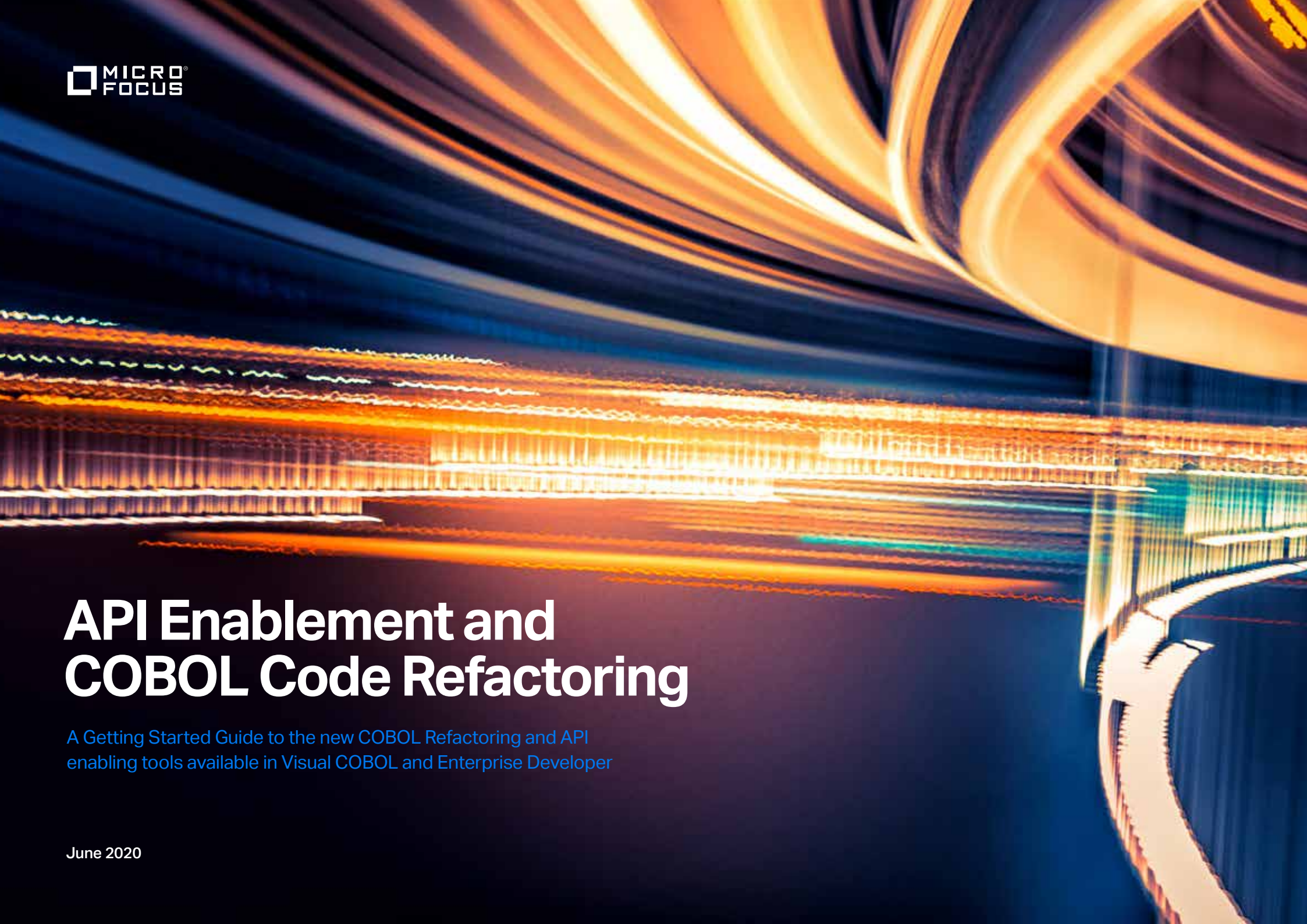


MICRO FOCUS

API Enablement and COBOL Code Refactoring

A Getting Started Guide to the new COBOL Refactoring and API enabling tools available in Visual COBOL and Enterprise Developer

June 2020

About this getting started guide

Welcome to this technical guide, thanks for picking it up. This is a technical guide to help you learn about the new code refactoring tools that are now available in the latest release of Micro Focus Visual COBOL and Enterprise Developer.

