checking. Unlike Java, methods do not define the list of exceptions they might throw. This mostly simplifies your code, but there is one case where it can cause problems: when you have Java code calling Visual COBOL code, the Java compiler will complain when you try to catch a checked exception from COBOL. The Java compiler can't see a list of checked exceptions declared on the COBOL method, so it marks it as an error.

The simplest workaround for this is to use `RuntimeException` as the base class for implementing exception classes in COBOL. Because these are unchecked exceptions, you can put them in Java catch blocks without the compiler errors.

Summary

In this chapter, we briefly looked at the Visual COBOL language and the similarities and differences between it and Java. In the next chapter, we will look at procedural (non-object oriented) COBOL, which is the kind of code that the majority of COBOL applications written over the last 40 years are written in.

CHAPTER 4



A Short Guide to Procedural COBOL

This chapter is aimed at the Java programmer who does not yet know much about procedural COBOL; the kind of code found in COBOL applications that need modernizing. In this chapter, you'll learn:

- An overview of COBOL applications
- A brief explanation of how Visual COBOL solves the problem of modernizing these applications so they can be run in environments like application servers or as Spring Boot applications
- A brief outline of a procedural COBOL program
- Datatypes and records in procedural COBOL
- Copybooks

Although we are referring to “procedural COBOL” here, it is only to emphasize the difference between COBOL and Visual COBOL. Visual COBOL is a superset of COBOL that can compile any COBOL program but includes full support for object-orientation when the target platform is either the JVM or .NET CLR. Throughout the rest of this chapter we will simply refer to “COBOL”.

The goal of this chapter is to help you to read COBOL code rather than to write it. This will help you when you start modernizing applications.

COBOL Applications

COBOL applications consist of one or more COBOL programs. A COBOL program looks very different to a Java class. All the variables in a COBOL program are declared at the start in the `working-storage` section. Working-storage is allocated when the program is loaded. One program can load another by calling it (with the `call` verb). Working-storage is only deallocated when the application ends (with a `stop run` statement) or when the program is cancelled with the `cancel` verb.

A COBOL program can be called with arguments and these arguments can be passed either by value, by reference, or by content (this is explained in the section “Program Parameters and Entry Points”). Any parameters must be declared in the `linkage` section.

Some dialects of COBOL (including Micro Focus COBOL) enable re-entrant code (code that can be called more than once concurrently). These dialects allow you to declare variables in the `local-storage` section. There is a new allocation of local-storage on the stack each time the program is called.

The ANSI 85 standard introduced structured programming to COBOL with block structures like `if...else...end-if`, `perform... end-perform`, and `evaluate... end-evaluate` (similar to the `switch` statement in Java).

COBOL has some very important strengths as a data processing language. It implements high-precision decimal arithmetic (up to 38 places), which is important when the numbers you are crunching represent somebody else’s money. COBOL code for carrying out financial calculations is much easier to understand than the equivalent in Java, where you will have to use the `BigDecimal` class rather than just writing simple arithmetic expressions.

COBOL is also very efficient at representing data records, allowing the programmer to create a record structure that resembles the way it will be stored in a file. COBOL implementations support a file type known as Indexed Sequential Access Method (ISAM), originally invented by IBM. Each record consists of a number of fields, one of which must be a primary key. Optionally, other fields can be designated as secondary keys. ISAM enables random access to records via primary or secondary keys.

Although ISAM has largely been supplanted by relational databases (and COBOL also has support for SQL), it is still in use for some applications and can offer very high performance for batch applications that have to process large numbers of records in a short time. Micro Focus also provides tooling that enables you to migrate your ISAM data to a relational database while still accessing it using the existing code.

Modernizing Applications with Visual COBOL

The object-oriented Visual COBOL code we looked at in the previous chapter represents a significant set of enhancements to the traditional COBOL that is found in most existing applications. However, the Visual COBOL compiler can compile existing COBOL

applications to run on the JVM and Visual COBOL classes and COBOL programs can interoperate seamlessly with each other. There are three problems you need to solve when using existing COBOL code to provide functionality to a Java application.

The first problem is the most basic one – how can I call a COBOL entry point from Java? This is solved by the Visual COBOL compiler compiling your existing programs as Java byte code. Because it is byte code rather than native code, you don't need to use Java Native Interface (JNI) to call it and you can also run it inside environments that will not support calling native code from Java.

The second problem is that the record structures defined in COBOL don't make much sense to Java. As we'll see later in this chapter, COBOL programs often describe complex record structures and lay them out byte by byte. This second problem is solved by the Visual COBOL language itself. You can declare the same record structures in a Visual COBOL class that you can in a COBOL program and access the individual fields so that you can remap them to fields and data types a Java program can work with. Visual COBOL enables you to create an API that is easily consumed by Java applications. There is also a Visual COBOL compiler feature known as "smart linkage" that automatically creates wrapper classes for the records in a COBOL application; in simple cases, this enables you to access COBOL data from Java without writing any new Visual COBOL code yourself.

The third problem is that each COBOL program in an application allocates all its memory the first time it is loaded. This memory is global to the program and effectively shared. This does not work well in modern application servers that are multithreaded and concurrent. The Visual COBOL run-time helps solve this problem with run-units, which are described in more detail later in this book.

Structure of a COBOL program

A procedural COBOL program has four divisions, each of which is identified by a header as shown in the following snippet:

```
IDENTIFICATION DIVISION.  
PROGRAM-ID. Program-1.  
* Contains information identifying program, author, date, etc  
ENVIRONMENT DIVISION.  
* Optional section which contains information specific to the OS  
* and/or machine the program will run on.  
DATA DIVISION.  
* Declares variables for use in the program  
WORKING-STORAGE SECTION.  
01 A-VARIABLE          PIC X(20).  
LINKAGE SECTION.  
01 A-PARAMETER         PIC S9(9) COMP-5.  
PROCEDURE DIVISION.  
* The statements making up the actual program.
```

In the Micro Focus COBOL dialect, the division headers are optional; with the exception of the procedure division header that can include parameters to be passed to the program, they have no actual effect. However, code that has been written for the mainframe will include some or all of them.

We'll cover the two most important divisions, the data division and procedure division, in the next two sections.

Data Division

The data division is where all the data is declared for a program. It consists of any of the following sections:

- File section: defines the structure of the records stored in those files.
- Working-storage section: Variables declared here are allocated when the program is first loaded. In procedural COBOL, all variables are global to the program they are declared in.
- Local-storage section: Storage for variables declared here is allocated on the stack each time the program is called. Not all dialects of COBOL have this feature.
- Linkage section: When you declare any kind of variable or record in the linkage section, you are only creating space for a memory pointer; you are not allocating the actual storage for the variable. The linkage section enables you to address arguments passed to your program by reference. (If you are compiling to managed code, the linkage item will contain an object reference rather than a memory pointer).

Each data division section is optional, but they must appear in the order shown above. Micro Focus COBOL allows you to omit the working-storage section header, but not the others.

COBOL Data Declarations

COBOL data declarations look rather different than the way they look in Java. The programmer not only declares the data but how it is stored.

Listing 4-1 shows a simple program with three variables. Items `anInteger` and `aDifferentInteger` are both 32-bit unsigned integers - `pic x(4)` allocates 4-bytes. One is declared as `comp-5`, which means it uses the native byte-ordering for the processor it is running on. `Comp-x` data uses big-endian byte ordering (the most significant byte is stored first). COBOL will handle the conversion from one to the other (when we move `anInteger` to `aDifferentInteger`, for example).

There can be a small overhead to using `comp-x` data (for example, on Intel processors `comp-x` items are converted to little-endian format before use in calculations). But records that will be written to file should always use `comp-x` fields rather than `comp-5` for portability.

For example, if you write a file from a processor with a big-endian format (for example, IBM RS/6000) and then read it with code running on a little-endian processor (like Intel processors), data represented as comp-5 will be scrambled.

There are several computational formats in COBOL that define different ways of representing numeric data. For example, comp-3 is decimal data (two decimal digits per byte) and is often used in financial calculations because it avoids the rounding errors involved whenever decimal fractions are represented in binary. All these formats are represented using pic strings containing '9' to represent a decimal digit, 'S' to represent the presence of a sign, and 'V' to represent an implied decimal point. For example, PIC S999V99 USAGE DISPLAY represents a 5 digit signed numeric field where the last 2 digits are considered as two decimal places, and where the field is stored in character format.

Variable aString simply declares 100 bytes of storage and the program moves a short literal into it.

Listing 4-1 Declaring and using data

```
program-id. DataItems.  
*> No DATA DIVISION header required in Micro Focus COBOL  
working-storage section.  
01 anInteger          pic x(4) comp-5.  
01 aDifferentInteger  pic x(4) comp-x.  
01 aString            pic x(100).  
  
procedure division.  
  move 99 to anInteger  
  move anInteger to aDifferentInteger  
  move "Hello there" to aString  
  
  display anInteger space aDifferentInteger space aString  
  stop run
```

Group Items

A COBOL group item (or record) enables the programmer to define a number of fields that are grouped together under a single data item name. This enables you to define a record (which you can then write to a file and read back). Although this seems like a rather low-level way to do things compared to more modern languages, it is very efficient and is one of the reasons why COBOL performs so well at all kinds of batch data processing.

A group item is very similar to a struct in the C language. A COBOL group item can specify the layout and sizes of fields down to individual bytes, although compiler directives enable you to align fields on set boundaries to make access more efficient on processors with different word lengths. A COBOL group item can also contain subgroups and you can redefine the layout of records.

For example, you can define an “employee” record with a number of common fields (name, address, date of birth), but redefine other parts of the record to enable you to store different information against different types of employee. The value of a flag field determines which kind of record it is. You can then write all your employee records to a file and use it each month when you run the payroll (for example).

Listing 4-2 shows a simple group item that represents a customer record. It has an eight-character zip-code field that is redefined as a UK postcode with two separate sub-fields. The `redefines` clause is similar to a union in C; the same storage is defined as a different type and/or a different name.

The address-line is declared with an `occurs` clause. This means the item is repeated the number of times defined by the `occurs` clause and can be referenced by index as shown in the second line of the procedure division.

Listing 4-2 A simple group item

```
program-id GroupItems.  
Working-storage section.  
01 cust-record.  
    03 cust-name          pic x(80).  
    03 cust-address.  
        05 address-line    pic x(80) occurs 3.  
        05 zip-code         pic x(8).  
        05 uk-post-code     redefines zip-code.  
            07 outward-code   pic x(4).  
            07 inward-code    pic x(4).  
    03 age                pic 9(4) comp-x.  
  
procedure division.  
    move "Director General" to cust-name  
    move "BBC" to address-line(1)  
    move "W1A" to outward-code  
    move "1AA" to inward-code.
```

Procedure Division

The procedure division holds the program statements and can optionally be divided into sections and paragraphs, where sections can contain multiple paragraphs.. By default, execution falls through from one section and paragraph to the next, but when you use the `perform` verb to execute a section or paragraph, control returns at the end of the section or paragraph or when an `exit section` or `exit paragraph` is executed. A section header is a section name followed by the word `section` and a period, whereas a paragraph header is just a name followed by a period. The sentence before a section or paragraph header must be terminated by a period. The period can appear at the end of a statement on the same line or on a line on its own.

Listing 4-3 shows a short program with two sections and one paragraph in the first section. This program executes thefirst and thesecond section, then exits.

Listing 4-3 Paragraphs and sections in the procedure division

```
program-id. Program1.

procedure division.
  perform thefirst
  perform thesecond
  goback
*   You must terminate the previous sentence with a period before
*   starting a new section.

.

thefirst section.
  display "thefirst"
  perform onea
  exit section.

*   Paragraph onea
onea.
  display "one A".

thesecond section.
  display "thesecond"
  exit section
```

Program Parameters and Entry Points

When one COBOL program calls another, it can pass some data and get a result back. You specify the parameters a program expects with the optional using clause in the procedure division header. The procedure division is the primary entry point for a COBOL program and the one that is executed when the program is called. However, you can declare other entry points inside a program using the entry header and these entry points can also have parameters.

By default, all arguments in a call statement are passed by reference. So the called program actually gets passed pointers to the data item in the caller's memory space. In the called program, these parameters must be declared in the linkage section. If an argument is passed to the callee by reference, it can modify the value in order to pass a result back to the caller. In fact, you can pass more than one result back since you can have several by reference parameters.

You can also pass items by value, in which case the data is copied onto the stack and accessed there by the called program.

You might occasionally see a `call` statement that passes an argument by content. This copies the data into a separate area of memory, then passes a pointer to the new area to the called program. To the called program it looks the same as an item passed by reference, but any changes made are not passed back to the calling program. You might occasionally see `omitted` or `by reference omitted`. This is the equivalent of passing a null to the called program.

Finally, if you have a returning clause on the `call` statement, the called program can return a numeric value using the special `return-code` register. The callee can either move data into `return-code` or add a returning clause to the `exit` program or `goback` statement used to return control to the caller. The `return-code` register is a signed numeric variable of type `comp` and is four bytes long by default, although it is only two bytes long in some dialects. The exact binary representation of a `comp` data item depends on the compiler and platform.

Listing 4-4 shows two programs that illustrate most of the points explained above. `MainProgram` loads `Program2` and then calls the entry points in it. Because `paramByRef` is passed by reference, `Program2` is able to change the value of the string before returning control to the caller. The last call passes `aNumber` by content; to the called program this looks like a call by reference, but the reference is a pointer to a copy of the original data so it remains unchanged when control returns to the caller. A result is passed back using the `return-code` register.

There is generally no type checking in procedural COBOL, so you have to be very careful when coding calls and entry points. And if a `call` statement doesn't provide arguments that the called program is expecting as `by reference` parameters, you will get an access violation at run-time as the called program will be trying to read or write memory without a valid pointer. If the program is compiled as managed code, this will show up as a null pointer exception.

 **Micro Focus COBOL enables you to define CALL prototypes, and if you use CALL prototypes, the compiler will validate call signatures at compile time. See the Micro Focus documentation for more information.**

Listing 4-4 Calls, parameters, and entry points

program-id `MainProgram`.

working-storage section.

```
01 aNumber      pic s9(9) comp-5.  
01 answer       pic s9(9) comp-5.  
01 aString      pic x(60).
```

```
procedure division.  
*   Load the other program  
call "Program2"  
  
move 32 to aNumber  
call "EntryOne" using by value aNumber  
          returning answer  
display answer  
move "Hello world" to aString  
call "EntryTwo" using by reference aString  
display aString *-> String value has been changed  
  
move 99 to aNumber  
call "EntryThree" using by content aNumber  
display aNumber *-> aNumber unchanged because it was passed  
          *> by content  
display return-code  
display aString  
  
program-id. Program2.  
  
working-storage section.  
01 result          pic s9(9) comp-5.  
  
linkage section.  
01 paramByRef      pic x(60).  
01 numberByRef     pic s9(9) comp-5.  
01 paramByValue    pic s9(9) comp-5.  
  
procedure division.  
display "Program loaded"  
goback.  
  
entry "EntryOne" using by value paramByValue.  
  
display "Entry one " paramByValue  
multiply 2 by paramByValue giving result  
exit program returning result.  
entry "EntryTwo" using by reference paramByRef.  
display paramByRef  
move spaces to paramByRef  
move "goodbye world" to paramByRef  
exit program.  
  
entry "EntryThree" using by reference numberByRef.  
multiply numberByRef by 2 giving numberByRef
```

```
move numberByRef to return-code  
exit program.  
  
end program Program2.
```



The exit program and goback statements are both ways of returning control from called code to the caller. In Visual COBOL the exit method and goback statements can be used to exit a method. Goback is a simpler and more consistent way to return control, but it isn't available in all COBOL dialects, so you might see exit statements in applications you are modernizing.

Copybooks

In the earlier section on COBOL data declarations, we showed an example of a simple record defined in the working-storage section. But traditional COBOL doesn't have a way of defining this as a new type, so what happens when we want to use our new record definition in more than one place?

COBOL has the "copybook". A copybook (also known as a "copy file") is a source code file that can be included inside another source file by using the copy verb. When a copybook is used with the replacing clause, it's possible to change the names of the items declared so that they are unique inside a single program, effectively providing a reusable type definition.

Copybooks are often shared between programs in an application because they contain definitions of all the data that will be used. If you are familiar with C, you can think of them as similar to header files: text the compiler inserts whenever it sees the copy statement. Copybooks can be used to hold code as well as data.

Listing 4-5 shows a copybook that defines a very simple record that can represent a date. The (PREFIX) on every line is not valid as part of a COBOL data name but will be replaced by some other text every time the copybook is added to a program with the copy... replacing statement. Listing 4-6 shows a simple program that returns the number of days in a month. The linkage section uses a copy... replacing statement to bring in the DATE.cpy copybook, replacing every occurrence of (PREFIX) with LNK. The effect is that the linkage section now contains declarations for LNK-DATE, LNK-YEAR, etc. Any client program that wants to use the date format defined here can also use copy... replacing statements to declare date records.

In our example, we've made a call to GET-DAYS-IN-MONTH from the procedure division, and used copy... replacing in the working-storage section to create a WS-DATE variable.

Listing 4-5 The DATE.cpy copybook defining a simple date format

```
* DATE
* YYYYMMDD format
01 (PREFIX)-DATE.

03 (PREFIX)-YEAR          PIC 9(4).
03 (PREFIX)-MONTH         PIC 9(2).
03 (PREFIX)-DAY           PIC 9(2).
```

Listing 4-6 A program using DATE.cpy

```
program-id. Calendar.

working-storage section.
copy "DATE.cpy" replacing ==(PREFIX)== by ==WS==.
78 GET-DAYS-IN-MONTH      value "GET-DAYS-IN-MONTH".

01 MOD-RESULT            pic 99 comp-5.
linkage section.
copy "DATE.cpy" replacing ==(PREFIX)== by ==LNK==.
01 LNK-RESULT            pic 99 comp-5.

procedure division.
move "20190704" to WS-DATE
call "GET-DAYS-IN-MONTH" using WS-DATE MOD-RESULT
display MOD-RESULT
goback.

ENTRY GET-DAYS-IN-MONTH using by reference LNK-DATE LNK-RESULT.
evaluate LNK-MONTH
when 1
when 3
when 5
when 7
when 8
when 10
when 12
move 31 to LNK-RESULT
when 2
compute mod-result = function mod (LNK-YEAR, 4)
if mod-result = 0
```

```
compute mod-result = function mod(LNK-YEAR, 100)
if (mod-result = 0)
    compute mod-result = function mod(LNK-YEAR, 400)
    if mod-result = 0
        move 29 to LNK-RESULT
    else
        move 28 to LNK-RESULT
    end-if
else
    move 29 to LNK-RESULT
end-if
else
    move 28 to LNK-RESULT
end-if
when other
    move 30 to LNK-RESULT
end-evaluate
goback.

end program Calendar.
```

 **Micro Focus COBOL and some other dialects have a `typedef` clause that can be used to define types that can be reused across programs, similar to `typedef` in C. However, they are not commonly used in legacy applications.**

Summary

In this chapter, we looked at the COBOL language used in existing applications that need modernizing, covering the use of group-items (or records) and copybooks. We also discussed how Visual COBOL simplifies modernizing these applications. In subsequent chapters we will show different ways COBOL code can be modernized to open it up for reuse as part of new applications or to extend the reach of existing ones.

CHAPTER 5



An Example Application

In the previous chapter, we talked about the challenges of modernizing COBOL applications. This chapter introduces a set of simple COBOL programs that you can imagine forming part of a larger application suite. In subsequent chapters, we will look at different ways we can deploy and reuse this code.

In this chapter:

- Introducing the example
- Storing records
- Calculating interest
- Generating example data
- Calling COBOL from Java

Introducing the Example

Our example consists of three separate COBOL programs that work together. You can think of them as a fragment of a much larger suite of programs that would form a complete application for managing credit card accounts. These programs don't provide any kind of UI on their own – they provide some business logic we want to reuse in a number of different ways.

The three programs are:

- ACCOUNT-STORAGE-ACCESS is a collection of routines for writing, reading and finding records for Customers, accounts, and transactions.
- Interest-Calculator is a program that calculates monthly interest for an account
- Calendar is a program that is used to find out how many days there are in a particular month.

The application stores data in three COBOL indexed files (**customer.dat**, **account.dat** and **transaction.dat**). The record structure for each file is defined in separate copybooks – so client programs can include the copybooks to be able to understand the data. The code itself is written as structured COBOL. The example is not as complex as a real COBOL application, but it is intended to be easy to understand and to illustrate some different approaches we can take to modernization.

This chapter describes our example and shows a very simplified example of calling COBOL directly from Java using the Smart Linkage feature. In subsequent chapters, we'll show it modernized in different ways, with the functionality exposed as a REST API for example or over a message queue. We'll also look at different ways it could be deployed: using Docker containers, serverless computing, or the Cloud Foundry platform. We'll also take a short look at how you could provide a modern UI for the application.

Importing the Example Projects

The procedural COBOL programs are in the **BusinessRules** project. There are two other projects as well (**DataBuilder** and **SmartLinkage**) and we will import all three projects together. Download the examples for Chapter 5 (see “Downloading the Examples” in Chapter 1) and import them into a new Eclipse workspace as follows:

1. Start Eclipse and create an empty workspace by clicking **File**, **Switch Workspace** and naming a new workspace.
2. Temporarily disable automatic builds in workspace by clicking **Project**, **Build Automatically** (so that the **Build Automatically** option is no longer checkmarked).

The projects you are going to import won't build without errors until you change the **Custom Builder** properties in the **BusinessRules** project. This shouldn't cause a problem, but occasionally Eclipse will get stuck in a loop where it keeps repeating the project builds and starts again each time they fail.

3. Click **File > Import**, select **General > Existing Projects into Workspace** from the **Import** wizard, and then click **Next**.
4. In the **Select root directory** field, enter the folder with the Chapter 5 examples. The projects **BusinessRules**, **DataBuilder** and **SmartLinkageClient** should appear in the **Projects** list.
5. Deselect **SmartLinkageClient** and then click **Finish**.

We will import this project later, but it won't compile until we change directive settings on the BusinessRules project later in this chapter.

6. In the **COBOL Explorer**, right-click the **BusinessRules** project and click Builders from the **Properties** window.
7. In the Builders pane, click **mvn_businessrules** and then click the **Edit** button.
8. Click the **Environment** tab and change the value of **M2_HOME** to the location of your Maven installation.
9. If you are running on Linux, click the **Main** tab and change the value of the **Location** field so that it ends in **mvn** instead of **mvn.cmd**.
10. In the **Edit Configuration dialog**, click **OK**.
11. In the Properties for Business Rules dialog, click **Apply and Close**.
12. Now re-enable Automatic Builds for the project.

The projects should now all build successfully, but if the **mvn_businessrules** builder fails, select **BusinessRules** in the COBOL Explorer and then click **Project, Clean**, and clean the **BusinessRules** project. The builders use Eclipse variables like **project_loc**, and sometimes Eclipse doesn't pick up the correct context to set them for the right project.

Storing Data in Indexed Files

COBOL has its own syntax for reading and writing data from files, including support for indexed files. Each record in an indexed file has a primary key and optionally one or more secondary keys. You can access records randomly by key. Secondary keys can include duplicates.

IBM ISAM (Indexed Sequential Access Method) is probably the earliest implementation of COBOL indexed files (mid-1960s) and predates relational databases by about a decade. Although most COBOL applications use an RDBMS for persisting data, there are also many that still use indexed files; for some workloads, indexed files will deliver a higher performance.

Our example uses indexed files to begin with as they bring their own set of problems for application modernization – although later in the book, we recode the access data layer to use a relational database instead. An application that uses a relational database runs better in cloud environments and is easier to scale.

Our BusinessRules project stores data in three indexed files: **customer.dat**, **account.dat**, and **transaction.dat**. There are copybooks in the BusinessRules project that define the record layout for each type of file.

Listing 5-1, Listing 5-2, and Listing 5-3 show the copybooks for each record type. As explained in the section on copybooks in Chapter 4, the compiler will substitute a string specified by the programmer for the (PREFIX) in each of the record names when the copybooks are included in a program by a copy... replacing statement.

We've deliberately kept the format of the records in these files very simple – a real application would have more fields in each record and might use redefines clauses to enable a single file to store several slightly different record formats.

For example, we haven't provided anywhere to store a postal address in this set of files, which is something a real application would want. The customer addresses could either be stored as a set of fields in the CUSTOMER-RECORD or you might add an ADDRESS-ID field and store them in a separate address file. A separate address file is closer to the way it would be structured in a relational database, but COBOL applications using indexed files do not always normalize data in this way, so in practice you might see data structured in either way.

Listing 5-1 *The CUSTOMER-RECORD.cpy copybook*

```
* CUSTOMER-RECORD
01 (PREFIX)-CUSTOMER-RECORD.
  03 (PREFIX)-CUSTOMER-ID      PIC X(4) COMP-X.
  03 (PREFIX)-FIRST-NAME      PIC X(60).
  03 (PREFIX)-LAST-NAME      PIC X(60).
```

Listing 5-2 *The ACCOUNT-RECORD.cpy copybook*

```
* ACCOUNT-RECORD
01 (PREFIX)-ACCOUNT.
  03 (PREFIX)-ACCOUNT-ID      PIC X(4) COMP-X.
  03 (PREFIX)-CUSTOMER-ID      PIC X(4) COMP-X.
  03 (PREFIX)-BALANCE        PIC S9(12)V99 COMP-3.
  03 (PREFIX)-TYPE          PIC X.
  03 (PREFIX)-CREDIT-LIMIT    PIC S9(12)V99 COMP-3.
```

Listing 5-3 *The TRANSACTION-RECORD.cpy*

```
* TRANSACTION-RECORD
01 (PREFIX)-TRANSACTION-RECORD.
  03 (PREFIX)-TRANSACTION-ID    PIC X(4) COMP-X.
  03 (PREFIX)-ACCOUNT-ID      PIC X(4) COMP-X.
  03 (PREFIX)-TRANS-DATE.     *gt; yyyyymmdd
    05 (PREFIX)-YEAR          PIC 9(4).
    05 (PREFIX)-MONTH         PIC 9(2).
    05 (PREFIX)-DAY           PIC 9(2).
  03 (PREFIX)-AMOUNT        PIC S9(12)V99.
  03 (PREFIX)-DESCRIPTION    PIC X(255).
```

The record layouts only describe the way data is arranged; they don't tell you which fields are the primary and secondary keys. That is done where the files are declared inside a program. We'll look at that in the next section.

Accessing Files

In this section, we'll look at the ACCOUNT-STORAGE-ACCESS program, which contains all the code for reading and writing records. Rather than putting it all into one single listing that would run across three pages of this book, we've split it into sections and will describe each one in turn.

If you are familiar with COBOL already, you can skim through the next few sections to get an idea of what the program does, but you won't need the detailed explanations. If your background is in Java, you should find the explanations helpful in understanding how COBOL applications are structured.

Declaring the Files and Data

Listing 5-4 shows the start of the ACCOUNT-STORAGE-ACCESS program, where the files are declared, and also the data-division, where all the data used by the program is declared.

Listing 5-4 Declaring the files

```
program-id. ACCOUNT-STORAGE-ACCESS.  
file-control.  
    select Account-File assign to external accountFile  
        file status is file-status  
        organization is indexed  
        access mode is dynamic  
        record key is FILE-ACCOUNT-ID OF FILE-ACCOUNT  
        alternate record key is FILE-CUSTOMER-ID OF FILE-ACCOUNT  
            with duplicates  
  
    select Customer-File assign to external customerFile  
        file status is file-status  
        organization is indexed  
        access mode is dynamic  
        record key is FILE-CUSTOMER-ID OF FILE-CUSTOMER-RECORD  
        alternate record key is FILE-LAST-NAME with duplicates  
  
    select Transaction-File assign to external transactionFile  
        file status is file-status
```

```
organization is indexed
access mode is dynamic
record key is FILE-TRANSACTION-ID
alternate record key is FILE-ACCOUNT-ID
    of FILE-TRANSACTION-RECORD
    with duplicates
alternate record key is FILE-TRANS-DATE with duplicates
.

data division.
file section.
fd Account-File.
copy "ACCOUNT-RECORD.cpy" replacing ==(PREFIX)== by ==FILE==.
fd Customer-File.
copy "CUSTOMER-RECORD.cpy" replacing ==(PREFIX)== by ==FILE==.
fd Transaction-File.
copy "TRANSACTION-RECORD.cpy" replacing ==(PREFIX)== by ==FILE==.
working-storage section.
01 displayable          pic x(255).
78 MAX-ID              value 2147483648.

01 file-status.
03 file-status-1 pic x.
03 file-status-2 pic x.

01 library-status-code  pic xx comp-5.
copy "PROCEDURE-NAMES.cpy".

linkage section.
01 LNK-STATUS.
03 LNK-FILE-STATUS-1      PIC X.
03 LNK-FILE-STATUS-2      PIC X.
copy "FUNCTION-CODES.cpy".
copy "ACCOUNT-RECORD.cpy" replacing ==(PREFIX)== by ==LNK==.
copy "CUSTOMER-RECORD.cpy" replacing ==(PREFIX)== by ==LNK==.
copy "TRANSACTION-RECORD.cpy" replacing ==(PREFIX)== by ==LNK==.
```

The program-id statement at the top identifies the program. The file-control paragraph declares three files using the select statement. It is part of the environment division (although our program omits the environment division header). The first name in the select statement declares the variable name by which the file is referred to inside this program. The assign to external clauses map the variable names to actual physical filenames. There are different ways of assigning files and they are implementation-dependent because they can differ from platform to platform.

Our external mappings are done via environment variables that will be explained later. There are several alternatives for mapping to external files with Micro Focus COBOL; these are explained in the product documentation (look for information on the External File Mapper).

The file status clause assigns a working-storage variable to receive the file status code after each operation (success is “00”). The organization clause declares all our files as indexed (other possibilities are relative or sequential). Setting the access mode to dynamic means the files can be opened for input, output, or input-output. Finally, there are two record key clauses: the first determines the primary key and the alternate record key enables you to declare one or more secondary keys. Secondary keys can have duplicates.

The file section in the data division is where the actual record formats for the files are declared. There is an fd (file-descriptor) for each of our three files. Each file-descriptor is followed up by copy... replacing statement that brings in the actual record description. In each of these, we've replaced ==(PREFIX)== with FILE, so that we end up with variable names FILE-CUSTOMER, etc. Each time we read or write a record, these variables are where the data is moved between file and program memory.

The working-storage section declares some variables the program needs to run. You'll remember that the file-control paragraph made file-status the location for storing the status of each file operation; working-storage is where the data item is actually declared and storage allocated.

The working-storage section also includes copybook **PROCEDURE-NAMES.cpy**. This defines some constants for naming all the entry points in the program. This is a piece of defensive coding; each entry point could be defined using a literal instead of a constant. For example, entry “OPEN-ACCOUNT-FILE” and entry OPEN-ACCOUNT-FILE (where OPEN-ACCOUNT-FILE has been declared as a constant) are equally valid. But using constants in the entry points and the call statements (call OPEN-ACCOUNT-FILE) means we'll spot any mistakes at compile time rather than at run-time when a statement like call “OEN-ACCOUNT-FILE” fails because the RTS can't find the entry point.

Finally, the parameters to all the entry points in the program are declared in the linkage section. Callers need to know the success or otherwise of file operations, so there's a LNK-STATUS to return file status codes. And so that we can pass records in and out of the program, the copybooks defining the different record types are all copied in, but this time with LNK replacing ==(PREFIX)== so that the variables have different names to the ones declared in the file section.

Procedure Division Entry Point

The procedure division header is the main entry point to any program and is executed when you call program-name. Calling a program loads it into memory, making all its entry points available. Calling an entry point before the program has been loaded causes a run-time error.

The procedure division in ACCOUNT-STORAGE-ACCESS logs some useful information on the console. It can be called more than once without having any other effects; this is useful since any client program can call ACCOUNT-STORAGE-ACCESS to access its entry points and it won't matter if it has already been called before.

Listing 5-5 shows the procedure division and the display-file-names section. Although display-file-names appears at the end of the program, we've shown the two together here.

The perform verb has a number of different meanings in COBOL; here it means execute a named section or paragraph.

The display-file-names section prints the names of environment variables dd_customerfile, dd_accountFile and dd_transactionFile to the console. These environment variables contain the path and filename of each of the data files set up in the file-control paragraph – this is one of the ways the Micro Focus external filerMapper maps to an external name. Displaying the names of the environment variables helps with troubleshooting.

Listing 5-5 AccountStorageAccess procedure division and display-file-names section

```
procedure division.  
    perform display-file-names  
        goback.  
        ...  
    display-file-names section.  
        display "dd_customerFile" upon environment-name  
        accept displayable from environment-value  
        display "Customer      file = " displayable  
        move spaces to displayable  
  
        display "dd_accountFile" upon environment-name  
        accept displayable from environment-value  
        display "Account      file = " displayable  
        move spaces to displayable  
  
        display "dd_transactionFile" upon environment-name  
        accept displayable from environment-value  
        display "Transaction file = " displayable
```

Displaying Logs on the Console

In our example, we've made a conscious decision to write some logging information directly to the console – mainly to make it easier to see what's happening if the program can't open the data files for example. In a real application, you should use a logging library to write log files, but there is no common standard for logging in COBOL. Micro Focus provides a Consolidated Trace Facility for logging, but it is not likely to be found in many legacy applications. We will talk more about logging later in this book when we look at cloud deployments.

Reading and Writing Files

Most of the rest of ACCOUNT-STORAGE-ACCESS is code to open files and to either write or find records. This program happens to manage three different files, but often you might have a single program for each file. For each of our three files there is a separate entry point to:

- Open a file (the open file entry points can also be used to close a file).
- Write a record.
- Read a record (either by primary or secondary key).
- Read the very last record (by primary key) in the file.

Listing 5-6 shows the code for the customer file. The code for the account and transaction files is very similar, so we'll only look at the customer code. You can examine the other code for yourself when you download the examples for this chapter.

***Listing 5-6** Reading and writing customer records*

```
ENTRY OPEN-CUSTOMER-FILE using by VALUE LNK-FUNCTION
                                by reference LNK-STATUS
    evaluate LNK-FUNCTION
        when OPEN-READ
            open input Customer-File
        when OPEN-WRITE
            open i-o Customer-File
        when OPEN-I-O
            open i-o Customer-File
        when CLOSE-FILE
            close Customer-File
    end-evaluate
    move file-status to LNK-STATUS
    goback.
```

```
ENTRY WRITE-CUSTOMER-RECORD using by value LNK-FUNCTION
    by reference LNK-CUSTOMER-RECORD
    LNK-STATUS.

move LNK-CUSTOMER-RECORD to FILE-CUSTOMER-RECORD
evaluate LNK-FUNCTION
when WRITE-RECORD
    write FILE-CUSTOMER-RECORD
when UPDATE-RECORD
    rewrite FILE-CUSTOMER-RECORD
when other
    move "88" to file-status
end-evaluate
move file-status to LNK-STATUS
goback.

ENTRY DELETE-CUSTOMER-RECORD using by reference LNK-CUSTOMER-RECORD
    LNK-STATUS.

move LNK-CUSTOMER-RECORD to FILE-CUSTOMER-RECORD
delete Customer-File record
move file-status to lnk-status
display file-status
goback.

* find account by customer last name
ENTRY FIND-CUSTOMER-NAME using BY value LNK-FUNCTION
    by reference LNK-CUSTOMER-RECORD
    LNK-STATUS.

move "00" to LNK-STATUS
evaluate LNK-FUNCTION
when START-READ
    move LNK-CUSTOMER-RECORD TO FILE-CUSTOMER-RECORD
    start Customer-File key is equal FILE-LAST-NAME
when READ-NEXT
    read Customer-File next
    move FILE-CUSTOMER-RECORD to LNK-CUSTOMER-RECORD
end-evaluate
move file-status to LNK-STATUS
goback.

* find account by customer ID
ENTRY FIND-CUSTOMER-ID using BY value LNK-FUNCTION
    by reference LNK-CUSTOMER-RECORD
    LNK-STATUS.

move "00" to LNK-STATUS
move LNK-CUSTOMER-RECORD to FILE-CUSTOMER-RECORD
```

```

read Customer-File key is FILE-CUSTOMER-ID
      of FILE-CUSTOMER-RECORD
move FILE-CUSTOMER-RECORD to LNK-CUSTOMER-RECORD

move file-status to LNK-STATUS
goback.

ENTRY READ-CUSTOMER-RECORD using by value LNK-FUNCTION
      reference LNK-CUSTOMER-RECORD
      LNK-STATUS

evaluate LNK-FUNCTION
when START-READ
      move LNK-CUSTOMER-RECORD TO FILE-CUSTOMER-RECORD
      start CUSTOMER-File key >= FILE-CUSTOMER-ID
            of FILE-CUSTOMER-RECORD
when READ-NEXT
      read CUSTOMER-FILE next
end-evaluate
move FILE-CUSTOMER-RECORD to LNK-CUSTOMER-RECORD
move file-status to LNK-STATUS
goback
.

ENTRY READ-LAST-CUSTOMER-RECORD using
      by reference LNK-CUSTOMER-RECORD
      LNK-STATUS
move MAX-ID to FILE-CUSTOMER-ID of FILE-CUSTOMER-RECORD
start Customer-File key
      < FILE-CUSTOMER-ID OF FILE-CUSTOMER-RECORD

read Customer-File previous
move FILE-CUSTOMER-RECORD to LNK-CUSTOMER-RECORD
move file-status to LNK-STATUS
goback.

ENTRY OPEN-ACCOUNT-FILE using by VALUE LNK-FUNCTION
      by reference LNK-STATUS
evaluate LNK-FUNCTION
when OPEN-READ
      open input Account-File
when OPEN-I-O
      open i-o Account-File
when OPEN-WRITE
      open output Account-File
when CLOSE-FILE

```

```
        close Account-File
end-evaluate
move file-status to LNK-STATUS

goback.
```

Opening a File

A file can be opened for input, output, or i-o. It is possible for a file to be opened from more than one process at a time, but when a process opens a file for output, individual records are locked while they are being written. This can cause performance issues if there is a lot of concurrency. Micro Focus FileShare can improve concurrent performance for file handling, but it is not available for either JVM or .NET compiled programs.

The entry point OPEN-CUSTOMER-FILE takes a function code as an argument to determine the file open mode and returns the status of the operation. All the function codes used by ACCOUNT-STORAGE-ACCESS are defined in copybook FUNCTION-CODES.cpy (see Listing 5-7).

Listing 5-7 The FUNCTION-CODES.cpy copybook

```
* function codes
01 LNK-FUNCTION          pic x.
    78 OPEN-WRITE           value "W".
    78 OPEN-READ            value "R".
    78 OPEN-I-O              value "B".
    78 CLOSE-FILE            value "C".
    78 START-READ           value "S".
    78 READ-NEXT             value "N".
    78 WRITE-RECORD          value "T".
    78 UPDATE-RECORD         value "U".
```

Writing a Record

Entry point WRITE-CUSTOMER writes a record into the file. FILE-CUSTOMER-ID is the primary key and duplicates are not allowed. The WRITE-RECORD function code writes a new record; UPDATE-RECORD uses the rewrite verb to update an existing record (one with the same primary key).

Reading a Record

There are three entry points that provide different ways of finding a particular customer record. Entry FIND-CUSTOMER-ID returns the record that matches a customer ID. There is more than one data item with the name FILE-CUSTOMER-ID – it is a field in both the customer

and account records, so whenever we refer to this data item, we have to explicitly reference the group it is a member of. For example, the read statement in FIND-CUSTOMER-ID is:

```
read Customer-File key is FILE-CUSTOMER-ID of FILE-CUSTOMER-RECORD
```

The next entry point for reading a record is FIND-CUSTOMER-NAME. This enables you to search for customers by their last name. FILE-LAST-NAME is declared as an alternative key in the select statement for the customer-file, and allows duplicates, so there could be more than one matching record.

We start the file on a record matching the last name we are searching for, with function-code START-READ. This doesn't return any actual data, although if there are no matching records the file status code will be "23". Then we keep calling the function to read records with function code READ-NEXT. The file-status will be "02" (the next record in sequence has the same key) if there is a further matching record to read. The file status for the last matching read will be "00".

Finally, we have an entry point for READ-LAST-CUSTOMER-RECORD. This reads the record with the highest value of the primary key. This is useful mainly for determining what value customer ID to give the next record when we are writing new records. We start reading records with a primary key less than the maximum possible value, but do a read previous, effectively reading the file backwards.

File Status Codes

All COBOL file operations return a 2-byte file status code, where "00" is a successful operation. All ANSI 85- or ANSI 74-compliant COBOL systems have standardized codes up to "49" and each of the two bytes is an ASCII numeric character. Micro Focus COBOL also has extended character codes where the first byte is ASCII "9" and the second byte a binary number; these codes are for situations that are more specific to the Micro Focus COBOL run-time system than the standard codes.

Using status codes like this might seem odd to Java programmers where idiomatic programming makes extensive use of exceptions to signal non-success conditions, but is very common in COBOL. Non-zero status codes do not always indicate an "exception"; they can also indicate non-error conditions that can occur during normal processing (for example record not found). You could think of them as analogous to http status codes that also signal both errors and non-error conditions that are of interest to a client.

The Visual COBOL documentation includes a full list of error codes. You can also find them by searching for "Micro Focus COBOL file status codes" on the web.

Interest Calculator Program

The Interest-Calculator program is the core piece of business logic we want to reuse as we modernize our application. It reads one month's transactions from the transaction file for a given account id and calculates the total interest payable at the end of the month based on an initial balance and a daily interest rate.

Interest is calculated based on the balance each day throughout the month and accumulated to a total figure at the end. The Interest-Calculator also calculates a minimum payment. Although Interest-Calculator is less than 200 lines of code, we've split it into pieces to make it easier to follow.

Interest-Calculator is a good example of the kind of business logic that is easier to write in COBOL than Java. Financial calculations require high-precision decimal arithmetic to avoid the rounding errors that occur when converting between decimal and binary fractions. You can use Java BigDecimal for these sorts of calculations, but the resulting code is not always easy to follow. Listing 5-8 shows the data declarations for the program.

Listing 5-8 Data division and procedure division header for Interest-Calculator

```
program-id. INTEREST-CALCULATOR.

data division.

working-storage section.
copy "FUNCTION-CODES.cpy".
copy "PROCEDURE-NAMES.cpy".
copy "TRANSACTION-RECORD.cpy" replacing ==(PREFIX)== by ==WS==.
01 WS-DEBUG                                PIC 9 VALUE 1.
01 WS-DAY-INTEREST                         PIC 9(8)v9(8) comp-3.
01 WORKING-BALANCE                          PIC S9(12)V9999 comp-3.
01 DAILY-BALANCE                           PIC S9(12)V99 OCCURS 31.
01 DAILY-BALANCE-INDEX                     PIC 99 COMP-5.
01 FUNCTION-CODE                           PIC X.
01 INTEREST-PAYABLE                        PIC S9(12)V9(8) COMP-3.
01 FILE-STATUS.
    03 FILE-STATUS-BYTE-1                  PIC X.
    03 FILE-STATUS-BYTE-2                  PIC X.
01 DAYS-IN-MONTH                           PIC 99 COMP-5.
01 DISPLAY-CASH                            PIC -Z(12)9.99.

linkage section.
copy "DATE.cpy" replacing ==(PREFIX)== BY ==LNK-START==.

01 LNK-DAY-RATE                            PIC 99v9(8) comp-3.
01 LNK-ACCOUNT-ID                          PIC X(4) COMP-X.
01 LNK-AMOUNT                             PIC S9(12)V99.
01 LNK-MINIMUM-PAYMENT                    PIC S9(12)V99.
```

```

01 LNK-INTEREST          PIC S9(12)V99.
01 LNK-STATUS.
03 LNK-FILE-STATUS-1    PIC X.
03 LNK-FILE-STATUS-2    PIC X.

procedure division.
  goback.