Is this guide suitable for me?

This guide is aimed at developers. It is perfect for COBOL developers but also for developers of other languages that want to understand more about modernizing COBOL applications.

If you're have an architect role, you might find this guide useful to skim through to see how this technology could support your modernization plans.

Do I need any experience or skills before I begin?

You'll need to be fairly comfortable using Visual Studio and the Micro Focus COBOL tools. Don't worry if you're not, you can get up to speed with some free training available here.

If you don't know COBOL, don't worry, if you're already a developer, COBOL is perhaps one of the easiest programming languages to learn. If you'd like to learn the language, you'll find free training available here.

What will I get out of this guide?

In this guide, we're going to cover the following aspects of working with modern COBOL tooling:

1. The new refactoring tools available in the latest release of Micro Focus COBOL products and how they can be used to identify and extract business logic from an existing COBOL application
2. How to create a REST API for an existing COBOL application using the Micro Focus REST web services framework
3. How to create an automated unit test for a COBOL program that can be run in a Continuous Integration platform

What software do I need?

You will need either Visual COBOL or Enterprise Developer 6.0. You can get a trial license if you don't already have the software installed

The screenshots and descriptions in this guide are based on Visual COBOL for Visual Studio 2019 but you can also use Enterprise Developer. If you prefer to use Eclipse instead of Visual Studio, you can still do so, although some of the steps and screens will be different but the same functionality is still available.

If you plan to follow along using Visual Studio 2019, you can obtain a trial here. Install this first before installing the Micro Focus software.

You can download Visual COBOL 6.0 here.

Something isn't working, where do I get help?

Let's be honest, if things don't go wrong then you're probably not learning anything. But if you get truly stuck where should you go to get help?

Well, there's plenty of help available here on our Community Website. Sign-up and post your question and we'll be sure to help.

Now, let us begin…

To get started, unpack the .zip file that contains the sample COBOL application we'll be working on.

In the extracted folder, open BookStoreDemo.sln to launch Visual Studio
The solution contains one project called BookStore -- a very simple COBOL application that maintains a stock list of books in an indexed data file.

Within the project there are two programs, a copy file and a data file

bookmain.cbl – provides a simple green screen user interface that allows the user to add, delete or view books in the inventory.

bookstore.cbl – the main functionality of the application that manages the book inventory

book-rec.cpy – a common source file that defines the structure of the data file records

bookfile.dat – an indexed data file that contains the details of a few books

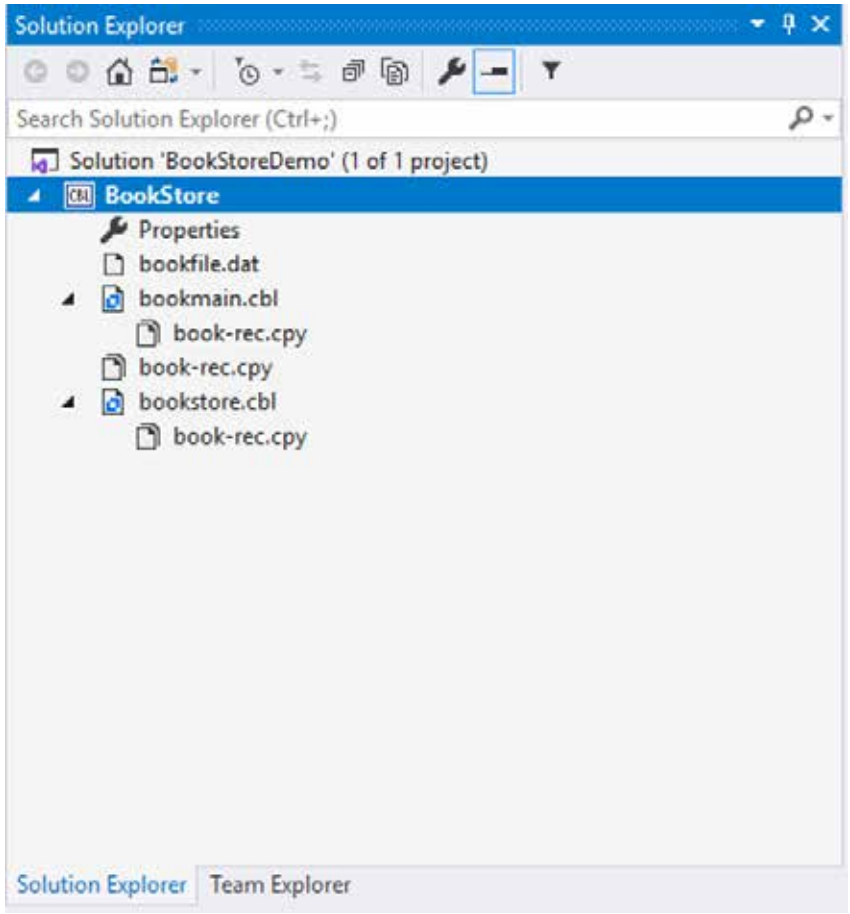


Fig. The solution explorer showing our sample project and code

Run the BookStore application…

Press CTRL-F5 to run the application and see how the application works.

1. Press 4 <enter> to read through the books in the inventory and see their details.
2. Press 2 <enter> and add your favourite book
3. Read through the book list to see it has been added successfully

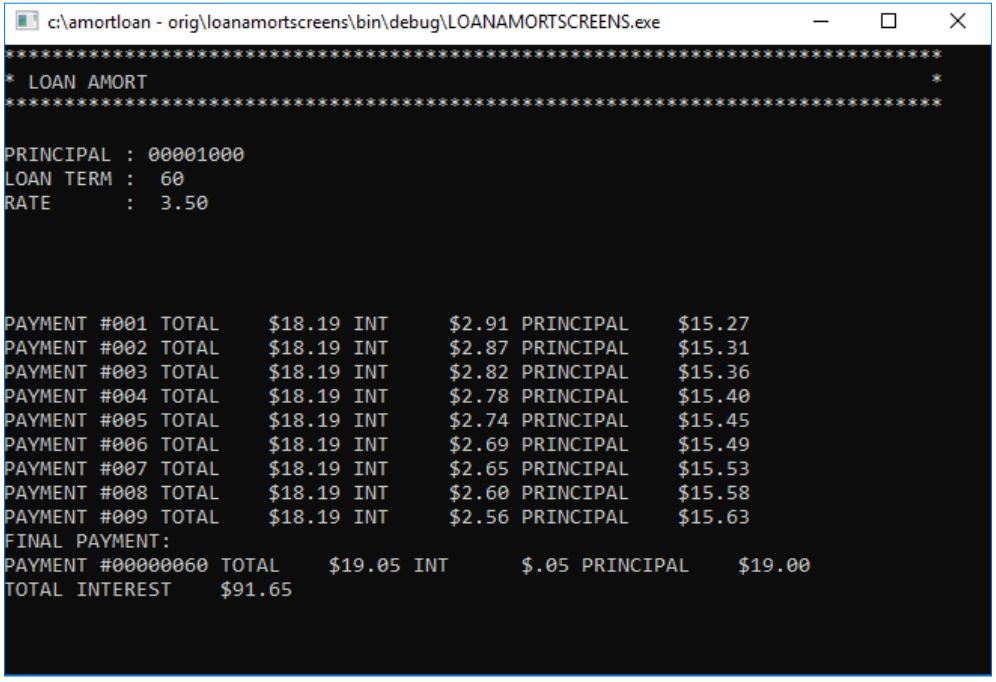


Fig. The main UI for the Book Store, maybe looking a little dated by today's standards

Review the code

Let's take a closer look at the code.

Open bookmain.cbl

This program contains the simple UI and sits in a loop until 9 is pressed. Each command is sent to the subprogram bookstore.cbl to process. The two programs exchange a data structure which holds various fields about a given book depending on the function.

Open bookstore.cbl.

The bookstore program takes in 3 parameters

1. The function code to execute: read, add, delete, etc
2. Data required for the operation e.g. when added or retrieving a book
3. A status code indicating success or error

The main section evaluates the function code requested and then performs the appropriate section to do the job. Each section opens and closes the data file to process the operation and returns information into the main program in the linkage section parameters.

Spend a minute or two examining the code.