```

Calling the program itself doesn't execute any code as the procedure division just contains a goback statement; as with ACCOUNT-STORAGE-ACCESS, this gives us a simple and safe way to load the program. The data division includes copybooks FUNCTION-CODES.cpy, PROCEDURE-NAMES.cpy and TRANSACTION-RECORD.cpy, as well as declaring some data used to carry out the calculations.

Some of the data is declared as comp-3, which is a decimal format (two decimal digits per byte) that is used for accurate decimal calculations. Binary formats like Java's float and double introduce rounding errors when used for decimal calculations, which comp-3 avoids. The precision is set by the picture clause for the data declaration. For example, 01 INTEREST-PAYABLE PIC S9(12)V9(8) COMP-3 declares a signed decimal with 12 places before the point and 8 digits precision after the decimal point.

Listing 5-9 shows the only other entry point (apart from the procedure division) for this program, and this contains all the code for actually calculating the monthly interest for an account.

Listing 5-9 Calculate-Interest entry point

```
*****
* LNK-DAY-RATE - Daily interest rate
* LNK-START-DATE - Assumed to be first of month.
* LNK-AMOUNT      - on entry: Start balance
*                  on exit: Total balance excluding interest
* LNK-INTEREST     Interest payable
*****
ENTRY CALCULATE-INTEREST using by value LNK-START-DATE
                                LNK-ACCOUNT-ID
                                by reference LNK-DAY-RATE LNK-AMOUNT
                                LNK-INTEREST
                                LNK-MINIMUM-PAYMENT
                                LNK-STATUS.

*   INITIALIZE DATA
  perform DISPLAY-START
  perform varying DAILY-BALANCE-INDEX FROM 1 by 1
    until DAILY-BALANCE-INDEX > 31
  move zero to DAILY-BALANCE(DAILY-BALANCE-INDEX)
```

```
end-perform
move LNK-AMOUNT to WORKING-BALANCE
move "00" to LNK-STATUS

call GET-DAYS-IN-MONTH using by reference LNK-START-DATE
                                DAYS-IN-MONTH

* OPEN TRANSACTION FILE
move OPEN-READ to FUNCTION-CODE
call OPEN-TRANSACTION-FILE using by value FUNCTION-CODE
                                by reference FILE-STATUS
if FILE-STATUS <> "00"
    move FILE-STATUS to LNK-STATUS
    goback
end-if

*> INITIALIZE READ FOR SELECTED ACCOUNT
move LNK-ACCOUNT-ID to WS-ACCOUNT-ID
move 0 to WS-TRANSACTION-ID
move START-READ to FUNCTION-CODE
call FIND-TRANSACTION-BY-ACCOUNT using by value FUNCTION-CODE
                                by reference WS-TRANSACTION-RECORD
                                FILE-STATUS
if FILE-STATUS <> "00"
    move FILE-STATUS to LNK-STATUS
    perform CLOSE-TRANSACTION-FILE
    goback
end-if

* First loop:
* Read all transactions for month, and
* Add each day's transactions to the balance for that day.
move READ-NEXT to FUNCTION-CODE
move "99" to FILE-STATUS
perform until FILE-STATUS = "00"
    call FIND-TRANSACTION-BY-ACCOUNT using
        by value FUNCTION-CODE
        by reference WS-TRANSACTION-RECORD
        FILE-STATUS
    if FILE-STATUS <> "00" and FILE-STATUS <> "02"
        exit perform
    end-if
    if WS-MONTH <> LNK-START-MONTH OR WS-YEAR <> LNK-START-YEAR
*         IGNORE TRANSACTIONS FOR OTHER MONTHS
        exit perform cycle
```

```

    end-if
    perform DISPLAY-TRANSACTION
    move WS-DAY to DAILY-BALANCE-INDEX
    add WS-AMOUNT to DAILY-BALANCE(DAILY-BALANCE-INDEX)
    end-perform

    if FILE-STATUS <> "00"
    *      Unexpected file status - can't complete calculation
        move FILE-STATUS to LNK-STATUS
        perform CLOSE-TRANSACTION-FILE
        goback
    end-if

    *      PERFORM INTEREST CALCULATION
    *      Second loop: for each day in the month calculate running
    *      total, and calculate interest for each day.
        add WORKING-BALANCE to DAILY-BALANCE(1)
        move 0 to INTEREST-PAYABLE
        perform varying DAILY-BALANCE-INDEX from 1 by 1
            until DAILY-BALANCE-INDEX > DAYS-IN-MONTH
        *      calculate the daily interest and add it to the daily balance
            multiply DAILY-BALANCE(DAILY-BALANCE-INDEX) by LNK-DAY-RATE
            giving WS-DAY-INTEREST
            add WS-DAY-INTEREST to DAILY-BALANCE(DAILY-BALANCE-INDEX),
                INTEREST-PAYABLE
            if DAILY-BALANCE-INDEX < DAYS-IN-MONTH
                *      Balance for next day starts with current day balance
                add DAILY-BALANCE(DAILY-BALANCE-INDEX)
                    to DAILY-BALANCE(DAILY-BALANCE-INDEX + 1)
            end-if
        end-perform
        move INTEREST-PAYABLE to LNK-INTEREST
    *      Last daily balance is now total for month
        move DAILY-BALANCE(DAYS-IN-MONTH) to LNK-AMOUNT
        multiply LNK-AMOUNT by .05 giving LNK-MINIMUM-PAYMENT
        if LNK-MINIMUM-PAYMENT < 5 and WORKING-BALANCE > 5
            move 5 to LNK-MINIMUM-PAYMENT
        else if WORKING-BALANCE < 5
            move WORKING-BALANCE to LNK-MINIMUM-PAYMENT
        end-if
    end-if
    perform DISPLAY-RESULT
    perform CLOSE-TRANSACTION-FILE
    goback.

```

```

DISPLAY-TRANSACTION SECTION.
  if WS-DEBUG > 1
    move WS-AMOUNT to DISPLAY-CASH
    display WS-DAY “,” with no advancing
    display DISPLAY-CASH “,” with no advancing
    display WS-DESCRIPTION
  end-if
  .

DISPLAY-START SECTION.
  if WS-DEBUG > 0
    move LNK-AMOUNT to DISPLAY-CASH
    display “*** Statement for account “ with no advancing
    display LNK-ACCOUNT-ID with no advancing
    display “ Start value “ DISPLAY-CASH
  end-if
  .

DISPLAY-RESULT SECTION.
  if WS-DEBUG > 1
    add 0 to LNK-INTEREST giving DISPLAY-CASH rounded
    display “account “ Lnk-account-id “ Interest “ DISPLAY-CASH
  end-if
  .

CLOSE-TRANSACTION-FILE SECTION.
  move CLOSE-FILE to FUNCTION-CODE
  CALL OPEN-TRANSACTION-FILE using by value FUNCTION-CODE
    BY reference FILE-STATUS
  .

```

The CALCULATE-INTEREST entry point calculates the interest day-by-day for the account. It takes the following parameters:

- LNK-START-DATE is an eight-character literal in yyyyymmdd format declared in the linkage section by copy... replacing of DATE.cpy. We have made the simplifying assumption that the calculator only deals with whole months (for example, balances don't run from the third of one month to the second of the next).
- LNK-ACCOUNT-ID is the ID of the account to read.
- LNK-DAY-RATE is the daily interest rate (for example, an annual interest rate of 25% is $25/100/365 = 0.00068493$).
- LNK-AMOUNT is used to input the start balance and return the total balance excluding interest.
- LNK-INTEREST returns the total interest payable at the end of the month.

- LNK-MINIMUM-PAYMENT returns the minimum payment due at the end of the month – the greater of 5% of the balance or \$5 (unless the balance is \$5, in which case the whole amount).
- LNK-STATUS is the file status for the operation. Any value other than “00” indicates a problem with retrieving data.

The algorithm is very simple. First, get the number of days in the month (by calling the CALENDAR program – see Listings 4-5 and 4-6 in Chapter 4). Then retrieve all the transactions on the selected account for that month and store the total transactions for each day in a separate table entry (DAILY-BALANCE in working-storage).

Once we have all the transactions for the month, we can calculate the total interest. Pass in the balance at the start of the month. Add the start balance to the first day of the month and calculate the daily interest. Add the balance of the first day to the transactions for the second day to get the new running total and calculate the interest for that day. We loop around this until we get to the end of the month, then return the amount, interest, and minimum payment to the caller.

Listing 5-9 performs some sections not shown in the listing. With the exception of CLOSE-TRANSACTION-FILE (which closes the **transaction.dat** file used by the program), these sections are used simply to display some data for debugging/troubleshooting. See the sidebar earlier in this chapter on logging to the console.

However, there isn't a single consistently used logging system available for procedural COBOL programs. Micro Focus provides the Consolidated Trace Facility (CTF), but this is not something you are likely to see in existing legacy applications. Applications that you are recompiling for JVM have the option of using any of the logging frameworks available for Java.

Generating Example Data

To be able to use the programs we looked at in the Accessing Files section, we need to create some data files we can use for testing. The DataBuilder project in the downloaded examples for this chapter generates customer, account, and transaction records from some raw data in comma separated value (CSV) files that are supplied in the csvData subdirectory of the DataBuilder project.

The DataBuilder project uses two Visual COBOL classes (AccountsBuilder and TransactionsBuilder) that read the CSV files and generate COBOL indexed files. A MainClass processes the command line and invokes the AccountsBuilder and TransactionsBuilder. There are several other classes in the project for processing the command line and parsing the CSV files, but we are not going to explain the workings of the DataBuilder in detail since it is supplied only as a utility to create some sample data.

Since you have the source code, you are free to examine it and use the debugger if you want to understand exactly how it works.

The **DataBuilder** uses the data in two CSV files (you can open these with spreadsheet applications like Excel or Google Sheets) to create 1,000 customers, 1,000 accounts, and one month of randomized transactions for each account (approximately 15,000 transactions altogether).

Once you have imported and built the Chapter 5 examples as explained in the section entitled “Importing the Example Projects,” you generate the customer data, account data, and one month of transactions as follows:

1. Create a COBOL JVM Application launch configuration. Select the **DataBuilder** project and click **Run, Run Configurations** to display the Run Configurations dialog.
2. In the left pane of the Run Configurations dialog, select **COBOL JVM Application** and then click the New Configuration icon.
3. Name the configuration **DataBuilder** and then set the Main class as **com.mfcobolbook.databuilder.MainClass** (click the Search button next to the Main class field to pick it from a list).
4. Click on the **Arguments** tab and provide Program Arguments:
`-new data-builder-project/csvData 20190701`
(where `data-builder-project` is the full path to the **DataBuilder** project). The numeric argument is a date in `yyyymmdd` format and will be used as the month for which to generate random transactions.
5. Click on the **Environment** tab and add the following three environment variables (where `chapter-5-examples` is the folder where you installed the example files):

```
dd_CUSTOMERFILE  
      = chapter-5-examples/Data/customer.dat  
dd_ACCOUNTFILE  
      = chapter-5-examples/Data/account.dat  
dd_TRANSACTIONFILE  
      = chapter-5-examples/Data/transaction.dat
```
6. Click **Run** to run the program. You should have **account.dat**, **customer.dat** and **transaction.dat** files in the data directory specified in the environment variables in Step 5.

You can optionally add an extra month of transaction data any time after you've created the base data. This isn't necessary to work through the examples in this book, but is offered in the event you want to work with more than one month of transaction data:

1. Duplicate the **Data Generator** run configuration. Click **Run, Run Configurations**, select **Account Generator**, and then click the **duplicate** icon.
2. Rename the new configuration as **Transaction Generator**.
3. Click on the **Arguments** tab and change the arguments to:
`-add data-generator-project/csvData 20190801`
The second argument is the start date (yyyymmdd format).
4. Click **Run**.



The Data Generator will not create two sets of transactions for the same month. If you have transactions for July 2019, a date parameter of 20190701 will cause an error. The day part of the date index is always ignored; transactions always run from the first to the last specified day of the month. The Data Generator randomizes the number of transactions and the days they appear on for each customer account.

Calling COBOL from Java

In this section we will use Visual COBOL's Smart Linkage feature to generate classes that wrap the linkage section of the ACCOUNT-STORAGE-ACCESS program. This gives you a simple way of calling a COBOL program directly from Java without writing any wrapper code yourself. Smart Linkage is a very quick way to get started and is often all you need.

However, writing your own wrapper classes provides a greater flexibility and can provide an API that is easier to consume from Java. We'll describe a set of wrapper classes in the next chapter, "A COBOL-Based REST Service."

Smart Linkage is a compiler option controlled by a set of directives. When you compile a program with Smart Linkage the compiler creates a .class file for each data item in the linkage section. The classes provide getters/setters for the child items in each group item. These getters and setters also map the COBOL data types to data types that can be consumed directly from Java. For example, a pic x(80) is converted to a java.lang.String. The COBOL product documentation contains a full list of the mappings between COBOL and managed data types.

Smart Linkage also renames items on the fly to fit Java rules and conventions; all hyphens are removed and names are folded to camel-case. For example, ACCOUNT-NUMBER will appear as getAccountNumber and setAccountNumber.

In the "Generating Data" section, the DataBuilder classes used the COBOL call verb to call entry points in ACCOUNT-STORAGE-ACCESS and were able to interpret COBOL data directly. We are now going to change the compiler directives for the Business Rules

project and make one small source code change to AccountStorageAccess.cbl so that we can access it directly from Java code.

To update the directives:

1. In the COBOL Explorer, right-click **BusinessRules** and click **Properties**.
2. Expand **Micro Focus** and select **Build Configuration**.
3. Under General, scroll down the **Settings** until you get to **Additional Directives**.
4. Enter the following into the **Additional Directives** field:
`ilsmartlinkage ilnamespace(com.mfcobolbook.businessrules)
ilcutprefix(lnk)`
5. Click **Apply and Close**.

The compiler creates a JVM class called ACCOUNT-STORAGE-ACCESS when compiling the ACCOUNT-STORAGE-ACCESS program (this is in the program-id header), but this name contains hyphens, which are not legal in Java identifiers. We are going to change the program-id header to create a legal Java identifier so we can call it from our Java client class. To do this:

1. Open **AccountStorageAccess.cbl** and locate the program-id header.
2. Add the following clause to the program-id header before the full-stop:
as "AccountStorageAccess"
The complete program-id should now read:
program-id. ACCOUNT-STORAGE-ACCESS as "AccountStorageAccess".
3. Save the changes. Eclipse rebuilds the project.

The BusinessRules project in this chapter is now configured with the following compiler directives:

- `ilsmartlinkage` turns on smart linkage
- `ilnamespace(com.mfcobolbook.businessrules)` specifies that all the generated .class files are inside package `com.mfcobolbook.businessrules`
- `ilcutprefix(lnk)` specifies that the prefix `lnk` should be removed from data names when mapping to getters and setters

For the full list of directives that affect Smart Linkage code generation, see the Micro Focus Visual COBOL documentation.

Listing 5-10 shows a short Java program that reads all the customer records using the ACCOUNT-STORAGE-ACCESS program. The `as "AccountStorageAccess"` clause added to the program-id in the previous set of steps means the class generated is

AccountStorageAccess, which is a valid identifier in Java, and this is the name the Java program uses to call it.

In subsequent chapters, all Java calls to ACCOUNT-STORAGE-ACCESS will go through Visual COBOL classes, where ACCOUNT-STORAGE-ACCESS is a legal identifier; we won't use the as clause again after this chapter. You need to understand these subtleties when using Smart Linkage as COBOL program names often include hyphens.

Listing 5-10 *The Java RecordReader class*

```
package com.mfcobolbook.smartlinkageclient;

import com.mfcobolbook.businessrules.AccountStorageAccess;
import com.mfcobolbook.businessrules.CustomerRecord;
import com.mfcobolbook.businessrules.Status;

public class RecordReader
{
    public static void main(String[] args)
    {
        AccountStorageAccess access = new AccountStorageAccess();
        Status s = new Status();

        access.OPEN_CUSTOMER_FILE("R", s);
        if (s.getStatus().equals("00"))
        {
            CustomerRecord record = new CustomerRecord();
            record.setCustomerId(0);
            access.READ_CUSTOMER_RECORD("S", record, s);
            while (s.getStatus().equals("00"))
            {
                access.READ_CUSTOMER_RECORD("N", record, s);
                if (s.getStatus().equals("00"))
                {
                    System.out.println(
                        String.format("Customer Record %d %s %s",
                            record.getCustomerId(),
                            record.getFirstName(),
                            record.getLastName()));
                }
            }
            access.OPEN_CUSTOMER_FILE("C", s);
        }
    }
}
```

```
        System.out.println(
            String.format("Could not open file, %s",
                s.getStatus())));
    }
}
}
```

The program creates an instance of AccountStorageAccess, which is in package com.mfcobolbook.businessrules. A procedural COBOL program like AccountStorageAccess does not have any way of defining a namespace or a package, but Eclipse tooling often works better when you do. So the `i1namespace` directive in the BusinessRules project defines a package name.

Each entry point in AccountStorageAccess appears as a public instance method. Some of the parameters to these instance methods are COBOL group items (which do not have a direct equivalent in Java) and some of them are passed by reference (which you can't do in Java). This is why `ilsmartrlinkage` generates classes based on the data definitions in the `Linkage` section of a program.

For example, every entry point in AccountStorageAccess is passed, by reference, a two-byte argument for the file status. If you look in the bin/com/mfcobolbook/businessrules directory of the BusinessRules project, you can see a `Status.class` file. The compiler has generated this as follows:

- The entry points refer to a parameter called `LNK-STATUS`. The `ilcutprefix(LNK)` directive removes `LNK` from the name. The naming rules for Smart Linkage then camel-case the rest of the name (leaving us with `Status`).
- The `Status` class has getters and setters for `FileStatus1` and `FileStatus2`, as well as for `Status` (which is a string created by concatenating `FileStatus1` and `FileStatus2`). If you look at the definition of `LNK-STATUS` in Listing 5-4 you can see how these have been arrived at.

So to invoke the `OPEN_CUSTOMER_FILE` method, we create a `Status` object and pass it as the second argument (the first is a string representing the file open mode). The called program can set the value of the `FileStatus` fields on this object, so the caller can read them afterwards.

To read from the file, we pass in a `CustomerRecord` object and `AccountStorageAccess` sets the results into it so that they can be read back by the caller. The compiler has generated wrapper classes for all the parameters passed to and from the program, and objects are returned with the expected values set. Smart Linkage enables us to call procedural COBOL programs directly from Java without Java needing to understand COBOL data structures, and without us needing to change the original COBOL code.

However, if we want to use our COBOL code in any kind of multi-threaded environment (like an application server), we need something extra. The COBOL JVM run-time

has to replicate all the semantics of legacy procedural COBOL, and that means that there is some shared state between each instance of a running COBOL program that is not directly visible to the programmer.

We can handle this using run units, which is something that we will cover in the next chapter. We will also create some wrapper classes in Visual COBOL that will provide a Java friendly API to the procedural code of our example. Although Smart Linkage gives us a very quick way of reaching COBOL code from Java, it's a style that may feel a little odd to the Java programmer. Visual COBOL provides a very natural bridge between Java and COBOL; you can code call statements to access the COBOL code, but wrap them inside methods and classes that provide a more natural API to Java.

To import the SmartLinkageClient program:

1. Open the Eclipse workspace where you imported the other example code for this chapter.
2. Click **File > Import**, select **General, Existing Projects into Workspace** from the **Import** wizard, and then click **Next**.
3. In the **Select root directory** field, enter the folder with the Chapter 5 examples, select SmartLinkageClient, and then click **Finish**.
4. In the COBOL Explorer, right-click SmartLinkageClient and click **Maven, Update Project**. This ensures all the Maven dependencies in the POM file (which include our BusinessRules project) are added to the Eclipse project as explained in Chapter 2.

To run the SmartLinkageClient program:

1. Click **Run > Run Configurations** to display the Run Configurations dialog.
2. In the left pane of the **Run Configurations** dialog, select **Java Application** and click the **New Configuration** icon.
3. Name the configuration **SmartLinkageClient** and set the Main class as **com.mfcobolbook.smartlinkageclient.RecordReader** (click the **Search** button next to the Main class field to pick it from a list).
4. Click on the **Environment** tab and add the following three environment variables (where chapter-5-examples is the folder where you installed the example files):

```
dd_CUSTOMERFILE  
= chapter-5-examples/Data/customer.dat  
dd_ACCOUNTFILE  
= chapter-5-examples/Data/account.dat
```

```
dd TRANSACTIONFILE  
= chapter-5-examples/Data/transaction.dat
```

The ACCOUNT-STORAGE-ACCESS program will need these environment variables to be able to find the data files.

5. Click the **Run** button.

The SmartLinkageClient reads all the customer records and displays them on the console.

Summary

In this chapter, you have learned about a small suite of COBOL programs that use indexed files. You then learned how to use Smart Linkage to generate wrappers for the COBOL data and read some records using a small Java program. In the next chapter, you will learn how to write a simple interop layer using Visual COBOL and then create a REST service that uses the business logic defined by our simple example.

CHAPTER 6



A COBOL-Based REST Service

In this chapter, we will use the example COBOL code from the previous chapter as the back end to a self-contained REST service running on Spring Boot. This chapter covers:

- Visual COBOL wrapper classes
- Run Units
- Spring Boot

The Application

We are going to create a simple REST (Representational State Transfer) service that enables access to the COBOL data and to the functionality that calculates monthly interest for an account. A REST service provides a set of HTTP endpoints that can be used to read or write information. This is the first step in making COBOL functionality available as microservices. We'll discuss microservices later in the book.

In a REST application, each endpoint is structured to reflect the resources it provides access to. Requests use HTTP verbs to determine which Create (POST), Read (GET), Update (PUT), or Delete (DELETE) operation is to be carried out on a resource. For example, carrying out a GET on `http://myserver/service/customer/1` retrieves the customer record with ID 1.

The application consists of three projects, each of which builds to a jar file. As in the previous chapters, we'll use Maven to store our COBOL projects as artifacts in the local repository so that they can be used from a Maven Java project. We use Spring Boot to

create a Web server application; Spring Boot applications are always built using either Maven or Gradle.

Figure 6-1 shows the main components of the application. Each large box denotes a separate project that builds into a jar file.

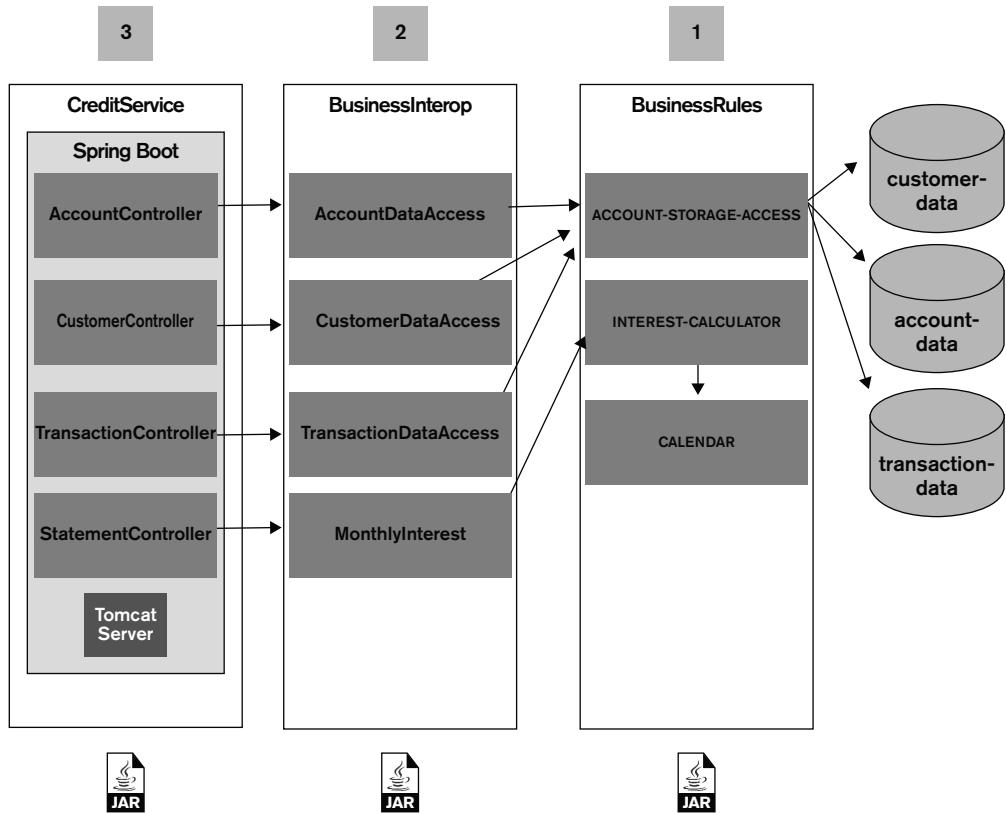


Figure 6-1 Components of the REST service

The dependencies run from left to right on the diagram. BusinessRules, labeled as Box 1, is the procedural code covered in the previous chapter. Apart from recompiling to JVM, the original procedural COBOL code is left untouched.

BusinessInterop, labeled as Box 2, is an interoperation layer written in Visual COBOL. As well as making the COBOL data accessible from Java (which we used Smart Linkage for in the previous chapter), it provides an API that translates COBOL semantics and idioms into ones more easily consumed by Java.

CreditService, labeled as Box 3, is the Spring Boot web application with the controllers that provide the REST endpoints. The diagram is slightly misleading; Box 3 reflects the code in the CreditService project, but when packaged into a jar file by Maven, the Spring Boot Maven plugin ensures that all the dependencies needed to run the application are

inside one jar file (including those in Box 1 and Box 2 in the diagram). This simplifies the deploying and running of the application.

The following section shows you how to import and run the application; subsequently, we'll spend more time going through the actual details of what's inside Box 2 and Box 3 in the diagram.

Controllers here are Java classes with methods that define our REST endpoints. Search for “spring boot getting started restful service” on the web to find a good tutorial on the spring.io website about creating a REST service.

Running the Application

Download the Chapter 6 examples (for instructions to download the examples, see “Downloading the Examples” in Chapter 1) and import BusinessRules and BusinessInterop by following the procedure in the Chapter 5 section entitled “Importing the Example Projects.” Update the Custom Builder for each of these projects and make sure they are building correctly; then import the CreditService project. If it doesn't build, try updating the Eclipse project from Maven (right-click **CreditService** and click **Maven > Update** on the **context** menu).

Now you can run the CreditService. First, create a Run Configuration:

1. Click **Run > Run Configurations**.
2. Select **Java Application** and then click the **New** button.
3. Name the configuration **CreditService** and then specify project **CreditService**.
4. Set the Main class to **com.mfcobolbook.creditservice.webservice.WebserviceApplication**.
5. Click the **Environment** tab and then create three new environment variables: **dd_ACCOUNTFILE**, **dd_CUSTOMERFILE** and **dd_TRANSACTIONFILE**, respectively. They should point to **account.dat**, **customer.dat** and **transaction.dat** files, respectively. If you don't already have these files (they are not included in the example), go back to the section entitled “Generating Data” in Chapter 5 and follow the instructions to create them.
6. Click **Run** to start the application and then save the configuration. The application starts an embedded Tomcat Server, which is listening on port 8080.

You can change the port number by editing **src/resources/application.properties**. Change the value of the **server.port** property.

The application displays a lot of console messages as it starts up. If it starts successfully, you should see something like this at the end (we've omitted the timestamps and other preamble from the logger output):

```
Tomcat started on port(s): 8080 (http) Started WebserviceApplication in 3.593
seconds (JVM running for 4.126)
```

7. To check the application is running correctly, open a web browser and try the following URL: <http://localhost:8080/service/customer/1>

If the application is working correctly, you should see some JSON returned:

```
{"customerId":1,"lastName":"Parrot",
    "firstName":"Worthington"}
```

If you don't get any output, or you get an HTTP error code, look at the application console in Eclipse for error messages or exceptions to help you trace the problem. The most likely problem is that the application can't find the data files.



For simplicity, we are using http with our examples; otherwise we will have to create and install SSL certificates. Any production application should always use https for data transport.

The Interoperation Layer

In this section we'll look at Box 2 in Figure 6-1. In the previous chapter, we used Smart Linkage to call our legacy COBOL directly from Java. Smart Linkage gives you a quick way to map COBOL data in group items to objects that can be used in Java, but it doesn't do anything to make procedural COBOL code easy for Java programs to work with. By writing an interoperation layer in Visual COBOL code you can bridge the gap between COBOL and Java semantics.

Visual COBOL classes can call COBOL entry points directly and can use the existing copybooks to access the record structures used in COBOL programs. This enables you to create a modern API that works more naturally with Java (for example, throwing exceptions to signal errors instead of requiring clients to check status codes after every invocation).

In the following sections, we will look how the interoperation layer provided by the BusinessInterop project mediates between procedural COBOL and Java. It provides:

- Data access classes to manage file operations
- Data transfer classes that provide us with types matching the records in the files
- Some exception classes for signaling errors
- A class for calculating interest called MonthlyInterest

Data Access and Transfer Classes

Figure 6-2 shows the classes for reading and writing to the data files. The AbstractBusinessAccess class encapsulates some common code for connecting to the ACCOUNT-STORAGE-ACCESS program (see the previous chapter) and for opening and closing files. The AccountDataAccess, CustomerDataAccess, and TransactionDataAccess classes provide methods for reading, writing, and searching account, customer, and transaction records.

Three other classes provide a Java-friendly abstraction for the account, customer, and transaction records, defined in the copybooks examined in the previous chapter. These are the AccountDto, CustomerDto, and TransactionDto classes.

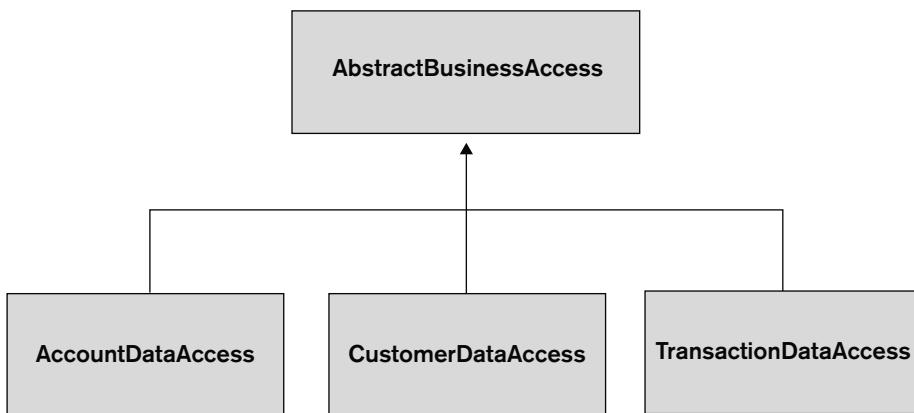


Figure 6-2 The data access classes

All three data access programs work in quite similar ways, so we will examine only AccountDataAccess and AbstractBusinessAccess in depth.

The AbstractBusinessAccess Class

The AbstractBusinessAccess class provides common logic for opening and closing files, as this is the same for all three subclasses. You might find it helpful to look at this code in the context of the ACCOUNT-STORAGE-ACCESS program that is part of the Business-Rules project you imported in the section entitled “Running the Application” previously in this chapter. The ACCOUNT-STORAGE-ACCESS program was described in Chapter 5.

Opening a File

Listing 6-1 shows the code for the AbstractBusinessAccess class, followed by a short walk-through of the logic for opening a file. This class provides an API for opening and

closing files that more closely matches the conventions and semantics of Java programs than the original procedural COBOL code. We'll call out some of the ways this is done during the explanations that follow the listing.

Listing 6-1 *The AbstractBusinessAccess class*

```
class-id com.mfcobolbook.businessinterop.AbstractBusinessAccess
    public abstract implements type AutoCloseable.
copy "PROCEDURE-NAMES.cpy".
copy "FUNCTION-CODES.cpy".
01 fileOpened          condition-value.

method-id open(openMode as type AbstractBusinessAccess+OpenMode).
    call "ACCOUNT-STORAGE-ACCESS"
    declare opcode as string
    declare allowedStatus = "00"

    evaluate openMode
        when type AbstractBusinessAccess+OpenMode::read
            move OPEN-READ to opcode
        when type AbstractBusinessAccess+OpenMode::write
            move OPEN-WRITE to opcode
            move "05" to allowedStatus
        when type AbstractBusinessAccess+OpenMode::rw
            move OPEN-I-O to opcode
    end-evaluate
    invoke openFile(opcode, allowedStatus)
    set fileOpened to true
end method.

method-id openFile(opcode as string, allowedStatus as string)
    returning result as string protected.
copy "FUNCTION-CODES.cpy".
01 pPointer           procedure-pointer.
01 fileStatus.
    03 statusByte1      pic x.
    03 statusbyte2      pic x.

    if size of opcode <> 1 then
        raise new Exception("Opcode should be one character")
    end-if
    if size of allowedStatus <> 2 then
        raise new Exception("FileStatus should be two characters")
    end-if

    invoke openEntryPointer(by reference pPointer)
```

```
call pPointer using by value opcode
      by reference fileStatus

if fileStatus <> "00" and fileStatus <> allowedStatus
  raise new Exception("Unexpected file status "
    & statusToString(fileStatus))
end-if
set result to fileStatus

end method.

method-id openEntryPointer abstract protected.
linkage section.
01 pPointer          procedure-pointer.
procedure divison using by reference pPointer.
end method.

method-id close().
if fileOpened
  invoke openFile(CLOSE-FILE, "00")
  set fileOpened to false
end-if
end method.

method-id statusToString(statusCode as string)
  returning result as string static public.
01 displayable      pic 999.
if size of statusCode <> 2
  raise new Exception("Status codes must be two characters")
end-if
if statusCode[0] <> "9"
  set result to statusCode
else
  move statusCode[1] to displayable
  set result to statusCode[0] & displayable
end-if
end method.

enum-id OpenMode.
78 #read.
78 #write.
78 #rw.
end enum.

end class.
```

The class-id header includes the abstract keyword. Just as in Java, abstract classes cannot be instantiated. This class also implements the Java AutoCloseable interface. This interface enables our data access classes to be used with the Java try-with-resources statement that closes a resource at the end of a code block.

The class has a working-storage section, which includes two copybooks and one member variable, fileOpened. The copybooks declare constants but don't allocate any member variables. PROCEDURE-NAMES.cpy declares a literal constant for each entry point defined in the ACCOUNT-STORAGE-ACCESS program; FUNCTION-CODES.cpy declares single-character function codes used by those entry points. Using constants for these values enables the compiler to flag errors that would otherwise not be picked up until runtime.

The open() method opens a file. The file that is opened is determined by the sub-class of the instance in use; this is an abstract class, so there are never any instances of AbstractAccountAccess and the method will always be executed as an instance of AccountDataAccess, CustomDataAccess, or TransactionDataAccess. This is the sequence of events to open a file:

1. A client invokes the open() method with an openMode argument.

The OpenMode enumeration is defined inside the AbstractBusinessAccess class just before the end class header (you can see it at the bottom of the listing). It provides a type-safe way for the client of the interoperation layer to specify that the file should be opened for read, write, or read/write.

2. The call "ACCOUNT-STORAGE-ACCESS" statement loads the ACCOUNT-STORAGE-ACCESS program into memory and executes its procedure division.

The procedure division of ACCOUNT-STORAGE-ACCESS does nothing, so there are no side effects to calling it every time we open a file. However, the first time the call is executed, it loads ACCOUNT-STORAGE-ACCESS, which also makes all its entry points known to the COBOL RTS. Without this step, the first call to any of those entry points would cause a COBOL RTS 114 error (program not found).

3. The next two statements declare the opcode and allowedStatus variables; these are used later in the method.
4. The evaluate statement converts the OpenMode enumeration into a single-character opcode as expected by the ACCOUNT-STORAGE-ACCESS program.

This is an example of converting from a Java semantic (use an enumeration to specify a choice) to the one used by the original COBOL code.

5. The `openFile()` method is invoked with the parameters defined by the preceding body of this method.
6. The member variable `fileOpened` is set to true.

If the `openFile()` method fails for any reason, it throws an exception and this last statement does not get executed.

The `openFile()` method contains the code to open or close a file (the same entry point in ACCOUNT-STORAGE-ACCESS does both, depending on the value of the opcode). This is the sequence of events for `openFile()`:

1. The first few lines of the method sanity-check the arguments passed in; they are both strings and this code checks that at least they are the expected size. It throws exceptions if they are not.
2. The method `openEntryPointer()` is invoked.

This is declared as an abstract method in this class; classes `AccountDataAccess`, `CustomerDataAccess`, and `TransactionDataAccess` each define an implementation of this method that provides a procedure-pointer for the entry point in ACCOUNT-STORAGE-ACCESS that opens account, customer, or transaction files, respectively.

3. The procedure-pointer is called to open the file.
4. The `fileStatus` is checked for an acceptable result; an exception is thrown if it appears there was an error

A file status of “00” indicates success. But not all non-zero codes indicate failure. For example, a code of “05” (optional file not present) is returned when a non-existent file is opened for write, but it isn’t an error because the file will be created as a result of being opened for write.

The `close()` method is required for this class’ implementation of the `AutoCloseable` interface and closes a file (if one has been opened). This is another example of the InteroperationLayer providing a Java semantic, which doesn’t exist in the original COBOL code, to clients of the API.

The last method is `statusToString()`, which converts the two-byte COBOL file-status into a string for inclusion in exception messages. In the next section, we’ll look at the code in the `AccountDataAccess` class, which actually reads and writes records.

Procedure-pointers

A procedure-pointer is a pointer that points to a piece of code; calling the procedure-pointer executes the code. Java uses callback mechanisms like interfaces, anonymous classes, and anonymous functions for late-binding to code rather than procedure-pointers.

The AccountStorageAccess Class

In this section, we'll take a look at how the AccountStorageAccess class provides a Java-friendly API to the business logic defined in our procedural COBOL program. Rather than listing the entire program in one go, it's split it into smaller pieces so that the explanations aren't too far from the code.

Setting a Procedure-Pointer

Listing 6-2 shows just the beginning of the class, together with the first method, `openEntryPointer()`.

Listing 6-2 The `AccountDataAccess` class

```
class-id com.mfcobolbook.businessinterop.AccountDataAccess
    inherits AbstractBusinessAccess public.

    working-storage section.
    copy "PROCEDURE-NAMES.cpy".
    copy "FUNCTION-CODES.cpy".

    method-id openEntryPointer override protected.
    linkage section.
    01 pPointer procedure-pointer.
    procedure division using by reference pPointer.
        set pPointer to entry OPEN-ACCOUNT-FILE
    end method.
```

This class inherits from `AbstractBusinessAccess`, so it must implement the abstract method `openEntryPointer()` declared in that class. It could appear anywhere in the class, but we've put it at the top. There are three entry points in the procedural ACCOUNT-STORAGE-ACCESS program for opening each of the three files the program manages (accounts, customers, and transactions).

The AbstractBusinessAccess class has all the code to open files, but it needs to call the appropriate entry point for the type of the file it is opening. We are using COBOL procedure-pointers to call the appropriate entry point. Each of our concrete data access classes returns a procedure-pointer set to the appropriate entrypointer.

The syntax of the `openEntryPointer()` method looks a little different to the other ones in this book; the arguments and return values aren't defined as part of the `method-id` header. Instead, there is a `linkage section` declaring the arguments and they are named in the `procedure division` header. This is an older flavor of object-oriented COBOL syntax; we've used it here because the newer version doesn't work with arguments that don't have direct equivalents in the Java type system (or don't map to Java types in the way that some COBOL types are mapped automatically to Java Strings or numerics).

Adding, Updating and Deleting Records

The `addAccount()`, `updateAccount()`, and `deleteAccount()` methods enable a client to add, update, or delete records. Listing 6-3 shows the code for these three methods. There are similar methods in the `CustomerDataAccess` and `TransactionDataAccess` classes for updating customer and transaction records. Before invoking any of these methods, the file must be opened for write or read/write.

Listing 6-3 Methods for updating the account file

```
method-id addAccount (account as type AccountDto)
    returning accountId as binary-long.
copy "ACCOUNT-RECORD.cpy" replacing ==(PREFIX)== by LS.

01 functionCode pic x.
01 fileStatus.
03 statusByte1 pic x.
03 statusByte1 pic x.

declare nextId as binary-long
declare lastAccount = self::getLastAccount()
if lastAccount = null
    set nextId = 1
else
    set nextId = lastAccount::accountId + 1
end-if
move WRITE-RECORD to functioncode
invoke account::getAsAccountRecord(LS-ACCOUNT)
move nextId to LS-ACCOUNT-ID
call WRITE-ACCOUNT-RECORD using by value functionCode
    by reference LS-ACCOUNT
```

```
fileStatus
if fileStatus <> "00" and fileStatus <> "02"
    raise new RecordWriteException(
        "Couldn't add new account record")
end-if
set accountId to nextId
end method.

method-id updateAccount (account as type AccountDto)
    returning success as condition-value.
copy "ACCOUNT-RECORD.cpy" replacing ==(PREFIX)== by LS.
01 functionCode pic x.
01 fileStatus.
03 statusByte1 pic x.
03 statusByte1 pic x.

move UPDATE-RECORD to functioncode
invoke account::getAsAccountRecord(LS-ACCOUNT)
call WRITE-ACCOUNT-RECORD using by value functionCode
    by reference LS-ACCOUNT
        fileStatus
if fileStatus <> "00" and
    fileStatus <> "02" and
    fileStatus <> "23"
    raise new RecordWriteException(
        "Couldn't update record")
end-if
set success to fileStatus <> "23"
end method.

method-id deleteAccount (accountId as binary-long)
    returning success as condition-value.
copy "ACCOUNT-RECORD.cpy" replacing ==(PREFIX)== by LS.
01 fileStatus.
03 statusByte1 pic x.
03 statusByte1 pic x.
move accountId to LS-account-ID
call DELETE-ACCOUNT-RECORD USING by reference LS-ACCOUNT
        fileStatus
set success to (fileStatus = "00")
end method.
```

All three of these methods start with a copy statement that declares a COBOL group item for the account record, called LS-ACCOUNT. This is part of the local storage for the method.

The `addAccount()` method takes a single parameter, an instance of `AccountDto`. `AccountDto` is a wrapper for the COBOL account record defined in the `ACCOUNT-RECORD.cpy` copybook that was described in the Chapter 5 section entitled “Storing Data in Indexed Files.” The `AccountDto` class provides a property corresponding to each field in the record and some helper methods.

The `addAccount()` method has been designed to always add a new account record with an account ID 1 greater than the last account ID in the file (like a database table where the primary index is set to autoincrement). It goes through the following steps to add a new account record:

1. Invokes the `getLastAccount()` method to find the record with the highest numbered account ID. If there are no records, the account ID is set to 1; otherwise it is set to the last account ID + 1.
2. Populates the `LS-ACCOUNT` record with the data passed in in the `AccountDto` object, using the `getAsAccountRecord()` helper method.
3. Calls the `WRITE-ACCOUNT-RECORD` entry-point in the `ACCOUNT-STORAGE-ACCESS` program to add the new record.
4. Checks the file status. A status of “02” indicates an allowed duplicate alternate key. The customer ID is the alternate key; duplicates are allowed as the system allows for a customer to have more than one account. A status of “00” indicates success. Any other status indicates that the operation has failed, which throws an exception. The caller doesn’t need to check status codes; errors are signaled through exceptions as is usual for a Java API.
5. The method returns the ID of the new account to the caller.