Code Refactoring

Refactoring is a coding activity that is often undertaken before commencing any new enhancement to the application. The purpose of refactoring is to prepare the code so that it can accommodate the new enhancement more easily but it's also the case that post-refactoring, the application should still behave and function just as it did before.

Think of refactoring like changing the oil in your car. If you don't do it, the car usually still works, for a time. But eventually, the engine will become dirty, clogged, and sluggish and eventually stop working altogether. In the world of software, it's exactly the same but we usually call this software dirt and grime, technical debt. If it's allowed to build up, it slows down progress and managers then start asking "why does it take so long to get anything done?"

Well, perhaps managers will always ask this question, we probably can't fix this quickly. But we can provide some useful automated tools that developers can use to help reduce technical debt and modernize applications. From small changes like renaming fields, to bigger things like code slicing, we're going to take a closer look.

Getting a lay of the land

Before we start refactoring the code, let's get a view on what we're dealing with. After all, we don't change the oil without knowing what oil the engine needs.

Open the bookstore.cbl and position the cursor on the first line of the procedure division

Bring up the editor context menu and select the "Show Program Flow Graph"

The program flow chart helps you see what programs are called and what sections are performed in the program. In our sample, it's a fairly trivial matter. For larger programs, you can quickly explore the application and focus the chart on the context you're interested in.

Try selecting and hovering over different nodes in the chart to see the code.

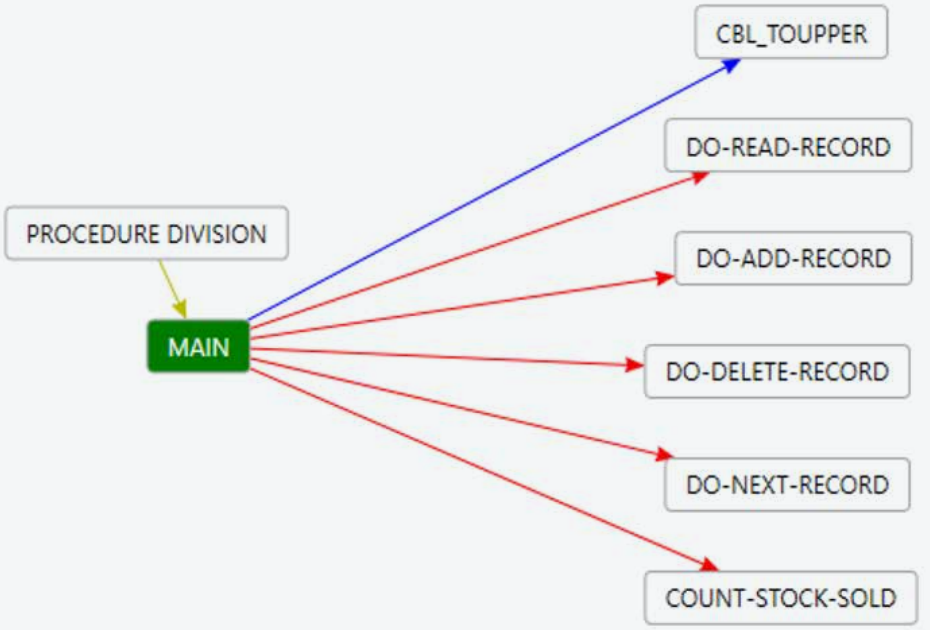


Fig. The Program Flow Graph for the Book Store program

Simple, simplification…

Let's explore some of the refactoring tools.

Locate the Do-Read-Record section and take a closer look at the evaluate statement.

Next, locate the do-delete-record section and look at the evaluate statement there.

Notice we have some duplicated code. Let's pull this code out into a new section and update the existing code to use our new section.

Go back to the do-read-record section and select the evaluate block up until the end-evaluate

Bring up the context menu, click "COBOL Refactoring" and select "Extract to Section"

The Visual Studio editor will move the code into a new section. You just need to give the new section a name.

To name the section, just type something appropriate like, read-book-file <enter>.

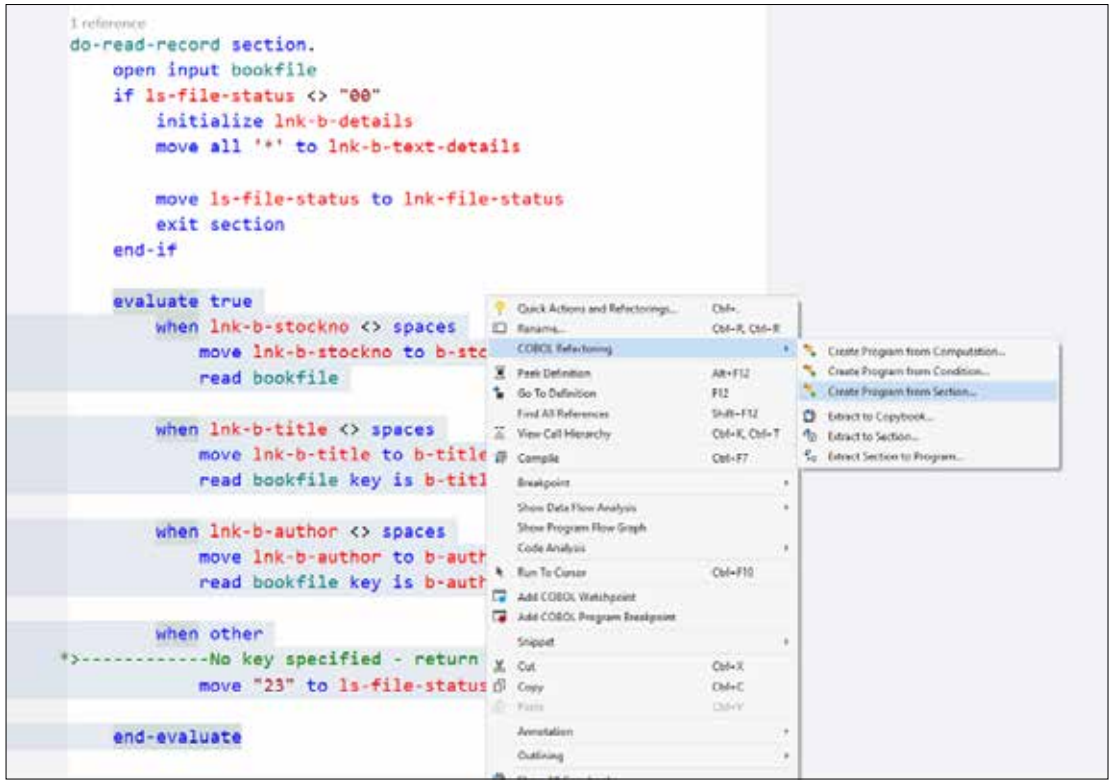


Fig. Extracting code into a new section using the refactoring tools

Now that we've moved the code into a new section, we can delete the evaluate block from the do-delete-record and replace this with a PERFORM to the new read-book-file section.

Extracting code into a new section is a simple refactoring tool that you can use not only to deal with duplicated code but just as useful to break down large sections into smaller units. If it makes sense, you can extract the code into a copybook instead.

Try the Program Flow Graph again, it should look something like this

In the next step, we'll take a look at one other simple refactoring tool that makes all the difference when you're faced with code you didn't write…

Fig. The Program Flow Graph now shows the new section

Breaking down the monolith

Generally speaking, COBOL systems are huge. With decades of development and engineering investment they run into millions of lines of code. Often, these applications are described as monolithic—meaning that one program serves many functions. Decomposing monolithic applications into smaller pieces can help simplify maintenance tasks, make the job of writing automated tests feasible and opens up options to use the smaller application pieces in new ways, such as APIs. Our sample application is hardly a monolith but we can use it to see how we could break down larger systems, write automated tests and even APIs. This is what we'll do next.

Making independently callable programs from sections

In this task, you're going extract a portion of logic from the bookstore program and move that into a self-contained COBOL program. We'll also update the original bookstore program to use the new program. Sounds like a lot of work? Well, yes it can be—especially for big programs. But this next refactoring tool will help do most of the heavy lifting, let's see how it works.