The `updateAccount()` method is similar, but this returns a success or failure Boolean. This is because attempting to update an account that does not already exist will fail, but depending on the semantics of your system, it might not count as an error. Any status that isn’t OK, any duplicate alternate key, or any record not found still throws an exception.

The `deleteAccount()` method also returns a condition-value (Boolean) to indicate success. Again, it isn’t necessarily an error condition if you can’t delete a record that doesn’t exist.

Reading and Finding Records

In this section, we will look at the methods that enable us to search and read the account file. All our files have been set up for dynamic access. This gives us the ability to read a

random record but also to read a group of records sequentially. So we can both read the record for account ID 13 but also read all the records in order starting from any given id.

This flexibility does mean that the logic for reading files is a little more complex than the logic for writing them. Listing 6-4 shows the section of code in our original procedural program to read a record,

Listing 6-4 Procedural code to read an account record

```
ENTRY READ-ACCOUNT-RECORD using by value LNK-FUNCTION
                                by reference LNK-ACCOUNT LNK-STATUS
    evaluate LNK-FUNCTION
        when START-READ
            move LNK-ACCOUNT TO FILE-ACCOUNT
            start ACCOUNT-File key >= FILE-ACCOUNT-ID
                                of FILE-ACCOUNT
        when READ-NEXT
            read ACCOUNT-File next
    end-evaluate
    move FILE-ACCOUNT to LNK-ACCOUNT
    move file-status to LNK-STATUS
    goback
.
```

This entry point enables you to either start or read... next from the file. The start verb positions a cursor at the first record either equal to or greater than the ID passed in with the LNK-ACCOUNT argument. The read verb reads the record and moves the cursor. If you want to continue reading records in account ID order, keep repeating the READ-NEXT operation.

The code has been designed to allow all the records to be read even if you don't know the first account ID; just pass in a value of 0 for the ID and the start will place the cursor at the first record in the file (assuming IDs are always greater than zero). However, that does mean if you want a specific record (and no other record), you need to check the id of the first record returned. For example, if you attempt to read account ID 7 and there is no such account, you will get returned the first record with an ID greater than 7.

The job of our interoperation layer is to turn this very COBOL-centric logic into something that looks like the API a Java programmer might expect. Listing 6-5 shows the code for retrieving accounts by ID.

Listing 6-5 Interoperation code for reading accounts

```
method-id getAccount (accountId as binary-long)
                    returning result as type AccountDto.
    perform varying result through getAccount(accountId, false)
        goback
    end-perform
```

```
end method.

iterator-id getAccounts () yielding result as type AccountDto.
    perform varying result through getAccount(1, true)
        goback
    end-perform
end iterator.

iterator-id getAccount (accountId as binary-long,
                        getAll as condition-value)
    yielding result as type AccountDto
    protected.

01 done condition-value.
01 fileStarted condition-value.
01 opcode string.
01 fileStatus string.

    perform until done
        if not fileStarted
            move START-READ to opcode
            invoke readFileById(accountId, opcode, getAll,
                                by reference result)
            set fileStarted to true
        end-if
        move READ-NEXT to opcode
        set fileStatus to readFileById(accountId, opcode, getAll,
                                         by reference result)
        if result = null
            stop iterator
        else
            if fileStatus = "00" and getAll = false
                set done to true
            end-if
            goback
        end-if
    end-perform
end iterator.

method-id readFileById (#id as binary-long, opcode as string,
                        getAll as condition-value,
                        by reference dto as type AccountDto)
    returning result as string
    protected.

copy "ACCOUNT-RECORD.cpy" replacing ==(PREFIX)== by LS.
01 fileStatus.

03 statusByte1 pic x.
```

```
03 statusByte1 pic x.  
01 accountType binary-char.  
    move #id to LS-ACCOUNT-ID  
    call READ-ACCOUNT-RECORD using by value opCode  
                                by reference LS-ACCOUNT  
                                fileStatus  
  
    set result to fileStatus  
    if fileStatus = "23" or fileStatus = "10" or  
        (LS-ACCOUNT-ID <> #id and not getAll)  
        set dto to null  
    else  
        if fileStatus = "00" or fileStatus = "02"  
            move LS-TYPE to accountType  
            set dto to new AccountDto(LS-ACCOUNT-ID, LS-CUSTOMER-ID,  
                LS-BALANCE, accountType,  
                LS-CREDIT-LIMIT)  
        else  
            raise new FileReadException("Could not read file "  
                & super::statusToString(fileStatus))  
        end-if  
    end-if  
  
end method.  
  
method-id getLastAccount () returning result as type AccountDto.  
copy "ACCOUNT-RECORD.cpy" replacing ==(PREFIX)== by LS.  
01 fileStatus.  
    03 statusByte1 pic x.  
    03 statusByte1 pic x.  
        call READ-LAST-ACCOUNT-RECORD using by reference LS-ACCOUNT  
                                fileStatus  
  
        evaluate fileStatus  
        when "00"  
            declare accType as binary-char = LS-TYPE  
            set result to new AccountDto(LS-ACCOUNT-ID,  
                LS-CUSTOMER-ID  
                LS-BALANCE,  
                accType,  
                LS-CREDIT-LIMIT)  
        when "46"  
            set result to null  
        when other  
            declare fs as string  
            set fs to fileStatus  
            raise new FileReadException(fs)
```

```
end-evaluate  
end method.
```

The `getAccount()` method retrieves one specific account by account ID. The `getAccounts()` iterator returns a `java.lang.Iterable<AccountDto>` that enables a client to iterate over all the account records using a `for` loop or by using the `forEach()` method of `Iterable<?>`. We'll explain Visual COBOL iterators in more detail later in this chapter, but first we'll look at the code for retrieving records.

The first `getAccount()` method takes a single ID as the argument and returns either the matching account or null if no record is found. It returns the first object returned by the `getAccount()` iterator (by doing a `perform varying` but returning as soon as the first record is found). This iterator, together with a helper method called `readRecordById()`, wraps up all the logic to start the file at a particular location and then retrieves a record, together with raising exceptions for any unexpected file status codes.

The `getAccount()` iterator consists of a `perform` loop that is only complete when the `done` flag is set to true. Because it is defined as an iterator, it returns the value in result each time it executes the `goback` statement at the bottom of the loop. The first time this code is executed, it calls the file-handling logic in the procedural program to start the file at the specified index; it then reads the record found there. Iterators are explained in more detail in the next section, but in effect, this code returns the next record in the sequence each time it is executed and maintains its local state between invocations. When it hits the `stop iterator` statement, there are no more records to return.

The `readByFileId()` method has the logic that calls the procedural program and translates unexpected file status values into an exception. If the `getAll` flag is set to false, it will return a record only if it matches the ID originally requested. This is how we differentiate between fetching all the records from a particular index and fetching only the exact record at an index. If we ask for account ID 5 and there is no record with that ID, but there is one with ID 7, it will be retrieved by the original logic, although the ID in the record returned will not be the same as the one originally requested and the file status will be "02" rather than "00".

The `getAccounts()` iterator retrieves all the records in the file by starting from account ID 1 and calling the `getAccount()` iterator with the `getAll()` flag set to true.

Iterators

Iterators in Visual COBOL always return an `Iterable<?>` object, with the generic type defined by the returning clause of the iterator. Listing 6-6 shows an example class with an iterator that is called by the main method. The iterator has a returning clause of `binary-long`, so it returns an `Iterable<Integer>` (`binary-long` is equivalent to the Java `Integer` type).

The iterator uses `perform varying` to read an array of Fibonacci numbers, but it returns only the even values. The `getEven()` iterator returns the contents of `result` each time it hits the `goback` statement. Its position in the `perform varying` statement is maintained between invocations.

Listing 6-6 Simple iterator program

```
class-id com.mfcobolbook.examples.Iterators public.  
01 fibonacciArray binary-long occurs any value  
    table of binary-long (1 2 3 5 8 13 21 34 55 89 144 233  
                           377).  
method-id main (arg as string occurs any) static.  
    declare ic as type Iterators = new Iterators()  
  
    perform varying evenNumber as binary-long through ic::getEven()  
        display evenNumber  
    end-perform  
end method.  
  
iterator-id getEven yielding result as binary-long.  
    perform varying i as binary-long through fibonacciArray  
        if i b-and 1 = 0 *-> Binary AND i with 1.  
            *-> Result is zero for even values of i  
            set result to i  
            goback  
        end-if  
    end-perform  
end iterator.  
  
end class.
```

What is really happening here? The compiler generates some hidden code to make all this work so elegantly. It constructs an implementation of `Iterable<Integer>` that will be instantiated and returned from the method. `Iterable` is a Java interface that provides `forEach()` and `iterator()` methods.

The `iterator()` method returns a `java.langIterator<Integer>` (in this case) and provides methods `hasNext()` and `next()`, which enable a client to retrieve all the elements from a list, or array, or whatever data structure backs the iterator. Visual COBOL takes the code in the `iterator` method, adds some code to maintain state between invocations, and puts it into the `Iterable` it has generated, which can now support the `hasNext()`, `next()`, and `forEach()` methods of the `Iterable<?>` and `Iterator<?>` interfaces.

This enables a client to iterate through all the accounts (using a `for` loop) or use the `forEach()` method to apply an action to each element.

The iterator feature in Visual COBOL is similar to the `yield` keyword that simplifies writing iterators in C#. Visual COBOL adopted the feature as part of the implementation of .NET; it is available in JVM to maintain language parity between Visual COBOL .NET and Visual COBOL JVM.

Visual COBOL iterators work well with Java because in COBOL JVM, they compile to interfaces that are part of the `java.lang` namespace, so they support Java language constructs like the `for` loop and the `forEach()` method introduced in Java 8. Listing 6-7 shows a Java method that uses `forEach()` and a lambda to display all account records using the iterator methods in the `AccountDataAccess` class.

Listing 6-7 Using `foreach` to display all account records

```
public void forEachAllAccounts() {
    try (AccountDataAccess cda = new AccountDataAccess()) {
        cda.open(AbstractBusinessAccess .OpenMode.read);
        cda.getAccounts().forEach( dto -> System.out.println(
            String.format("%d %s",
                dto.getAccountId(),
                dto.getCustomerId(),
                dto.getCreditLimit().toString())));
    }
}
```

The MonthlyInterest Class

Box 2 in Figure 6-1 showed a `MonthlyInterest` class as well as the three classes for data access. As shown in Listing 6-8, `MonthlyInterest` calls the `CALCULATE-INTEREST` program examined in the previous chapter to work out how much interest is owed on a particular account in a given month.

Listing 6-8 The `MonthlyInterest` class

```
$set ilusing(java.time)
$set ilusing(com.microfocus.cobol.runtimeservices)
class-id com.mfcobolbook.businessinterop.MonthlyInterest public.
copy "PROCEDURE-NAMES.cpy".
01 valuesCalculated          condition-value.
01 dayRate                   decimal.
01 startingAmount            decimal.
01 endingAmount              decimal.
01 startDate                 type LocalDate.
01 accountId                 binary-long.
01 minimumPayment            decimal.
```

```
01 interest          decimal.
01 initialized      condition-value.
01 runUnit          type RunUnit.

method-id init (dayRate as decimal, startingAmount as decimal,
                startDate as type LocalDate,
                accountId as binary-long).
    set self::dayRate to dayRate
    set self::startingAmount to startingAmount
    set self::startDate to startDate
    set self::accountId to accountId
    set initialized to true
end method.

method-id close.
end method.

method-id calculate().
copy "DATE.cpy" replacing ==(PREFIX)== BY ==START==.
01 tempResult          PIC S9(12)V99.
01 tempDayRate         PIC 99v9(8) comp-3.
01 tempInterestPayment PIC S9(12)V99.
01 tempMinimumPayment PIC S9(12)V99.
01 fileStatus.

03 statusByte1          pic x.
03 statusByte2          pic x.

if not initialized
    raise new UninitialisedObjectException("No data provided")
end-if

if not valuesCalculated
    call "INTEREST-CALCULATOR"
    set START-YEAR of START-DATE to startDate::getYear()
    set START-MONTH of START-DATE to startDate::getMonthValue()
    set START-DAY of START-DATE to startDate::getDayOfMonth()
    move startingAmount to tempResult
    move dayRate to tempDayRate
    call CALCULATE-INTEREST using by value START-DATE
                                accountid
                                by reference tempDayRate
                                tempResult
                                tempInterestPayment
                                tempMinimumPayment
                                fileStatus

    if fileStatus <> "00"
        if fileStatus = "23"
            raise new RecordNotFoundException(

```

```
        "No transactions for account " & accountId)
    else
        raise new Exception("Could not calculate result")
    end-if
end-if
set endingAmount to tempResult
set interest to tempInterestPayment
set minimumPayment to tempMinimumPayment
set valuesCalculated to true
end-if

end method.

method-id getMinimumPayment() returning result as decimal.
if not valuesCalculated
    invoke calculate()
end-if
set result to minimumPayment
end method.

method-id getEndingAmount() returning result as decimal.
if not valuesCalculated
    invoke calculate()
end-if
set result to endingAmount
end method.

method-id getInterestPayment() returning result as decimal.
if not valuesCalculated
    invoke calculate()
end-if
set result to interest
end method.

method-id getStatementDto() returning result as type StatementDto.
if not valuesCalculated
    invoke calculate()
end-if
set result to new StatementDto (accountId, startDate,
                                minimumPayment,
                                endingAmount, interest)

end method.

end class.
```

To use this class, initialize it with the daily interest rate, initial balance, start date, and account ID. All the result values are available as individual properties or in the form of a StatementDto object. The calculation is carried out the first time any of the accessors is called.

You might be wondering why this class doesn't take all the values it requires in the constructor rather than requiring a separate init() method. It will become clearer when we look at Run Units later in this chapter.

Creating a REST Interface

In this section, we will look at Box 3 in Figure 6-1, the REST layer. So far in this chapter we've looked at creating an interoperation layer that turns the COBOL idioms in our original procedural program into an API that makes sense to a Java programmer. Now we are going to use Java and Spring Boot to create a small web application that enables us to read and write data and calculate monthly interest.

What Is Spring Boot?

Spring Boot is described as an “opinionated view of the Spring platform and third-party libraries.” Spring Boot makes it easier to create Java applications by providing curated collections of dependencies for popular technologies and libraries and by providing sensible defaults for many of the things you might otherwise have to choose for yourself. It builds on top of the popular Spring Framework, which has largely supplanted J2EE as a framework for building complex applications. Spring and Spring Boot (<https://spring.io/projects/spring-boot>) are both open source software, licensed under the Apache 2.0 license, and can be used without license fees.

The REST controller introduced later in this chapter is based on the Spring Boot example for Building a RESTful Web Service (<https://spring.io/guides/gs/rest-service/>). You can generate the template for a Spring Boot application using the Spring Initializr (<https://start.spring.io>). Tell Initializr what kind of dependencies your application needs and it will create the outline of a Maven or Gradle project that includes all those dependencies.

The “opinionated view” mentioned in the first paragraph of this section enables you to reduce much of the boiler-plate code often associated with Java applications and enables you to create applications that are easy to deploy and configure.

By default, all Spring Boot web applications build into a single jar that includes all dependencies including an embedded Tomcat Server. “Deployment” means copying the jar file to an environment that can run Java and starting it up. Within a few seconds, your application is running and able to respond to http requests.

The WebServiceApplication Class

Our web service application consists of a Spring Boot Application class and the AccountController, CustomerController, StatementController, and TransactionController classes. There are also three classes used for serializing accounts, customers, and transactions (these are in the com.mfcobolbook.creditservice.forms namespace).

Listing 6-9 shows the WebServiceApplication. This contains a main method that is the entry point to start the entire application. It is annotated with @SpringBootApplication, which marks it as a Spring Boot Application so that when it is started with the SpringApplication.run() method, Spring wires up the configuration and then carries out any dependency injection needed to make the application run.

Listing 6-9 The WebServiceApplication

```
package com.mfcobolbook.creditservice.webservice;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class WebserviceApplication {

    public static void main(String[] args) {
        SpringApplication.run(WebserviceApplication.class, args);
    }
}
```

The AccountController Class

Each controller class provides a set of endpoints that enable callers to read, write, or delete records (or calculate monthly interest in the case of the StatementController). Listing 6-10 shows the AccountController class.

Listing 6-10 The AccountController class

```
package com.mfcobolbook.creditservice.webservice;

@RequestMapping("/service/account")
@RestController
public class AccountController {
    @GetMapping
    public ArrayWrapper<AccountForm> accounts() {
        List<AccountForm> accountList = new ArrayList<>();
        try (RunUnit<AccountDataAccess> ru = new RunUnit<>(
                AccountDataAccess.class)) {
            for (AccountDataAccess acc : ru) {
                accountList.add(acc.getAccount());
            }
        }
        return new ArrayWrapper<AccountForm>(accountList);
    }
}
```

```
try (AccountDataAccess accessor = (AccountDataAccess) ru
     .GetInstance(AccountDataAccess.class, true)) {
    accessor.open(AbstractBusinessAccess .OpenMode.read);
    for (AccountDto next : accessor.getAccounts()) {
        accountList.add(new AccountForm(next));
    }
}
}

return new ArrayWrapper<>(accountList);
}

@GetMapping(value = “/{id}”)
public AccountForm getAccount(@PathVariable(“id”) int id)
    throws ResourceNotFoundException {
try (RunUnit<AccountDataAccess> ru = new RunUnit<>(
    AccountDataAccess.class)) {
    try (AccountDataAccess accessor = (AccountDataAccess) ru
         .GetInstance(AccountDataAccess.class, true)) {
        accessor.open(AbstractBusinessAccess .OpenMode.read);
        AccountDto dto = accessor.getAccount(id);
        if (dto != null) {
            return new AccountForm(dto);
        } else {
            throw new ResourceNotFoundException(
                String.format(“Could not find record %d”, id));
        }
    }
}
}

@PostMapping
public ResponseEntity<AccountForm> addAccount(
    @RequestBody AccountForm account) {
try (RunUnit<AccountDataAccess> ru = new RunUnit<>(
    AccountDataAccess.class)) {
    try (AccountDataAccess accessor = (AccountDataAccess) ru
         .GetInstance(AccountDataAccess.class, true)) {
        accessor.open(AbstractBusinessAccess .OpenMode.rw);
        AccountDto dto = account.createAccountDto();
        int id = accessor.addAccount(dto);
        account.setId(id);
        return ResponseEntity.ok(account);
    }
}
}
```

```

@DeleteMapping(value = "/{id}")
public ResponseEntity<?> deleteAccount(@PathVariable("id")
                                         int id)
    throws ResourceNotFoundException {
    HttpStatus status = null;
    try (RunUnit<AccountDataAccess> ru = new RunUnit<>(
        AccountDataAccess.class)) {
        try (AccountDataAccess accessor = (AccountDataAccess) ru
            .GetInstance(AccountDataAccess.class, true)) {
            accessor.open(AbstractBusinessAccess .OpenMode.rw);
            status = accessor.deleteAccount(id)
                ? HttpStatus.NO_CONTENT
                : HttpStatus.NOT_FOUND;
        }
    }
    return new ResponseEntity<String>(status);
}

@PutMapping
public ResponseEntity<String> updateAccount(
    @RequestBody AccountForm account) {
    try (RunUnit<AccountDataAccess> ru = new RunUnit<>(
        AccountDataAccess.class)) {
        try (AccountDataAccess accessor =
            (AccountDataAccess) ru.GetInstance(
                AccountDataAccess.class, true)) {
            accessor.open(AbstractBusinessAccess .OpenMode.rw);
            AccountDto dto = account.createAccountDto();
            HttpStatus status = accessor.updateAccount(dto)
                ? HttpStatus.NO_CONTENT
                : HttpStatus.NOT_FOUND;
            return new ResponseEntity<>(status);
        }
    }
}

```

The imports are omitted to make the listing a little shorter. The class is annotated with `@RestController`, which is a Spring framework annotation that marks this class as a REST controller. It is also annotated with `@RequestMapping("/service/account")`, which indicates that all the endpoints defined in this class start with the path `/service/account`.

The first method, `accounts()`, returns an array of all the account records. It's annotated with `@GetMapping`, so to return all the accounts, a client would make an HTTP GET request to `http://hostname/service/account`. In a real-world application, you would make this a

pageable request to limit the number of records returned from a single request. The only reason this hasn't been done here is to keep things as simple as possible.

The body of the method creates a `ListArray` to hold the results and then uses the `try... with resources` statement to construct a `RunUnit<AccountDataAccess>`. A Run Unit is an object supplied by the Visual COBOL run-time to safely wrap procedural COBOL programs so that they can be called in multi-threaded environments.

It's constructed in a `try... with resources` so that it will always get closed at the end of the block; otherwise the application will have a memory leak. The `RunUnit.GetInstance()` method constructs an `AccountDataAccess` object. The `AccountDataAccess` object itself calls procedural COBOL, which is why a Run Unit is needed. The sidebar on Visual COBOL Run Units provides some extra information.

The logic of the `accounts()` method is very simple:

1. Create a Run Unit in a `try... with resources`.
2. Get an instance of `AccountDataAccess` from the Run Unit in a `try... with resources` (this ensures the underlying file resource is closed when we've finished reading).
3. Open the accessor for Read.
4. Iterate through all the accounts.
5. Create an `AccountForm` for each `AccountDto` (the `AccountForm` is our serialization object) and then store it in the `ArrayList`.
6. Return the list of accounts.

There's one small wrinkle: rather than returning the array itself, it's returned wrapped inside another object. This is to avoid a Web vulnerability known as JSON hijacking. A detailed discussion of this is not within the scope of this book, but it's a side effect of the fact that arrays in JavaScript are executable. You can find out more by searching for JSON hijacking on the web.

The result of all the Spring magic encapsulated inside a Spring RestController is that the object returned from our method is serialized and returned to the HTTP GET request as JSON.

The `AccountController` also has methods mapped to the POST, DELETE, and PUT verbs (using Spring's `@PostMapping`, `@Delete` and `@PutMapping` annotations). So all the CRUD operations of the original COBOL ACCOUNT-STORAGE-ACCESS program are now available over an HTTP REST interface. This makes it easy to create a Web UI using any modern JavaScript framework (a later chapter shows a UI written in React). It also means the functionality is available to any other application that can make HTTP calls.

Visual COBOL Run Units

The Visual COBOL run-time has to do two different things:

- Enable procedural COBOL to run exactly as it always has done, even though it is now being executed as instances of an object in a Java Virtual Machine. This means that all the global run-time state has to be preserved between calls to the procedural code.
- Enable procedural COBOL to be accessed from application servers which use a thread pool to service incoming requests efficiently.



Wrapping each instance of our procedural program in a Run Unit enables the Visual COBOL runtime to satisfy these two conflicting requirements. The Run Unit is a lightweight container that enables each procedural COBOL program to run as it always has done, without memory clashes and conflicts between different threads.

Testing the Endpoints

Start the application up as described in the section entitled “Running the Application.” There’s an explanation of automated tests for the application later in this book, but right now we just want to try some of the different endpoints to get a feeling for how the application works.

There are two free and widely used tools for manually testing an endpoint:

- cURL (Command-line URL) is an open-source command-line tool that can be downloaded from <https://curl.haxx.se>.
- Postman is a commercial tool with a GUI. Postman provides a free-use tier that does everything you need to follow along with the exercises in this book. You can download it at <https://www.getpostman.com>.

Postman is easier to use, so we’ll concentrate on that in the following examples,, but in the examples folder, there is a curl folder with a text file showing cURL commands with some sample data in separate JSON files.

To add a new account:

1. Start Postman and create a new request called **addAccount**. If prompted, create a new collection to save the request to accountService. Figure 6-3 shows the Postman UI as the new request is created.

2. Click the **GET** drop-down and change the verb to **POST**.
3. Fill in the URL as `http://localhost:8080/service/account`.
4. Click **Headers** and then add a header with the KEY **Content-Type** and VALUE **application/json**.
5. Click **Body**, click **Raw**, and then add the following JSON:

```
{
  "customerId":2001,
  "balance":999.05,
  "type":3,
  "creditLimit":4000.00 }
```

6. Click **Send**. You should see a response from the server with the Body:

```
{
  "id": 1001
  "customerId":2001,
  "balance":999.05,
  "type":3,
  "creditLimit":4000.00
}
```

The actual ID returned will depend on the last record in the file.

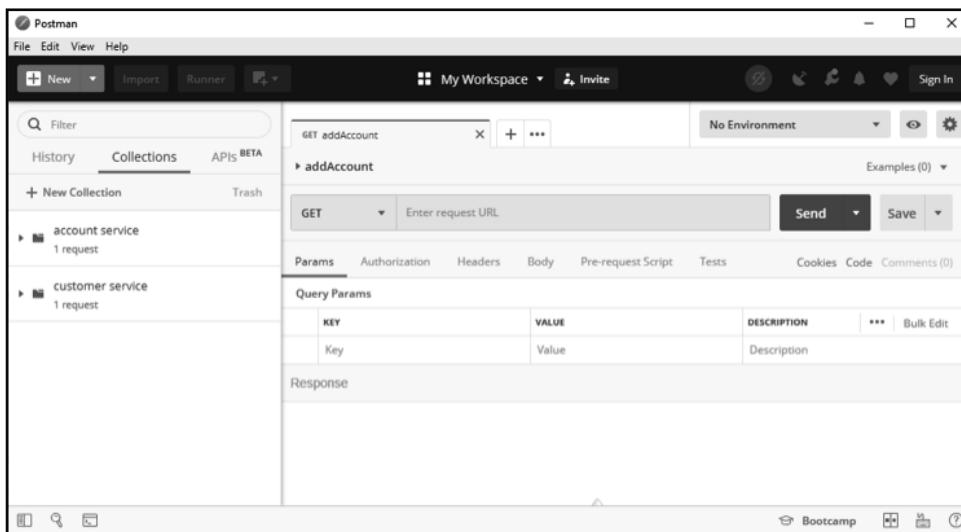


Figure 6-3 Adding a new request to Postman

To read the account:

1. Create a new request in Postman called **readAccount**.
2. Set the URL to `http://localhost:8080/service/account/1001`.

3. Click **Send**. You should get back a Body with the value of the account.

To update the account:

4. Create a new PUT request called **updateAccount**.
5. Set the URL to `http://localhost:8080/service/account`.
6. Click **Headers** and then add a header with the KEY **Content-Type** and VALUE **application/json**.
7. Click **Body**, click **Raw**, and then add the following JSON:

```
{ "id":1001,
  "customerId":2001,
  "balance":777.05,
  "type":3,
  "creditLimit":4000.00}
```

8. Click **Send**. If you reread the record using the **readAccount** request, you should see it with the updated values.

To delete the account:

1. Create a DELETE request called **deleteAccount**.
2. Set the URL to `http://localhost:8080/service/account/1001`.
3. Click **Send**. If you try to read the record back again, you will get a 404 (not found) error.

At this point, you've exercised the functionality of the account service. You can try the same sorts of things with the customer and transaction services.

The StatementController Class

The StatementController provides only one endpoint, which enables you to get the monthly interest calculation for an account. Listing 6-11 shows the StatementController code.

Listing 6-11 The StatementController

```
@RestController
@RequestMapping("/service/account")
public class StatementController {
    private static final Logger LOGGER = LoggerFactory
        .getLogger(StatementController.class);

    @GetMapping(value = "/{id}/statement/{date}")
    public ResponseEntity<StatementDto> calculateStatement(
```

```
    @PathVariable("id") int id,
    @PathVariable("date") String date,
    @RequestParam("rate") String rate,
    @RequestParam("initialBalance") String balance) {
    BigDecimal dailyRate = null;
    BigDecimal initialBalance = null;
    LocalDate localDate = LocalDate.parse(date,
        DateTimeFormatter.BASIC_ISO_DATE);
    try {
        dailyRate = new BigDecimal(rate).divide(
            new BigDecimal(365 * 100)), 10,
            RoundingMode.HALF_UP);
        initialBalance = new BigDecimal(balance);
    } catch (Exception e) {
        return new ResponseEntity<>(HttpStatus.BAD_REQUEST);
    }
    try (RunUnit<MonthlyInterest> ru = new RunUnit<>(
        MonthlyInterest.class)) {
        MonthlyInterest statement = (MonthlyInterest) ru
            .GetInstance(MonthlyInterest.class, true);
        try {
            statement.init(dailyRate, initialBalance, localDate,
                id);
            return ResponseEntity.ok(
                statement.getStatementDto());
        } catch (Exception e) {
            LOGGER.error(e.getMessage());
            return null;
        }
    }
}
```

The StatementController picks up the account ID and starting date from the path since these are arguments that determine the resource we are going to access. But the data for the daily rate and the starting balance to begin the calculation from are passed in as query string parameters. They are picked out using the @RequestParam notation. Start the CreditService application and then enter this URL into a web browser:

```
http://localhost:8080/service/account/1/statement/20190701?rate=25
&initialBalance=100
```

You should get some JSON back (exact values might be different as the transaction data has been generated by a pseudo random process):

```
{  
    "minimumPayment":58.85,  
    "endingAmount":1177.06,  
    "interestAmount":13.68,  
    "accountId":1,  
    "startDate":"2019-07-01"  
}
```

You might wonder why we need to pass in the starting balance – couldn't we get it from the account record? The answer is that we need to pass in the balance at the start of the month for which we are doing the calculation. It could be calculated from the values in the transaction data file, but in the interests of keeping the example simple, we are just providing a value instead.

Summary

In this chapter, we wrapped some legacy COBOL code inside a Spring Boot application to make it available over HTTP/HTTPS. In the next chapter, we'll use MFUnit and JUnit to run a set of test cases to cover the different layers of the application.

CHAPTER 7



Automated Testing

In this chapter, we will test our application. This chapter covers:

- Strategies for testing
- MFUnit
- Testing the BusinessRules Layer
- Testing the Interoperation Layer
- Testing the Application End-to-End

We will use several different technologies in this chapter; MFUnit for testing procedural COBOL, JUnit for testing Visual COBOL, and REST Assured for testing our Web service. By the end of the chapter we will have a mix of unit and integration tests that cover every layer in our application.

Strategies for Testing

This chapter builds up a comprehensive set of tests for the application we looked at in the previous chapter. Although we haven't looked at any tests so far, these tests were created alongside the application used in this book. Although it isn't a "real" application, it evolved as the book was written and went through many changes (and bug fixes).

Having a test suite for all the different parts gave me the confidence to keep making changes as I went along, and know that I hadn't broke anything. And when things do break, tests make it easier to quickly find out why and what.

Test-driven development (TDD) is a methodology where the developer writes a unit test before writing the implementation code that will satisfy the test. For example, before writing a method to calculate sales tax, you write a unit test that exercises the calculation

functionality, and asserts that the method returns the correct values. At this point, the test calls a stub method with no implementation.

The test at this point will always fail when run. The developer then fills out the method with the actual implementation so that the test now passes. If you haven't used TDD, it might seem that you are doing a lot more work before you deliver any working code. However, as you build out the functionality for the application, you are also creating an extensive automated test suite that will make it easy to spot any regressions or new bugs. It also ensures that any APIs you build remain stable for clients consuming the functionality. The unit tests also serve as a form of documentation, since they spell out explicitly the expected behavior of your code.

Unit tests are not usually enough on their own, as they are testing only individual pieces of functionality in isolation. Often unit tests will make use of mocks, fakes, or stubs to replace other parts of the system, which are required for the test to run, but are not part of the actual functionality on test. For example, persistent databases are often replaced in unit tests by in-memory databases that enable you to test your business logic without worrying about the state left over from previous tests.

Mocks are also used for any other part of the system that might be slow to respond; unit test suites are expected to run fast. Developers will run the unit test suite often. They will run the unit tests every time before they commit code back to the source repository; they will also run the unit tests regularly as they are changing code to make sure nothing has broken.

Unit tests provide one level of confidence, but applications also need integration or end-to-end tests that test the system in its entirety. Integration tests take longer to run and often require more set up (for example, providing a known set of test data before they are started), so they are not run quite as often as unit tests.

However, integration tests should also be automated so that they can be run regularly by Continuous Integration systems. Probably the best known CI system is the open-source Jenkins, but there are many other systems both proprietary and open-source.

Downloading the Test Examples

The test examples are organized according to the following folders:

- MFUnit introduces MFUnit for testing COBOL Logic
- CreditServiceApplication is the REST service application from Chapter 6, plus:
 - MFUnit tests for the COBOL Business Logic
 - JUnit tests for the interoperation layer
 - JUnit integration tests for the entire application

Test Doubles—Mocks, Fakes and Stubs

A Test Double is a substitute for a part of your system that simplifies testing a particular part of your system. Fakes and stubs mimic in some way the behavior of another part of your system (a part that you aren't testing, but that is necessary to enable the part under test to work). Mocks are a variant that enable you to verify that a particular piece of code was called the expected number of times and with the expected values.

A lot of these techniques are enabled by designing APIs around interfaces rather than concrete classes; it's much easier to replace another part of your system if it is always represented in your code by an interface. Unfortunately, the kind of "legacy" code we are using in our example, and that you are likely to encounter in the real world, was designed and written before these techniques were common. This makes it difficult to remove it and replace it with mock functionality for testing, so we are forced in this chapter to use some different techniques to make our tests repeatable and fast.

Download the example code; the text will tell you when to import individual projects into an Eclipse workspace. The example code in this chapter contains six projects; it is easier to take care of errors by importing them as they are needed and in the right order.

Introducing MFUnit

Most Java developers are familiar with JUnit, the Java Unit testing framework. Java IDEs such as Eclipse and IntelliJ support JUnit and provide easy ways to run JUnit tests and display the results. Maven also supports JUnit and has a project structure that includes a test folder. By default, Maven will run those tests every time you run the build. Maven also enables you to mark dependencies in the POM file as only being needed for testing, so that you don't bloat deployables with libraries not needed at run-time.

Micro Focus has provided a test framework for Visual COBOL called the Micro Focus Unit Testing Framework (MFUnit). MFUnit tests provide a convenient way to test existing procedural code and can be used for COBOL whether it is compiled to native code or JVM or .NET. Visual COBOL code that is being compiled to JVM can be tested with MFUnit but it can also be tested with JUnit.

However, it is easier to test procedural code with MFUnit because test code written in COBOL shares the same semantics and data structures as the code you are testing. The BusinessInterop project examined in the previous chapter provides a Java API, so although it is written in COBOL, it feels more natural to test it using JUnit.

This chapter describes both MFUnit and JUnit tests. But before diving into the test code for our application, we'll illustrate the basics of MFUnit with a simple program that carries out multiplication and division.

Testing a Simple Calculator with MFUnit

There is a lot of legacy COBOL code in the world that doesn't have very much in the way of automated testing. It was written at a time when it was common to have large teams of testers who would work through test scripts to exercise the whole system. However, if you want to migrate and evolve this legacy code to the new world of microservices, you need fast feedback to know when you have broken something; that means automated tests that can be run quickly (ideally in a few seconds or less) every time you make changes.

MFUnit is ideal for this purpose. It enables a COBOL developer to write tests using standard Micro Focus COBOL. You can use it to test procedural code that has been compiled as native code. Then recompile the code and the tests as JVM code, and run them again to verify that you are still getting the expected behavior. And then when you start modifying your JVM based code, the tests will help you quickly spot any breakages. MFUnit enables you to test procedural code by calling entire programs or by calling individual entry-points.

Running the Test Suites

There is an MFUnit test suite for the BusinessRules project, but first we'll look at a very simple program and test suite. Download the examples for this chapter. There are two subfolders: MFUnit and CreditServiceApplication. Import everything under MFUnit into an Eclipse workspace. You should see four projects:

- NativeCalculator
- NativeCalculatorTest
- Calculator
- CalculatorTest

The source code in the two native projects is identical to the source code in the other two projects, but one pair of projects builds to native code and the other pair to JVM. The source code for the calculator is shown in Listing 7-1. It has a single entry-point that takes four parameters by reference. It multiplies or divides operand1 by operand2 depending on the value of function-code and places the answer into result.

Errors are signaled by a non-zero value of return-code (a COBOL special register). The special register is a 16-, 32- or 64-bit signed integer; the size depends on the bitism of the compiler and can be overridden by the RTNCODE-SIZE directive (see the Micro Focus documentation). For COBOL JVM, it defaults to 64 bits.

Listing 7-1 The calculator program

```

program-id. MainProgram.

data division.
working-storage section.
linkage section.
01 operand1          pic s9(15)v9(10).
01 operand2          pic s9(15)v9(10).
01 result            pic s9(15)v9(10).
01 function-code     pic x.
procedure division using by reference operand1 operand2
                      function-code result
                      returning return-code.
evaluate function-code
when "M"
  multiply operand1 by operand2 giving result
when "D"
  divide operand1 by operand2 giving result
when other
  move -1 to return-code
end-evaluate
display "wait"
goback.
```

There are (at least) three test cases required to test this program:

- Multiplication
- Division
- Invalid function-code (one that isn't "D" or "M").

Listing 7-2 shows the test suite for our calculator program.

Listing 7-2 The calculator test suite

```

copy "mfunit_prototypes.cpy".
program-id. MainProgramTest.

data division.
working-storage section.
78 TEST-MultiplyTest value "MultiplyTest".
78 TEST-DivideTest value "DivideTest".
78 TEST-InvalidOpTest value "InvalidOpTest".

copy "mfunit.cpy".
01 operand1          pic s9(15)v9(10).
```

```
01 operand2          pic s9(15)v9(10).
01 operator-code    pic x.
01 result           pic s9(15)v9(10).
01 displayable      pic x(28).
01 msg              pic x(100).
78 program-under-test   value "Calculate".
procedure division.

entry MFU-TC-PREFIX & TEST-MultiplyTest.
  move 33 to operand1
  move 3.1 to operand2
  move "M" to operator-code
  call program-under-test using by reference operand1 operand2
                                operator-code result
                                returning return-code
  if result <> 102.3
    display result
    call MFU-ASSERT-FAIL-Z using z"Expected 102.3"
  end-if
  goback returning return-code
  .

entry MFU-TC-PREFIX & TEST-DivideTest.
  move 33 to operand1
  move 3.3 to operand2
  move "D" to operator-code
  call program-under-test using by reference operand1 operand2
                                operator-code result
                                returning return-code
  if result <> 10
    display result
    call MFU-ASSERT-FAIL-Z using z"Expected 10"
  end-if
  goback returning return-code
  .

entry MFU-TC-PREFIX & TEST-InvalidOpTest.
  move 33 to operand1
  move 3.1 to operand2
  move "X" to operator-code
  call program-under-test using by reference operand1 operand2
                                operator-code result
                                returning return-code
  if return-code = 0
    call MFU-ASSERT-FAIL-Z using
      z"Expected non-zero return code"
```

```

        else
*      Zero return code before completing or test will be marked
*      as failed.
        move 0 to return-code
    end-if
    goback returning return-code
.

$region Test Configuration
entry MFU-TC-SETUP-PREFIX & TEST-MultiplyTest.
entry MFU-TC-SETUP-PREFIX & TEST-DivideTest.
entry MFU-TC-SETUP-PREFIX & TEST-InvalidOpTest.
$if JVMGEN set
    call "MainProgram"
$else
    call "NativeCalculator"
$endif
goback returning 0.

entry MFU-TC-TEARDOWN-PREFIX & TEST-MultiplyTest.
goback returning 0.

entry MFU-TC-METADATA-SETUP-PREFIX & TEST-MultiplyTest.
    move "This is a example of a dynamic description"
        to MFU-MD-TESTCASE-DESCRIPTION
    move 4000 to MFU-MD-TIMEOUT-IN-MS
    move "smoke" to MFU-MD-TRAITS
    set MFU-MD-SKIP-TESTCASE to false
    goback.
$end-region

```

MFUnit uses conventions to define the format of the entry-points in a test suite:

- Each test case entry-point starts with the name MFU-TC-PREFIX (this is a literal defined in `mfunit.cpy`).

A test case is assumed to pass unless it sets a non-zero value into the return-code special register or calls either MFU-ASSERT-FAIL or MFU-ASSERT-FAIL-Z during execution. The -Z variant indicates that this routine expects a null-terminated string (C style string), which you can create in Micro Focus COBOL by prefixing z to a literal. For example `z"Hello World"` creates the string Hello World with a null byte at the end.

- Each setup function starts with the prefix MFU-TC-SETUP-PREFIX followed by the name of a test case.
- Each teardown function starts with the prefix MFU-TC-TEARDOWN-PREFIX followed by the name of a test case.

- Metadata functions start with the prefix MF-TC-METADATA-SETUP-PREFIX followed by the name of a test case.
- MFUnit also provides some entry-points for asserting failures and defining metadata. These are defined in `mfunit_prototypes.cpy`, included at the top of the program.

Setup, teardown, and metadata definition are all optional. This test suite shows teardown and metadata for the TEST-MultiplyTest test case only. But all three testcases share the same setup code – the setup entry-points for each test case follow each other consecutively, with a single block of code beneath; executing any of these entry-points executes the same code.

Figure 7-1 shows the order of execution of entry-points for an individual test case called T1. The steps in dotted lines are optional; if you don't create entry-points for a particular test case there is no error. But if you add any of these metadata points with a name that does not also appear in a test case, MFUnit will give an error and will not run the test suite. For example, if you add entry `MFU-TC-SETUP-PREFIX-T1` but don't add entry `MFU-TC-PREFIX-T1`, MFUnit displays an error and does not run the test suite.

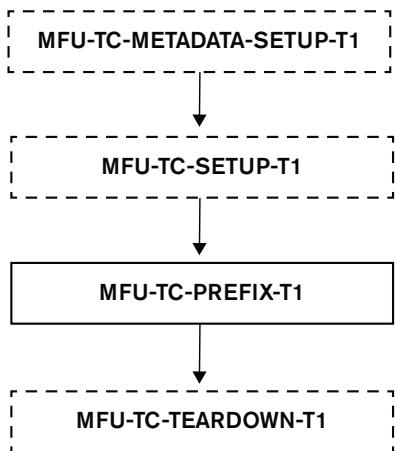


Figure 7-1 Execution of T1 test case

The CalculatorTest code itself uses conditional compilation to provide different code depending on whether we are executing as JVM or native code. The setup code is needed only to load the calculator into memory so that its entry-point is available. However, because the executable artefacts for JVM and Native code are different, we need different calls here.

When this project is built as native code, we get an artefact called `NativeCalculator.dll` (Windows) or `NativeCalculator.so` (Linux), but when it is built as JVM, the artefact is `MainProgram.class`. Code following `$if JVMGEN` set up to the `$else` is only compiled when the `JVMGEN` directive is set. The code between `$else` and `$end` is compiled when the `JVMGEN` directive is not set. In this case the assumption is that if `JVMGEN` is not set, we are compiling for native code.

To run the native code test suite:

1. In the COBOL Explorer, right-click **NativeCalculatorTest** and then click **Run As > COBOL Unit Test**.
2. After a moment, you should see the Micro Focus Unit Testing pane as shown in Figure 7-2. It shows the three test cases and a pass for each one.

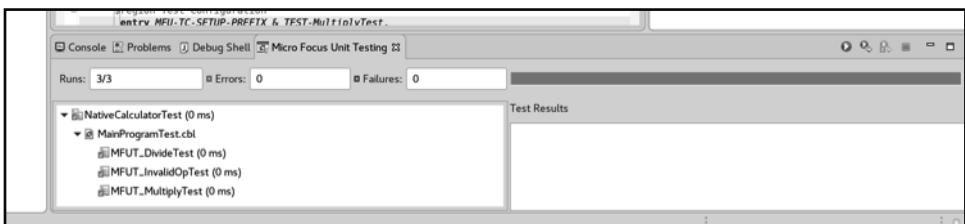


Figure 7-2 The Micro Focus Unit Testing pane

You can also run the JVM test suite:

1. In the COBOL Explorer, right-click **CalculatorTest** and click **Run As > COBOL JVM Unit Test**.
2. You should see the Micro Focus Unit Testing pane as before, showing the same test case and that each one passes.

Spend a little time playing with the two test suites. Verify that the code being tested in the two cases is identical and that the only difference is how it is compiled, and then experiment with adding new test cases of your own.

Testing the BusinessRules Layer

The previous section introduced MFUnit with a test suite for a simple program. In this section we look at a test suite for the original legacy procedural code of the application. The procedural part of the CreditService application has logic for the CRUD operations for customers, accounts, transactions, and calculating interest. This is what the tests focus on.

The value of these tests will become apparent later in the book when we start refactoring the BusinessRules project; only when all the tests pass can we be sure our refactored project is working like the original.

The BusinessRules project is mainly concerned with data persistence, so any test that is run needs to run against a known state of stored data. With Java projects these sorts of tests can often be run using an in-memory database like HSQL, which is initialized from scratch every time the tests are run. This option isn't available for our legacy COBOL code, which is using COBOL files (which, in turn, use the file system).

To keep these tests robust, simple, and fast, the test setup methods simply delete the backing files before the start of every test case. Every test starts from the known state of “no data”.

To run the BusinessRules tests:

1. Start Eclipse with a new empty workspace.
2. Import the **BusinessRules** and **BusinessSystemTests** projects from the CreditServiceApplication of the Chapter 7 examples.
3. Right-click the **BusinessRules** project, click **Properties > Builders**, and then edit **mvn_businessrules** for your local setup (as explained in the “Importing the Example Projects” section in Chapter 5).
You don’t need this builder to work in order to run the tests, but it will need to work later on the chapter; fixing it now will remove a source of error messages.
4. Click **Run > Run Configurations**, select **COBOL JVM Unit Test**, and then click the **New** icon.

The test suite for the calculator didn’t need any specific configuration, which is why we could run it as a standard COBOL JVM Unit Test without any extra settings.

5. Set the name to **BusinessSystemTests** and then click the **Browse** button to select the **BusinessSystemTests** project as the COBOL JVM Unit Test Project.
6. Click the **Environment** tab and add three environment variables:

```
dd_ACCOUNTFILE =  
application-download/BusinessSystemTests/testData/account.dat  
dd_TRANSACTIONFILE=  
application-download/BusinessSystemTests/testData/transaction.dat  
dd_CUSTOMERFILE=  
application-download/BusinessSystemTests/testData/customer.dat
```

Do *not* set these environment variables to the files where your “production” data (generated in Chapter 5) is stored. These files are deleted at the start of every test case.

7. Click **Run**.

You should see the Micro Focus Unit Testing pane with the **BusinessSystemTests** test suite, showing 19 tests with no failures or errors.

The **BusinessSystemTests** project contains the following programs (you can see them under `src/default` package in the COBOL or Project Explorer):

- TestAccountStorage.cbl
- TestCustomerStorage.cbl
- TestTransactionStorage.cbl
- TestInterestCalculator
- TestCalendar.cbl
- Helper-Functions.cbl

The first three programs provide test cases for the CRUD functions supplied by the ACCOUNT-TEST-STORAGE program. **TestInterestCalculator.cbl** tests the calculation functions in InterestCalculator. **TestCalendar.cbl** tests the functionality provided by the Calculator program.

The last program, **Helper-Functions.cbl**, contains common code shared between the other test suites, in particular the setup functions that ensure all tests start from a known state.

We'll start with a look at the **TestAccountStorage** program, organized into separate sections to make it easier to follow the explanations. Listing 7-3 shows the beginning of the **TestAccountStorage** program, together with the **setup-account-test** section from near the end of the program.

Listing 7-3 Start of the TestAccountStorage program

```

copy "mfunit_prototypes.cpy".
copy "cblproto.cpy".

program-id. TestAccountStorage.

data division.
working-storage section.
copy "PROCEDURE-NAMES.cpy".
copy "FUNCTION-CODES.cpy".
copy "HELPER-FUNCTIONS.cpy".
78 TEST-WriteAccount      value "TestWriteAccount".
78 TEST-ReadLastAccount   value "TestReadLastAccount".
78 TEST-ReadRecords       value "TestReadAccountRecords".
78 TEST-UpdateAccount    value "TestUpdateAccount".
01 write-mode             pic x.

78 FK-CUSTOMER-ID        value 888.
78 FK-CUSTOMER-FIRST-NAME value "Verity".
78 FK-CUSTOMER-LAST-NAME value "Talkington".

78 TEST-ID-1              value 200.

```

```

78 TEST-BALANCE-1           value 400.21.
78 TEST-TYPE-1             value "C".
78 TEST-CREDIT-LIMIT-1    value 1000.00.
78 TEST-ID-2               value 201.
78 TEST-BALANCE-2          value 702.31.
78 TEST-TYPE-2             value "C".
78 TEST-CREDIT-LIMIT-2    value 2000.00.
78 TEST-ID-3               value 210.
78 TEST-BALANCE-3          value 99.37.
78 TEST-TYPE-3             value "C".
78 TEST-CREDIT-LIMIT-3    value 750.00.

01 WS-ID-TABLE.

 03 WS-ID-ROW              pic x(4) comp-5 OCCURS 3 TIMES.
 01 i                      pic x(2) comp-5.
 01 msg                     pic x(200).
 01 function-code           pic x.
 01 file-status.
 03 status-byte-1           pic x.
 03 status-byte-2           pic x.
copy "CUSTOMER-RECORD.cpy" replacing ==(PREFIX)== by WS.
copy "ACCOUNT-RECORD.cpy"  replacing ==(PREFIX)== by WS.
copy "ACCOUNT-RECORD.cpy"  replacing ==(PREFIX)== by TEST.

copy "mfunit.cpy".
procedure division.

$region Test Configuration
entry MFU-TC-SETUP-PREFIX & TEST-UpdateAccount.
entry MFU-TC-SETUP-PREFIX & TEST-WriteAccount.
entry MFU-TC-SETUP-PREFIX & TEST-ReadRecords.
entry MFU-TC-SETUP-PREFIX & TEST-ReadLastAccount.
  perform setup-account-test
  goback returning 0.

$end-region
*> Omitted code
*>...
setup-account-test section.
  call HELPER-FUNCTIONS
  call INIT-ACCOUNT-TEST using by reference function-status
  goback.

```

Most of this listing is the data declarations, including the copybooks referenced elsewhere in the program. As with the Calculator test, from earlier in this chapter, it also includes two copybooks used by MFUnit: `mfunit_prototypes.cpy` and `cblproto.cpy`.

Test Case Setup Code

The code in the Test Configuration region of Listing 7-3 contains the setup for all the test cases in this program: `perform setup-account-test`. The code in the `setup-account-test` section loads the `HELPER-FUNCTIONS` program and then calls the `INIT-ACCOUNT-TEST` entry-point. Listing 7-4 shows just this entry-point from `HELPER-FUNCTIONS` together with the `setup-test-data` section.

Listing 7-4 The setup code from the `HELPER-FUNCTIONS` program

```
*> code fragment
ENTRY INIT-ACCOUNT-TEST using by reference lnk-function-status.
    perform init-helper
    set open-ppointer to entry OPEN-ACCOUNT-FILE
    set write-ppointer to entry WRITE-ACCOUNT-RECORD
    set read-one-record-ppointer to entry FIND-ACCOUNT-ID
    set read-last-ppointer to entry READ-LAST-ACCOUNT-RECORD
    set read-records-ppointer to entry READ-ACCOUNT-RECORD
    display "dd_ACCOUNTFILE" upon environment-name
    perform setup-test-data
    goback.
*> Omitted code
*> ...
setup-test-data section.
    move spaces to ws-filename
    accept ws-filename from environment-value
    if ws-filename equals spaces
        move z"Environment variable not set" to msg
        call MFU-ASSERT-FAIL-Z using msg
        goback
    end-if
    call CBL-CHECK-FILE-EXIST using ws-filename
                    file-details
    if return-code = 0
*>        delete the file before running the test
        display "Deleting file"
        call CBL-DELETE-FILE using ws-filename
    end-if
    set succeeded to true
    move function-status to lnk-function-status
    exit section.
*> Omitted code
*> ...
init-helper section.
```

```

move 0 to lnk-function-status
move spaces to VERIFICATION-RECORD
exit section.

```

Figure 7-3 shows the order of initialization and test cases. The entry INIT-ACCOUNT-TEST sets up some procedure pointers to entry-points in the ACCOUNT-STORAGE-ACCESS program, then performs the setup-test-data section that deletes the data file for the current test. The init-helper section zeroes the lnk-function-status flag. This section is called at the start of every entry-point in HELPER-FUNCTIONS.

All callers of the HELPER-FUNCTIONS program check the status flag on return from any entry-point and fail the current test case if the status is set to failed. A function-status item is defined in copybook HELPER-FUNCTIONS.cpy as shown in Listing 7-5. Level 88 items define condition-names. You can set their parent item to any of the conditions defined and use any of the condition-names inside a conditional clause.

Listing 7-5 The function-status flag

```

01 function-status          pic 9.
  88 failed                value 0.
  88 succeeded             value 1.
  88 no-more-records       value 2.

```

init-helper also sets the VERIFICATION-RECORD data item to spaces. The VERIFICATION-RECORD data item is used for comparing actual results to expected values; not all tests use it but clearing it every time reduces a possible cause of errors.

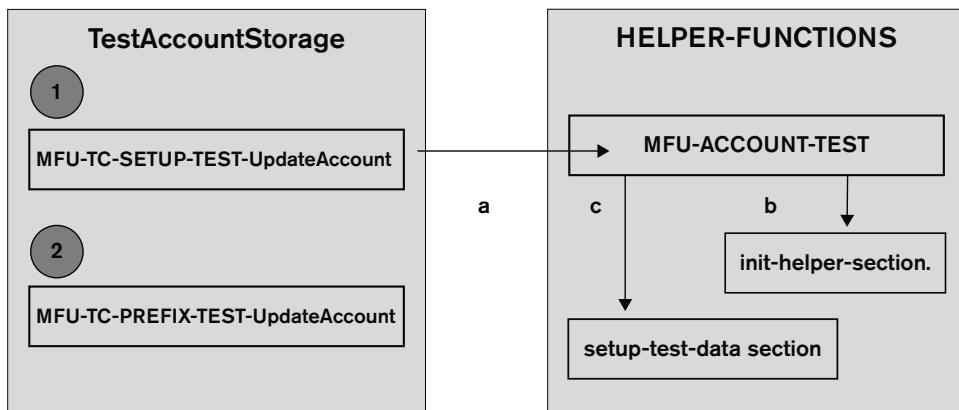


Figure 7-3 Flow for initialization and running a test case

Account Storage Test Case Code

We'll look at just one of the test cases for the account storage code; TEST-UpdateAccount. Listing 7-6 is the code for this test case from TestAccountStorage.cbl, together with the write-an-account-record section, which is used by several of the test cases.

Listing 7-6 Test case *TEST-UpdateAccount*

```

entry MFU-TC-PREFIX & TEST-UpdateAccount.
    if failed perform test-failed end-if *> check for setup failure
    move TEST-ID-1 to TEST-ACCOUNT-ID
    move FK-CUSTOMER-ID to TEST-CUSTOMER-ID
    move TEST-BALANCE-1 to TEST-BALANCE
    move TEST-TYPE-1 to TEST-TYPE
    move TEST-CREDIT-LIMIT-1 to TEST-CREDIT-LIMIT
    move WRITE-RECORD to write-mode
    perform write-an-account-record
    move TEST-BALANCE-2 to TEST-BALANCE
    move TEST-CREDIT-LIMIT-2 to TEST-CREDIT-LIMIT
    move UPDATE-RECORD to write-mode
    perform write-an-account-record
    move spaces to WS-ACCOUNT
    move TEST-ID-1 to WS-ACCOUNT-ID
    call COMPARE-RECORDS using by value
        length of WS-ACCOUNT
        by reference WS-ACCOUNT
        TEST-ACCOUNT
        function-status
    goback returning return-code.
*> Omitted code
*> ...
write-an-account-record section.
    move OPEN-I-0 to function-code
    call OPEN-TEST-FILE using by value function-code
        by reference function-status
    if failed perform test-failed end-if
    call WRITE-TEST-RECORD using by value write-mode
        by reference TEST-ACCOUNT
        function-status
    if failed perform test-failed end-if
    call CLOSE-TEST-FILE using by reference function-status
    if failed perform test-failed end-if
    exit section.

```

The test case starts by checking the function-status flag for the failed condition. Why do we start by checking for a failure before we've run any test-code? Because MFUnit runs

the initialization code shown in Figure 7-3 before each test case and this might fail for any one of several reasons (for example, if the environment variable for the data file wasn't set). If the initialization code doesn't succeed, then the test won't behave as expected; and a pass or fail isn't really telling you whether the code under test is working or not.

Interest Calculator Test Case Code

All we really want to test with InterestCalculator program is whether it carries out interest calculations correctly. However, it depends on ACCOUNT-STORAGE-ACCESS for reading transaction records that have the data it needs. A test will require some actual transaction records to operate on.

With modern Java applications and test frameworks, there are different options for mocking out a storage layer so that we can supply test data without actually writing it to files. With legacy COBOL, however, this is rarely an option.

To make our test cases for the Interest Calculator independent of each other and easily rerunnable, each one starts by writing some test transaction records so that they can be read back again by ACCOUNT-STORAGE-ACCESS for the tests. This means these tests are closer to being integration or system tests than they are unit tests; in effect, we are testing our entire storage layer every time we run these tests as well as the Interest-Calculator itself.

When possible, this is something that you want to minimize. You get the best test coverage by having many unit tests that cover all the edge cases and exception paths as well as the "happy path." Unit tests should be small, independent (so you can run them in order or in isolation from each other), and run fast so that it is practical to run them on every code commit. The more complex your test, and the more dependencies on the rest of the system needed in order to run it, the longer they will take to run.

For our example test suite, we have three test cases for the InterestCalculator, all of which run on the same set of test data that is created at the beginning of every test case. A more comprehensive test suite would create some different sets of test-data so that other edge-cases could be tested. And adding regression tests as bugs are found would also require different test datasets.

The four cases tested here are:

- Basic interest calculation ("the happy path")
- Interest calculation when we set the rate to zero
- Interest calculation when there are no transactions for the account and period selected
- Interest calculation with initial balance of zero and no transactions

Because TestInterestCalculator.cbl is going to create test data at the start of every test case, it uses **HELPER-FUNCTIONS** in the same way as the storage tests we looked at earlier.

Listing 7-7 shows the initialization code for the InterestCalculator test suite.

Listing 7-7 The initialization code for TestInterestCalculator

```
entry MFU-TC-SETUP-PREFIX & TEST-TestInterestCalculation.  
entry MFU-TC-SETUP-PREFIX & TEST-TestZeroInterestCalculation  
entry MFU-TC-SETUP-PREFIX & TEST-NoTransactionsCalculation.  
entry MFU-TC-SETUP-PREFIX & TEST-ZeroBalanceTransactionsCalculation.  
    perform test-setup  
    perform write-multiple-records  
    goback. test-setup section.  
    call "INTEREST-CALCULATOR"  
    call HELPER-FUNCTIONS  
    call INIT-TRANSACTION-TEST using by reference function-status  
    if failed perform setup-failed end-if  
    .  
  
write-multiple-records section.  
    move OPEN-WRITE to function-code  
    call OPEN-TEST-FILE using by value function-code  
        by reference function-status  
    move WRITE-RECORD to function-code  
    move TEST-TRANSACTION-ID-1 to TEST-TRANSACTION-ID  
    move FK-ACCOUNT-ID to TEST-ACCOUNT-ID of TEST-TRANSACTION-RECORD  
    move TEST-AMOUNT-1 to TEST-AMOUNT  
    move TEST-TRANS-DATE-1 to TEST-TRANS-DATE  
    move TEST-DESCRIPTION-1 to TEST-DESCRIPTION  
    call WRITE-TEST-RECORD using by value function-code  
        by reference TEST-TRANSACTION-RECORD  
        function-status  
    if failed perform test-failed end-if  
    move TEST-TRANSACTION-ID-2 to TEST-TRANSACTION-ID  
    move FK-ACCOUNT-ID to TEST-ACCOUNT-ID of TEST-TRANSACTION-RECORD  
    move TEST-AMOUNT-2 to TEST-AMOUNT  
    move TEST-TRANS-DATE-2 to TEST-TRANS-DATE  
    move TEST-DESCRIPTION-2 to TEST-DESCRIPTION  
    call WRITE-TEST-RECORD using by value function-code  
        by reference TEST-TRANSACTION-RECORD  
        function-status  
    if failed perform test-failed end-if  
    move TEST-TRANSACTION-ID-3 to TEST-TRANSACTION-ID  
    move FK-ACCOUNT-ID to TEST-ACCOUNT-ID of TEST-TRANSACTION-RECORD
```

```

move TEST-AMOUNT-3 to TEST-AMOUNT
move TEST-TRANS-DATE-3 to TEST-TRANS-DATE
move TEST-DESCRIPTION-3 to TEST-DESCRIPTION
call WRITE-TEST-RECORD using by value function-code
    by reference TEST-TRANSACTION-RECORD
        function-status
if failed perform test-failed end-if
call CLOSE-TEST-FILE using by reference function-status
.

setup-failed section.
call MFU-ASSERT-FAIL-Z using by reference z"Test setup failed"
goback.

test-failed section.
call MFU-ASSERT-FAIL-Z using
    by reference z"Test helper function failed"
goback.

```

The setup code uses the **HELPER-FUNCTIONS** to initialize a transaction test (since transaction records are what we need to run the test) and then writes three test transaction records. All are for the same time period (August 2019) and all belong to the same account.

Listing 7-8 shows the simplest test case – calculate the interest for the month for which we have transactions (with a non-zero starting balance and a non-zero rate of interest) and check the results.

Listing 7-8 Test case for the happy path test case

```

entry MFU-TC-PREFIX & TEST-TestInterestCalculation.
if failed perform setup-failed end-if
divide 1 by 3650 giving WS-DAY-RATE */> 10% interest rate
move "20190801" to TEST-TRANS-DATE
move START-AMOUNT to WS-AMOUNT
move FK-ACCOUNT-ID to WS-ACCOUNT-ID
call CALCULATE-INTEREST using by value TEST-TRANS-DATE
    WS-ACCOUNT-ID
    by reference WS-DAY-RATE
    WS-AMOUNT
    WS-INTEREST
    WS-MINIMUM-PAYMENT
    WS-STATUS
if WS-STATUS <> "00"
    move "File Status of " & WS-STATUS & x"00" to msg
    call MFU-ASSERT-FAIL-Z using msg
end-if

```

```
if WS-MINIMUM-PAYMENT <> 18.55
    move "Expected Minimum payment 18.55, actual "
        & WS-MINIMUM-PAYMENT & x"00" to msg
    call MFU-ASSERT-FAIL-Z using msg
end-if
if WS-AMOUNT <> 371.01
    move "Expected balance 371.01, actual "
        & WS-MINIMUM-PAYMENT & x"00" to msg
    call MFU-ASSERT-FAIL-Z using msg
end-if
if WS-INTEREST <> 3.08
    move "Expected interest 371.01, actual "
        & WS-INTEREST & x"00" to msg
    call MFU-ASSERT-FAIL-Z using msg
end-if
goback.
```

Testing Legacy Code with MFUnit

MFUnit brings some of the testing techniques that have been proven with frameworks such as JUnit to legacy COBOL. It gives you a structured way to write individual tests, together with setup and teardown code where needed. The metadata functions (which we didn't discuss here) also provide you with ways of adding descriptive data to tests and grouping them together.

Even a very small test suite like the one shown here provides some confidence to make refactoring changes. If this were a real application though, it would need to be more comprehensive and cover several more edge and error conditions. However, we will see it again later in the book when we make some major changes to the BusinessRules layer. In the next section, we look at testing the interoperation layer.

Testing the Interoperation Layer

The next project to test is `BusinessInterop`. Although this is written in COBOL, the tests are written in Java using JUnit. The `BusinessInterop` project API is object-oriented, intended for use by Java applications, and JUnit and Java feels like a better fit for testing it. The tests are in a separate project: `interoptests`.

To import the `BusinessInterop` and its tests:

1. Start Eclipse with the workspace you used when importing projects in section Testing the BusinessRules layer.
2. Import the **BusinessInterop** and **interoptests** projects.

3. Right-click the **BusinessInterop** project, click **Properties > Builders**, and then edit **mvn_businessinterop** for your local setup (as explained in the “Importing the Example Projects” section in Chapter 5).
4. Click **Run > Run Configurations**, select **JUnit**, and then click the **New** icon.
5. Set the name to **InteropTests**, select the **interoptests** project for the **Run all tests in the selected project, package or source folder** field, and in the **Test runner** field, select **JUnit 4**
6. Click the **Environment** tab and then add three environment variables:
`dd_ACCOUNTFILE =
application-download/JavalInteropTests/testData/account.dat
dd_TRANSACTIONFILE=
application-download/JavalInteropTests/testData/transaction.dat
dd_CUSTOMERFILE=
application-download/JavalInteropTests/testData/customer.dat`

Do *not* set these environment variables to the files where your “production” data (generated in Chapter 5) is stored. These files are deleted at the start of every test case.

7. Set the **Test runner** field to **JUnit 4**.
8. Click **Run**.

You should see the JUnit test pane with the InteropTest test suite, showing 28 tests with no failures or errors.

The BusinessInterop project depends on the BusinessRules project, with which it interacts through COBOL CALLs. This is another codebase that is tied very closely into physical file storage in a way that makes it hard to mock out in any of the usual ways (by replacing an interface or using a library like Mockito or JMock).

So once again, we’ve taken the line of least resistance and accepted that the persistence layer is going to be involved in our tests even though that is not ideal. The difference this time is that we have three fixed files of test data (you can see them in the `src/main/resources` folder) and they are named `account.testdat`, `customer.testdat` and `transaction.testdat`. They are regular Micro Focus COBOL indexed files, but they have a `.testdat` extension to differentiate them from files with the usual `.dat` extension. This is for two reasons:

- It is easy to distinguish the test data from other data files that might be on the machine.
- If your source control system has been set to ignore files of type `.dat` (because they are not part of the source code), you can still store the `.testdat` files alongside the source for the test suite.

For example, I used git as source control when developing the examples in this book, so my `.gitignore` file had the line:

```
*.dat
```

This stopped me from adding `.dat` files to my repository, but I was able to store the `.testdat` file.

The setup for each test copies them to the locations indicated by the `dd_` environment variables setup in Step 6. Because the files are small, this is a fairly fast setup step; on a fast laptop, all 28 tests run in less than a second. Listing 7-9 shows the setup code. The setup method can be any `public void` method annotated with `@Before`. JUnit has only one setup method per class.

Listing 7-9 The setup code for the `InteropTests`

```
public class InteropTest {
    private static BigDecimal DAILY_RATE = new BigDecimal(0.1)
        .divide(new BigDecimal(365), 10, RoundingMode.HALF_UP);

    @Before
    public void initTestData() throws IOException {
        copyDataResource("account.testdat", "dd_ACCOUNTFILE");
        copyDataResource("customer.testdat", "dd_CUSTOMERFILE");
        copyDataResource("transaction.testdat",
            "dd_TRANSACTIONFILE");
    }

    private void copyDataResource(String source, String target)
        throws IOException {
        URL url = InteropTest.class.getClassLoader()
            .getResource(source);
        String path = url.getPath();
        assertNotNull(String.format(
            "No resource found for %s", source), path);
        Path sourcePath = new File(path).toPath();
        String environmentValue = System.getenv().get(target);
        assertNotNull(String.format(
            "No value found for %s", target),
            environmentValue);
        assertNotNull(sourcePath);
        File targetFile = new File(environmentValue);
        Files.copy(sourcePath, targetFile.toPath(),
            StandardCopyOption.REPLACE_EXISTING);
    }
}
```

Listing 7-10 shows a single test case. It is annotated with @Test.

Listing 7-10 The updateCustomer test case

```
@Test
public void updateCustomer() {
    int id = 2;
    String newLastName = "Vonnegut";
    try (CustomerDataAccess cda = new CustomerDataAccess()) {
        cda.open(AbstractBusinessAccess.OpenMode.rw);
        CustomerDto dto = cda.getCustomer(id);
        assertNotNull(dto);
        dto.setLastName(newLastName);
        assertTrue(cda.updateCustomer(dto));
    }
    try (CustomerDataAccess ada = new CustomerDataAccess()) {
        ada.open(AbstractBusinessAccess.OpenMode.read);
        CustomerDto dto = ada.getCustomer(id);
        assertTrue(dto != null);
        assertEquals(newLastName, dto.getLastName());
    }
}
```

This test reads a particular customer, changes the last Name, and then rereads it to verify the change has been performed. In the next section, we'll look at testing our application end-to-end.

Testing the Application End-to-End

We have been working our way through, testing the different layers of our application. We are now going to create some integration tests that actually test the entire application end-to-end. These tests are going to start up our CreditService web front end, make HTTP calls, and then verify that the expected results come back.

To make this easier, we are going to use a testing library named REST Assured. This does all the heavy lifting of calling our REST end-points and validating the responses. Before we can use REST Assured in the CreditService project, it must be added as a dependency to the pom.xml for the project.

Listing 7-11 shows just the new dependency added to pom.xml for REST Assured. There is no version number for this dependency because the Spring Boot plugin in this project will work out the appropriate version based on the version of Spring we are using. The scope of the dependency is test, which means the dependency is not included in the jar file for distributing the application; Maven uses it only when running the application tests.

Listing 7-11 The REST Assured dependency

```
<dependency>
    <groupId>io.rest-assured</groupId>
    <artifactId>rest-assured</artifactId>
    <scope>test</scope>
</dependency>
```

As with the earlier sets of tests in this chapter, we need some test data. We are creating integration tests that make sure all parts of the application function as expected when put together at run-time.

In the tests we used for the BusinessInterop and BusinessRules layers, we started with empty data files and added a few records for testing. And then, in the previous section, we copied some data files with a known set of records as the data for running the tests. We are repeating this approach here, and as before, we're going to copy the data as part of the setup for every single test case so that test cases can be run individually and so that we don't build dependencies between our test cases.

The test data is stored as part of the CreditService, under src/test/resources. You can see three files: account.testdat, customer.testdat and transaction.testdat.

The test suite itself is under src/test and is in package com.mfcobolbook.creditservice.webservice. It consists of four classes:

- WebserviceAccountTests
- WebserviceCustomerTests
- WebserviceTransactionTests
- WebserviceStatementTests

Listing 7-12 shows a fifth class (import statements are omitted to save space), WebServiceApplicationTests, which contains all the common setup code used by the other tests.

Listing 7-12 The abstract test class

```
package com.mfcobolbook.creditservice.webservice;

public abstract class WebServiceApplicationTests {
    private static boolean hasInitialized = false;

    public static void setup() {
        if (!hasInitialized) {
            SpringApplication application =
                new SpringApplication(WebServiceApplication.class) ;
            RestAssured.port = 8088;
            application.run(new String[] {});
        }
    }
}
```

```
        hasInitialized = true;
    }
}

protected void initTestData() throws IOException {
    copyDataResource("account.testdat", "dd_ACCOUNTFILE");
    copyDataResource("customer.testdat", "dd_CUSTOMERFILE");
    copyDataResource("transaction.testdat",
                     "dd_TRANSACTIONFILE");
}

private void copyDataResource(String source, String target)
    throws IOException {
    URL url = WebserviceApplicationTests.class.getClassLoader()
        .getResource(source);
    String path = url.getPath();
    assertNotNull(String.format(
        "No resource found for %s", source), path);
    Path sourcePath = new File(path).toPath();
    String environmentValue = System.getenv().get(target);
    assertNotNull(String.format("No value found for %s", target),
                  environmentValue);
    assertNotNull(sourcePath);
    File targetFile = new File(environmentValue);
    Files.copy(sourcePath, targetFile.toPath(),
               StandardCopyOption.REPLACE_EXISTING);
}
}
```

The static `setup()` method starts the `WebserviceApplication` test and sets the port REST Assured will use to connect to the webserver to 8088. If you look at the application.properties file in `src/test/resources`, it contains the line:

```
server.port=8088
```

This is different to the default port (8080) on which the application usually runs. It means we can run the tests even when the application is already running in a separate process; two processes can't listen on the same port.

This abstract class also has two helper methods, `initTestData()` and `copyDataResource()`, that copy the `.testdat` files over the `.dat` files used for running the test cases. These are called by the `setup` methods in the subclasses. All of the methods in this class are called by the subclasses that contain the test cases.

Listing 7-13 shows the start of the class that has the tests for the Accounts controller. It is annotated with `@RunWith(SpringRunner.class)` and `@SpringBootTest`. These annotations make sure that the spring context is loaded before the tests are run.

Listing 7-13 The start of the WebserviceAccountTests class

```

package com.mfcobolbook.creditservice.webservice;

@RunWith(SpringRunner.class)
@SpringBootTest
public class WebserviceAccountTests extends WebserviceApplicationTests {

    private static final String ID = "id";

    private static final String CUSTOMER_ID = "customerId";
    private static final String BALANCE = "balance";
    private static final String CREDIT_LIMIT = "creditLimit";

    private static final int CUSTOMER_ID_DATA = 300;
    private static final float BALANCE_DATA = 5432.0f;
    private static final byte TYPE_DATA = 'C';
    private static final float CREDIT_LIMIT_DATA = 10000.00f;
    private static final BigDecimal BIG_DECIMAL_ERROR =
        new BigDecimal(.001);

    @BeforeClass
    public static void setup() {
        WebserviceApplicationTests.setup();
    }

    @Before
    public void initTestData() throws IOException {
        super.initTestData();
    }
}

```

There are two methods in this excerpt of the test class. The static `setup()` method is annotated with `@BeforeClass`. This annotation ensures that the method is called only once, the first time the class is loaded by the test runner. This calls the `setup()` method from Listing 7-12 that starts the application. We want to do this only once for the test cases in a class because it is a comparatively expensive (slow) operation.

The `initTestData()` method is annotated with `@Before` so it will get called before every single test case. This calls the method in the superclass that reinitializes the test data to a known state. This doesn't slow the test cases by very much and it gives us the benefit that all the tests can be completely independent of each other.

Listing 7-14 shows a single test case from this class. It is annotated with `@Test`, which tells JUnit that this method is a test case. The code inside the method is a test using REST Assured.

Listing 7-14 A single REST controller test case

```
@Test
public void shouldLoadAccount() {
    when().get("/service/account/1")
        .then().assertThat()
            .body(ID, equalTo(1)).body(CUSTOMER_ID, equalTo(1))
            .body(BALANCE, equalTo(989.99f))
            .body(CREDIT_LIMIT, equalTo(3000.00f));
}
```

`when()` is a static method of the `RestAssured` class that returns a `RequestSender` object. The `get()` method sends an HTTP GET request on the specified URL to the Web Server. The remainder of this statement enables us to assert that there are certain things we expect to see in the body of the response. REST Assured is clever enough to unpick the JSON response from our application and check the fields contain the expected values. `ID`, `CUSTOMER_ID`, `BALANCE`, and `CREDIT_BALANCE` are `static final Strings` with the names of the fields (you can see them defined at the start of Listing 7-13).

That's a very simple test that just does a GET. Listing 7-15 shows a slightly more complicated test that does a POST. This test creates an `AccountForm` object and then adds it as a new account by posting to the `/service/account` path in the application.

Listing 7-15 A test that does a POST

```
@Test
public void shouldAddNewAccount() {
    AccountForm account = new AccountForm(new AccountDto(0,
        CUSTOMER_ID_DATA, new BigDecimal(BALANCE_DATA),
        TYPE_DATA, new BigDecimal(CREDIT_LIMIT_DATA)));

    given().body(account)
        .headers(Collections.singletonMap("Content-Type",
            "application/json"))
        .when().post("/service/account/").then().assertThat()
            .statusCode(equalTo(200)).body(ID, equalTo(22))
            .body(CUSTOMER_ID, equalTo(CUSTOMER_ID_DATA))
            .body(BALANCE, equalTo(5432))
            .body(CREDIT_LIMIT, equalTo(10000));

    try (RunUnit<AccountDataAccess> ru = new RunUnit<>(
        AccountDataAccess.class)) {
```

```
try (AccountDataAccess accessor = (AccountDataAccess) ru
     .GetInstance(AccountDataAccess.class, true)) {
    accessor.open(AbstractBusinessAccess .OpenMode.read);
    AccountDto dto = accessor.getAccount(22);
    assertNotNull(dto);
    assertThat(dto.getCustomerId(),
               equalTo(CUSTOMER_ID_DATA));
    assertThat(dto.getBalance(),
               closeTo(BigDecimal(BALANCE_DATA),
                      BIG_DECIMAL_ERROR));
    assertThat(dto.getCreditLimit().intValue(),
               equalTo((int) CREDIT_LIMIT_DATA));
}

}
```

This test starts with the `given()` method (another static method of the `RestAssured` class), sets up the body we wanted posted, together with the headers that inform the application the data will be in json format, and then posts the request. As with the other test case, it makes some assertions about the expected response from the Web server. It also has a second part where it uses the `AccountDataAccess` interoperation layer to read back the information posted to see whether it has actually been stored.

The style of programming for REST Assured might look unusual if you haven't seen it before. REST Assured uses a fluent interface; each method invocation returns an object of the type that will be useful for the next part of the operation so that your code looks like it is written in paragraphs of (slightly stilted) English. This makes the code easy to read and easy to compose.

The paragraphs you write for REST Assured follow a paradigm known as Behavior Driven Development (BDD). BDD is an extension of Test Driven Design (TDD). In TDD, you write the test cases first, then write application code that enables the test cases to pass.

In BDD, the desired behavior is provided as a specification that can be executed as a test against the code you write to provide that behavior. These integration tests are following a pattern of `GIVEN x WHEN y THEN z`. The REST Assured library is designed to enable us to write tests that follow this format.

To run the `CreditController` tests:

1. Import the **CreditService** project into the workspace.
2. Click **Run > Run Configurations**, select **JUnit Test**, and then click the **New** icon.
3. Set the name to **CreditServiceTests** and then click the **Browse** button to select the **CreditService** project **Test Project**.

4. Set the **Test runner** field to **JUnit 4**.
5. Click the **Environment** tab and then add three environment variables:

```
dd_ACCOUNTFILE =  
application-download//CreditiService/testData/account.dat  
dd_TRANSACTIONFILE=  
application-download//CreditiService /testData/transaction.dat  
dd_CUSTOMERFILE=  
application-download//CreditiService /testData/customer.dat
```

Do *not* set these environment variables to the files where your “production” data (generated in Chapter 5) is stored. These files are replaced by the .testdat files in the test resources at the start of every test case.

6. Click **Run**.

You should see 23/23 tests pass in the Eclipse JUnit pane. You will also see the Spring Boot banner and startup output scrolling in the console as the Spring Boot application for the web server starts up for each class in the test suite.

You have now run an integration test suite that has tested our entire application from end-to-end.

Summary

In this chapter, we used MFUnit to test procedural COBOL code and JUnit to test COBOL JVM code and Java code. The JUnit tests for the Java REST service were integration tests that used REST Assured to call our Web service through HTTP and because they did not use any mock objects, they called all the way to the back end.

Although it is desirable to have unit tests as well as integration tests to provide more in-depth code coverage of edge cases, we didn’t do that here because it is difficult to mock out COBOL components. However, other ways of structuring the application would make this easier. For example, if we had based the InteroperationLayer around interfaces rather than classes, it would be easier to replace everything from the interoperation layer with a set of dummy objects or mocks for testing. We haven’t done that here because it would have complicated the example, but when modernizing applications or building new ones, we should always consider designing for testability.

As we begin changing our legacy code in the following chapters, even the limited test suite built for this book will give us confidence that the application is continuing to behave as we want it to.

CHAPTER 8



User Interface Modernization

In this chapter, you will learn about creating user interfaces (UI) for COBOL applications, including:

- UI Choices
- The Credit Service UI Application
- Creating Web UIs with the React Javascript library

UI Choices

COBOL applications are often associated with old-fashioned “green-screen” user interfaces that are text heavy and have no graphics. In the 1960s and 1970s, these were run on dedicated IBM 3270 terminals; since the 1980s, terminal hardware has been gradually replaced by software emulators that run on PCs.

These days, there are lots of choices for updating the user interface for a COBOL application. The old 3270 console screens have now mostly been replaced by Web interfaces, often using software that automatically converts the screen maps.

Remapping a console interface to Web pages can extend its usefulness as it can now be accessed from a browser, but this isn’t really modernization in the sense of providing new paradigms for communicating with the old functionality. In this chapter, we are going to look at how modernizing the back end of our application (by providing it with a REST API) enables us to create a completely new front-end that communicates with the new API. Specifically, we are going to look at the type of UI known as the “single-page Web application.”

We are not going to look at traditional desktop UIs in this chapter. As Web interfaces have become faster and more capable, there are fewer and fewer use cases that require building a dedicated UI program that runs on a computer's desktop (as opposed to in a Web browser). Web applications have a lot of advantages over desktop applications:

- The end user doesn't need to install anything.
- The application developer can deploy a new version by updating the website (no action is required by the end user).
- It is much easier to carry out canary or blue-green deployments so that new releases can be staged gradually.
- Most enterprise applications rely on data stored centrally; Web applications are naturally client-server, with the Web browser and Web server handling communication.

The only applications that are still better suited for the desktop than the Web are those that require very low latency combined with high computational requirements; image and video editing is still mainly done on the desktop, although there are Web versions of these types of applications, too.

Singe-Page Web Applications

Single-page Web applications are increasingly common these days. Google docs, Outlook.com, and Facebook are all examples of single-page Web applications. They are different from older styles of Web programming in that the user never loads a "new" page into the Web browser as they proceed through the application.

Instead, they download a single page that renders different content through Javascript manipulation of the Web browser's Document Object Model (DOM). The JavaScript is regularly sending and fetching data to and from a Web browser, to either update what the user sees or to save new information back to the Web server.

Figure 8-1 shows the basic flow of operation of a single-page Web application.

1. The user's Web browser requests the application from the Web server.
2. The Web server sends the page requested back and the page, in turn, requests the Javascript files that contain all the UI logic.
3. The Javascript manipulates the DOM, controlling the content rendered and shown to the user.
4. The Javascript can request further data from the Web server or post data back, depending on the user's interactions with the rendered content.

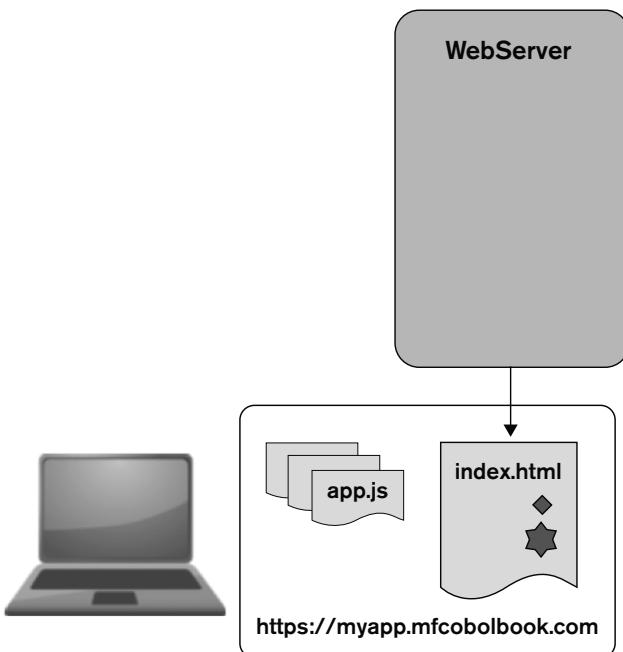


Figure 8-1 Single-page Web application

A single-page Web application has a different architecture to older-style Web applications like JSP (Java Servlet Pages) and ASP (Active Server Pages, which is the .NET equivalent to JSP). In these applications, the server generates the HTML the user sees before sending it down. Some user actions cause the server to send down a different page, which causes the browser display to refresh. Even when these applications are running well over low-latency connections with good bandwidth, this continual re-rendering gives a subjective impression that the application is not as responsive as a desktop application.

Javascript is an interpreted rather than a compiled language; the source code written by the developer can be executed exactly as it is by a Web browser. This can make the actual code for Javascript applications quite large compared with a binary format. Every byte in a Javascript application has to be downloaded to a Web browser before the application can run, whereas traditional Web applications download only one page at a time.

To minimize download and startup times, single-page Web applications are usually put through a build process that will, among other things, *minify* the Javascript. This removes all unnecessary whitespace and shortens function and variable names to one or two characters. The resulting Javascript is unintelligible to a human reader but is interpreted by the browser in exactly the same way as the original.

Single-page applications can still sometimes take longer to render the first page (even when minified, there can be a lot of code to download), but after that, all interactions feel

smoother because only changed sections of the page are rendered – much like a desktop application.

Single-page applications also have the benefit of forcing a clear separation between server logic for saving, retrieving, and manipulating data and the UI logic for presenting it to the user. A single-page application is dependent on access to a REST API for all the functions that are provided by the back-end server; this, in turn, helps developers of the Web services adopt an API-first approach where they think about building a consistent and simple API for consumption by other clients. Once you start adopting an API first approach, other possibilities open up; the same service can be consumed by a UI application or by another microservice.

There are a lot of common functions needed to write single-page applications (for example, to fetch or post data, to display dialog boxes on the Web page, and so on). Not all Web browsers have identical capabilities, so handling this in your application code is not straightforward.

Consequently, there are several Javascript libraries and frameworks to smooth out the differences between various browsers and versions and to provide the application developer with a consistent set of functions and semantics to help create a polished experience without having to write a lot of low-level code.

At the time of writing, two of the most popular frameworks are React (developed by Facebook) and Angular (developed by Google). Both are open-source which means you can use them without paying license fees. Both frameworks are very capable and make it possible to create complex applications.

Angular provides more functionality out of the box but is also more complex to learn. For this book, we've chosen React because it is quicker to get started with.

Javascript for Desktop Applications

Although we're concentrating on applications for the Web, Javascript is also being used to build desktop applications using frameworks like Electron. It's very easy to create cross-platform applications with Electron because the same code can be run on Windows, MacOS, or Linux.

The Credit Service UI Application

The application consists of a React front-end named credit-service-form and the Credit Service application we have been looking at over the last few chapters. The React front-end and the Credit Service application run on different Web servers. The architecture now looks like Figure 8-2. The interactions are similar to our previous diagram, but now

the Javascript communicates with the CreditService Web service instead of the server that supplies the form.