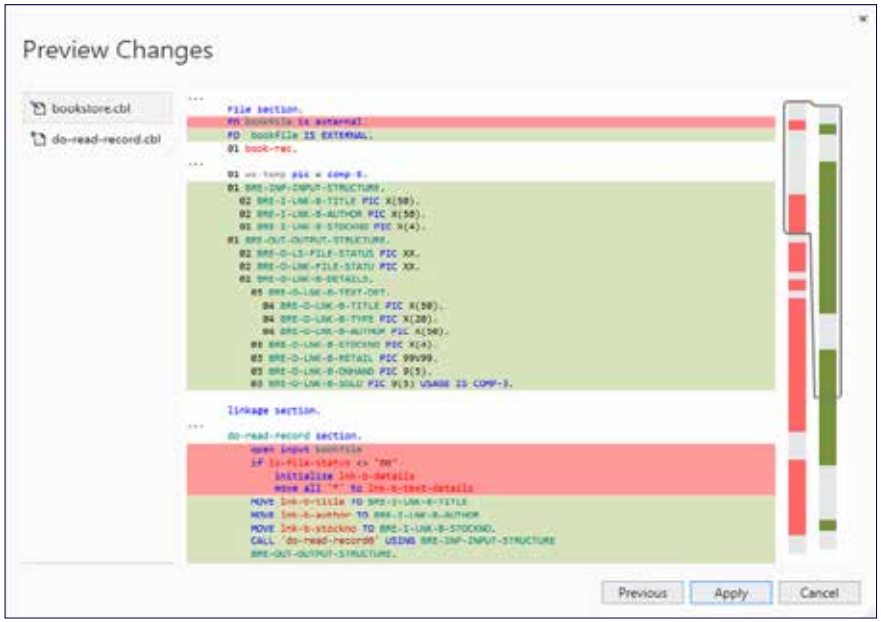


Fig. The COBOL code refactoring preview

Locate the do-read-record section

From the COBOL Refactoring context menu, select Extract Section to Program...

In the dialog box, click next to see a preview of the code changes

Study these if you like but we'll review the code changes next anyway.

Click apply to complete the refactoring

A new program has been created called *do-read-record.cbl*

The original bookstore.cbl program has been updated to call do-read-record. Let's take a closer look, we'll start with do-read-record.cbl.

If you scroll through this program you'll notice that all of the fields and sections needed by the original code have been brought into this new program. This includes the file definition and working storage fields. In addition, you'll also see that a linkage section has been created that allows the caller to pass in the parameters required and for the result to be returned.

In the procedure division:

- The first PERFORM statement is unpacking the input fields into working storage
- The second statement performs the actual processing
- The final perform statement, moves the working storage items into the linkage section.

Whether you're working on a small program like this one or something more complex, the same pattern will be applied by the refactoring tools.

If you scroll further down in the code you'll see the original section we extracted and even the new read-book-file section we created in a previous step. All-in-all, we have a self-contained program that will read a record from the file.

Make sure the program still runs as you expect.

Hit F5 to build and run the application

Test the READ function to ensure you can see the records in the file.

```
PROCEDURE DIVISION USING BRE-INP-INPUT-STRUCTURE
    BRE-OUT-OUTPUT-STRUCTURE.
0 references
main section.

    PERFORM BRE-COPY-INPUT-DATA-0.
    PERFORM PAR-DO-READ-RECORD.
    PERFORM BRE-COPY-OUTPUT-DATA.
    GOBACK.
```

Fig. The linkage section of the newly created program.

Wait a minute! I've changed my mind

Whenever I go out for dinner, I need to look at everything on the menu, twice, and still end up changing my choice. Well, if you also end up changing your mind about the refactored code, don't worry—you can restore it. Let's try it out.

Hit CTRL-Z to UNDO your changes.

The original code in the bookstore.cbl program is restored to before the refactoring.

Hit CTRL-Y to reinstate the refactoring.

When you finish up here, leave the code WITH the refactored do-read-record code in place.

Code slicing

In the previous step, we extracted code from the bookstore program into a separate program and updated bookstore to use the new code. In this step, we'll explore a new type of refactoring called code slicing.

We use code slicing to pull out useful business logic from an existing program but we leave the original intact. This can be useful for different purposes as we'll come to see.

There are 3 type of code slicing operations you can perform:

1. Section slicing – just like the previous step except we don't modify the existing program
2. Conditional slicing – we extract only the code that would execute if a field contained a specified value
3. Computational slicing – we extract the code needed to calculate a field at a particular point in the code

To a larger extent, we've already covered the section slicing operation, so let's move onto conditional slicing.