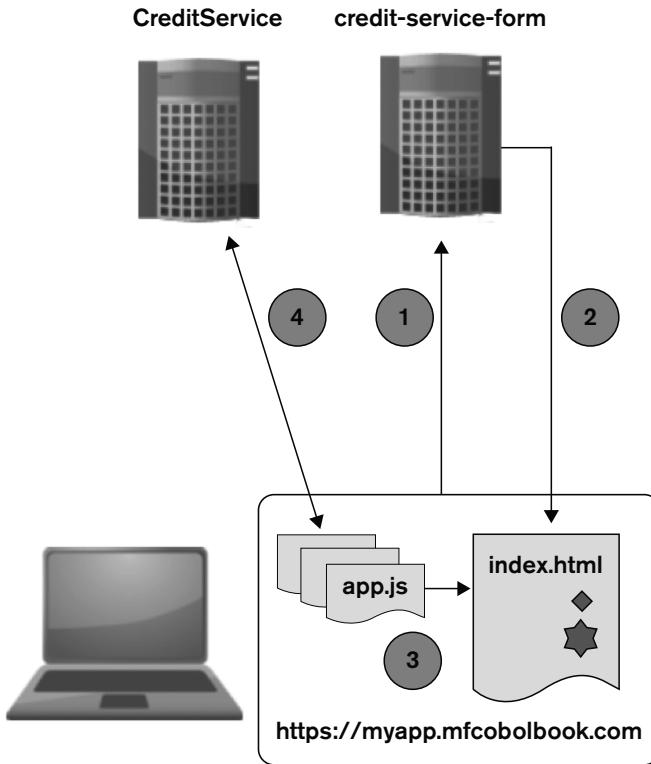


Figure 8-2 CreditService and credit-service-form

This is a common pattern with single-page applications, as the Web server that supplies the back-end REST API is doing a very different job to the Web server that hosts the UI application for download to the client. The REST API must be hosted on an application server that is running application logic, connecting to databases, and that must also be secure from unwanted intrusion.

The Web server hosting the UI has to supply only some static content to a Web browser. In the case of Web sites with high demand, this Web server might be hosted by a third-party Content Delivery Network (CDN). Alternatively, the application owners might use a high-performance server like Nginx to serve up the static content for the UI and a Java Web server for the REST service. The different parts of the application will almost certainly have different scaling needs.

In the next section, we'll download and start up the application.

Running the Application

We've provided two different versions of the application in this chapter:

- A worked example that takes you through some changes you need to make to get the entire application working.
- A complete example that shows you the finished code.

React uses the Node Package Manager (NPM) to fetch all the Javascript dependencies needed to make React applications run. Before you start, you need to install Node.js on your computer. At the time of writing, you can download Node.js from <https://nodejs.org/en/download/>. This page provides an installer for Windows; to install on Red Hat or SUSE, follow the links for installing using a package manager.

Once you have installed Node.js, start the credit-service-form application by following these steps:

1. Download the examples for Chapter 8.
2. Open a command prompt or terminal and then change to the **worked\credit-service-form** directory in the Chapter 8 examples download.

3. This directory contains the sources for the credit-service-form application, but you must install the React modules before it will run. Type the command:
`npm install`

It will take a couple of minutes for all the dependencies to download, but eventually you should see a message indicating the number of packages that have been downloaded.

4. You can now run the application in development mode. Type the command:

```
npm start
```

Node.js includes its own Web server, which deploys the application and starts your default Web browser on `http://localhost:3000`.

Your Web browser should display something similar to Figure 8-3. No data is displayed because, at the moment, there is no server application running for it to get the data from.

The screenshot shows a web application with a clean, modern design. It features a header with the title 'Credit Service Form' and a subtitle 'A simple application for managing customer accounts and transactions.' Below the header, there are three main sections: 'Customers' on the left, 'Account details' in the center, and 'Transactions' on the right. The 'Customers' section is currently empty. The 'Account details' section contains a sub-instruction 'Select a customer to see account details.' followed by two input fields: 'AccountId' and 'Balance'. The 'AccountId' field is empty, while the 'Balance' field contains the value '0'. The 'Transactions' section is also currently empty.

Figure 8-3 The credit-service-form application

To display some data, start up the Web server:

1. Import the **BusinessInterop**, **BusinessRules** and **CreditService** projects into a new Eclipse workspace.
2. For the **BusinessRules** project and the **BusinessInterop** project, right-click each project, click **Properties > Builders**, and then edit **mvn_businessrules** for your local setup (as explained in the “Importing the Example Projects” section in Chapter 5).
3. Create a Java Application Run Configuration and give it the name **CreditService**. Set the **Project** field to **CreditService** and the main class to **WebserviceApplication**. Then add environment variables dd_ACCOUNTFILE, dd_CUSTOMERFILE and dd_TRANSACTIONFILE to point to the data files you created in the “Generating Data” section in Chapter 5.
4. Click **Run** to run the application.
5. Open a browser tab and point it to <http://localhost:8080/service/customer/1>. Provided the Web server started and the environment variables are set correctly, some data should be returned.
6. On the browser tab where the React application is running credit-service-form, click **Refresh**. It still shows no data. In your browser, open the developer tools and look at the console. You will see an error similar to this:

```
Access to fetch at 'http://localhost:8080/service/customer/' from origin
'http://localhost:3000' has been blocked by CORS policy: No 'Access-Control-
Allow-Origin' header is present on the requested resource.
```

Browser Developer Tools

Most modern browsers include developer tools that help with debugging Web applications. The developer tools include tabs showing page elements, a console, a Javascript debugger, and a network viewer. The tools are similar across most browsers; Chrome, Edge, Firefox, and Opera can be opened with the keyboard shortcut **Ctrl+Shift+I**.

Cross-Origin Resource Sharing

Web browsers do not allow Javascript to retrieve data from arbitrary addresses; by default, Javascript can retrieve data from only the same host as the Javascript itself was served from. The port is part of the hostname, so the browser treats localhost:8080 and localhost:3000 as different hosts.

This is to prevent cross-site scripting (XSS) exploits. For example, imagine you have logged into an application that serves data from www.secureapplication.com. The pages downloaded from this site enable you to access the secure data. But visiting www.villainousbehavior.com downloads a page with some Javascript that also attempts to access www.secureapplication.com using your credentials. To make these exploits more difficult to execute, a browser will, by default, refuse to fetch data from one domain for a script hosted from another domain.

To enable our React application to fetch data from our CreditService application, we need to set a Cross Origin Resource Sharing (CORS) policy on CreditService. When a script requests data from a different domain, the browser sends a request to that domain to find out if this is permitted. If the server returns an Access-Control-Allow-Origin header that matches the requesting domain, the browser will provide the data to the requesting Javascript.

Spring Boot makes it relatively easy to set this policy for your application. To enable CORS on CreditService:

1. Open the WebserviceApplication class in Eclipse.
2. Add the code in Listing 8-1 after the main() method.
3. Rebuild and restart the application.

Listing 8-1 Configuring CORS on the CreditService

```
@Bean
public WebMvcConfigurer corsConfigurer() {
    return new WebMvcConfigurer () {
        @Override
        public void addCorsMappings(CorsRegistry registry) {
            registry.addMapping("/**")
                .allowedOrigins("http://localhost:3000");
        }
    };
}
```

This policy enables applications from <http://localhost:3000> to access all URLs in the CreditService application. You can also enable CORS policies on individual controllers using a CrossOrigin annotation and you can use wildcard matching as well as exact names. Consult Spring documentation for more information about CORS, WebMvcConfigurer and REST services.

Now that you have added the new configuration to the CreditService, refresh the browser tab with the React application; you should now see a list of customers down the left side of the page. Click on a customer to see that customer's balance and transactions (see Figure 8-4).

Customers			Account details			
1	Worthington	Parrot				
2	Mira	Jaggard				
3	Evie	Ariss				
4	Carlina	Cranidge				
5	Inock	Bearfmore				
6	Saxe	Dibbert				
7	Zenia	Kift				
8	Alleen	Sackes				
9	Skelly	Kwiatak				
10	Andreana	Bread				
11	Henri	Laurant				
12	Fidel	Maleby				
13	Shellysheldon	Battson				
14	Aylmer	Nornable				
15	Corilla	Aldington				
16	Elisavetka	Enssle				

Transactions				
74	20190706	61.36	Beauty	
75	20190709	17.8	Clothing	
76	20190710	13.42	Outdoors	

Figure 8-4 Application showing data

To stop the application, go to the command prompt where you started the application with `npm start` and press **Ctrl+C**.

In the next section we will look at the React application in more detail.

Getting Started with React

In this section, we'll take a closer look at the React code to get some understanding of how the credit-service-form application interacts with the CreditService that provides the data. This documentation outlines some React basics, but it isn't a substitute for the React documentation or a full React tutorial.

If you want to learn React and write your own applications, there is an excellent tutorial on the React website (<https://reactjs.org/tutorial/tutorial.html#setup-for-the-tutorial>) or try searching the Web for "React JS Tutorial." There are also many online courses for React.

Development Environments for React

Eclipse as supplied with Visual COBOL doesn't provide great functionality for working with Javascript applications. There are some plugins that can help, but most of them require commercial licensing. I recommend using Visual Studio Code for working with applications like React. If you aren't familiar with Visual Studio Code (not to be confused with Visual Studio), it is a free code editor from Microsoft that can be installed on Windows or any of the Linux desktop environments supported by Visual COBOL.

Visual Studio Code includes syntax highlighting and intellisense for Javascript applications, plus you can open the folder containing your application and navigate it using Visual Studio Code's Explorer. You can use any code editor of your choice to work through the rest of this chapter, but Visual Studio Code is a good choice if you have no strong preference already. You can download it at <https://code.visualstudio.com> or just search for "VS Code" on the Web. Figure 8-5 shows Visual Studio Code after opening the folder containing the credit-service-form application.

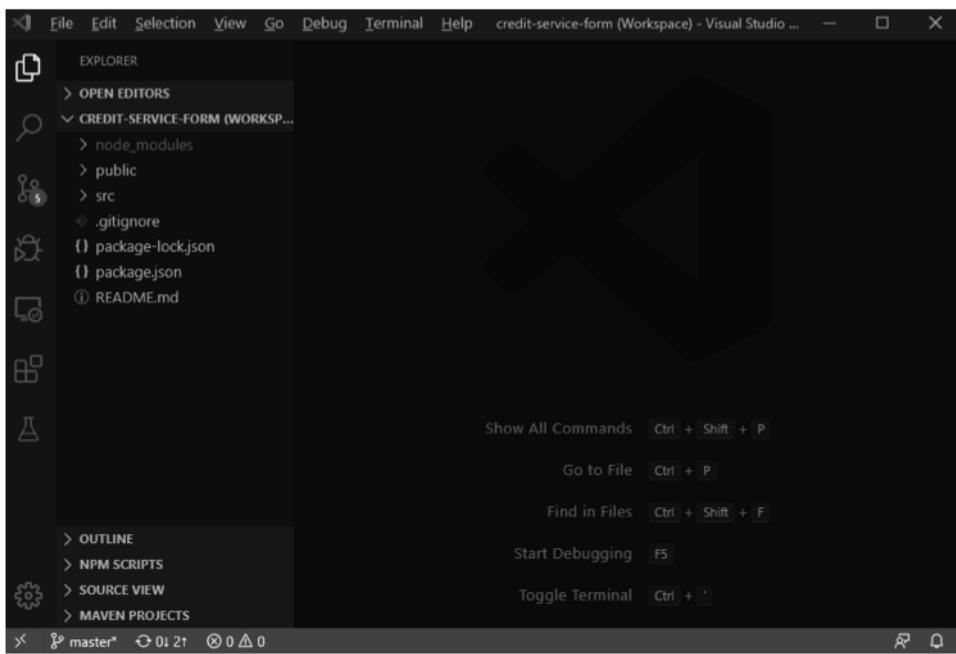


Figure 8-5 Visual Studio Code view of credit-service-form

Structure of a React Application

Figure 8-6 shows the file structure of a React application (like credit-service-form). The JS code for the application is in the src sub folder, while the main assets (an HTML page and some icons) are in the public folder. Figure 8-2 showed the single HTML page that is sent to the browser to render the application; that is index.html inside the public folder.

The package.json file is a list of dependencies and scripts for the application. The package-lock.json looks like a more complicated version of package.json; it is the full list of all the transitive dependencies needed to run the application. *Do not edit* this file by hand, but *do* store it as part of the version control for an application. For more information, see the React documentation.

The application directory is created as a local git repository and includes an appropriate .gitignore file. This is the structure created when you run the command npx create-react-app my-application. It is also located under credit-service-form when you download the examples for this chapter.

When you ran the npm install command to start the application running earlier in the chapter, NPM created a node_modules folder with all the dependencies needed to run your application. Visual Studio Code understands the structure of a React application and shows this in gray rather than white because you wouldn't normally edit any of the files in here or store them in your source code control system.

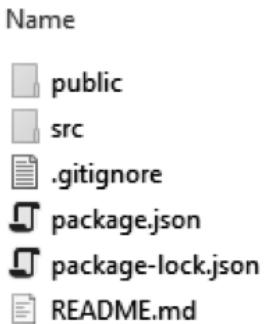


Figure 8-6 Directory structure of credit-service-form application

The `npm start` application that runs the application starts a local instance of the Node server on your computer and opens the application in your default Web browser. When you make a change to the source code, npm automatically rebuilds the application and the results are visible immediately in the browser.

This works very well when you are working on the application but is not so good for publishing it on the Web because there are several Javascript files in the `node-modules` directory, so the overall download size is large. The `npm run build` command creates a compact minified version of the application in the `build` directory for distribution. This uses a set of default tools for the build chain, but the React documentation explains how to create your own build chain if you want more flexibility in how the application is built.

React Philosophy

Compared with Javascript frameworks like Angular, React is a relatively simple library that is focused on making it easy to build the visual components of an interface. The learning curve is not as steep, but you have to decide which other technologies and libraries to include to build a fully functioning application.

Angular includes functionality to bind controls directly to server APIs; with React, you can use other libraries to do this or you can do it yourself. React focuses on:

- Rendering the application visually
- Making it easy to manage state changes
- Handling events

Even handling forms controls is left to the programmer in React; this can lead to you writing a lot of boilerplate code that doesn't add any value to your application. However, React has a rich ecosystem of libraries provided by other developers and vendors that enables you to build elegant and robust applications.

Dependencies for the credit-service-form Application

I am not a full-time Javascript developer, so I followed the line of least resistance to build the credit-service-form, selecting some other well-known libraries to supplement React:

- Formik handles all the repetitive code otherwise needed to work with forms in React. Formik is another library managed through NPM that I added to the application during creation with the command `npm install formik`. Since Formik is already part of credit-service-form, you don't need to install it again. At time of writing, the home page for Formik is <https://jaredpalmer.com/formik/> and you can also download the sources from [github](#).
- Bootstrap provides a set of Cascading Style Sheets (CSS) that gives your application a professional look and feel. Bootstrap (<https://getbootstrap.com>) also includes some Javascript libraries to provide functionality like dialog boxes. At time of writing, the home page for Bootstrap is <https://getbootstrap.com>. I didn't actually download Bootstrap to use it; credit-service-form includes references to Bootstrap CSS and Javascript files on the index.html root page of the application, taking them from a third-party Content Distribution Network (CDN). One of the advantages of this approach is that because so very many websites use Bootstrap, there are many opportunities for the library to be cached by your Web browser or by your ISP, improving load times. Most CDNs also provide server nodes in many regions across the globe, meaning that data provided by a CDN can often be supplied by a server geographically close to a client located anywhere.
- The Fetch API used to connect the Javascript application to the REST API provided by CreditService. The Fetch API is available in most modern browsers, although not in Internet Explorer. I've assumed that all readers will have access to at least one of Microsoft Edge, Chrome, Firefox, or Opera, all of which support Fetch. You can check which versions of popular browsers support different API calls at <https://developer.mozilla.org>. The Mozilla site also provides comprehensive documentation on browser APIs available to developers.

If you need to support an older browser like Internet Explorer, you can often find "polyfill" libraries that provide newer Javascript APIs for older browsers. For example, search the Web for "javascript polyfill fetch."

Application Components

Our React application consists of a few separate components, each of which is responsible for rendering one part of the page shown in Figure 8-4. As shown in Listing 8-2, we will start by looking at App.js, which is the main application file.

*****Listing 8-2***** *The start of the App.js file*

```
import React from "react";
import "./App.css";
import CustomerList from "./CustomerList.js";
import AccountDetails from "./AccountDetails.js";
import TransactionList from "./TransactionList.js";
import RESTApi from "./RESTApi";

function App() {
  return <MainPage value="http://localhost:8080" />;
}


```

Listing 8-2 shows only the first few lines of this file. The `import` statements pull in the other components for the application, which are defined in separate Javascript files. You do not need to define each component in a separate file because Javascript doesn't really have any rules about this, but you should consider this approach because larger applications are easier to work on when the code is logically organized into separate files.

The next statement defines a function called `App()`, which is the main entry point for the application and is called when the `index.html` page is first loaded by the browser. It consists of one statement, which might look a bit odd if this is your first encounter with React. The value returned by the function is `<MainPage value="http://localhost:8080" />`, which looks closer to HTML than Javascript, especially because it is not quoted the way a string literal would be. It is actually an extension to Javascript syntax called **JSX**, which enables you to define how your application is rendered using React components. React components are either predefined ones that correspond to HTML elements (for example, `<p>` or `<h1>`) or they are components you define yourself. In this case, `<MainPage>` is a component defined as part of the application. You can define a new React component either with a single Javascript function with the same name as the component or as a Javascript class with the same name as the component.

You do not have to use JSX when writing React applications; JSX is converted into plain Javascript when you build a React application. The React documentation explains how you can write this Javascript yourself instead of using JSX, but JSX makes it much easier to understand and write React applications, so we will use JSX throughout this chapter.

Each React component in your application is instantiated as a Javascript object at run-time, which contains two separate data objects: `props` and `state`. Each of these data objects can consist of any number of fields, each of which can hold data or functions.

Anything declared inside a JSX element as an attribute is passed through to the component as a field inside `props`. Figure 8-7 represents the `MainPage` element with the value passed through to it in Listing 8-2.



Figure 8-7 The `MainPage` element with `props` and `state`

The `MainPage` Component

Our `MainPage` component is defined inside `App.js` as the `MainPage` class, which extends `React.Component`. `React.Component` provides all the base functionality for elements to render and update. Listing 8-3 shows the `render()` function for `MainPage`, which is executed each time it needs to be rendered on the page.

Listing 8-3 The `MainPage` `render()` function

```
render() {
  return (
    <div className="row">
      <div className="col">
        <div>
          <h1>Customers</h1>
          <div>
            <CustomerList
              className="container scrollable"
              value={{
                onCustomerSelected: x =>
                  this.handleCustomerSelected(x),
                customerData: this.state.customerData
              }}
            />
          </div>
        </div>
      </div>
      <div className="col">
        <div className="row">
          <div className="container">
            <h1>Account details</h1>
            <p>Select a customer to see account details.</p>
            <AccountDetails value={this.state.accountData} />
          </div>
        </div>
      </div>
    </div>
  )
}
```

```

<div className="row">
  <div className="container">
    <h1>Transactions</h1>
    <TransactionList value={this.state.transactionData} />
  </div>
</div>
</div>
);
}
}

```

The `render()` function returns JSX describing what should appear in the Browser. It consists mostly of `div` and `h1` elements, which will be rendered as HTML `div` and `h1`. However, there are some differences between JSX and HTML – one of which is that the HTML class attribute is replaced by `className` in JSX. Here we are using Bootstraps `row` and `col` styles to provide a two-column layout for the page – customers down the left and accounts and transactions down the right.

The `render()` function also returns three other JSX elements defined by the application: `CustomerList`, `AccountDetails`, and `TransactionList`. Figure 8-8 shows the layout that will be rendered by `MainPage` – the headings and the three components that make up the application itself.

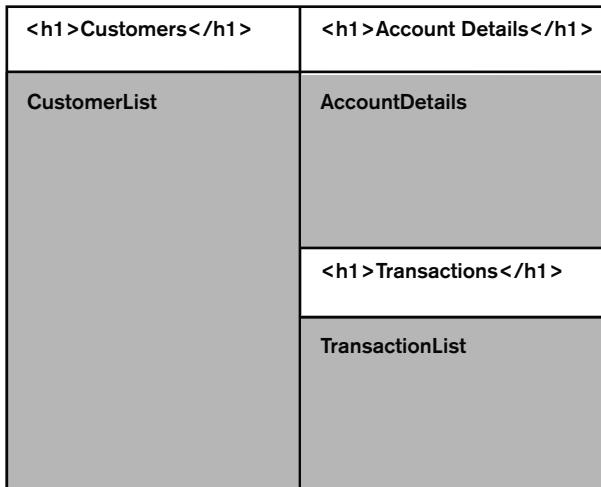


Figure 8-8 Layout of the `MainPage` element

Each of the components of the application receives a `props.value` that is part of the application state of `MainPage`. The props being passed through to each component define the data that those components will actually display in their render methods.

This is a typical pattern for React applications. A higher-level component will pass data to lower-level components for rendering. When the data to be rendered changes, an event sent to the parent component will trigger the `setState()` function on the component. The `setState()` function enables you to change the state of the component and also triggers the `render()` function, causing the component and all of its children to be rerendered with the new data. Listing 8-4 shows the `MainPage` class minus the event handler and render functions.

Listing 8-4 The `MainPage` class

```
class MainPage extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      customerData: null,
      customerId: null,
      accountData: {
        id: "",
        customerId: "",
        balance: 0.0,
        type: "",
        creditLimit: 0.0
      },
      transactionData: []
    };
    this.RESTApi = new RESTApi(props.value);
    this.accountFetched = this.accountFetched.bind(this);
    this.accountFetchFailed = this.accountFetchFailed.bind(this);
    this.transactionsFetched = this.transactionsFetched.bind(this);
    this.transactionsFetchFailed = this.transactionsFetchFailed.bind(
      this
    );
  }

  componentDidMount() {
    fetch(this.RESTApi.serviceUrl("service/customer"))
      .then(response => {
        return response.json();
      })
      .then(myJson => {
        this.setState({ customerData: myJson.array });
      });
  }
}
```

```
// Event handlers omitted from listing
. . .
// render() function omitted from listing
. . .
}
```

The constructor receives a single parameter, `props`, which contains the data passed into the component (see Listing 8-2, where the only data passed in was the URL for the REST application). The constructor passes this to the superclass and then defines the `props` object for the component. This has fields for `customer`, `account`, and `transaction` data, which are all set to “empty” values. But we want to display actual data fetched from the REST API. This is done by the `componentDidMount()` function. `React.Component` defines an empty implementation of this, which the `MainPage` class overrides. `React` calls this function after a component is loaded and rendered.

It has been overridden here to call the REST API using `fetch()` to retrieve the entire list of customers. The lambdas defined inside the `then()` method calls are executed as each previous operation is completed. The final lambda calls `setState()` to update `MainPage` `props` with the array of customers that has been retrieved. Because the `props` of `MainPage` has been changed, the `render()` function is called and the `CustomerList` component now gets the list of customers to render.

The `fetch()` function itself is asynchronous; because accessing resources over HTTP is comparatively slow, `fetch()` returns before the HTTP network call has actually completed; this function does not actually return with the data you are interested in. What you get back is an object called a promise.

Each `promise.then()` function takes a lambda – effectively a function that will be executed later, when the HTTP call has completed and the promise can be fulfilled.

If you are unaccustomed to asynchronous programming, this style of programming can be a little confusing to begin with. Figure 8-9 shows the sequence of events – `componentDidMount()` is executed, initiates the `fetch`, and provides lambda functions to the promises returned by the `fetch()` and `then()` functions. Then when the executions represented by the promises complete, the lambda code is executed.

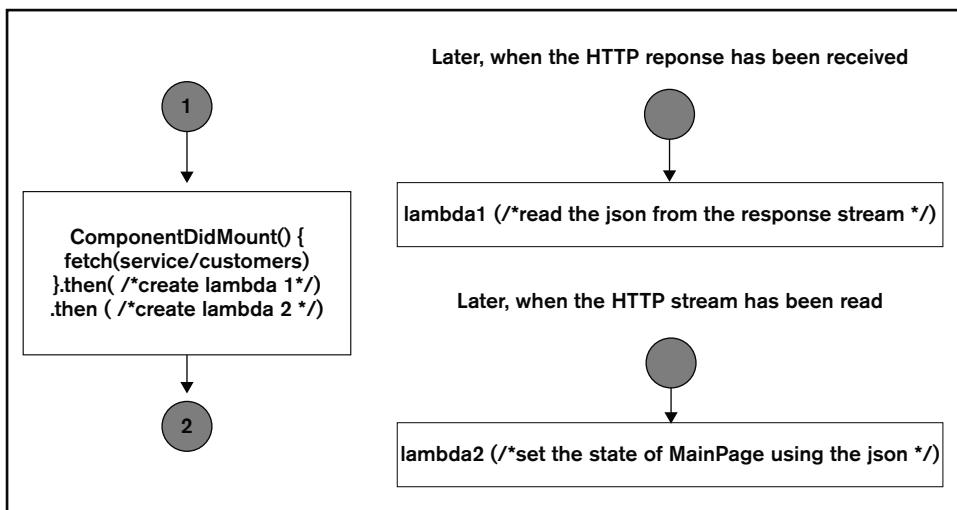


Figure 8-9 Asynchronous code execution

The CustomerList Component

The `CustomerList` component actually renders a list of customers on the browser and is defined inside `CustomerList.js`, together with a `Customer` component (which renders an individual customer). Listing 8-5 shows the `CustomerList` component.

Listing 8-5 The `CustomerList` component

```

class CustomerList extends React.Component {
  customerSelected(customerId) {
    if (this.props.value.onCustomerSelected) {
      this.props.value.onCustomerSelected(customerId);
    } else {
      console.debug(
        "No onselected handler provided to CustomerList"
      );
    }
  }

  render() {
    const customers = [];
    if (this.props.value.customerData) {
      this.props.value.customerData.map((x, index) =>
        customers.push(
          <Customer
            key={index}
            value={{ customer: x, index: index }}>
        )
      );
    }
  }
}

```

```

        onClick={() => {
            this.customerSelected(x.id);
        }}
    >
    </Customer>
)
);
}

return <div className="scrollable">{customers}</div>;
}
}

```

If you look at Listing 8-3 again, you can see that `CustomerList` gets passed a `props` value object with fields `customerData` and `onCustomerSelected`. The `customerData` field is expected to be an array of customer objects. The `render()` code uses the Javascript `map()` function, which iterates over the array invoking the supplied lambda function with each element and element index.

This is used to create an array of `Customer` elements. Each `Customer` element has a `props` object that defines the index in the array, the data for one customer, and an `onClick` function handler that will call the `customerSelected()` function of `CustomerList`.

`Customer` itself is defined by the `Customer()` function also in `CustomerList.js` and shown in Listing 8-6.

Listing 8-6 The Customer function

```

function Customer(props) {
    return (
        <div
            className={
                "padded highlightable row " +
                (props.value.index % 2 === 0 ? "grey" : "white")
            }
            onClick={props.onClick}
        >
            <div className="padded col">{props.value.customer.id}</div>
            <div className="col">{props.value.customer.firstName}</div>
            <div className="col">{props.value.customer.lastName}</div>
        </div>
    );
}

```

This is the other way of defining a renderable element in React – by providing a `render` function with the name of the component. This function uses the `customer` object passed in as part of the `props` to display the id, first name, and last name in three columns; the

onclick handler will be called every time the row is clicked. It also uses styles grey and white (defined in the App.css file) to provide the alternate shading for each row (even rows are grey and odd rows are white).

The Remaining Components

By now you should be starting to get a feel for how the application works (built out of components which are passed down data from MainPage). When you click on a customer, the event goes the other way, up from the Customer component, via the CustomerList, until finally triggering the MainPage.handleCustomerSelected(). This uses functionality in the RESTApi class (RESTApi.js) to fetch the related account and the related list of transactions.

You can examine the code in AccountDetails.js and TransactionList.js to see how the rest of the application works. In the next section, we will extend the application functionality to enable us to add new customers.

Extending the Application

In this section, we will create an AddCustomerForm component, which is a modal dialog that enables us to add new customers. Use the codebase in the Chapter 8 worked example to do this. If you want to see the finished application, it is in the Chapter 8 complete example.

To make this work, we need to add the following items to the application:

- A button to display an Add Customer form
- An AddCustomerForm dialog box component
- New event handlers:
 - To toggle display of the dialog box on and off
 - To handle the user clicking Save on the dialog box
- Code to POST the new user details back to the REST API

In the next few sections, we will add the new code to our UI application. We'll take advantage of the fact that React enables us to edit the application while it is running in the NPM development server, so as we make changes, you will see the running application change. To make it easier to find the places where new code should be added, there are comments in the application Javascript files that start "Step 1", "Step 2", etc.

The Add Customer Button

Start both the UI application (credit-service-form) and the COBOL back end (CreditService) as explained previously in this chapter in the "Running the Application"

section. As you change the React code, the UI will update, showing your changes. And if you make a mistake, error messages will indicate what's wrong – both in the command prompt where you are running `npm start` and the Web browser. Figure 8-10 shows the Add New button.

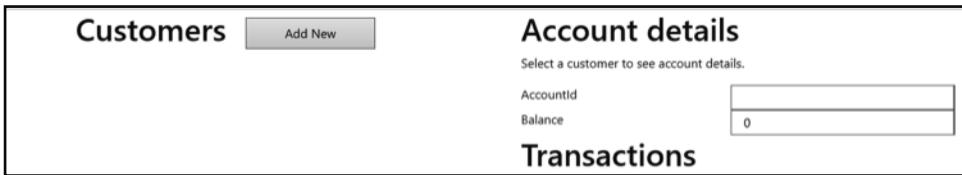


Figure 8-10 The Add New button

To add a button labeled *Add New*:

1. Open **App.js** in your editor of choice.
2. Find the `MainPage render()` function. At the moment, this function starts with three `<div>` elements in succession. The first two use style classes to create a row and column, respectively. The last div is a container for the `Customers` heading and the `CustomerList` component (shown on the left side of Figure 8-8). We are going to make this third div into another nested row and make the `<h1>` and new `<button>` elements columns so that they appear next to each other.
3. Find the Step 1 comment just before `<h1>Customers</h1>`. Replace `<h1>Customers</h1>` with the code in Listing 8-7.
4. Save your changes. NPM rebuilds your application and the changes display in the Web browser. Any error messages will be displayed in the console where you are running NPM as well as in the Web browser.

Listing 8-7 Code to render a button on the form

```

<div className="row">
  <h1 className="col">Customers</h1>
  <button
    style={{ margin: 10 + "px", height: 40 + "px" }}
    className="col"
    onClick={this.toggleAddCustomerDlg}
  >
    Add New
  </button>
  <div className="col" />
</div>

```

There is some extra styling added to the button to set its height and the distance between it and other elements. There's also an `onClick` handler, but the function it points to has not been added yet, so clicking the button does nothing right now.

The AddCustomerForm Dialog

In this section, we add the code for the `AddCustomerForm` dialog. Since this is a new React component, it is best to put it in its own separate source file.

1. Create a file called **AddCustomer.js** in the `src` directory of the `credit-service-form` application.
2. Insert the contents of Listing 8-8 into `AddCustomer.js` and then save the file.

This file contains a single class, `AddCustomerForm`, which is a `React.Component`. The import statements at the top pull in the `React` and `Formik` libraries as well as some extra styles defined for the application in `App.css`. There is also an `export` statement at the end that exports `AddCustomerForm` so it can be imported into other application files.

Listing 8-8 `AddCustomer.js`

```
import React from "react";
import "./App.css";
import { Formik, Field, Form } from "formik";

class AddCustomerForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      customerData: null,
      show: props.show
    };
    this.hide = this.hide.bind(this);
  }

  hide() {
    this.setState({ show: false });
  }

  render() {
    if (!this.props.show) {
      return null;
    }

    return (

```

```
<div className="modal-dialog modal-dialog-centered">
  <Formik
    onSubmit={ (values)=>
    {
      this.props.service.postCustomer(values,
        this.props.success,
        this.props.failure);
    }
  }
>
<Form>
  <div className="modal-content">
    <div className="modal-header">
      <h5 className="modalTitle">Add New Customer</h5>
      <button
        type="button"
        className="close"
        data-dismiss="modal"
        aria-label="Close"
        onClick={this.props.onClose}
      >
        <span aria-hidden="true">&times;</span>
      </button>
    </div>
    <div className="modal-body">
      <div className="row">
        <label className="col">First Name</label>
        <Field
          style={{ marginRight: 20 + "px" }}
          className="col"
          type="input"
          name="firstName"
          value={this.props.firstName}
        />
      </div>
      <div className="row">
        <label className="col">Last Name</label>
        <Field
          style={{ marginRight: 20 + "px" }}
          className="col"
          type="input"
          name="lastName"
          value={this.props.lastName}
        />
      </div>
    </div>
  </div>
```

```
<div className="modal-footer">
  <button
    type="button"
    onClick={this.props.onClose}
    className="btn btn-secondary"
    data-dismiss="modal"
  >
    Cancel
  </button>
  <button type="submit" className="btn btn-primary">
    Save changes
  </button>
</div>
</div>
</Formik>
</div>
);
}
}

export default AddCustomerForm;
```

The constructor at the top of the file passes the props up to the superclass and initializes a props object with two fields: customerData and show. The customerData field contains the input from the end user when the Save changes button is clicked. The show field is Boolean; when true, the dialog is rendered and when false, it is not rendered. You can see the render() function returns null when this.props.show is not true.

The dialog box itself is a <div> element styled with the Bootstrap modal-dialog and modal-dialog-centered styles. Within this is a <Formik> element, which contains an onSubmit event handler that will be triggered when the button marked as type="submit" is clicked. Most of the code for the modal dialog is taken directly from the Bootstrap example for a modal dialog, but the <div> styled as modal-content is wrapped inside a <Form> control. The <Field> elements are Formik components that take their value from the component's props and pass the values in them to the onSubmit handler.

The AddCustomerForm component has the code to render the dialog and to handle its events, but we need to add it to the application page in order to display it. We will add to the MainPage.render() function. When state.show==true, it will render, concealing the rest of the page; when it is false, it will be invisible.

A render() function can only return a single root element, so to add the dialog to MainComponent, we have to wrap both the new dialog code and the existing code inside a single <div> element.

To add the form:

1. Open **App.js** for editing.
2. Find the Step 2 comment near the top of the file and follow the instruction to uncomment the import statement below it. This imports AddCustomerForm so that we can refer to it from App.js.
3. Find the Step 3 comment at the top of the render() function and insert the code in Listing 8-9 below it.
4. This code includes the start of a new <div> element that needs to be closed or the application will no longer run. If you save the file now, you will see “Parsing error: Unexpected JSX Contents”.
5. Go to the bottom of the render() method. Immediately below the comment that starts Close div element..., add </div> and then save the file.

The code compiles, but the following warning displays: Comments inside children section of tag should be placed inside braces. This is because by adding the closing </div>, the comment is now part of JSX rather than Javascript. If you either delete the comment or format it like the ones at the top of the render() method, the warning disappears.

Listing 8-9 Code to render AddCustomerForm on MainPage

```
<div>
  <AddCustomerForm
    show={this.state.addCustomer}
    onClose={this.toggleAddCustomerDlg}
    success={(customer) => {
      this.toggleAddCustomerDlg();
      this.state.customerData.unshift(customer);
      this.setState({
        customerData: this.state.customerData,
      });
    }}
    failure={() => {
      alert("Couldn't add customer");
    }}
    service={this.RESTApi}
  />
```

We now have all the code to render the form, but it still does not appear when the Add New button is clicked. In the next section, we will add the event handler that makes this work.

Adding Event Handlers

At this point, there is an Add New button and a new form, but the form doesn't display and we still can't add a new customer. We now have to wire up the Add New button to an event handler to make it work.

1. Look for the Step 4 comment in App.js, which is about a third of the way down the file.
2. Insert the code in Listing 8-10 and then save the file.

Listing 8-10 Event handler to display and hide dialog

```
toggleAddCustomerDlg() {  
  this.setState({ addCustomer: !this.state.addCustomer });  
}
```

If you now click the Add New button, the following error displays in your browser: `TypeError: this is undefined`. The `toggleAddCustomerDlg()` function is passed as a parameter to the `onClick` event-handler of the Add New button (and also to the `AddCustomerData` form as part of its props for the `onClose` and `Submit` events). When `toggleAddCustomerDlg()` actually gets executed, it has lost the `this` context it would otherwise have had as part of the `MainPage` component.

To make it work, we need to bind the correct `this` context to the function so it is always defined:

1. Find Step 5 inside the `MainPage` constructor function and uncomment the specified lines.
2. Save the file.

Now when you click the Add New button, the dialog appears. Clicking either the Cancel button or the Close button (the small x in the top-right corner) closes the dialog. Clicking the Save Changes button displays the following error: `this.props.service.postCustomer` is not a function. We'll define this in the next section.

Posting the New User Details to the Application

If you look back at the code in Listing 8-9, one of the values passed in as part of the `props` for `AddCustomerForm` is `service={this.RESTApi}`. The `RESTApi` object defines all the functions for communicating with the COBOL powered REST service that provides all the back-end functions for this application. However, the `AddCustomerForm` tries to call a function `postCustomer()` when you submit the data (see Listing 8-8), but this doesn't yet exist.

1. Open the RESTApi.js file.
2. Find the Step 6 comment and insert the code in Listing 8-11.
3. Save the file.

Listing 8-11 POST new customer to REST API

```
/** 
 * Call REST service to add a new customer in data.
 * succeeded and failed functions called depending on
 * success
 * @param {*} data
 * @param {*} succeeded
 * @param {*} failed
 */
async postCustomer(data, succeeded, failed) {
  var url = this.serviceUrl("service/customer");
  const response = await fetch(url, {
    method: "POST", // *GET, POST, PUT, DELETE, etc.
    mode: "cors", // no-cors, *cors, same-origin
    cache: "no-cache", // *default, no-cache, reload, force-cache, only-if-
    cached
    headers: {
      "Content-Type": "application/json",
    },
    redirect: "follow", // manual, *follow, error
    referrerPolicy: "no-referrer", // no-referrer, *client
    body: JSON.stringify(data), // body data type must match "Content-Type"
  header
  });
  if (response.status === 200) {
    succeeded(await response.json());
  } else {
    failed(data);
  }
}
```

The postCustomer() function expects three parameters:

- The customer data itself (an object with firstName and lastName fields).
- A function to call whenever the POST is successful (the server returns HTTP status 200).
- A function to call whenever the POST fails.

The success and failure functions are already defined as lambdas inside the props for the AddCustomerForm element in Listing 8-9. If you now click Add New, fill in a first and last name, and then click Save Changes, the application adds a new customer and saves the details using the COBOL CreditService application.

You will also notice that the new customer appears at the top of the Customers list. This is actually done inside the render() function of MainPage. The success lambda code includes these statements:

```
this.state.customerData.unshift(customer);
this.setState({
  customerData: this.state.customerData,
});
```

`unshift()` adds an item to the top of the `customerData` array in `MainPage` and then `setState()` rerenders the component with the new data. If you refresh the page in the browser after adding a new customer, the data will be reread from the REST service and the new customer details display at the end (as it will be the last record in the file).

Summary

This chapter demonstrated that once you have converted a COBOL application into a REST service, it is simple to fit a modern Web UI. Although this chapter used React, you can use any technology that can make REST calls. In the next chapter, we will look at containerization of the application and start refactoring the application to make it work better in the cloud.

CHAPTER 9



Containerizing COBOL Applications

In this chapter, we refactor the CreditService application so that it can be built and deployed as a container. This is a stepping stone to deploying it in different cloud environments (which you will learn about in the next chapter). Coming up:

- Containerizing Applications for the Cloud
- Changing from ISAM to a Database
- Running the Revised CreditService Application
- Containerizing the CreditService

Containerizing Applications for the Cloud

In this chapter, we are going to refactor the CreditService application to make it run well “in the cloud.” To achieve this, we are going to:

- Refactor it to use a relational database rather than COBOL ISAM files
- Containerize it so that it is deployable from a single image that contains its dependencies

Although you can run almost any application in a container, not every application in a container is cloud-native. Cloud-native applications are those that best exploit the capabilities of cloud computing. They have the following characteristics:

- Horizontally scalable — you can increase application capacity by adding more instances
- Fast startup and shutdown

- No reliance on local state

There is a worked example for this chapter that takes the now-familiar CreditService application, with its COBOL (Visual COBOL), and Java projects for refactoring. Whether you are using Windows or you are using Linux, you can follow all the examples until the final major section of this chapter, “Containerizing the Application”. You will need a Docker capable OS to be able to carry out the practical parts of this section.

You can install Docker on both SUSE and Red Hat Enterprise Linux. If you are using Windows, you can install Docker Desktop (which will also enable you to follow the containerization examples) as long as you have a Docker capable version of Windows. The section “Install as a Docker Container” provides more information about which versions of Windows run Docker Desktop.

In the next two sections, we’ll discuss containers and cloud-native applications before getting into the practical exercises.

What are Containers?

Containers started in the Linux world. A container is a way of isolating processes and operating system resources (like a slice of the filesystem) inside a namespace. The containerized process can access only the resources that have been shared inside its namespace.

You can build a container image for an application that contains all the application dependencies and run that container on any OS that has a compatible container run-time. They are called containers as an analogy to the steel shipping containers used to transport freight throughout the world. Before standardized shipping containers, moving freight was a labor-intensive process. To move freight onto trains and lorries for transport to its final destination, several workers were needed to load the freight onto ships at one end and then off-load it at the other end,

Standardized shipping containers enabled the use of standard machinery to move containers on and off ships. Instead of the freight then being moved by hand into a lorry or train, the container is simply fitted to a compatible trailer or carriage.

Thus, the time and labor required is reduced. Computer containers provide a similar benefit.

Figure 9-1 shows three applications installed on a server, each application with different run-time requirements: one needs Java 8.0, one needs Java 11, and one needs Dotnet Core 2.3. Before you can install each application, you have to install its run-time. You also need to make sure that each application is correctly configured to pick up the appropriate run-time. Sometimes different applications depend on different versions of the same shared libraries on the host OS, which adds even more complexity. (Windows programmers even have a name for this: “DLL hell.”)

Just like moving the freight before shipping containers were introduced, installing applications directly onto the OS can be labor-intensive and difficult to automate.

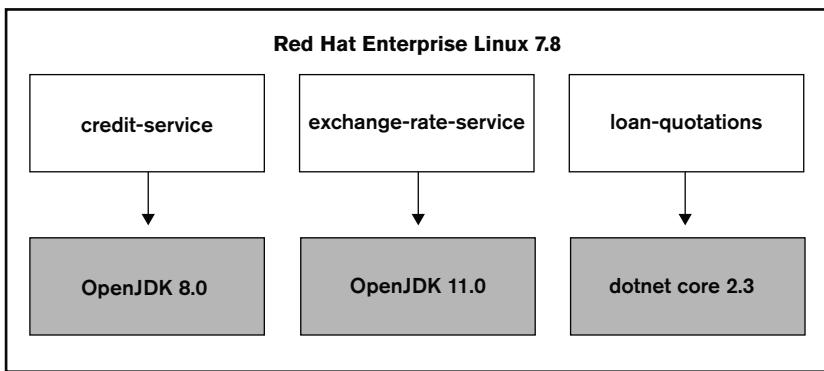


Figure 9-1 Applications running directly on a server

Figure 9-2 shows the same applications running inside containers. Each container includes the application's run-time dependencies, including an OS. (There are a number of slimmed down Linux distributions available for building containers.) The only thing needed on the host OS is the run-time to support the containers.

Each container is built from its own image stored on a container repository. Installing the applications is now just a matter of starting the container from its image. To remove an application and all its dependencies, just delete the container. Installing applications is now straightforward and easy to automate. Containerization is one of the key technologies that enables cloud-native computing.

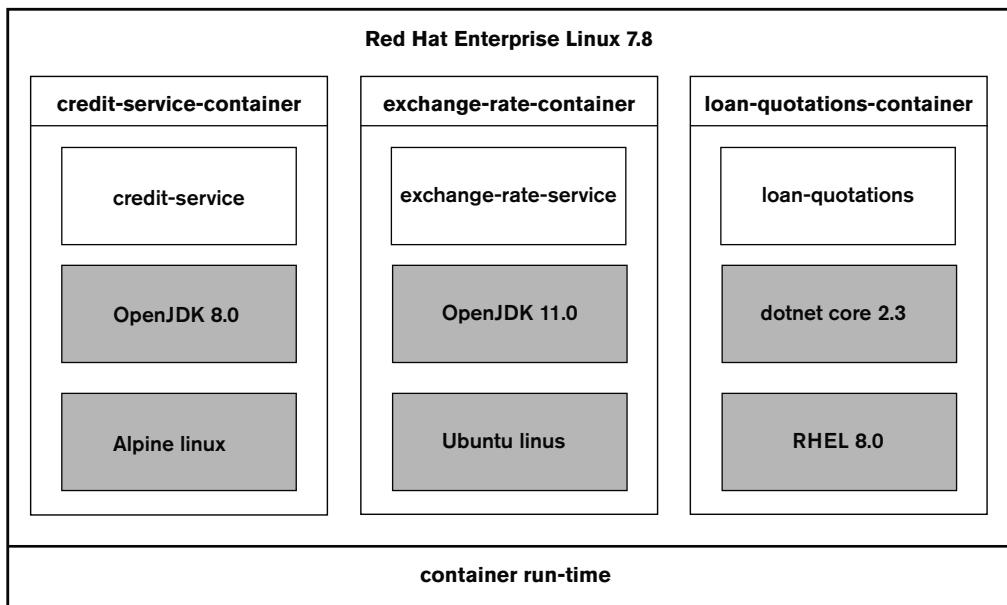


Figure 9-2 Containerized applications

Cloud-Native Applications

The term “cloud computing” has many possible meanings. We are going to talk specifically about cloud-native applications and Platform-as-a-Service (PaaS). PaaS itself is a broad category that covers vendor-specific offerings such as Amazon Elastic Beanstalk and Microsoft Azure App service as well as open-source projects like Cloud Foundry. There are commercially backed versions of the open-source projects as well, like Tanzu Application Service (which is VMware’s implementation of Cloud Foundry).

Amazon Elastic Computer Cloud (EC2) was one of the first publicly available commercial cloud offerings. EC2 provides Infrastructure-as-a-Service (IaaS). IaaS enables you to provision servers “in the cloud,” but you are responsible for the server after you have provisioned it. An EC2 server boots up with the OS of your choice, but after that it’s up to you to manage it, install the software you want, keep it patched, and so on. In terms of running applications, you still have the situation shown in Figure 9-1.

With PaaS, you provide the application and leave the platform to manage running it. It’s much closer to Figure 9-2. With some PaaS systems, you provide the application code and the platform builds a container for you; with others, you create the container image and provide that to the platform.

Whether you are using PaaS or you are using IaaS, the basis of cloud computing is Virtual Machines (VMs), which can be provisioned and started up very quickly. Networking is software-defined; you can automate and version the way your servers and connectivity are created.

Figure 9-3 shows the relationship between hardware, IaaS, and PaaS. PaaS provides a higher level of abstraction than IaaS, so you don’t have to do as much work, but the environment is more constrained than IaaS. The VMs provided by IaaS aren’t that different in terms of their capabilities than the physical servers; you can run almost anything that the virtual hardware can support.

The convenience of PaaS comes at a cost of some flexibility; your applications will have to follow some rules to be deployable on the platform. The specifics differ between platforms.

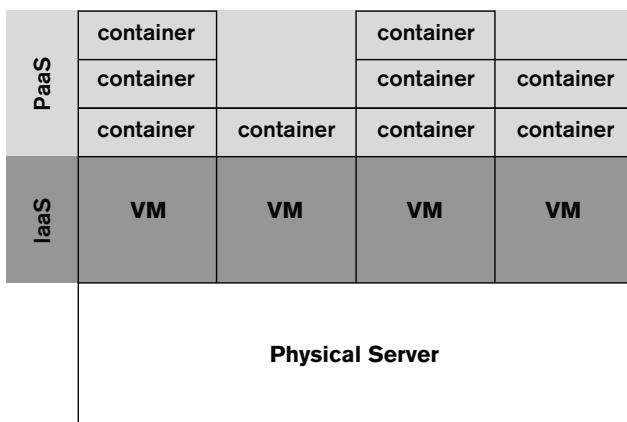


Figure 9-3 IaaS and PaaS

If you engineer an application to be “cloud-native,” it is likely to fit in with PaaS restrictions. The changes we are going to make to Credit Service mean that it would run easily on a number of cloud platforms (for example, Elastic Beanstalk or Azure App Service).

Cloud-native applications are sometimes referred to as “12 factor applications”, after the list of principles outlined at <https://12factor.net>. We aren’t going to discuss all 12 factors here, but we are going to get our application into a state where we can run and scale it on the cloud. It’s been a few years since the 12 factors were first defined, so the very top tier for cloud-native is now the “15-factor application,” but any application that meets the first 12 factors is well architected and should be easy to deploy on cloud platforms.

At the moment, our application stores all its data on the file system using COBOL ISAM files. ISAM files are not always a good fit for cloud computing; it is difficult to scale the data store on a file system across multiple instances of the application without running into locking issues and contention.

We will refactor the application to use a database. (The fourth factor is “Treat backing services as attached resources.”) This externalizes the data from the application instance and makes it easy for many instances to share the same data. Relational databases have been tuned and optimized over decades of development to work well when serving multiple clients. ISAM can still provide good performance for data that is read-only or updated infrequently.

The CreditService application is already well behaved in another way: it doesn’t maintain any state between different invocations of the service. This makes horizontal scaling easy because we can add new instances or delete existing ones at any time.

Application architectures that embrace the ephemeral nature of individual application instances are key to maximizing the benefits of cloud computing. For example, being able to scale up on demand keeps applications performant, and being able to scale down keeps them cost effective.

When you accept that an individual instance might go away at any time (either because of scaling down or some kind of failure), you design your system on that basis and it becomes much more robust. For example, having more than one application instance (ideally running on different servers in different data centers) makes your application more resilient.

One of the functions in our CreditService is the monthly interest calculation. This is a comparatively compute-intensive operation that carries out several high-precision decimal arithmetic multiplications to calculate the result. Sending statements to all the customers at the end of the month requires a lot of interest calculations. By making our application horizontally scalable, we can run more instances at month-end when we have lots of calculations to do. We then scale back down again when they are no longer needed.

Microservice Architectures

Microservice architectures aim to solve some of the problems associated with older monolithic styles of application. Monolithic applications often provide several services that are tightly coupled and they are often large and complex to deploy. Consequently, they are not updated very often – perhaps only once a quarter or once every six months. This means it takes longer to introduce new features or patch security holes.

Microservice architectures break down monolithic applications into smaller services that are loosely coupled and can be deployed independently of each other. For example, instead of using a monolithic application to run a shopping website, you could have deployed a service that manages and provides your catalog. A separate service runs the shopping cart; a third service provides customer login and enrollment. Each service has different scaling requirements, plus you might want to deliver new versions at different times.

In a monolithic application that manages all these functions in a single database, you can only ever deliver the entire application at one time. That can be problematic when one team has changes ready to go and another team is only part way through testing their new functionality. You can only scale up or scale down the entire application at the same time.

By breaking the application into separate microservices developed by separate teams, you make the dependencies clearer and easier to manage. Also, you can scale the shopping cart up at busy times without needing to scale all your other services. Microservice architectures encourage an “API-first” style of development whereby each microservice aims to provide a simple and usable API. This has other benefits: it makes it easy to do black-box testing on a service, and that, in turn, makes it easy to see when an API has changed its contract.

Microservice architectures can be very effective, but they aren’t the best solution in every case. For one thing, you get new complexities that don’t exist with monolithic architectures. Microservices need to be able to communicate with each other and they need discovery mechanisms to find each other. It’s often best to start out by making sure your monolith is composed of well-structured and well-behaved modules with clear dependencies. Start moving functionality into individual microservices when you need to address specific issues, like scalability or speed of deployment.

Tools such as the Micro Focus Enterprise Analyzer can help you understand the complexities of an existing application so that you can start breaking it into smaller pieces. Visual COBOL Eclipse can also help you better understand the structure of your application and programs through its built-in analysis and outline functionality.

Developing and deploying microservice architectures generally implies using a PaaS that has good mechanisms for managing and orchestrating the individual containers. It’s a big subject and beyond the scope of this book, so we won’t be discussing it any further detail. For some introductory articles, search the Web for “Martin Fowler microservices.” To better understand where the dividing lines between microservices are, look for books

and articles about Domain Driven Design. The seminal work is “Domain-Driven Design: Tackling Complexity in the Heart of Software” by Eric Evans. Although this book predates microservice architectures, if you can identify the domains in your application, you can see where the microservice boundaries should be.

In the next section, you will learn how to refactor the CreditService to get it to a point where it is a cloud-friendly application.

Changing from ISAM to a Database

We are going to refactor the COBOL part of our application so that it uses a relational database to store data instead of COBOL indexed files. This externalizes the storage used by the application so that when it is deployed, we no longer have to consider the disk location where the data will be stored. That is now the concern of the database. This becomes important when we begin using containers for deployment.

It also makes it easier to scale the application horizontally because multiple instances of the application can all connect to the same database. Although multiple COBOL processes can read and write concurrently to shared files, it is difficult to make this solution perform well. Micro Focus Fileshare can provide good performance for concurrent access, but this is only supported for JVM COBOL if you use the native file-handler with your application. The native file-handler cannot run in all the environments that the JVM application can run in, however; for example, it’s not easy to bundle it into a containerized Java application or run it from a Java Webserver.

For this chapter, we have taken the familiar Credit Service application and changed all the file access code in the ACCOUNT-STORAGE-ACCESS program to Open ESQL code that reads and writes database records. To run the samples in this chapter, you will need to install PostgreSQL on your machine.

PostgreSQL

PostgreSQL is open source and has a license similar to the MIT open-source license. It is a viable alternative to commercially licensed database servers. All examples in this chapter were run using PostgreSQL Version 12, although they will probably work with versions at least as far back as PostgreSQL Version 10. You can install Postgres directly onto your host OS or you can install it as a Docker container.

If you want to install PostgreSQL as a container, you need to install Docker first. You can install Docker on Linux or you can install Docker Desktop on some versions of Windows. For more details, see the section entitled “Installing a Docker Container.”.

Installing on Native OS

Download and installation instructions are at <https://www.postgresql.org/download/>.

For Windows, an installer program will guide you through installation and run the server as a Windows service. It also prompts you to create a password for the root database username of `postgres`.

For Linux, you must add the correct package repository and then install PostgreSQL using your OS package manager. Instructions are included on the PostgreSQL site. After installation, you will have to initialize the database, start the Postgres service, and then set the password for the `postgres` user. On Linux installations, database authentication defaults to the currently logged-in user for client access. Although this is more secure, it is a little more complicated for use on a development system. The examples in this book assume that we will always log in as `postgres` using a password we have set.

To change the authentication method, you need to edit the `pg_hba.conf` file located in the PostgreSQL data directory and change the authentication method from `peer` to `md5`. For more information, go to <https://www.postgresql.org/docs/12/auth-pg-hba-conf.html>.

 **md5 is the easiest method to set up for a development machine but it is no longer considered secure. For a secure non-peer authentication method, use scram-sha-256. You will have to update the password encryption in postgresql.conf. See the PostgreSQL documentation for more information.**

Installing as a Docker Container

Install Docker on your host OS first. For SUSE or Red Hat Enterprise Linux, follow the product documentation for instructions. On Windows, you can install Docker Desktop from <https://www.docker.com>. Documentation is located at <https://docs.docker.com/docker-for-windows/install/>).

You can install Docker on versions of Windows 10 with the Hyper-V role enabled. (Hyper-V is not available for Windows 10 home. However, you can install Docker Desktop on Windows 10 Home using the Windows Subsystem for Linux 2 [WSL 2].) WSL 2 is available on Windows 10 Version 2004 and later. See the Microsoft documentation for more information about installing WSL 2. See the Docker documentation for information about using Docker Desktop with WSL 2.

Once you have installed Docker on your OS, you can install PostgreSQL in a container using the following command (this is a single command running across several lines because the page isn't wide enough):

```
docker run --name postgres-server  
-e POSTGRES_DB=application-db -e POSTGRES_PASSWORD=password  
-p 5432:5432 -d postgres:12.3
```

This is what the command line means:

- run downloads and starts a container process.
- --name postgres-server provides the container with an easily identifiable name.
- -e POSTGRES_DB=application-db sets the environment variable POSTGRES_DB to application-db. PostgreSQL creates an empty database with this name when the container starts.
- -e POSTGRES_PASSWORD=password sets the environment variable POSTGRES_PASSWORD. PostgreSQL sets the password for user postgres to the value of this environment variable.
- -p 5432:5432 maps a network port (left of the colon) on the host OS to a port on the container (right of the colon).
- -d runs the container in detached mode. Leave this switch off and it runs connected to std in and std out in the console where it was started.
- postgres:12.3 is the name and version of the docker image to create the container. By default, the image will be fetched from the Docker hub repository.

The database files are created inside the container file system, so if the container is deleted, the data is lost. This is acceptable for running the example application, but if you want to use a containerized database to store data permanently, you should mount a host volume into the container for storing files. The documentation for this image (https://hub.docker.com/_/postgres/) explains how to do this using the PGDATA environment variable and -v command line switch.

The container gives you a PostgreSQL server, but you need to install the PostgreSQL client tools on your host OS so that you can view and administer the database. When you want to administer the database, log into it as user postgres and use the password you set when you started the docker container. The command line we used to create and start the container makes the database available on port 5432 of the host OS. (This is the default port number for PostgreSQL).

Creating a Database

Once you have installed PostgreSQL, you need to create two databases on your local server; one for testing the application and one for running it. (If you installed PostgreSQL as a container, you already created application-db when you started up the container.) To do this from the command line using the client tools:

```
createdb -U postgres application-db
```

```
createdb -U postgres test-db
```

The createdb command prompts you for the postgres password each time. You now have two empty databases on your local server. The application will create the tables and data.

Remember, you need to install the PostgreSQL tools in your host OS to be able to run these commands, even if you have installed PostgreSQL server using a container.

The OpenESQL Version of Credit Service

Most of the changes made to the application are in the ACCOUNT-STORAGE-ACCESS program (AccountStorageAccess.cbl), which now uses Open ESQL syntax to connect to the database and access the records. The three data files — account.dat, customer.dat and transaction.dat — have been replaced by three tables: account, customer, and transaction.

Each transaction belongs to an account and each account belongs to a customer, so the account id field in transaction is defined as a foreign key pointing to a record in the account table and the customer id in account is defined as a foreign key pointing to a record in the customer table (see Figure 9-4).

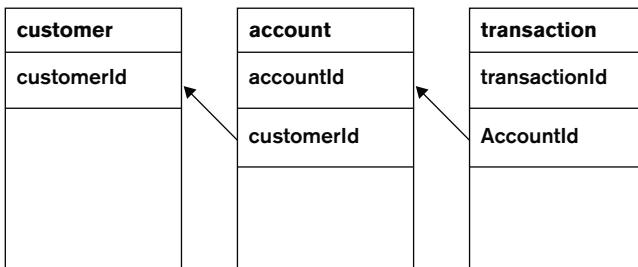


Figure 9-4 Tables with foreign keys

To minimize changes elsewhere in the application, ACCOUNT-STORAGE-ACCESS still has all the same entry-points as before and they all behave in (mostly) the same way as before. A legacy application can often have more than one downstream application that depends on it, in which case you need to keep the new version compatible with the old version.

Figure 9-5 shows what has changed in the new version of the application (refer back to Figure 6-1 to see the original version of the application). The BusinessRules project has been renamed to BusinessRules-oesql to differentiate it from the original. This was for my benefit when working on the different examples rather than for any technical reason.

Grey shading indicates what has changed. ACCOUNT-STORAGE-ACCESS is where nearly all the changes are. Nearly all of the logic in this program that originally read and wrote records in ISAM files has been replaced by SQL statements. As shown in the SQL

Preprocessor in the project properties for BusinessRules-oesql, the OpenESQL processor has been enabled and the driver manager set to JDBC.

The small gray dots on three of the classes in BusinessInterop indicate that some minor changes were made here. This was in places where the behavior of the revised ACCOUNT-STORAGE-ACCESS program didn't exactly match the original. You can diff these files against those from the Chapter 8 examples to see that little has changed. None of the existing code in CreditService itself has changed, but there is a new AdminController class that uses the new DatabaseInitializer module. We'll explain DatabaseInitializer later in this chapter, when we start running the code.

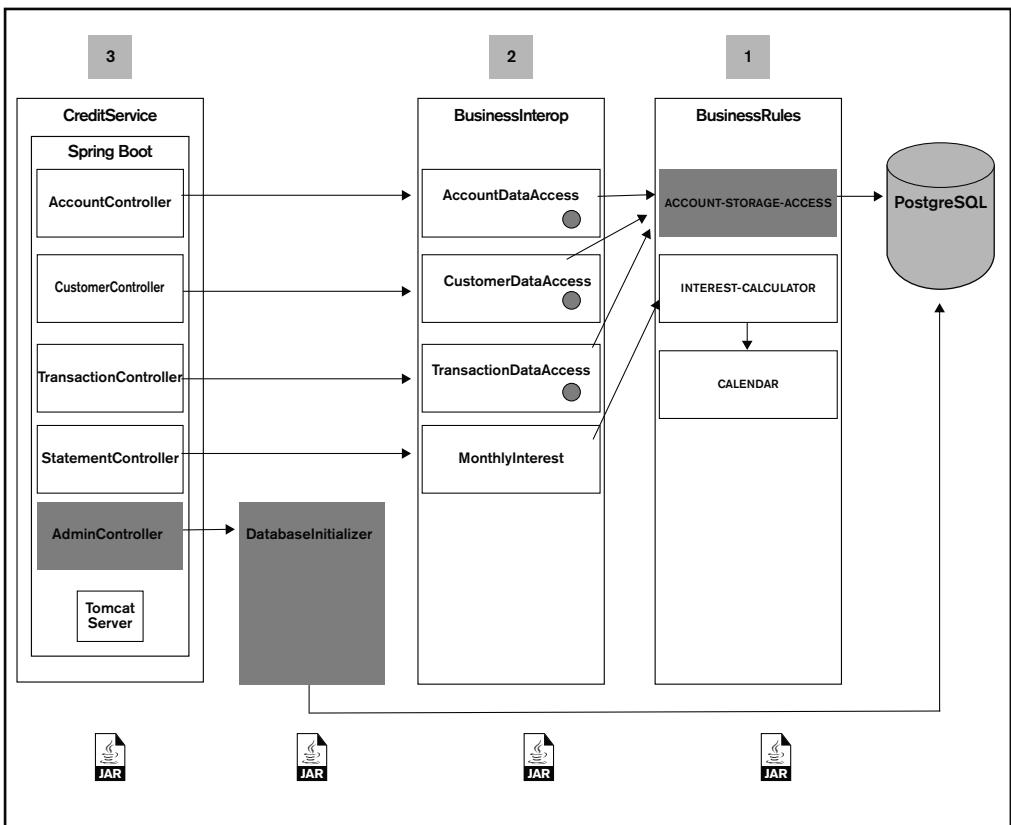


Figure 9-5 Changes to the Credit Service application

It's not always easy to know when you have inadvertently broken something during a refactoring like this. However, for this application, the three test suites we used in Chapter 7 made it much easier to see which parts weren't behaving as expected.

Download the examples for this chapter. There is a complete example and a worked example in this chapter. The worked example has some failing tests that are all fixed in the complete example.

Exporting Existing Data

All the data for our application is currently in COBOL indexed files, but it needs to be migrated to a database for the new application. There is more than one way to do this, but the approach taken here was to export all the data out of the indexed files into Comma Separated Variable (CSV) files. CSV files are plain-text files where each line is a record and the fields in each record are separated by commas. You can open CSV files in spreadsheet programs like Excel or Google Sheets.

To export the data, I wrote a small Visual COBOL application called DataMigrationTool. It's included as part of the Chapter 9 examples. You don't need to run it because the exported CSV files are actually included as part of the examples, but it's included so that you can run it against your own data files if you want to see how it works.

To run the DataMigration application:

1. Import it into an Eclipse workspace that includes the previous version of the BusinessRules project (for example, the one we used in Chapter 8).
2. Create a new COBOL JVM Run configuration for the DataMigration project.
3. Set the **Main Class** as **com.mfcobolbook.datamigration.Main**.
4. On the **Arguments** tab, provide a single **Program Argument** giving the full path to a directory for the output. The CSV files will be written here.
5. Add the dd_accountFile, dd_customerFile and dd_transactionFile environment variables to the **Run Configuration**, pointing towards the data files you want to migrate.
6. Click **Run**.

If everything is set up correctly, files account.csv, customer.csv, and transaction.csv are written to the output directory. This is how the data was extracted from the COBOL files. We'll see how it is used to initialize the database when we start running the tests.

Building the Revised Application

There is an extra step before building and running the Chapter 9 application; you need to download the PostgreSQL JDBC driver before it can connect to the database and add it to the JVM build path for the BusinessRules-oesql and DatabaseInitializer projects. The application will build without the driver jar being present, but it will fail at run-time as soon as the code tries to access the database.

You can download the PostgreSQL JDBC driver from <https://jdbc.postgresql.org>. The examples here were run using Version 42.2.12, but later versions should work provided they are compatible with Java 8.

To add the driver:

1. Import the projects in the worked folder of the Chapter 9 examples into a clean Eclipse workspace.
2. In the **COBOL Explorer**, right-click the **BusinessRules-oesql** project and choose **New Folder**. Create a new folder called **lib**.
3. Copy the downloaded JDBC jar file into the **lib** folder.
4. In the **COBOL Explorer**, right-click the **BusinessRules-oesql** folder and choose **Properties > Micro Focus > JVM Build Path**.
5. Click the **Libraries** tab and then click the **Add JARs button**. Navigate to the **lib** folder, select the PostgreSQL JDBC driver jar, and then click **Apply and Close**.
6. Repeat Steps 3-5 for the **DatabaseInitializer** project.
7. Once you have added the driver, ensure the Maven builders for projects **BusinessRules-oesql**, **BusinessInterop**, and **DatabaseInitializer** (as explained in previous chapters) run without any errors. If you get Maven dependency errors for **CreditService**, ensure that the COBOL RTS is in your local repository (see Chapter 2).

Running the Tests

We have three test suites for the application, in the following projects:

- **BusinessSystemTests** are MFUnit tests that run directly against the legacy COBOL
- **interoptests** are JUnit tests that run against the **BusinessInterop** project
- **CreditService** includes JUnit tests under the project's `src/test/java` folder that test the entire application end-to-end.

Running the **BusinessSystemTests**

We'll run the **BusinessSystemTests** first, as these interact directly with the **BusinessRules-oesql** project:

1. Click **Run > Run Configurations**.
2. On the left, select **COBOL JVM Unit Test** and then click the **New Configuration** button.
3. Name the configuration **BusinessSystemTests** and select **BusinessSystemTests** as the **COBOL JVM Unit Test Project**.

4. Click on the **Environment** tab and then add a new environment variable:

```
DB_CONNECTION_STRING=
Driver=org.postgresql.Driver;URL=jdbc:postgresql:test-
db?user=postgres&password=db-password
```

5. Click **Run**.

The tests should all run, with 11 passes and 8 failures. If there are any problems with the database or test setup, all the tests will fail with the same error. Make sure the password and database user match the password and database user that was set in the connection string in Step 4; also make sure you created the test-db schema as described in the “Creating a Database” section. If you see the error code IM001 against all the tests, you probably haven’t added the PostgreSQL JDBC driver as described in the previous section.

Once you have the expected number of passes and failures, you can develop an understanding of why they failed. All the test failures are in the TestAccountStorage and TestTransactionStorage programs; if you click on any of the failed test cases, the output in the Test Results pane of the Micro Focus Unit Testing tab finishes with a SQL msg error for either the account or transaction table (the following message is truncated; the last word should be “constraint”):

```
insert or update on table "account" violates foreign key constr
```

All BusinessSystemTests always start off with an empty database. In Chapter 7, we described the code that deleted the data files at the start of every test. In the examples for this chapter, the HelperFunctions program has been modified so that every test starts by dropping all the tables and then recreating them with the correct layout.

But now that we are using a relational database, there are constraints on the relationships between the tables. You can’t add an account record unless there is already a customer record that satisfies its foreign key constraint and you can’t add a transaction record without an account record that satisfies this transaction record’s foreign key constraint.

Before these tests will pass, we must update the account and transaction tests to add customer and account records that will ensure the database constraints are satisfied. But first, let’s understand what changed in HelperFunctions in the BusinessSystemTests project.

Listing 9-1 shows the setup-test-data section from HelperFunctions. The previous version of the test code deleted whichever file was pointed to by an environment variable. The new version loads the DatabaseInitializer program and calls the CREATE-TABLES entry point. DatabaseInitializer is a program in the DatabaseInitializer project; the CREATE-TABLES entry point drops the customer, account, and transaction tables and recreates them.

Listing 9-1 The setup-test-data section

```
setup-test-data section.
call DATABASE-INITIALIZER
call CREATE-TABLES
```

```
set succeeded to true
move function-status to lnk-function-status
exit section.
```

In HelperFunctions.cbl the setup-test-data section contains the original source code and the code shown in Listing 9-1 but uses conditional compilation so that the new one is active. In the Micro Focus Build Configuration in the project properties for Business-SystemTests, under Additional directives, you will see the constant setting OESQL-TEST (1). To return to the original version of the code that worked with ISAM files, change the constant from 1 to 0.

To simplify updating the code so the tests pass, the code to add additional customer and account records is already in the TestAccountStorage.cbl and TestTransactionStorage.cbl files but is omitted through conditional compilation. Listing 9-2 shows the setup-account-test code in TestAccountStorage.cbl.

Listing 9-2 *The setup-account-test section*

```
setup-account-test section.
  call HELPER-FUNCTIONS
  call INIT-ACCOUNT-TEST using by reference function-status
$if OESQL-TEST = 11
  perform add-fk-customer
$end
  goback.
```

The new code (perform add-fk-customer) is inside a conditional compilation block. Change the value in the test statement from 11 to 1 to include the code. It shows as an error because add-fk-customer isn't yet included in the compilation.

Search for another occurrence of 11 to find the code in Listing 9-3.

Listing 9-3 *The add-fk-customer section*

```
$if OESQL-TEST=11
*> A customer record is needed as the foreign key to an account
add-fk-customer section.
  move FK-CUSTOMER-ID to WS-CUSTOMER-ID of WS-CUSTOMER-RECORD
  move FK-CUSTOMER-FIRST-NAME to WS-FIRST-NAME
  move FK-CUSTOMER-LAST-NAME to WS-LAST-NAME
  call ADD-CUSTOMER using by reference function-status
                                         WS-CUSTOMER-RECORD
  if failed perform test-failed end-if
  exit section.
$end
```

This code creates a customer-record and adds it to the database by calling ADD-CUSTOMER. If any account record uses FK-CUSTOMER-ID as the Customer ID of any record added, the foreign key constraint is satisfied. Change the value in the test statement from 11 to 1 and run the tests again. This time, the tests in the TestAccountStorage suite pass.

Repeat the same procedure with TestTransactionStorage.cbl; find the conditional compilation blocks where OESQL-TEST=11 and change them to OESQL-TEST=1. Rerun BusinessSystemTests; all the tests now pass.

Behavior Changes in Updated Code

Our intention in updating BusinessRules was to keep the same behavior as before in order to maintain compatibility with any clients of the code. But despite that, we've had to modify the tests before they will pass. Similar issues can arise whenever you replace one technology with another. We could have sidestepped the issue by defining the account and transaction tables to not have foreign key dependencies. This would have kept the behavior closer to the original, but that is not the expected behavior for tables in a relational database and would mean losing one of the advantages of relational data: maintaining internal integrity. In this case, it is probably better to live with a small change in behavior than to have a database with an "unusual" design. Whenever you make changes like this, communicate the change, and the reasons for that change, to any other teams that rely on the code.

Running the Interoperation Tests

We'll run the interoptests next:

1. Click **Run > Run Configurations**.
2. On the left, click **JUnit** then click the **New Configuration** button.
3. Name the configuration **interoptests** and then select **interoptests** as the project
4. Set the **Test runner** to **JUnit 4**.
5. Click on the **Environment** tab and add a new environment variable:

```
DB_CONNECTION_STRING=
Driver=org.postgresql.Driver;URL=jdbc:postgresql:test-
db?user=postgres&password=db-password
```

(It's easiest to copy the value from the BusinessSystemTests Run configuration since you've already got that working).

6. Click Run.

This time there are two errors and two failures out of 28 tests. And as before, they are caused by attempted violations of foreign key constraints. Before fixing it, let's see what has changed in InteropTest.java in this version. Listing 9-4 shows almost all the changed code in this class.

Listing 9-4 *InteropTest.java initialization code*

```
public class InteropTest {
    private static BigDecimal DAILY_RATE = new BigDecimal(0.1)
        .divide(new BigDecimal(365), 10, RoundingMode.HALF_UP);

    private DatabaseInitializerWrapper databaseInitializerWrapper;

    @Before
    public void initTestData() throws IOException {
        databaseInitializerWrapper =
            new DatabaseInitializerWrapper();
        createEmptyTables();
        populateTables();
    }

    private void populateTables() throws IOException {
        String csvFile = get CsvPath("customer.csv");
        databaseInitializerWrapper.loadCustomerData(csvFile);
        csvFile = get CsvPath("account.csv");
        databaseInitializerWrapper.loadAccountData(csvFile);
        csvFile = get CsvPath("transaction.csv");
        databaseInitializerWrapper.loadTransactionData(csvFile);
    }

    private void createEmptyTables() {
        databaseInitializerWrapper.dropAndCreateTables();
    }

    private String get CsvPath(String source) throws IOException {
        URL url = InteropTest.class.getClassLoader()

            .getResource(source);
        String path = url.getPath();
        assertNotNull(path);
        assertTrue(path.length() > 0);
        File f = new File(path);
        return f.getAbsolutePath();
    }
}
```

Java lacks conditional compilation, so you can't see it directly against the original code. To compare it to the previous version, use a diff tool to compare it to the version of JavalInteropTest in the Chapter 7 examples.

These tests run against a small set of test data that is initialized before each test case is run (unlike the previous tests, which always run against an empty database). The previous version of the tests simply copied a new set of indexed files from a predefined test set. That approach is not so easy with PostgreSQL, especially because the test runner might not have permissions to rewrite the database files in the file system. So instead, the initialization code makes use of the DatabaseInitializer project to drop the tables from the test database, recreate them, and then initialize them with data stored in CSV files. The CSV files are in the project's resources folder so that they can be located from the classpath.

The COBOL program in the previous example called the DatabaseInitializer program directly. But for the benefit of Java, there's also a DatabaseInitializerWrapper class, which the code here uses.

Three of the tests in InteropTest.java start with no records at all; they started by invoking copyDataSource() to copy in empty data tables. These tests now call createEmptyTables(), which drops and recreates the tables in the test database. Those are the only other changes to the compiled code.

As before, we alter the tests so that they no longer violate foreign key constraints. This means updating the initialization data as well as some of the test code, adding one extra customer record and one extra account record. These extra records don't have dependent account or transaction records, so they can be used safely in the deleteCustomer and deleteAccount tests (both currently failing).

To change the tests so that they all pass:

1. In **worked/interoptests/src/main/resources/**, add the new records to the end of the **customer.csv** and **account.csv** files.

The easiest way to do this is to copy the **customer.csv** and **account.csv** files from **complete/interoptests/src/main/resources/**.

2. Open **InteropTest.java**. All the changes needed to fix the tests are in the worked version of this file but commented out. Search for the TODO comments in this file and follow the instructions in each one.
3. Save the changes and rerun the tests. They should now all pass.

Running the CreditService Tests

To run the CreditService tests:

1. Click **Run > Run Configurations**.
2. On the left, select **JUnit** and then click the **New Configuration** button.

3. Name the configuration **CreditService** and then select **CreditService** as the project.

4. Set the **Test runner** to **JUnit 4**.

5. Click on the **Environment** tab and add a new environment variable:

```
DB_CONNECTION_STRING=
Driver=org.postgresql.Driver;URL=jdbc:postgresql:test-
db?user=postgres&password=db-password
```

(It's easiest to copy the value from the **BusinessSystemTests** Run configuration since you've already got that working).

6. Click **Run**.

There are six failures out of 23 tests, and as in the previous cases, they are all foreign key errors. Like the interoptests, the only updates at this point are to the initialization, which works the same way as it does for interoptests; the tables are dropped from the database and reinitialized with the values in CSV files in the src/test/resources folder.

To change the tests so that they all pass:

1. In **worked/CreditService/test/main/resources/**, add the new records to the end of the **customer.csv** and **account.csv** files.

The easiest way to do this is to copy the **customer.csv** and **account.csv** files from **complete/ CreditService/test/main/resources/**.

2. There are failing tests in three files: `WebServiceAccountTests.java`, `WebserviceCustomerTests.java` and `WebserviceTransactionTests.java`. All the changes needed to fix the tests are in the worked versions of these files but commented out. Search for the `TODO` comments in this file and follow the instructions in each one.

3. Save the changes and rerun the tests. They should now all pass.

Now that all the tests for the application pass, we can be reasonably confident that it now functions the same as it did before we started changing the backend.

Running the Revised CreditService Application

Now that all the tests pass in our refactored application, we can run the application itself. But first we are going to populate the application database (`application_db`) with customer and account data migrated from the original application.

Administrative Tasks

Populating an application database before it is run the first time is an administrative task. You could ask a Database Administrator to do this for you, but one of the 12 factors for cloud-native applications (<https://12factor.net>) is “Run admin/management tasks as one-off processes.”

So a task like the initialization of a database schema should be an automated, repeatable task that ships along with the application code and can be run in the same way whether the application is in development, test, staging, or production. There are several ways you could approach this. You could include one or more admin programs that ship with your application to carry out these tasks. In the case of the example, we've simply added new endpoints to the application.

This makes it simpler to demonstrate the application as we move it to different environments but opens up a security vulnerability for a real production application, as anyone accessing those endpoints could delete or reinitialize the database.

A real application would enforce authentication and authorization to mitigate the risk of such attacks, but an even better practice is to separate administrative functions either to a separate application with more restricted access or applications intended to be run separately as one-off jobs. However, the principle holds that the programs carrying out the admin should be part of the deliverables for the application, and deployable in the same environment.

In the CreditService project, open the `com.mfcobolbook.creditservice.webservice` package to see the new `AdminController` class. It has one endpoint (`/admin/initialize-db`), which drops and recreates the tables and then loads data from CSV files in `/src/main/resources`. Storing initialization data along with the application is something you would not be likely to do with a real application. For one thing, the datasets for production will not be the same as for other environments, and shipping a different deployable for each environment violates cloud-native principles.

One way to make data available for initializing applications in real cloud environments is to put it into a bulk-storage medium. Amazon Web Services (AWS) provides S3, a service for storing large amounts of data cheaply in “buckets.” The S3 API has been implemented by several vendors for their own storage solutions so that the same code and libraries can be used regardless of whether your application will be deployed to AWS.

Starting the Application

To start the application:

1. Click **Run > Run Configurations**.
2. On the left, select **Java Application** and then click the **New Configuration** button.

3. Name the configuration **CreditService**, select **CreditService** as the Java project, and name **com.mfcobolbook.creditservice.webservice.WebServiceApplication** as the Main class.
4. Click on the **Environment** tab and then add a new environment variable:
DB_CONNECTION_STRING=
Driver=org.postgresql.Driver;URL=jdbc:postgresql:application-db?user=postgres&password=db-password
This assumes that you created application-db when you created **test-db**, in section Creating a Database earlier in this chapter.
5. Click **Run**.
6. Go to <http://localhost:8080/admin/initialize-db>. If everything is set up correctly, the browser displays the message **All tables initialized**. You need to do this only the first time you run the application against a new database.
7. Retrieve a customer: <http://localhost:8080/service/customer/1>
8. Carry out an interest calculation: <http://localhost:8080/service/account/25/state>ment/20190701?rate=25&initialBalance=300

At this point, we have an application we can easily containerize. It is no longer dependent on the local file system for storage. The database configuration is set by an environment variable so that we can deploy exactly the same code in every environment rather than editing configuration files. This satisfies the first of the 12 factors: “One codebase tracked in revision control, many deploys.”

Containerizing the CreditService

In this section, we will use Docker to create a container for our application. The container image contains all the dependencies needed to run the application, including a Java runtime. Figure 9-6 shows the application and database containers, both of which are inside a Docker network.

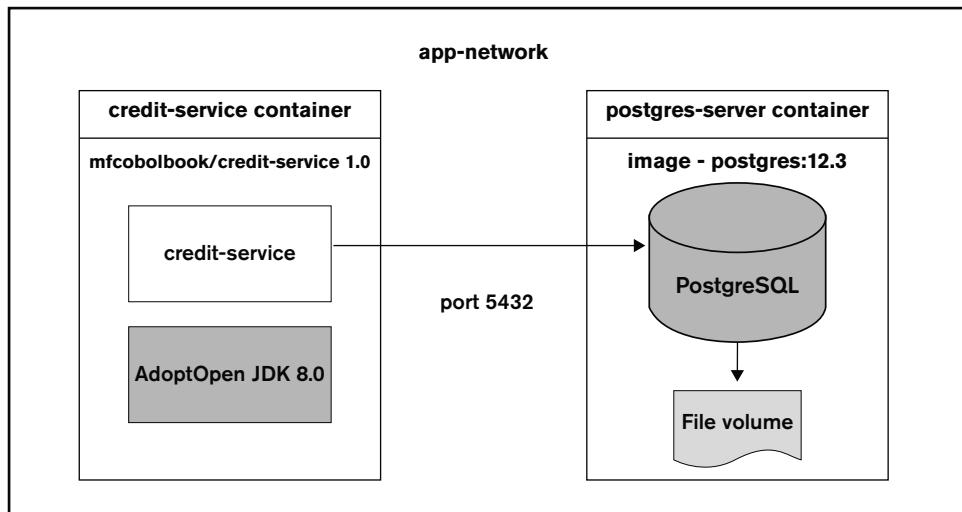


Figure 9-6 The containerized application

The Docker network is a virtual network created on the host that allows separate containers to communicate. By default, each container you start up on the host OS has no ports open to other processes running on the host OS. The `-p` switch that you can use with the `docker run` command enables you to map container ports to ports on the OS, making them visible in the same way as ports opened by applications running directly on the host OS.

However, containers can't see ports open on the host OS without being made part of a network that can also see those ports. The easiest way for an application made from separate running containers to communicate is to create a Docker network and specify it as the container network when a container is started. Docker provides a DNS service that enables containers on the same network to find each other by container name.

In the next sections we will:

1. Create a container image for the CreditService application.
2. Create the network and start the application running.

Building a Container Image

A container image is a standalone package that includes everything needed to run the packaged application: the executable code for the application, dependent libraries, system tools, and configuration. A container image can be run anywhere that hosts a suitable container run-time. We will use Docker because it is the best known and most widely available container run-time.

In Figure 9-6, our credit-service container contains both the CreditService application and a Java run-time. Because CreditService is a Spring Boot application, we can package

it as a single jar file that includes all the other Java libraries it depends on (this is known as an “uber-jar” or “fat jar”). Until now, we’ve always run the application through Eclipse, but now we will build the jar file and run that before packaging it up.

To build CreditService:

1. Start a command prompt and navigate to the Chapter 9 worked/CreditService directory.
2. Set a DB_CONNECTION_STRING environment variable to point to the test database on your PostgreSQL server. You can copy the setting from your Eclipse CreditServiceTests Run Configuration (see the “Running the CreditService Tests” section in this chapter).
3. Enter the command `mvn clean package`

Maven runs against the project’s pom.xml, compiles the application, and then runs all the project tests. If the tests all pass, it creates a jar in the target directory. This is why we set the DB_CONNECTION_STRING environment variable first; the tests will all fail without it.

You can skip running the tests by including `-Dmaven.test.skip=true` as part of the Maven command line, but it’s good practice to run tests as part of a build.

If the build ran successfully, you have a CreditService-0.0.1-SNAPSHOT.jar in the CreditService/target directory. You can run the application with command `java -jar CreditService-0.0.1-SNAPSHOT.jar`. (You will be connected to the test database unless you change the DB_CONNECTION_STRING environment variable first.)

Now we can package the jar file into a container by creating a Dockerfile, which is a text file that tells Docker how to build the image. Explaining the syntax and commands for a Dockerfile is beyond the scope of this book. There are plenty of online resources you can refer to, including the Docker documentation itself. The CreditService application is relatively simple and we’ve already built the jar file that includes all of its Java dependencies. To create the Dockerfile, we will use the example Spring Boot Dockerfile from the Spring Getting Started with Docker documentation, which you can find by searching the Web for “spring boot docker image”.

In the CreditService directory, create a text file called Dockerfile (no extension) and then copy in the text from Listing 9-5.

Listing 9-5 A Dockerfile

```
FROM openjdk:8-jdk-alpine
RUN addgroup -S spring && adduser -S spring -G spring
USER spring:spring
ARG JAR_FILE=target/*.jar
COPY ${JAR_FILE} app.jar
ENTRYPOINT ["java","-jar","/app.jar"]
```

This builds a docker image using the base image openjdk:8-jdk-alpine, which is an Alpine Linux base image that includes an OpenJDK Java. It is stored in a public Docker image repository.

The other instructions in the Dockerfile add the Spring user — a user the application will run as — to the container and copy our application jar file into the container as app.jar. The ENTRYPOINT statement provides the command that will be run when the container is started.

To create the image:

```
docker build . -t mfccobolbook/credit-service:1.0
```

This builds the image and tags it as mfccobolbook/credit-service with version 1.0. The image file doesn't appear in the CreditService directory; it is stored directly in the image repository on your local machine. To make this image available to other machines, you have to publish it to a public or private repository using the docker push command.

Docker repositories are analogous to the Maven repositories discussed in Chapter 2, but instead of containing Java jar files, they contain container image files. Like Maven, Docker searches your local repository first and then searches the other repositories it has been configured with. When you ran the docker build command, it displayed messages showing the repositories scanned in order to find the openjdk:8-jdk-alpine image used to build the application image.

In the next section, we will run the containerized application.

Running the Application

At this point, we have a container for our application. We are going to create the Docker network and put our database and application into the network. For the application to be usable, we will also have to expose its HTTP port outside the Docker network. We'll also expose the PostgreSQL default port outside the network so that it is still possible to administer it with tools running on the host OS. Figure 9-7 shows the containerized application, the Docker created virtual network, and the two ports that will appear as localhost ports on the Host OS.

A host OS only has one set of ports available; only one application can listen on port 8080 and only one application can listen on port 5432. Each container running inside a Docker virtual network has its own set of ports, but to expose those ports outside the virtual network, they must be mapped to ports on the host OS; it follows that only one container can be mapped to a given port on the host.