Conditional Slicing

Open up bookstore.cbl

Locate the definition of lnk-function in the linkage section

Place the cursor on the field name and bring up the context menu

Click COBOL Refactoring then Create Program from Condition

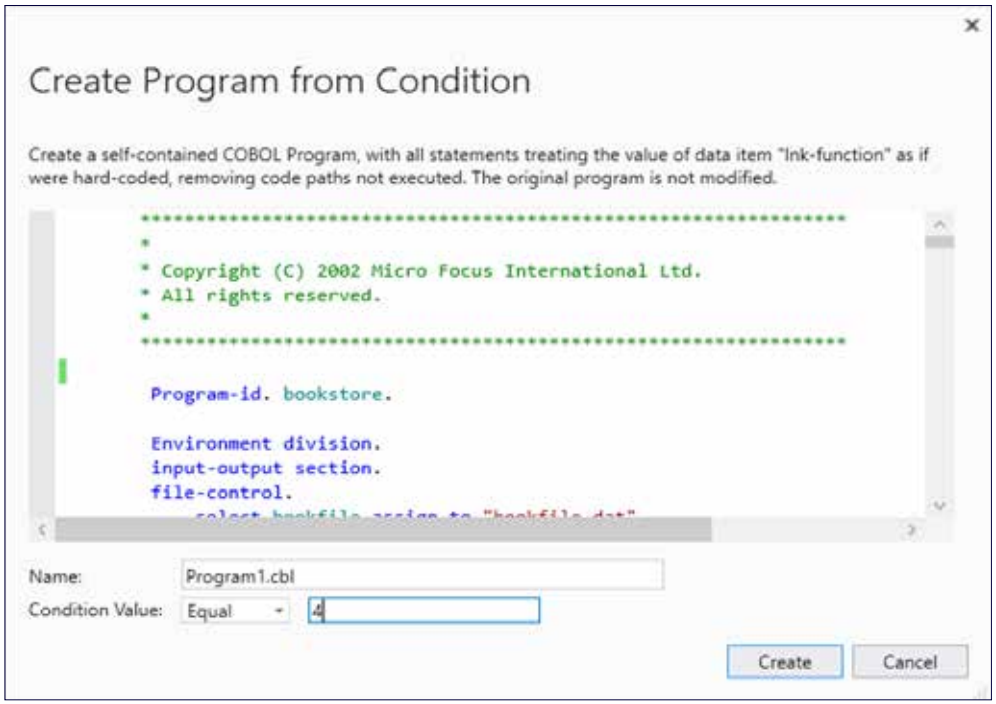


Fig. The Conditional refactoring tool

Notice the dialog refers to the lnk-function field as the subject. You can change the program name if you wish.

In the condition value input field, input 4. This corresponds to the READ-NEXT book operation.

Click Create to slice out the code

The finished result is a new program that includes only the code that would have be executed if the original bookstore program had been called with the READ-NEXT operation.

The original linkage section of the bookstore program has been preserved in the code slice. The idea here is that it should then be easier for you to reuse as the interface matches code you will already have. But it is certainly possible to perform further manual refactoring of the code. For example, you really don't need the lnk-function parameter as it's not used anywhere else in the program.

We're not going to use this code slice elsewhere in this guide. You can leave it in the project or delete it if you wish.

Using conditional slicing to remove obsolete code

Over time, some code in an application will become redundant. Take for example, an application that is cloned for different businesses and share much code in common but also have customized aspects. Sometime this customization is done in the code and over time, these customizations can become obsolete.

Recognizing this has happened is one thing but then removing it is altogether another task.

Open the bookstore.cbl file and locate the do-add-record section

Notice specific code that checks the store-id and then makes adjustments to the book price. However, all of these stores are no longer in existence and this code will now never be executed.

Invoke the conditional slicing refactoring tool once more but this time select the store-id field

Leave the value blank and create the new program

If you examine the code, you'll see that the special tax code is no longer present in the new program.

This can be a useful way to remove redundant code from the application and cut back on the amount of code you thought you needed to maintain.