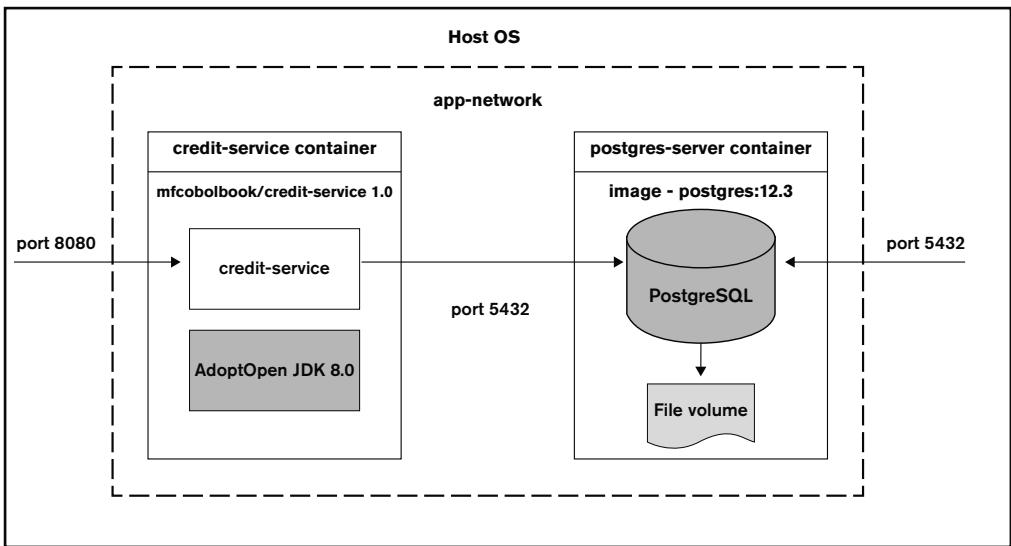


Figure 9-7 Application with network ports

Port mappings are defined when a container is started up the first time. A running container can be stopped and restarted, but it retains the same port mappings as when it was created.

Figure 9-8 shows the basic lifecycle of a container. A container is created from an image and starts running in response to the `docker run` command. When a container is no longer required, it is deleted. You can start several containers from the same image; each container has its own separate lifecycle.

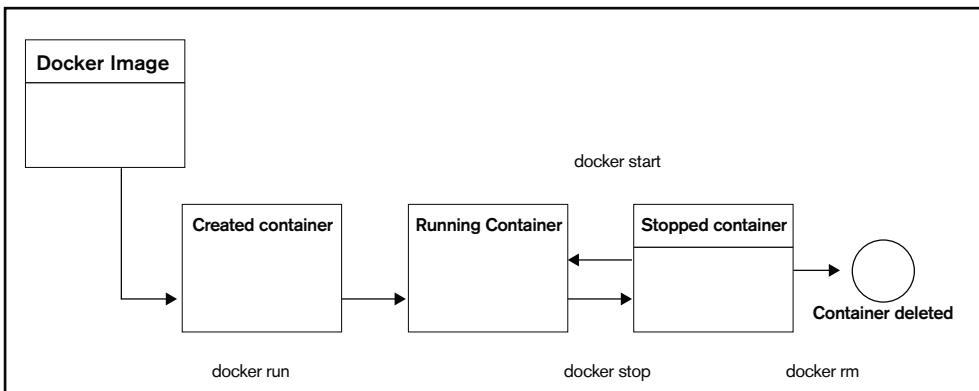


Figure 9-8 Container lifecycle

To create the network:

```
docker network create app-network
```

To remove the current database container and create a new one:

1. Run command `docker stop postgres`.

2. Run command `docker rm postgres`.

This removes the container. Perform Step 1 and Step 2 only if you created a `postgres` container previously. For more information, see the “Install as a Docker Container” section).

3. Run command `docker run --name postgres --network app-network`

```
-e POSTGRES_DB=application-db
```

```
-e POSTGRES_PASSWORD=password
```

```
-p 5433:5432 -d postgres:12.3
```

The only difference between this command and the one we used previously in the chapter is that we have specified that it should run inside `app-network`.

Finally, to start the application running:

```
docker run -p 8080:8080 --name credit-service-app --rm  
--network app-network  
-e DB_CONNECTION_STRING='Driver=org.postgresql.  
Driver;URL=jdbc:postgresql://postgres/application-db?user=postgres&password  
='password  
mfcobolbook/credit-service:1.0
```

There are some differences between the way we have started the application and database containers.

- It has been started without the `-d` flag, so it isn’t running in detached mode.
- It has been started with the `--rm` flag, so if you stop the container, it will be automatically deleted.

Not running in detached mode leaves the container attached to the console’s `stdout` and you will see all the Spring Boot startup messages. If there are any problems starting it up (like connecting to the database), the error messages will make it easier to debug. Once it’s working, you can stop the container and then use `docker run` with the `-d` switch to restart it in detached mode.

It’s very common to use `--rm` with application containers when there is no state that needs to be preserved. All our application’s state is in the attached database, and if we need to stop the container, there’s no need to clean it up by deleting it later. Creating a new container from the image is not much more expensive than restarting a stopped one.

Go to `http://localhost:8080` and you should see the “Hello World” message that indicates our Spring Boot application has started and is responding to requests. Try

initializing the database by going to <http://localhost:8080/admin/initialize-db>. If this completes successfully, the application is successfully running. You can use any of the other application REST calls to verify functionality.

At this point, we have a fully containerized application. Once it has been pushed to a repository, it can be run on any OS with a docker run-time that can pull the image from the repository. The only configuration it needs is a database connection string; that is read in from the environment, so exactly the same code and image can be used whether we are running the application in testing, staging, or production.

It's also possible to scale the application by starting multiple instances and putting them behind a load balancer. For example, the monthly interest-calculation is comparatively compute-intensive; you could use multiple instances of the application to run calculations in parallel when it's time to send monthly statements to millions of customers.

The docker-compose Command

The docker-compose command enables you to configure and start multiple containers with a single command. You create a `docker-compose.yml` file with the configuration for an application and then use `docker-compose up` to start the containers and use `docker-compose down` to dispose of them. When you have a multi-container application you want to distribute to end-users, docker-compose is a good way to hide the complexity of configuring and starting the application. However, it isn't as effective as Kubernetes for managing production-grade cloud-native applications. You will learn about Kubernetes in Chapter 10.

Building Scripts

Building the container and then deploying the application required several commands. Automating the process enables you to deliver more often and reduces the likelihood of errors. The CreditService application is very simple to build and containerize, but most real applications have more complex builds. Tests should be run as part of the build process and builds failed if tests fail.

Creating a Continuous Integration/Continuous Delivery (CI/CD) pipeline means that every change committed to your source code repository results in the application being built, tested, and deployed – even if the automated deployment is only to a test environment. The more mature your automation and testing process, the closer you should get towards full automation from commit through to production.

At time of writing, Jenkins is probably the most popular tool used for building CI/CD pipelines. Creating a CI/CD pipeline is not something that we are going to cover here, but

in the complete/CreditService folder of the Chapter 9 examples, you will find two shell scripts that carry out the build and deployment for CreditService. These could be used as part of a build pipeline.

Listing 9-6 shows a shell script that builds the CreditService jar and then builds the container. The statement that sets DB_CONNECTION_STRING has been split across several lines because of the limited page width but is only one line in the actual shell script file.

Listing 9-6 Simple build script

```
#!/bin/bash
if [ ! -z $DB_TEST_PASSWORD ]
then
    DB_CONNECTION_STRING=
        'Driver=org.postgresql.Driver;
        URL=jdbc:postgresql:test-db?
        user=postgres&password='$DB_TEST_PASSWORD
    echo $DB_CONNECTION_STRING
    if mvn package
    then
        docker build . -t mfcobolbook/credit-service:1.0
    else
        echo *** BUILD FAILED ***
        exit 1
    fi
else
    echo Set DB_TEST_PASSWORD TO value for test database.
fi
```

Because Maven runs the CreditService tests before creating the package, the script sets up a DB_CONNECTION_STRING environment variable. The script expects the database password to be set already (in DB_TEST_PASSWORD) and fails when this value is not already set. As a general principle, don't set secrets like passwords into scripts that will go into source control. Your CI/CD tool should provide a secure way for you to pass secrets to your build and test scripts.

The script runs the mvn package that will build CreditService and run its tests. If either the build or the test fails, the non-zero exit code will fail the build and the container won't get built. If mvn package runs successfully, the script builds the container image. If the docker build step fails, it exits with a non-zero code; the CI/CD system running this will be able to use a non-zero code to know that the build has failed and not to move to the next step in the pipeline.

If this script is embedded in a CI/CD pipeline, the next step might be to push the container image to a registry, deploy it to a test environment, and then run a larger test-suite.

Listing 9-7 shows an even simpler script. It has the same three docker commands we used to set up and run the application earlier, although as in the previous script, the database password has been externalized into another environment variable.

Listing 9-7 Deploy and run the application

```
#!/bin/bash
docker network create app-network
docker run --name postgres-server --network app-network
  -e POSTGRES_DB=application-db
  -e POSTGRES_PASSWORD=$DB_TEST_PASSWORD
  -p 5433:5432 -d postgres:12.3
docker run -p 8080:8080 --name credit-service --rm
  --network app-network
  -e DB_CONNECTION_STRING='Driver=org.postgresql.Driver;
    URL=jdbc:postgresql://postgres-server/application-db?
      user=postgres&password='$DB_TEST_PASSWORD
      mfcobolbook/credit-service:1.0
```

This script assumes that neither the docker network nor the containers are already created on the host; if you use this as part of an automated test environment deploy, you need a tear-down stage to delete these entities if already present.

Summary

This chapter covered several topics. We refactored the CreditService application to be much closer to “cloud-native” using the existing test-suites to ensure that it still behaves as expected. And then we ran it using Docker containers. Containers are an important building block for building cloud-native applications and applications built from micro-services, but they are only part of what you need to build and run a real application.

Even in its simplest form, our application requires its own network and separate containers for the application and the database. If we started scaling it up, the complexity would increase quickly. In the next chapter, you will learn about cloud deployment in more detail, including how to use Kubernetes for deploying and orchestrating containers. You will also learn about “serverless computing.”



COBOL and Microservices

In the previous chapter, we containerized the CreditService application and ran it using Docker. Containerizing the application made it easier to manage and deploy on different platforms. In this chapter, we'll take a look at two platforms you can use to deploy microservices:

- Kubernetes
- Serverless Computing

This is not an exhaustive list of platforms; the goal is to give you an idea of the possibilities. Although Kubernetes itself is an open-source project, there are several vendors who sell and support their own flavor of the platform. The core operating principles and APIs are the same in all of these; what differs are the add-ons that help with deploying, monitoring, and managing the platform.

Serverless computing is a particular class of PaaS; one where you deploy individual “functions” to the platform. When the functions aren’t in use, they aren’t running and need no compute or memory resources. The only resource needed is the storage for the container image. In this chapter, we will use AWS Lambda as our serverless platform, but there are others to choose from.

Why Do I Need a Platform?

If all the material presented in the last chapter is new to you, you might be feeling that life was simpler in the old days. You had an application, you commissioned a server, and you installed your application.

Important applications warranted big servers, with RAID arrays, UPS, and lots of engineering to make sure they didn't go down unexpectedly. Really important applications got a second server for failover, possibly located in a different data center.

There are several factors driving the move towards clouds. One of the biggest is the drive for rapid innovation. Deploying apps directly on servers is slow and expensive. It can take months to commission a server and get it installed before you can put your application on there (most of this time is administrative, but it still all has to be done). Even though servers are usually VMs rather than physical hardware, in many organizations the process can still be very slow, with separate requests for network and connectivity. Development teams should be able to concentrate on applications and business value, not infrastructure and plumbing.

With a capable cloud platform, whether private or public, a team can go from having an idea to deploying an initial version of the application in days. If they've already written the code, they could be up and running with something accessible to end-users in a few hours.

In the last chapter, we used Docker to containerize and deploy the CreditService application. However, we were deploying only two containers. For a robust and scalable microservice deployment, we want a minimum of two instances of the application running. Figure 10-1 shows a more realistic deployment. We've now added two more application containers, a load balancer, and we also need to open up the application's subnet so that the application can be accessed from outside. We might also want to have some kind of autoscaling set up so that the number of credit-service containers can be increased or decreased according to load.

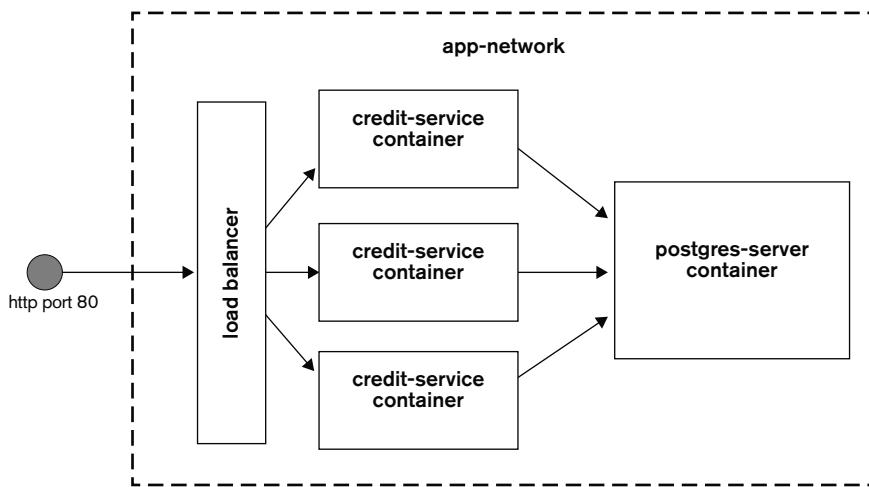


Figure 10-1 A scaled version of the CreditService application

Now imagine that your entire application consists of perhaps six or seven such microservices, which now also need mechanisms for internal discovery, for services, and for communication – suddenly deployment and management is a lot more complex.

What happens when you want to update the version of the deployed application? How do you monitor its health, examine application logs, and so on? These aspects of operating a production application are sometimes called “day two” issues; a capable platform will help you solve them.

In this chapter, you will learn about two alternative platforms for deploying our application:

- Kubernetes
- Serverless computing

This should start you thinking about the opportunities and problems that moving your applications from monolith to microservices are likely to bring. You won’t be a devops expert after reading this chapter, but you will have a better understanding of the technical issues.

Kubernetes

Kubernetes (often abbreviated to K8S) is an open-source container orchestration project for automating deployment and management of applications. Kubernetes was initially designed by Google, but is now maintained by the Cloud Native Computing Foundation (CNCF). A Kubernetes cluster consists of one or more nodes, each of which can run several pods.

The Kubernetes documentation defines a pod as the “smallest deployable unit of computing.” The pods in our example have only a single container, but pods can be defined with multiple containers. This is to support use cases in which two different processes work closely together. For example, you might deploy a proxy container together with your application container for traffic management and metrics collection.

Kubernetes also provides abstractions to help you manage applications. We’ll look at a few of these in this section, but Kubernetes provides a much larger set than we are going to cover here. Kubernetes provides a lot of flexibility and power, but it also means that you have to do a lot of the work in building a robust and scalable production platform that fits your requirements.

A lot of vendors also provide Kubernetes implementations that provide extra management and monitoring layers on top of the basic Kubernetes experience to make it easier to build and run a stable production platform.

Getting Kubernetes

Installing a version of Kubernetes suitable for developer use on your local machine is very straightforward using Minikube (which is available at <https://kubernetes.io>). Minikube uses virtualization to install a single node cluster on your machine.

You will need a hypervisor installed before you can use Minikube. If you are on Windows, you can use VirtualBox (which is available free at <https://www.virtualbox.org>) or Hyper-V.

VirtualBox is also available for Linux or you can install Linux Kernel Virtual Machine (KVM), which is available at <https://www.linux-kvm.org>.

Once you have installed a hypervisor, follow the instructions at <https://kubernetes.io/docs/tasks/tools/install-minikube/> (or just search the web for “install minikube”) to install Minikube on your local machine.

You will also need to install the Kubernetes Command Line Interface (CLI). This is also available at <https://kubernetes.io>. Modern cloud platforms put a heavy emphasis on being able to operate everything from the command-line tool. GUIs are provided mainly for monitoring and observation. This is because a command line makes it much easier to automate operations than a GUI.

Once you have installed Minikube and the Kubernetes CLI, follow the Minikube documentation to start up a single node cluster:

```
minikube start --driver=driver-name
```

For example, if you are using VirtualBox:

```
minikube start --driver=virtualbox
```

Or if you are running Docker for Windows and therefore using Hyper-V as your hypervisor:

```
minikube start --driver=hyperv
```

See the Minikube documentation for more details. At the time of this writing, creating multi-node clusters is a beta feature of Minikube, but we need only one node to work through the examples.

After the cluster is started, try the command:

```
kubectl get nodes
```

This should display output similar to:

NAME	STATUS	ROLES	AGE	VERSION
minikube	Ready	master	1m	v1.18.3

In practice, Kubernetes on its own does not provide a full PaaS experience, which is why there are so many vendors offering you their own Kubernetes distribution, built on top of open-source Kubernetes. Setting up and running a production-ready Kubernetes environment on either bare metal or VMs is a lot more complicated than installing Minikube on a laptop. However, from the developer’s point of view, the way you deploy applications doesn’t differ much between Minikube and any other Kubernetes distribution.

Running CreditService as a Kubernetes Application

Kubernetes defines several types of entities that act as the building blocks to deploy containerized cloud applications. These entities enable you to manage containers, scaling, networks, rollouts, and so on. Figure 10-2 shows the application running in Kubernetes and the Kubernetes entities used to create it.

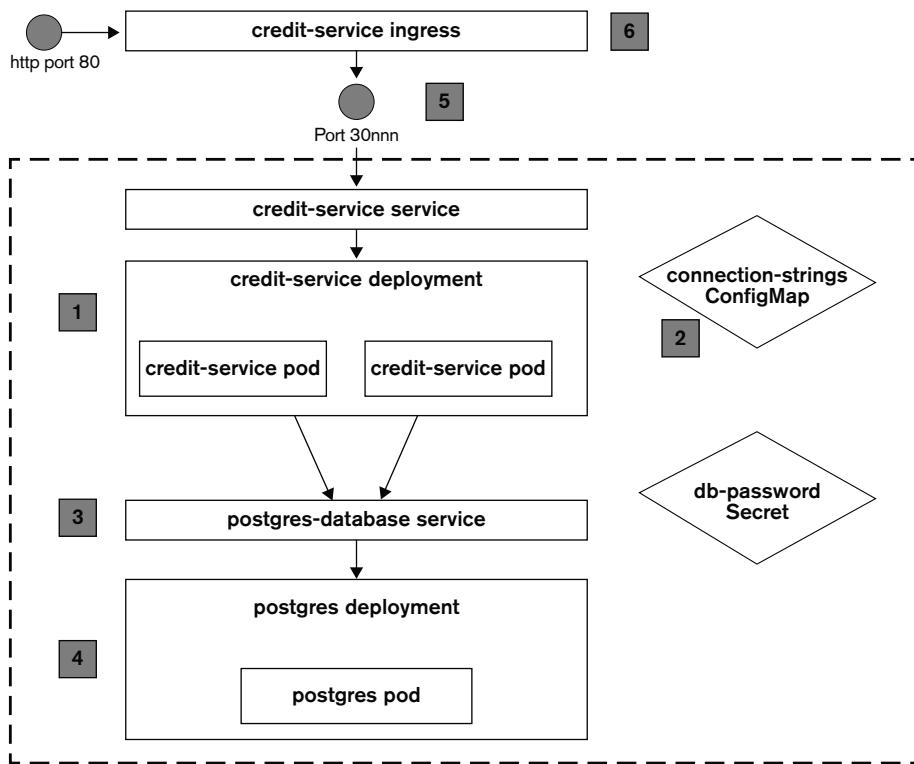


Figure 10-2 CreditService Kubernetes diagram

These entities are:

- **Pods.** A pod is the basic execution unit of your application. It defines a container together with some metadata. You never run a container directly in Kubernetes; you always run a pod. Most pods have a single container, but you can define multi-container pods where two (or more) separate containers must work closely together.
- **Deployments.** A Deployment is a declarative description of a ReplicaSet. A ReplicaSet is a group of identical pods. If we want to run five instances of our CreditService container, we can do it by creating a deployment that says we want 5 CreditService pods.
- **Services.** A service is an abstraction that provides access to a group of pods. Individual pods can come and go at any time. You might scale the number of pods

in a Deployment up or down according to demand or a pod might crash and be replaced by a new pod. Once you connect a Service to a Deployment, traffic sent to the service is sent to the deployed pods in round-robin fashion

- **ConfigMaps.** A ConfigMap enables you to provide a set of key-value pairs to a pod. You can use ConfigMaps in several ways, but we will use them to set environment variables in our container.
- **Secrets.** A Secret is like a ConfigMap, but the values are not shown in plain text when queried through Kubernetes tools. Secrets are useful to prevent inadvertent leaking of passwords or other credentials but they are not actually secure, as the information is simply encoded as a base64 string. To store secrets safely, you need to use secure encryption; there are a few products available to help with this. Our example uses a Kubernetes Secret to store the database password.
- **Ingress.** The ingress makes our application available outside the Kubernetes cluster on port 80 or port 443 for https. Our example application runs on http so that we don't need to generate and deploy certificates, but all production applications should use https.

You might feel that we have suddenly introduced a lot of new concepts to run a relatively simple application. These are all necessary to run any application in production. In the past they might have been dealt with by a separate IT or Operations team, but cloud technology is driving the move to DevOps – where developers get more closely involved with some of the processes necessary to make an application run in production.

Now developers run and deploy the same application in development and test environments that is deployed into production. This drives beneficial outcomes, which include:

- The application deployment that developers work on is much more closely aligned to the one in production.
- Development and Operations work closer together, which reduces silos of knowledge and time-consuming hand-offs between teams.
- It incentivizes building automated Continuous Integration/Continuous Deployment (CI/CD) pipelines that can be used for development, test, and production environments

Application Changes

The only change we have made to the CreditService application for Kubernetes is in the way we get the connection string for the database. Until now, we've been passing it the database connection string using the value of a single environment variable. From now on, we will construct the connection string inside the application from four separate variables:

- POSTGRES_HOST
- POSTGRES_DB
- POSTGRES_USER
- POSTGRES_PASSWORD

This makes it easier to keep the password secret and means we can share configuration between the database and the application.

Download the Chapter 10 example files. There are two subdirectories:

- Kubernetes
- Serverless

Import the projects under the Kubernetes directory into Eclipse if you want to look at the changes. However, you won't actually need to build the application to run this example – the container image is already in the Docker Hub registry; when you build the application it is downloaded from here.

We've added a static method, `setConnectionString()` to the `DatabaseInitializerWrapper` class, and it is called from the `open()` method in the `AbstractBusinessAccess` class. This method is called each time the Java code client code uses the COBOL application (`AbstractBusinessAccess` is part of the interoperation layer).

The `setConnectionString()` itself constructs a connection string from the environment variables listed above and then calls a new entry point in `ACCOUNT-STORAGE-ACCESS` to set the connection string for this program. Everything else is the same as it was in the previous chapter.

In the next section, we will create and run the application using Kubernetes.

Defining the Kubernetes Application

All the files to create the application are in the Chapter 10 example files under the `kubernetes/configuration` folder. Kubernetes works declaratively; you provide Kubernetes with a description of the desired state of your system and it does its best to achieve and maintain that state.

For example, the credit-service deployment creates the pods that host the `CreditService` container. The specification provided says there should be two replicas. If one of the pods fails for any reason (for example, it crashes), Kubernetes will detect the difference between the current state and the desired state and start a new pod.

You can create a deployment by using the `kubectl create deploy` command, but Kubernetes entities are usually defined using `.yaml` files. This enables you to store your configuration in source control along with the rest of the application.

YAML Files

YAML—Yet Another Markup Language (or YAML Ain’t Markup Language, according to preference)—is a structured text format. It is semantically similar to JSON (JavaScript Object Notation) but easier for humans to read.

The configuration directory contains the following files:

- credit-service-deploy.yaml
- credit-service-service.yaml
- postgres-deploy.yaml
- postgres-service.yaml
- cs-connection-strings.yaml
- pg-db-secrets.yaml

Listing 10-1 shows the resource definition for the credit-service deployment. The deployment definition includes the definition of the Kubernetes pods that run the credit-service application. It corresponds to Label 1 in Figure 10-2.

There are two spec sections; the first defines the deployment and the second spec subsection defines the pod. Most of the pod definition describes the container the pod will host.

Listing 10-1 The credit-service deployment resource definition

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: credit-service
    name: credit-service
spec:
  replicas: 2
  selector:
    matchLabels:
      app: credit-service
  template:
    metadata:
      labels:
        app: credit-service
    spec:
      containers:
        - image: mfcobolbook/credit-service:1.0
```

```

name: credit-service
ports:
- containerPort: 8080
env:
- name: POSTGRES_HOST
  valueFrom:
    configMapKeyRef:
      name: cs-connection-strings
      key: host
- name: POSTGRES_DB
  valueFrom:
    configMapKeyRef:
      name: cs-connection-strings
      key: db-name
- name: POSTGRES_USER
  valueFrom:
    configMapKeyRef:
      name: cs-connection-strings
      key: user
- name: POSTGRES_PASSWORD
  valueFrom:
    secretKeyRef:
      name: pg-db-secrets
      key: password

```

There are a few items in this file worth describing in more detail. The first is the `replicas: 2` field; this specifies that there should always be two pods for the credit-service application. If, for example, the container in one of the pods crashes, it will be closed down and Kubernetes will create a new pod in order to keep the replicas specification true. It's possible to edit the deployment to increase or decrease the number of pods. When you edit an object like a deployment, Kubernetes adjusts the state to meet the new specification. However, not all properties can be edited dynamically. There is also a Kubernetes autoscaler that can be used to change the number of pods in a deployment dynamically based on metrics like CPU or network.

The container image is defined as `mfcobolbook/credit-service:1.0`. This is a Docker container, built in the way described in the previous chapter. It has been pushed to the `mfcobolbook` repository on Docker Hub; when one of these pods is created for the first time on a Kubernetes cluster, the image is fetched automatically from the repository.

Enterprises often host their own container repository of images rather than using repositories on the public Internet. This enables tighter control over what is downloaded and run on clusters.

The container specification also states that port 8080 should be open; this is the port on which our Spring Boot application listens for requests. Finally, the `env` section specifies

four environment variables. Values for the first three are taken from a ConfigMap object called cs-connection-strings.

The last value is for the database password and it is taken from a Kubernetes Secret (defined in pg-db-secrets.yaml). If you now open postgres-service.yaml and look at that, you can see it uses the same PostgreSQL image we used in the previous chapter. But we are now setting the user, database name, and password from the same ConfigMap and Secret that we used to configure the credit-service.

How do the credit-service pods connect to PostgreSQL? The credit-service deployment definition gets a host name from the cs-connection-strings ConfigMap (see Listing 10-2). This corresponds to Label 2 in Figure 10-2.

Listing 10-2 The ConfigMap resource file

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: cs-connection-strings
data:
  host: "postgres-database"
  db-name: "application-db"
  user: "postgres"
  password: "LakeOrangeGarden"
```

The host is set to postgres-database. The host name for the database is defined by the postgres-database-service. Listing 10-3 shows the database service resource. There are three types of Service objects in Kubernetes; this one is the default type, ClusterIP. It not only acts as a load balancer if the deployment it connects to uses multiple replicas, it registers its name in the Kubernetes DNS service. This corresponds to Label 3 in Figure 10-2.

Listing 10-3 The database service resource

```
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: null
  labels:
    app: postgres-database
  name: postgres-database
spec:
  ports:
    - port: 5432
      protocol: TCP
      targetPort: 5432
  selector:
```

```
app: postgres-database
```

The ports specification in the ClusterIP service definition specifies the port on which this service is exposed and the target port used by the application running inside the container. The exposed port is mapped to the target port on the container. In this case, both ports are 5432 – the standard PostgreSQL server port. The service is associated with the deployment through the matching label app: postgres-database. The postgres deployment (Label 4 in Figure 10-2) shares the same label – the resource isn't listed here, but you can see it by opening postgres-deploy.yaml in the configuration directory. In the next section, we will deploy the application.

Deploying the Application

Deploying the application is very simple, as we have defined all the resources in configuration files already. To deploy the application and start it running:

1. Open a command prompt.
2. Change directory to the **kubernetes\configuration** directory in the chapter 10 examples.
3. Enter the following command:

```
kubectl apply -f .
```

The period at the end of this command means “use the files in the current directory.” The -f command tells Kubernetes to apply the configuration state defined in the specified file – in this case, all the files in the configuration directory.

4. It will take a little while for all the containers to start up and to connect to the services. You can check progress by giving the command:

```
kubectl get pods
```

When all the pods are running, the output should look something like this:

NAME	READY	STATUS	RESTARTS	AGE
credit-service-556b578-hpxdn	1/1	Running	0	25s
credit-service-556b578-nsxnb	1/1	Running	0	24s
postgres-database-78dd556-hg9bg	1/1	Running	0	30s

The pod names in your deployment will be slightly different; when Kubernetes creates a pod from a deployment, the first part of the name is the same as the deployment name and the second part is created randomly. If any of the pods don't start running successfully, you can use the kubectl describe command to show the state of the pod, including a list of the events as it tried to start the container.

Assuming everything is running, we can now take a look at the application running. The instructions here assume you are using Minikube running locally. To see the application:

1. Find the IP address Minikube has actually published your service on. From the command prompt, run:
`minikube service credit-service --url`
2. This displays a result that looks like: `http://192.168.99.101:30850`
This URL corresponds to Label 5 on Figure 10-2.
3. Browse to the URL in your Web browser to see the “Hello World” confirmation that the service is running.
4. This version of the application, like the one in Chapter 9, has its data defined in a resource, so you can now initialize the database by browsing to `http://192.168.99.101:30850/admin/initialize-db` (your IP and port number will be different than the one shown here). If the database initializes correctly, you should see the **All tables initialized** message in the web browser. You can retrieve the first customer by using the `/service/customer/1` path on the end of the URL.

At this point we have a Kubernetes application running and it is exposed outside the Kubernetes cluster (although it is on a non-standard port). If we wanted to scale up the number of instances for the credit service, we could do it easily by editing the credit-service deployment and changing the replicas value from 2. We could also add a Kubernetes autoscaler and add rules to increase or decrease the number of replicas according to load. If you want to learn more about Kubernetes, the <https://kubernetes.io> website has interactive tutorials in the Documentation section. At the time of writing, there are online video courses available from Pluralsight and Udemy.

In the next section, we will create a Kubernetes ingress, making the application available on port 80.

Creating the Ingress

This section is optional as we already have our containerized COBOL application running and deployed in a Kubernetes cluster. An ingress is a Kubernetes resource that enables you to publish your Web applications to the standard ports (80 and 443). Although the ingress resource provides you with a standard way of defining an ingress, the manner in which they are implemented depends on how a Kubernetes cluster has been deployed. A Kubernetes cluster deployed out of the box (like our Minikube cluster) does not include an ingress controller; without an ingress controller, creating an ingress does nothing.

To create an ingress controller on Minikube and then expose the credit-service application on port 80 (Label 6 on Figure 10-2):

1. From a command prompt, run:
`minikube addons enable ingress`
2. This deploys an NGINX ingress controller, which can take up to a minute or so to become fully available. This pod is deployed to the kube-system namespace rather than the default namespace (where credit-service is deployed). To see the pods in the kube-system namespace:
`kubectl get pods -n kube-system`
You are looking for a pod with a name that starts `ingress-nginx-controller-`. When this shows with status `Running`, you can move to the next step.
3. In the kubernetes/ingress folder there is an `ingress.yaml` file. Run:
`kubectl apply -f ingress.yaml`
4. This creates an ingress and assigns an IP address. It can take two or three minutes for IP address assignment. Use the command:
`kubectl get ingress`
5. The output shows the name of the ingress, a host name (`credit-service.info`), an IP address, and Ports. The IP address field will be blank until Kubernetes has finished assigning it.
6. Once you have a valid IP address, you need to make sure your OS can resolve the host-name (`credit-service.info`) to the IP address. If you were deploying this on the public web or inside an enterprise network, you would create a DNS entry. However, since we are running all of this locally, update the hosts file on your computer. On Linux, this is `/etc/hosts`. On Windows, it is `C:\Windows\System32\drivers\etc\hosts`. You will need administrator privilege to edit this file.
7. Add the following entry to your hosts file:
`192.168.99.101 credit-service.info`
The IP address should be the one displayed by the `kubectl get ingress` command in Step 4 – not the one printed here.
8. Go to your web browser and see the application running at
`http://credit-service.info`.

Viewing Application Logs

Kubernetes treats everything our containers write out to StdOut and StdErr as logs. Factor 11 from <https://12factor.net> is “Treat logs as event streams.” One of the things you

don't want to do when running at cloud scale is to connect directly to individual VMs or containers in order to look at log files.

The Java parts of our application are using a logging framework brought in as part of the Spring Boot dependencies. By default, Spring Boot configures it to send all log output directly to the console. We haven't set up our COBOL application with proper logging (this was discussed in Chapter 5); we have used display statements. This isn't good practice, but it was done here as the most straightforward way to get the example running. It isn't good practice because a logging framework can be configured to add useful extra information to every log statement (for example, a timestamp) as well as allow you to configure the verbosity of your log output. An application running in production might be configured only to output warnings and errors, whereas when you are debugging an application, you might want to see extra information.

We can see the recent log output for the credit-service application by running this command:

```
kubectl logs -l app=credit-service
```

This displays the logs for all pods labeled `app=credit-service`. You can see the output for an individual pod by giving the name of the pod instead, for example (your pod name will be different as they are randomly assigned):

```
kubectl logs credit-service-556b578777-hpxdn
```

You can also tail the logs so that you can see new output as it is written:

```
kubectl logs -l app=credit-service --follow
```

If you have followed all the steps in this section, you have deployed a COBOL application to Kubernetes, connected it to a database, and made it available (locally at least) through http. Although some of the details would be different, you would follow the same steps to build an application and run it on a Kubernetes cluster hosted in the public cloud or inside a datacenter. For simplicity, we've chosen to make our application pure JVM (it doesn't use any of functionality from the COBOL native run-time), but Micro Focus provides its own Docker images that can be used to build your own application container. This would give you the option, for example, of using the native file handler for higher performance when using ISAM files.

In the next section, you will learn a completely different approach to deploying an application: serverless computing.

Serverless Computing

Serverless computing is a way to deploy applications that is very different to anything we have looked at so far. In serverless computing, you deploy individual API functions to a service that will run those functions on demand. There is no server for you to maintain or container for you to build; keeping the underlying infrastructure patched and up-to-date is the responsibility of the vendor.

All major public cloud vendors provide their own versions of serverless computing. There also are some PaaS offerings that run inside the data center that offer serverless deployments. Serverless computing is very attractive for services that have widely varying demand requirements. When a serverless function isn't in use, it doesn't consume any compute or memory at all; it is loaded only when it is called. During periods of high demand, a serverless function can be scaled out to hundreds or thousands of concurrent instances as needed.

Serverless computing is a new form of cloud computing, but at the time of this writing, it is mature enough to be considered for any production jobs that fit its niche characteristics. Serverless computing has both its disadvantages and its benefits. There isn't a standardized model for serverless computing, so you might get locked into one vendor, particularly as your application becomes more complex. You don't have any visibility into the infrastructure where your application is deployed, so you are putting more trust into the vendor than when you are consuming IaaS.

The example here uses AWS Lambda and an API Gateway. If you want to follow along with the example, you will need an AWS account. Creating a new AWS account requires a credit card, but you can follow this example within the free allowance provided by AWS so that you don't incur any charges.

We will not deploy the entirety of the credit-service application to AWS Lambda. The set of REST endpoints provided by the credit-service application is better offered as a microservice than as a set of separate serverless functions. Instead, we will provide only the `StatementCalculator` function to work out monthly interest. This is a relatively compute-heavy function that has high load demands once a month. It is used very little the rest of the time, so it is a good candidate for a serverless function.

Changing the Application

We are going to refactor the application before deploying it. At the moment, the `StatementCalculator` is tightly bound to `ACCOUNT-STORAGE-ACCESS`, which retrieves all the transaction information. Connecting our serverless function to a database increases the complexity (and cost) of the deployment. Instead, this serverless function will receive a JSON object that includes all the transactions for calculating monthly interest for a statement.

We are not going to make any changes to the InterestCalculator itself – the business rules it embodies are well tested and well understood. Instead we are going to modify the InteroperationLayer and provide a different implementation of ACCOUNT-STORAGE-ACCESS. Figure 10-3 shows how the serverless function is implemented.

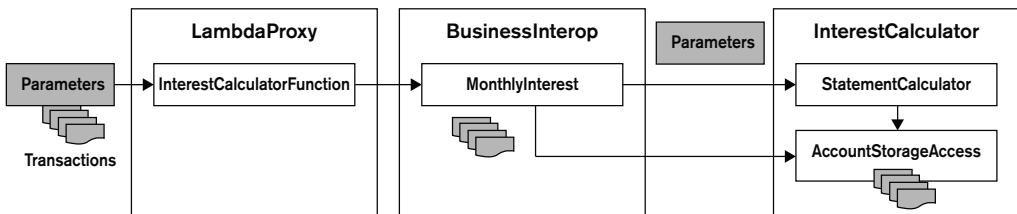


Figure 10-3 The StatementCalculator as a serverless function

There are three projects and each will be built into a single jar by Maven using the pom.xml in the LambdaProxy project. The LambdaProxy project provides an InterestCalculatorFunction class that has a handleApiRequest() method that will be invoked when the function is called.

The data to this function will include the same parameters that we passed for interest calculation in previous versions of the CreditService application (day-rate, starting amount, accountId, and start date). It will also include an array with all the transactions for the account and month we want to calculate. The only reason we pass an account Id and start date as well is for the original StatementCalculator code; this code remains unchanged and expects those parameters and won't function without them.

The handleApiRequest() method, in turn, calls the MonthlyInterest::init() method in the BusinessInterop project. This has been modified from the version in previous versions of the CreditService application with an extra argument – the array of transactions. It passes these to a modified version of the ACCOUNT-STORAGE-ACCESS program that stores them.

When the MonthlyInterest class invokes the StatementCalculator program, it passes the set of parameters the StatementCalculator expects. The StatementCalculator then performs the calculation, calling the OPEN-TRANSACTION-FILE and FIND-TRANSACTION-BY-ACCOUNT entry points in ACCOUNT-STORAGE-ACCESS as it always has done. These return the transactions one at a time from the list already passed to it. The StatementCalculator works through the business rules to calculate the interest just as it always has and returns the result. Listing 10-4 shows the modified init() method from the MonthlyInterest class in the interoperation layer.

Listing 10-4 The MonthlyInterest init() method

```

method-id init (dayRate as decimal, startingAmount as decimal,
                startDate as type LocalDate,
                accountId as binary-long

```

```

        transactions as type List[type TransactionDto]).
copy "TRANSACTION-RECORD.cpy" replacing ==(PREFIX)== by ==LS==.
declare recordList as type List[binary-char occurs any]
    = new ArrayList[binary-char occurs any]
declare recordBytes = GetByteArray
set self::dayRate to dayRate
set self::startingAmount to startingAmount
set self::startDate to startDate
set self::accountId to accountId
set initialized to true
call "ACCOUNT-STORAGE-ACCESS"
perform varying nextTransaction as type TransactionDto
        through transactions
    invoke nextTransaction::getAsTransactionRecord
        (LS-TRANSACTION-RECORD)
move LS-TRANSACTION-RECORD to recordBytes
    invoke recordList::add(recordBytes)
end-perform
call SET-TRANSACTION-DATA using by value recordList

end method.

```

You can see the extra argument at the end, the list of TransactionDto objects (each TransactionDto represents a single transaction). The method stores all the parameters it has been passed, converts each TransactionDto object into the COBOL record layout defined for transactions and puts them in a new list. It then calls the SET-TRANSACTION-DATA entry point in ACCOUNT-STORAGE-ACCESS.

Listing 10-5 shows the entire ACCOUNT-STORAGE-ACCESS program. What was formerly quite a long program — first for managing ISAM files and then for making database access look like ISAM files — is now very short.

Listing 10-5 The ACCOUNT-STORAGE-ACCESS program

```

$set ilusing(java.util)
program-id. ACCOUNT-STORAGE-ACCESS.

data division.
working-storage section.
copy "PROCEDURE-NAMES.cpy".
01 transaction-index      binary-long.
01 transaction-list       type List[binary-char occurs any]. 

linkage section.
01 LNK-STATUS.
03 LNK-FILE-STATUS-1      PIC X.

```

```

03 LNK-FILE-STATUS-2      PIC X.
01 LNK-TRANSACTION-LIST    type List[binary-char occurs any].
copy "FUNCTION-CODES.cpy".
copy "TRANSACTION-RECORD.cpy" replacing ==(PREFIX)== by ==LNK==.

procedure division.
goback.

ENTRY OPEN-TRANSACTION-FILE using by VALUE LNK-FUNCTION
                           by reference LNK-STATUS
move "00" to LNK-STATUS
goback.

ENTRY SET-TRANSACTION-DATA using by value Lnk-transaction-list.
move LNK-TRANSACTION-LIST to transaction-list
goback.

ENTRY FIND-TRANSACTION-BY-ACCOUNT using by value LNK-FUNCTION
                           by reference LNK-TRANSACTION-RECORD
                           LNK-STATUS
evaluate LNK-FUNCTION
when START-READ
  if transaction-list = null or transaction-list::size = 0
    move "23" to LNK-STATUS *> No records
  else
    move 0 to transaction-index
    move "00" to LNK-STATUS
  end-if
when READ-NEXT
  move "00" to LNK-STATUS
  if transaction-index < transaction-list::size
    declare next-record
      = transaction-list::get(transaction-index)
    set LNK-TRANSACTION-RECORD to next-record
    add 1 to transaction-index
    if transaction-index < transaction-list::size
      move "02" to LNK-STATUS *> more records
    end-if
  end-if
end-evaluate
goback.

```

It has an OPEN-TRANSACTION-FILE entry point that always returns a “00” success status (because there is no file or database to fail to open). It has a SET-TRANSACTION-DATA entry point that stores the list of transactions. And it has a FIND-TRANSACTION-BY-ACCOUNT entry point that returns the transactions from the list. The new code mimics the same behavior as the original version of the application that read the records from an ISAM file, returning the same status codes as before.

The Java Lambda Function Handler

Setting up a Lambda function in AWS consists of:

- Defining the function
- Uploading the code for the function
- Connecting the function to a trigger event

There are several ways of triggering a Lambda function. For example, you can create a new object in an S3 bucket or post a message onto a queue. The code that is called by AWS Lambda is effectively an event handler; it will receive information about the event that triggered it.

Different types of events require you to write different kinds of event handlers. AWS provides sample code that shows you how to write these different kinds of event handlers in the various languages supported by Lambda.

A serverless function is not the same as an HTTP endpoint; as previously stated, it is an event handler that responds to an event. We will first define the function and then test it using the AWS Lambda dashboard.

Then, to expose the functionality as an HTTP endpoint, we will connect our function to an AWS API Gateway. The API gateway will define a REST endpoint that expects an HTTP POST. The body of the POST request will contain JSON that defines all the data needed for the interest calculation.

When we POST a request to the endpoint, the Lambda function is invoked and it receives a payload of data from the Gateway; the payload includes the body of the POST request. Figure 10-4 shows the flow from the POST request hitting the endpoint defined in the API gateway, through the payload being sent to the Lambda function and then the actual code to do the work getting invoked.

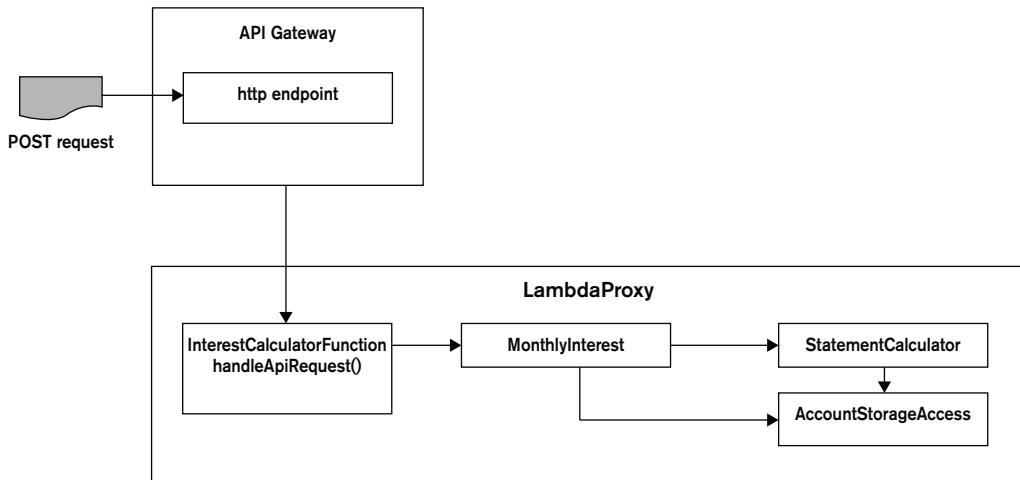


Figure 10-4 The API Gateway and Lambda function

The API gateway handles all the HTTP communication and response. It will also provide you with an encrypted HTTPS connection; you have the option of configuring different types of authentication and authorization for your endpoints. This enables you to secure your application with the gateway doing most of the heavy lifting; you are free to concentrate on writing your application code.

The `handleApiRequest()` method has three arguments:

- An `InputStream`
- An `OutputStream`
- A `Context` object

The `Context` object provides your function with some metadata about itself as well as a logger object. Listing 10-6 shows the `handleApiRequest()` method from class `InterestCalculatorFunction`.

Listing 10-6 The `handleApiRequest()` method

```

public void handleApiRequest(InputStream inputStream,
                             OutputStream outputStream, Context context)
                             throws IOException {
    BufferedWriter writer = new BufferedWriter(
        new OutputStreamWriter(outputStream));
    try {
        LambdaLogger logger = context.getLogger();
        logger.log("handleApiRequest");
        Gson gson = new GsonBuilder().setPrettyPrinting()
    }
  
```

```

        .create();
JsonObject o = JsonParser
        .parseReader(new InputStreamReader(inputStream))
        .getAsJsonObject();
JsonElement element = o.get("body");

if (element == null) {
    logger.log("No body element found");
} else {
    logger.log(element.toString());
}

String unescapedBody = StringEscapeUtils
        .unescapeJson(element.toString());
String unwrappedRequest = StringUtils
        .unwrap(unescapedBody, "'");
CalculationRequest request = gson.fromJson(
        unwrappedRequest, CalculationRequest.class);

if (request == null) {
    logger.log("No calculation request found");
} else {
    logger.log(request.toString());
}
StatementDto dto = doCalculation(request);
if (dto == null) {
    logger.log("No DTO returned");
}
String json = gson.toJson(dto);
writer.write(json);
} catch (Exception e) {
    logger.log(e.getMessage());
} finally {
    writer.close();
}
}
}

```

The `InputStream` passed to this function contains the text for a JSON object that includes the HTTP headers sent to the endpoint, a timestamp, and other information that might be useful. It also includes the request body; the API Gateway encodes this as a JSON object. The body we have passed in is already a JSON object, so the body has to be unescaped before it can be properly decoded.

Most of the code in Listing 10-6 is actually parsing the JSON, unescaping the body field, and then parsing that as a JSON object. The LambdaProxy project includes a class

called `CalculationRequest`; this is really just a Data Transfer Object (DTO) with fields that correspond to all the parameters needed to request a calculation statement.

The `Gson` object deserializes the JSON from the input stream into a `CalculationRequest` that is then used to call the Interoperation Layer. A separate `doCalculation()` method (not shown here) converts the `CalculationRequest` into the parameter format the `MonthlyInterest` class expects.

Testing the Code

If you haven't already done so, download the Chapter 10 example code and import all the projects under the serverless directory into Eclipse. As well as the projects to build our Lambda function, we've included the test projects from earlier chapters (although the tests themselves have now been slimmed down to include only tests for interest calculations).

We've also made some changes to the tests as they now have to pass in all the transactions to run a test rather than the business logic reading the data in from elsewhere. But because there are no files to open or databases to connect to, you can run the tests without needing to set up complex run configurations. For example, to run the `BusinessSystemTests`, just right-click the project and choose **Run As > COBOL JVM Unit Test**. The only thing you might need to fix (as in earlier chapters) is the Maven builders used to put the jars for the `BusinessInterop` and `InterestCalculator` projects into the Maven repository.

The `LambdaProxy` project also has a test class called `ProxyWithStreamTest`. This reads a JSON text file in from `resources/gateway-stream.json` and invokes `InterestCalculatorFunction.handleApiRequest()`. The contents of this JSON file follow the same format as the JSON input streams sent by the API gateway and give us an end-to-end test that helps us assess whether our code behaves as expected before we upload it to AWS.

Right-click the `LambdaProxy` project and choose **Run As > JUnit test** to run the tests. You can set breakpoints and debug these tests if you want to get a better feeling for how the code works. Getting the code to work was an iterative process; to find out exactly what the JSON sent from the API Gateway would look like, I deployed the AWS Lambda Java "Hello World" application and connected it to an API Gateway endpoint. Then I used Postman to send POST requests containing a JSON body for my calculation request and logged the information sent to the application so that I could see exactly what the data sent from the API Gateway looked like.

These logs were also used to create the sample data used for the test. In the next section, we will build and deploy the application.

Building and Deploying the Lambda Function

We don't need to build a container to deploy our Lambda function, but we do need to create a fat jar with all the Java dependencies it needs to run. From a Visual COBOL command prompt (on Windows) or a shell terminal with COBDIR set on Linux, go to the serverless/LambdaProxy directory and run:

```
mvn clean package
```

This builds the package, runs the tests, and creates the jar file.

To deploy the Lambda function:

1. Log in to your AWS Account.
2. From the AWS Management Console page, go to the Lambda page (if you can't find it, type **lambda** into the **Find Services** field and click the Lambda link).
3. Click the **Create Function** button.
4. Select **Author from scratch** and, in the **Function** name field, type **InterestCalculator**.
5. Select **Java 8** as the **Runtime**.
6. At the bottom of the page, click **Create function**. The Configuration page for InterestCalculator appears.
7. Scroll down to the **Basic settings** box and click the **Edit** button.
8. You must change the Handler field to specify the class and method to be invoked when the Lambda function is triggered:

```
com.mfcobolbook.lambda.InterestCalculatorFunction::  
handleApiRequest
```

(all on one line). You can also add a description like "Calculate Monthly Interest" into the **Description** field.
9. Click **Save**.
10. In the Function code box, click **Actions > Upload a .zip or .jar file**.
Ignore the message in this box indicating that the function editor does not support Java code. This is because Java functions are uploaded as compiled binaries that cannot be edited. When you use an interpreted language like Python or JavaScript for Lambdas, you upload source code that you can edit here.
11. Click **Upload**, select **LambdaProxy-1.0.0.jar** in the **File** dialog, click **Open** or **Ok** in the **File** dialog, and then click **Save** on the Web page.

You have now created a Java Lambda function (that is actually using COBOL business logic to perform a calculation). To test the function is configured correctly:

1. At the top of the **InterestCalculator** configuration page, note the **Select a Test Event** drop-down field to the right of the page, which says **Select a Test Event**.
2. Click the field and click **Select Test Event**. There are no events, so you will see a **Configure test event** dialog.
3. Select **Create new test event** and name your event **Calculation1**.
4. Using either Eclipse or a text editor, open the **gateway-stream.json** file from the LambdaProxy **test/resources** folder. Select the entire contents and then paste them into text edit area in the Configure test event dialog.
5. Click **Create**.
6. Now click the **Test** button to the right of the page. This invokes the function with the test data we just configured, and if everything has worked, you should see **Execution result: succeeded**.
7. You can expand the details to see what was returned from the function. You can also click the **Logs** link at the top to go through to a Cloudwatch page where you can drill down into the log stream for the application.
If the execution failed rather than succeeded, this should help you work out where things went wrong.

Once the function is running, you can go through to the next section and configure the API Gateway.

Configure the API Gateway

You now have a working Lambda function, but you need to make it accessible for it to be useful. We will create a single REST endpoint using the AWS API Gateway. Because we are doing this in the simplest possible way, your endpoint will be available on the public web and will not be secured (anyone can invoke the endpoint without needing to authenticate themselves). Because of this, I advise you not to leave the endpoint open after you have finished trying it out. The odds of it happening are low, but in theory, a malicious party could hit your endpoint repeatedly, running your function until you start to incur charges on your AWS account.

This is unlikely to happen while you are trying things out, but any public endpoint on the web is discovered by all sorts of crawlers and bots eventually, and public endpoints are often repeatedly invoked by automated scripts looking for vulnerabilities. So, if you leave the endpoint open for weeks and months, you might end up with a bill for Gateway and Lambda usage that you didn't expect.

To set up and test the Gateway:

1. From the **InterestCalculator** configuration page, at the right side of the Designer box, click the **Add trigger** button.
2. Select **API Gateway** from the drop-down.
3. Click **Create an API** in the API drop-down.
4. Select **HTTP API** as the API type.
5. Set the security to **Open**.
6. Click **Add**.
7. At the bottom of the **InterestCalculator** page, there is now an API Gateway box, with **InterestCalculator-API**.
8. Under details, there is an API endpoint. It ends in the path **/default/InterestCalculator**. Copy the endpoint.
9. Open up Postman. (We used Postman in earlier chapters to test our REST endpoints.)
10. Create a POST request in Postman with the endpoint you just copied.
11. Add a header **Content-Type** with value **application/json**
12. Under **Body**, select **raw**, and then paste in the contents of **serverless/calculation-request-data.json**.
13. Click **Send**.

You should see this response:

```
{  
  "minimumPayment": 22.29,  
  "endingAmount": 445.87,  
  "interestAmount": 1.96,  
  "accountId": 0,  
  "startDate": {  
    "year": 2019,  
    "month": 7,  
    "day": 1  
  }  
}
```

You have just run an interest calculation written in COBOL over the public Web using a serverless function! If you had to do 100,000 of these calculations in a short amount of time, as you sent in more and more requests, AWS would create many concurrent instances of your function to manage the load; the calculations would be completed faster than if you were relying on just one or two instances of the running application.

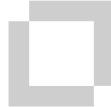
Don't forget to go to the API gateway page and delete your endpoints once you no longer want them to be accessible. Alternatively, you could experiment with adding JWT authorization to the API to secure it from unauthorized users.

Conclusion

In this chapter, you learned two ways of exploiting legacy COBOL code in scalable cloud environments. The example application is much simpler than most real COBOL applications, but at the end of this book, I hope you are left with a sense of the possibilities that recompiling COBOL to modern run-times can bring. The key points I would like readers to take away are:

- Visual COBOL greatly simplifies the consumption of legacy applications in modern architectures.
- MFUnit makes writing automated tests for procedural legacy COBOL easier – and automated tests make it possible to refactor code safely.
- You can refactor your applications to preserve complex business rules written in COBOL and run them in microservice architectures.

I hope you've enjoyed reading this book as much as I've enjoyed writing it.



Index

Symbols

12 factor applications (cloud-native applications) 159
15-factor application (cloud-native applications) 159

A

AbstractBusinessAccess class (REST Service)
 71–75
abstract test class 121–122
accessing data
 REST Service
 AbstractBusinessAccess class 71–75
 AccountController class 89–92
 AccountStorageAccess class 76–83
 application components 67–70
 data access and transfer classes 71
 interoperation layer 70
 iterators 83–85
 MonthlyInterest class 85–88
 Spring Boot 88
 StatementController class 95–97
 testing endpoints 93–95
 WebServiceApplication class 89–90
accessing files, indexed files 45–53
 declaring files and data 45–47
 opening a file 52–53
 procedure division entry point 47–48
 reading and writing files 49–51
 reading a record 52–53
 writing records 52
AccountController class (REST Service) 89–92
AccountDataAccess class 71

AccountDto wrapper class 79
ACCOUNT-RECORD.cpy copybook 44
accounts() method 91
account-SORAGE-ACCESS program
 accessing files
 reading and writing files 49–51
AccountStorageAccess class (REST Service)
 76–83, 77
 adding, updating, and deleting records 77–79
 procedure-pointers 76–77
 reading and finding records 79–83
ACCOUNT-STORAGE-ACCESS program 41
accessing files 45–53
 declaring files and data 45–47
 opening a file 52–53
 procedure division entry point 47–48
 reading and writing files 49–51
 reading a record 52–53
 writing a record 52
 calling COBOL programs from Java 61–65
account storage test case code
 BusinessRules Layer, automated testing 113–114
Active Server Pages (ASP) 129
addAccount() method 77
Add Customer button, React 146–148
AddCustomerForm component 146–148
AddCustomerForm dialog, React 148–151
Add New button 147
Amazon Elastic Computer Cloud (EC2) 158
Amazon Web Services (AWS), S3 174
Angular 130, 137
ANSI 85 COBOL standard 21
API Gateway
 Lambda function 204

- serverless computing 208–210
- App()** function (`App.js` files) 139
- `App.js` file 139
- applications 30
 - automated testing 99
 - BusinessRules Layer 107–117
 - downloading test examples 100–101
 - end-to-end tests 120–126
 - Interoperation Layer 117–120
 - MFUnit 101–107
 - strategies 99–100
 - containerizing 155
 - changing from ISAM to a database 161–173
 - cloud-native applications 158–159
 - containerizing the CreditService 175–183
 - containers 156–157
 - microservice architectures 160–161
 - running revised CreditService application 173–175
 - modernizing with Visual COBOL 30–31
- REST Service**
 - `AbstractBusinessAccess` class 71–75
 - `AccountController` class 89–92
 - `AccoutStorageAccess` class 76–83
 - application components 67–70
 - data access and transfer classes 71
 - interoperation layer 70
 - iterators 83–85
 - `MonthlyInterest` class 85–88
 - Spring Boot 88
 - `StatementController` class 95–97
 - testing endpoints 93–95
 - `WebServiceApplication` class 89–90
- sample application 41
 - calling COBOL from Java 61–65
 - data storage in indexed files 43–53
 - generating data 59–61
 - Interest-Calculator program 54–60
- user interface (UI) modernization
 - Credit Service application 130–135
 - React 135–154
 - Single-Page Web applications 128–130
 - UI choices 127–128
- architecture
 - microservices 160–161
- arguments
 - `handleApiRequest()` method 204
- arguments, calling COBOL programs 30
- artifact ID (identifiers) 10
- ASP (Active Server Pages) 129
- asynchronous programming 143–144
- `AutoCloseable` interface 74
- automated testing 99
 - BusinessRules Layer
 - account storage test case code 113–114
 - interest calculator test case code 114–117
 - legacy code 117
 - test case setup code 111–112
 - downloading test examples 100–101
 - end-to-end tests 120–126
 - Interoperation Layer 117–120
 - MFUnit 101–107
 - strategies 99–100
- AWS (Amazon Web Services), S3 174

B

- BDD** (Behavior Driven Development) 125
- `@BeforeClass` annotation 123
- Behavior Driven Development (BDD) 125
- binary large object. See `BLOB`
- bin directory 6
- block structures, structured programming 30
- Booleans 23
- Bootstrap 138
- browser developer tools 133
- Build Properties window 8
- BusinessRules Layer, automated testing
 - account storage test case code 113–114
 - interest calculator test case code 114–117
 - legacy code 117
 - test case setup code 111–112
- BusinessRules project 42
- BusinessSystemTests 167–170
- by content, passing arguments 36
- by reference, passing arguments 35
- byte code 20
- by value, passing arguments 35

C

- `calculateArea()` method 25
- calculator test suite 103–105
- Calendar 41
- CALL prototypes 36
- call statements, COBOL program structure 36
- cancel verb 30
- case sensitivity
 - source formats 23
- CDN (Content Delivery Network) 131
- CI/CD (Continuous Integration/Continuous Delivery)
 - pipeline 181
- CI (Continuous Integration) systems 100
- Circle class 24
- classes

HelloWorldProcedural.class 6
 object model 23
 Visual COBOL 24–25
 class headers 24
 CLI (Command Line Interface), Kubernetes 188
 close() method 75
 cloud environments
 containerizing applications 155–161
 changing from ISAM to a database 161–173
 cloud-native applications 158–159
 containerizing the CreditService 175–183
 containers 156–157
 microservice architectures 160–161
 running revised CreditService application
 173–175
 cloud-native applications 158–159
 Cloud Native Computing Foundation (CNCF) 187
 CloudService application
 containerizing for cloud environment
 cloud-native applications 158–159
 microservice architectures 160–161
 CNCF (Cloud Native Computing Foundation) 187
 cobjrun command 6
 COBOL command prompts 6
 COBOL dialects 19–20
 COBOL Explorer window 8
 COBOL Perspective 8
 COBOL programs
 calling from Java 61–65
 structure 31
 data division 32–34
 procedure division 34–38
 COBOL source formats 21–23
 cobsetenv command 8
 code
 creating JVM projects with Maven 13–14
 command 177
 Command Line Interface (CLI), Kubernetes 188
 Command-line URL (cURL) 93
 commands
 cobjrun 6
 COBOL command prompts 6
 cobsetenv 8
 Comma Separated Variable (CSV) files 166
 comments
 source formats 22–23
 compiler directives
 COBOL dialects 20
 componentDidMount() function 143
 components
 consuming
 creating Java projects with Maven 16–18
 components, React 138–140
 CustomerList component 144–146

MainPage component 140–144
 ConfigMaps, Kubernetes 190
 constructors (classes) 25
 consuming
 components
 creating Java projects with Maven 16–18
 container images 176–178
 containerizing applications 155–161
 changing from ISAM to a database 161–173
 creating databases 163–164
 exporting data 166
 installation on Native OS 162
 OpenESQL syntax 164–165
 PostgreSQL 161
 revised application 166–167
 running tests 167–170
 cloud-native applications 158–159
 containerizing the CreditService 175–183
 container images 176–178
 running the application 178–181
 scripts 181–183
 containers 156–157
 microservice architectures 160–161
 running revised CreditService application 173–175
 containers 156–157
 Content Delivery Network (CDN) 131
 Context argument (handleApiRequest() method) 204
 Continuous Integration (CI) systems 100
 Continuous Integration/Continuous Delivery (CI/CD)
 pipeline 181
 conventions
 MFUnit test suites 105
 copybooks 38–40
 copyDataSource() helper method 122
 copy files 38–40
 copy... replacing statements 38
 copy verb 38
 CORS (Cross Origin Resource Sharing) policy 134
 createdb command 164
 creating
 JVM projects
 Eclipse 7–9
 CreditController tests 125–126
 Credit Service application 130–135
 cross-origin resource sharing 133–135
 running the application 132–134
 CreditService application
 containerizing for cloud environment 155–161,
 175–183
 changing from ISAM to a database 161–173
 containers 156–157
 running revised CreditService application
 173–175
 running as Kubernetes application 189–190

credit-service-form application 132
 React 138
CreditService tests
 changing from ISAM to a database 172–173
 cross-origin resource sharing 133–135
 Cross Origin Resource Sharing (CORS) policy 134
 cross-site scripting (XSS) 134
 CSV (Comma Separated Variable) files 166
 cURL (Command-line URL) 93
 custom builders
 creating JVM projects with Maven 14–16
CustomerDataAccess class 71
Customer() function 145
CustomerList component, React 144–146
CUSTOMER-RECORD.cpy copybook 44
customerSelected() function 145

D

DAC (dedicated administrator connection). See **DAC**
data
 copybooks 38–40
 declarations 32–33
 enabling access with REST Service
 AbstractBusinessAccess class 71–75
 AccountController class 89–92
 AccountStorageAccess class 76–83
 application components 67–70
 data access and transfer classes 71
 interoperation layer 70
 iterators 83–85
 MonthlyInterest class 85–88
 Spring Boot 88
 StatementController class 95–97
 testing endpoints 93–95
 WebServiceApplication class 89–90
 group items 33–34
 storage
 indexed files 43–53
databases
 changing from ISAM to a database 163–164
DataBuilder project 42
data declarations 32–33
data division, COBOL program structure
 data declarations 32–33
 group items 33–34
DataMigrationTool 166
data records 30
day two issues 187
decimal arithmetic 30
declarations
 files and data (indexed files) 45–47
declaring data, COBOL programs 33

dedicated administrator connection. See **DAC**
deleteAccount() method 77
deploying
 Kubernetes application 195–196
 deployment entities, Kubernetes 189
 developer tools, browsers 133
 development environments, React 135–136
 Development Hub (Visual COBOL) 3
 dialects (Visual COBOL) 19–20
docker build command 178
docker-compose command 181
Docker Desktop 156
Dockerfiles 177
docker network create app-network command 179
docker push command 178
docker rm postgres command 180
docker run --name postgres --network app-network
 command 180
docker stop postgres command 180
Document Object Model (DOM) 128
Domain Driven Design 161
DOM (Document Object Model) 128
 downloading test examples 100–101

E

EC2 (Amazon Elastic Computer Cloud) 158
Eclipse 3
 creating COBOL JVM projects 7–9
 starting
 Linux 8
 Windows 8
Eclipse IDE 3
endpoints, testing 93–95
end-to-end tests, applications 100, 120–126
Enterprise Analyzer 160
entry points
 accessing files 47–48
 order of execution 106
entry points (COBOL programs) 35–38
environment division, COBOL program structure 31
event handlers 152–153
exceptions
 Visual COBOL 26–27
executable code 20
exit statements, COBOL program structure 38
Explorer window 8
exporting data
 changing from ISAM to a database 166
extending applications, React 146

F

fakes (Test Double) 100–101
 fat jar 177
 Fetch API 138
 fetch() function 143
 fields
 Visual COBOL 24
 file access, indexed files 45–53
 declaring files and data 45–47
 opening a file 52–53
 procedure division entry point 47–48
 reading and writing files 49–51
 reading a record 52–53
 writing records 52
 files
 .jar 12
 mfcobol.jar 13
 mfcobolrts.jar 13
 native code files 20
 pom.xml (Maven) 10
 file section, COBOL program structure 32
 forEach() method 85
 Formik 138
 FUNCTION-CODES.cpy 74
 function-status flag 112

G

generating data 59–61
 getAccount() method 83
 getAsAccountRecord() method 79
 getLastAccount() method 79
 @GetMapping annotation 91
 get() method 124
 .gitignore file 136
 given() method 125
 goback statements, COBOL program structure 38
 group ID (identifiers) 10
 group items 33–34

H

handleApiRequest() method 200
 arguments 204
 Hello World 5–7
 writing as a class 7
 HelloWorldProcedural.class 6
 HELPER-FUNCTIONS program 111–112
 helper methods
 abstract test class 122

high-precision decimal arithmetic 30
 horizontal scalability, cloud-native applications 155
 Hyper-V 187

I

IaaS (Infrastructure-as-a-Service) 158
 identification division, COBOL program structure 31
 identifiers (POM files) 10
 import statements (App.js files) 139
 indexed files 43–53
 accessing files 45–53
 declaring files and data 45–47
 opening a file 52–53
 procedure division entry point 47–48
 reading and writing files 49–51
 writing records 52
 reading records 52–53
 Indexed Sequential Access Method (ISAM) 30, 43
 Infrastructure-as-a-Service (IaaS) 158
 ingress, Kubernetes 190, 196–197
 init() method 200
 initTestData() helper method 122
 InputStream argument (handleApiRequest() method)
 204
 installation
 Kubernetes 187
 installing
 Maven 11
 integration tests 100
 IntelliJ 101
 Interest Calculator
 BusinessRules Layer, automated testing
 test case code 114–117
 Interest-Calculator 41
 InterestCalculatorFunction class (LambdaProxy project)
 200
 Interest-Calculator program 54–60
 internal modifiers 24
 internal repositories (Maven) 10
 interoperation layer
 accesing data with REST Service 70
 Interoperation Layer, automated testing 117–120
 interoperation tests
 changing from ISAM to a database 170–172
 InteropTests, setup code 119–120
 ISAM
 changing to a database 161–173
 creating databases 163–164
 exporting data 166
 installation on Native OS 162
 OpenESQL syntax 164–165
 PostgreSQL 161

revised application 166–167
 running tests 167–170
ISAM (Indexed Sequential Access Method) 30, 43
iterator() method 84
iterators 83–85
deploying the application 195–196
installation 187
running CreditService as Kubernetes application 189–190
viewing application logs 197–198
KVM (Kernel Virtual Machine) 188

J

.jar files 12
jar files 177
Java
 calling COBOL programs 61–65
Java Development Kit (JDK) 5
Java Lambda function handler 203–206
Java Native Interface (JNI) 31
Java projects
 creating with Maven
 adding COBOL component to repository 12–13
 adding code 13–14
 adding custom builders 14–16
 consuming the component from Java 16–18
 project structure 12
 running the application 18
Javascript 129
 desktop applications 130
Java Servlet Pages (JSP) 129
Java Virtual Machine (JVM) 1, 20
JDK installation 3
JDK (Java Development Kit) 5
Jenkins 181
JNI (Java Native Interface) 31
JSP (Java Servlet Pages) 129
JSX (Javascript syntax) 139
JUnit 101
JVM (Java Virtual Machine) 1, 20
JVM projects
 creating in Eclipse 7–9
jvmsourcebase directive 26
JVM test suite, running 107

K

Kernel Virtual Machine (KVM) 188
kubectl apply -f command 195
kubectl create deploy command 191
kubectl get nodes command 188
kubectl logs -l app=credit-service command 198
Kubernetes (K8S) 187–198
 application changes 190–191
 creating the ingress 196–197
 defining the application 191–195

deploying the application 195–196
installation 187
running CreditService as Kubernetes application 189–190
viewing application logs 197–198
KVM (Kernel Virtual Machine) 188

L

Lambda function
 API Gateway 204
 serverless computing 207–208
Lambda function handler 203–206
LambdaProxy project 200
legacy code
 testing with MFUnit 117
lifecycle, containers 179
linkage section (COBOL programs) 30, 32
Linux
 starting Eclipse 8
literals
 source formats 23
local repositories (Maven) 10
local-storage (COBOL programs) 30, 32
logs
 Kubernetes 197–198

M

main() method 25
MainPage class 140
MainPage component, React 140–144
MainPage render() function 147
managed code 20–21
map() function 145
Maven
 creating JVM projects
 adding COBOL component to repository 12–13
 adding code 13–14
 adding custom builders 14–16
 consuming the component from Java 16–18
 project structure 12
 running the application 18
 installation 11
 introduction 10–11
 support for JUnit 101
metadata 20
methods 24
 – calculateArea() 25
 main() 25
mfcobol.jar file 13

mfcobolrts.jar file 13
MF-TC-METADATA-SETUP-PREFIX (test case metadata function) 106
MFUnit
 testing legacy code 117
 testing procedural COBOL 101–107
 running test suites 102–107
MFU-TC-SETUP-PREFIX (test case setup function) 105
MFU-TC-TEARDOWN-PREFIX (tet case teardown function) 105
Micro Focus compiler
 support for COBOL dialects 20
Micro Focus Enterprise Analyzer 160
Micro Focus Enterprise Developer 8
Micro Focus Team Developer 8
Micro Focus Unit Testing pane 107
Micro Focus Visual COBOL 8
microservices 185
 architecture 160–161
 Kubernetes 187–198
 application changes 190–191
 creating the ingress 196–197
 defining the application 191–195
 deploying the application 195–196
 installation 187
 running CreditService as Kubernetes application 189–190
 viewing application logs 197–198
 platform necessity 185–187
 serverless computing 199–210
 building and deploying Lambda function 207–208
 changing the application 199–203
 configuring API Gateway 208–210
 Java Lambda function handler 203–206
 testing code 206
Microsoft SQL Server. See **SQL Server**
Minikube 187
 minikube addons enable ingress command 197
mocks (*Test Double*) 100–101
modernizing
 COBOL applications 30–31
 monolithic applications 160
MonthlyInterest class (*REST Service*) 85–88
MonthlyInterest::init() method (*BusinessInterop project*) 200
mvn clean package command 207
mvn package command 182

N

namespaces
 Visual COBOL 26–27
native code files 20
.NET Common Language Runtime (CLR) 20
Node.js 132
Node Package Manager (NPM) 132
npm install command 132
NPM (Node Package Manager) 132
npm start command 132
numeric data, representation 33
numerics 23

O

Object Linking and Embedding. See **OLE**
object model (Visual COBOL) 23–24
onClick function handler 145
openEntryPointer() method 75, 76
OpenESQL syntax 164–165
openFile() method 75
opening files (indexed files) 52–53
open() method 74
OpenSUSE
 Maven installation 11
Oracle JDK 8 3
order of execution, entry points 106
OutputStream argument (*handleApIRequest()* method) 204

P

PaaS (Platform-as-a-Service) 158
package.json file 136
package-lock.json 136
packages
 Visual COBOL 26–27
paragraphs, COBOL program structure (procedure division) 34–35
parameters
 COBOI programs 35–38
Personal Edition (Visual COBOL) 3
philosophy, React 137–138
Platform-as-a-Service (PaaS) 158
platforms, deploying microservices 185
 Kubernetes 187–198
 application changes 190–191

- creating the ingress 196–197
- defining the application 191–195
- deploying the application 195–196
- installation 187
- running CreditService as Kubernetes application 189–190
- viewing application logs 197–198
- platform necessity 185–187
- serverless computing 199–210
 - building and deploying Lambda function 207–208
 - changing the application 199–203
 - configuring API Gateway 208–210
 - Java Lambda function handler 203–206
 - testing code 206
- platform-specific code. See cross-platform programming
- pods, Kubernetes 189
- POM (Project Object Model) 10
- pom.xml files (Maven) 10
- port mappings 178
- postCustomer() function 152
- PostgreSQL 161
- PostgreSQL JDBC driver 166
- Postman 93
- prerequisites 3
- primary keys (ISAM) 30
- primary keys (records) 43
- private modifiers 24
- procedural COBOL 1, 29
 - applications 30–31
 - copybooks 38–40
 - structure of COBOL programs 31
 - data division 32–34
 - procedure division 34–38
 - testing with MFUnit 101–107
 - running test suites 102–107
- procedure division
 - accessing files 47–48
- procedure division, COBOL program structure 34–38
- PROCEDURE-NAMES.cpy 74
- procedure-pointers, AccountStorageAccess class (REST Service) 76–77
- programs
 - automated testing 99
 - downloading test examples 100–101
 - end-to-end testing 120–126
 - Interoperation Layer 117–120
 - MFUnit 101–107
 - strategies 99–100
 - running
 - Hello World 5–6
 - sample application 41
- calling COBOL from Java 61–65
- data storage in indexed files 43–53
- generating data 59–61
- Interest-Calculator program 54–60
- programs (COBOL)
 - structure 31
 - data division 32–34
 - procedure division 34–38
- Project Explorer 8
- Project Object Model (POM) 10
- project structure
 - creating Java projects with Maven 12
- promise objects 143
- promise.then() function 143
- properties
 - Visual COBOL 24
- property-id header 25
- props parameter 143
- ProxyWithStreamTest (lambdaProxy project) 206
- public modifiers 24

R

- raise verb 26
- React
 - Add Customer button 146–148
 - AddCustomerForm dialog 148–151
 - application components 138–140
 - CustomerList component 144–146
 - MainPage component 140–144
 - dependencies for the credit-service-form 138
 - development environments 135–136
 - event handlers 152–153
 - extending the application 146
 - philosophy 137–138
 - posting new user details 152–154
 - structure of application 136–137
- readByFileId() method 83
- reading files
 - accessing indexed files 49–51
- reading records, indexed files 52–53
- readRecordById() method 83
- read statements 22
- records
 - AccountStorageAccess class (REST Service)
 - adding, updating, and deleting records 77–79
 - reading and finding records 79–83
 - COBOL programs 33–34
 - reading (indexed files) 52–53
 - writing (indexed files) 52
- Red Hat Enterprise Linux
- Docker installation 156
- re-entrant code 30

- relational databases 43
- remote repositories (Maven) 10
- render() function 140, 151
- repositories
 - Docker 178
 - Maven hierarchy 10
- Representational State Transfer (REST) service.
 - See REST (Representational State Transfer) service
 - @RequestMapping annotation 91
 - REST Assured, automated testing 120–126
 - REST Assured dependency 121
 - @RestController annotation 91
 - REST controller test case 124
 - RESTful Web Service 88
 - REST (Representational State Transfer) Service
 - accessing data
 - AbstractBusinessAccess class 71–75
 - AccountController class 89–92
 - AccoutStorageAccess class 76–83
 - application components 67–70
 - data access and transfer classes 71
 - interoperation layer 70
 - iterators 83–85
 - MonthlyInterest class 85–88
 - Spring Boot 88
 - StatementController class 95–97
 - testing endpoints 93–95
 - WebServiceApplication class 89–90
 - return-code register 36
 - running
 - programs
 - Hello World 5–6
 - run-time environments
 - COBOL dialects 21
 - run-times
 - containers 157
 - run-time, Visual COBOL 93
 - RunUnit.GetInstance() method 92
 - @RunWith(SpringRunner.class) annotation 122

S

- S3 (Amazon Web Services) 174
- scripts
 - containerizing the CreditService 181–183
- secondary keys (ISAM) 30
- secondary keys (records) 43
- Secrets, Kubernetes 190
- sections, COBOL program strucure (procedure division) 34–35
- serverless computing 199–210
 - building and deploying Lambda function 207–208
 - changing the application 199–203
 - configuring API Gateway 208–210
 - Java Lambda function handler 203–206
 - testing code 206
- services, Kubernetes entities 189
- setConnectionString() method 191
- setState() method 143
- setter heading 25
- setup() method 122
- Single-Page Web applications 128–130
- SmartLinkage project 42
- smart linkage, Visual COBOL compiler 31
- sourceformat directive 22
- source format fixed 22
- source format free 22
- source formats
 - Visual COBOL 21–23
 - case sensitivity 23
 - comments 22–23
 - literals 23
 - source format variable 22–23
 - SpringApplication.run() method 89
 - Spring Boot 88
 - @SpringBootTest annotation 122
 - Spring Initializr 88
 - starting
 - Eclipse
 - Linux 8
 - Windows 8
 - StatementCalculator, serverless function 200
 - StatementController class (REST Service) 95–97
 - statements, COBOL program structure
 - copy... replacing 38
 - exit 38
 - goback 38
 - stop run 30
 - static method
 - writing Hello World as a class 7
 - static public method 25
 - statusToString() method 75
 - stop run statement 30
 - storage of data
 - indexed files 43–53
 - accessing files 45–53
 - strategies, testing applications 99–100
 - string literals 23
 - structure
 - COBOL programs 31
 - data division 32–34
 - procedure division 34–38
 - React 136–137
 - structured programming 30
 - stub method 100
 - stubs (test double) 100–101

stubs (Test Double) 100–101
 subgroups, COBOL group items 33
 SUSE
 Docker installation 156
 SUSE Enterprise
 Maven installation 11

T

TDD (test-driven development) 99–100
 TestAccountStorage program 109–110
 test case setup code
 BusinessRules Layer, automated testing 111–112
 Test Doubles 101
 test-driven development (TDD) 99–100
 testing
 changing from ISAM to a database 167–170
 BusinessSystemTests 167
 CreditService tests 172–173
 interoperation tests 170–172
 endpoints 93–95
 serverless computing 206
 testing (automated) 99
 BusinessRules Layer
 account storage test case code 113–114
 interest calculator test case code 114–117
 legacy code 117
 test case setup code 111–112
 downloading test examples 100–101
 end-to-end tests 120–126
 Interoperation Layer 117–120
 MFUnit 101–107
 strategies 99–100
 test suites, MFUnit 102–107
 then() method 143
 toggleAddCustomerDlg() function 152
 TRANSACTION-RECORD.cpy copybook 44
 transfer classes
 accessing data with REST Service 71
 AbstractBusinessAccess class 71–75
 AccountStorageAccess class 76–83
 try... catch... finally blocks
 Visual COBOL 26
 typedef clause 40

U

uber-jar 177
 UI (user interface) modernization
 Credit Service application 130–135
 cross-origin resource sharing 133–135

running the application 132–134
 React
 Add Customer button 146–148
 AddCustomerForm dialog 148–151
 application components 138–146
 dependencies for the credit-service-form 138
 development environments 135–136
 event handlers 152–153
 extending the application 146
 philosophy 137–138
 posting new user details 152–154
 structure of application 136–137
 Single-Page Web applications 128–130
 UI choices 127–128
 unit tests 99–100
 unshift() function 154
 updateAccount() method 77
 updateCustomer test case 120
 user interface (UI) modernization
 Credit Service application 130–135
 cross-origin resource sharing 133–135
 running the application 132–134
 React
 Add Customer button 146–148
 AddCustomerForm dialog 148–151
 application components 138–146
 dependencies for the credit-service-form 138
 development environments 135–136
 event handlers 152–153
 extending the application 146
 philosophy 137–138
 posting new user details 152–154
 structure of application 136–137
 Single-Page Web applications 128–130
 UI choices 127–128

V

value-types
 Visual COBOL object model 23
 version numbers (POM files) 10
 viewing
 Kubernetes application logs 197–198
 VirtualBox 187
 Virtual Machines (VMs) 158
 Visual COBOL 19
 classes 24–25
 COBOL dialects 19–20
 COBOL source formats 21–23
 exceptions 26–27
 managed code 20–21
 modernizing COBOL applications 30–31
 object model 23–24

packages 26–27
source formats
 case sensitivity 23
 comments 22–23
 literals 23
Visual COBOL – A Developer's Guide to Modern CO-BOL (*italic*) 20

Visual COBOL—A Developer's Guide to Modern CO-BOL (*italic*) 2–3

Visual COBOL compiler 31

Visual COBOL Development Hub 3

Visual COBOL for Eclipse 3

Visual COBOL Personal Edition 3

Visual Studio Code 135

VMs (Virtual Machines) 158

W

WebserviceAccountTests class 123

WebServiceApplication class (REST Service) 89–90

when() method 124

windows
 Build Properties 8
 COBOL Explorer 8

Windows
 starting Eclipse 8

working-storage (COBOL programs) 30, 32

writing files
 accessing indexed files 49–51

writing records, indexed files 52

X

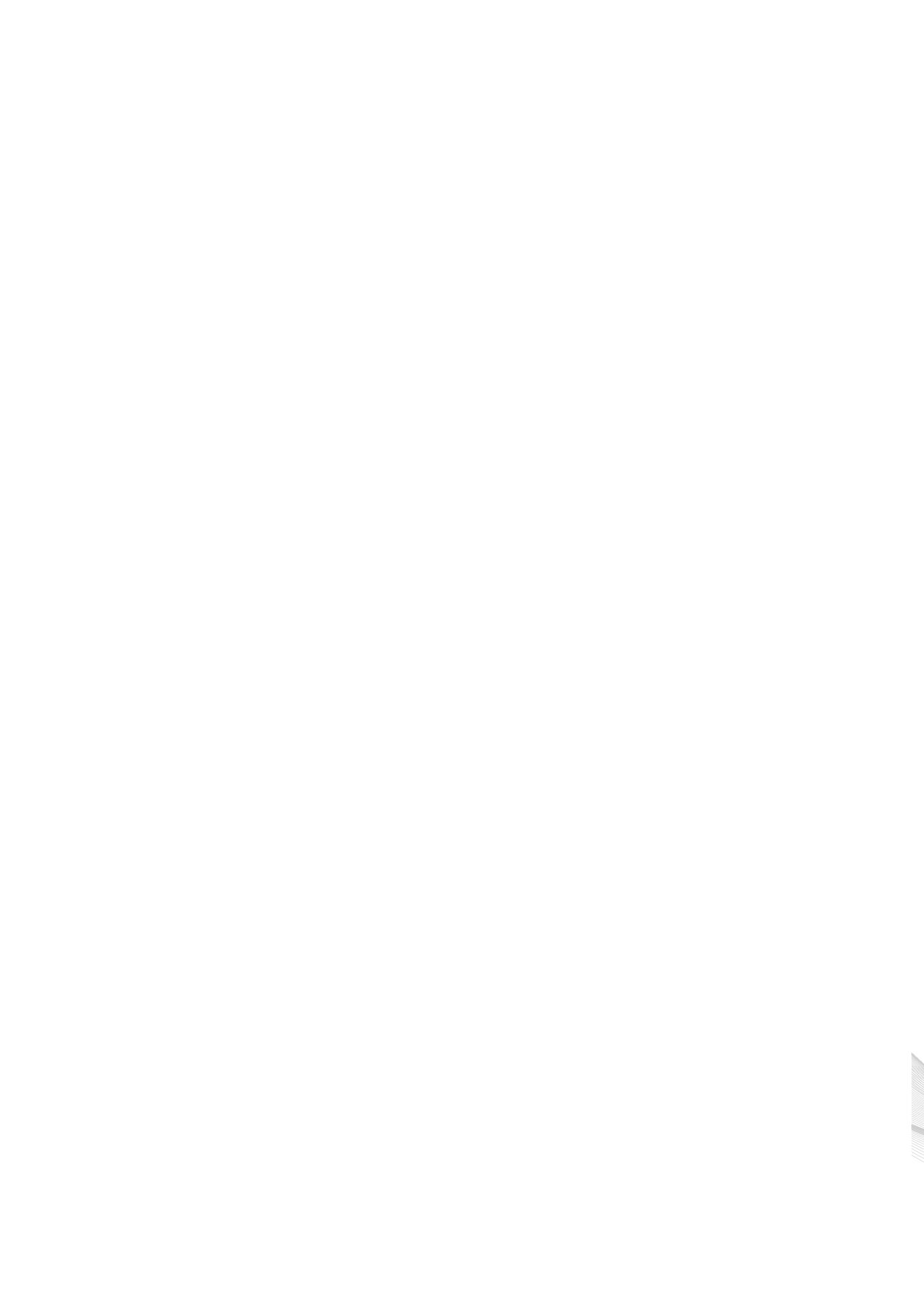
XSS (cross-site scripting) 134

Y

YAML Ain't Markup Language 192

YAML (Yet Another Markup Language) 192

Yet Another Markup Language (YAML) 192





**FREE
DOWNLOAD!**

The Future of COBOL is Here

TUNE YOUR SKILLS AND GAIN EXPERIENCE WITH THE NEXT GENERATION IN COBOL DEVELOPMENT—VISUAL COBOL.

Visual COBOL™ Personal Edition (PE) is here. Work within the industry-standard IDE of your choice. Take full advantage of the new capabilities available to you within this innovative development environment. Visual COBOL PE integrates with Microsoft Visual Studio and Eclipse giving you the choice to develop COBOL applications using the world's most popular integrated development environments. This is your opportunity to learn the enterprise language behind 70% of today's business transactions.

Visual COBOL Personal Edition has the same features as the commercial version. These include:

- Procedural and object-oriented COBOL development
- Build COBOL applications for native code, .NET and the Java Virtual Machine
- Mixed language interoperability with Java and C#
- Code assist tools such as Intellisense and auto-complete
- Visual Studio designer support for creating WPF, Winforms and ASP.NET applications

www.microfocus.com/products/visual-cobol/personal-edition