Computational Slicing

Within your own applications, there will be parts of the code that are responsible for calculating an important piece of information which is then stored in a field. It could be useful to provide that same calculated value for other purposes but it's currently trapped beneath a mound of application code and isn't easily accessible without running through a lot more of the application. This is where computational slicing can help.

Locate the count-stock-sold section

Position the cursor on the last line which updates the lnk-books-sold field

Invoke the COBOL Refactoring context menu and select Create Program from Computation…

Click the Create button and review the generated code

You can see that a new program has been created which includes the code path needed to calculate the total number of books sold by the store.

Note how extraneous code is generally removed from the resulting program. For example, the count-stock-sold section performed a separate section that displayed a message to the user. This code was not included in the new program as it did not affect the result of the calculation.

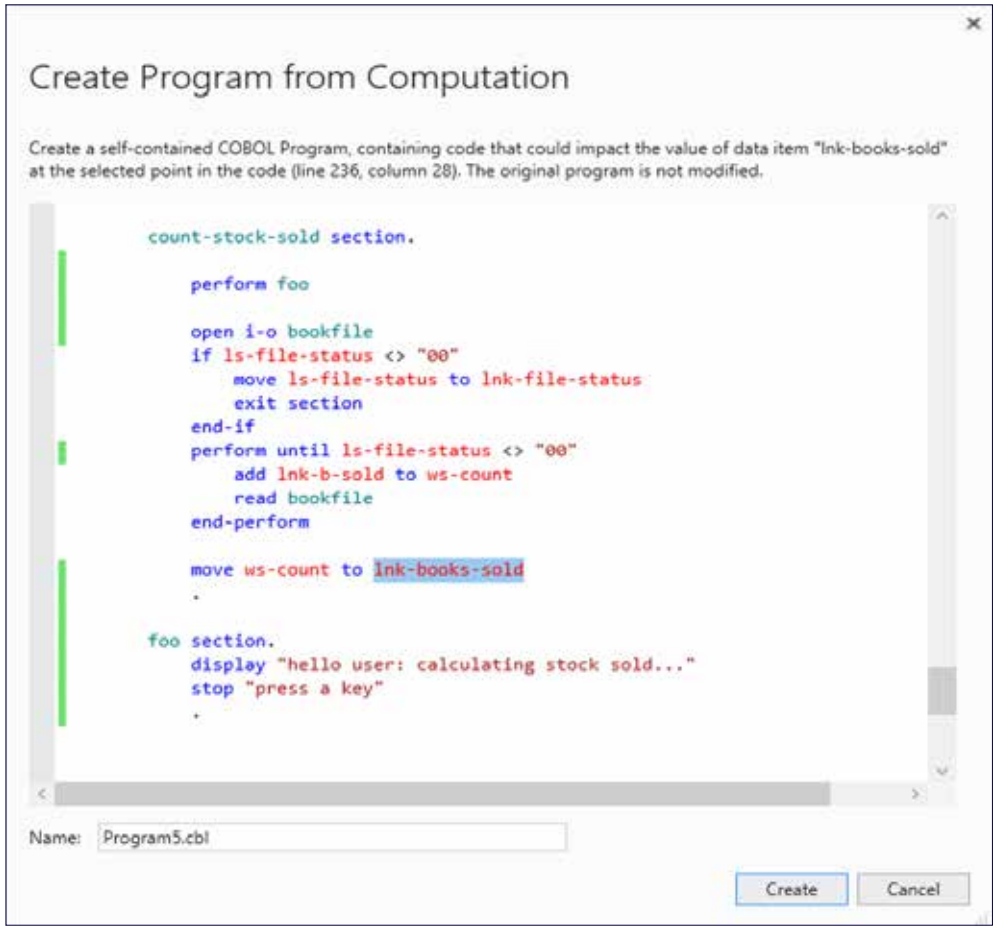


Fig. The Computational Refactoring tool

Dead code, your days are numbered.

Another consequence of long running applications, is dead code. Code can get side lined without engineers ever realising. This code soon mounts up and can even incur unnecessary but costly maintenance cycles.

So before we move on any further, why don't we clean up the code and see what's no longer needed.

Open the bookstore.cbl program

Using the context menu, select the Code Analysis option and then 'Within Entire Program'

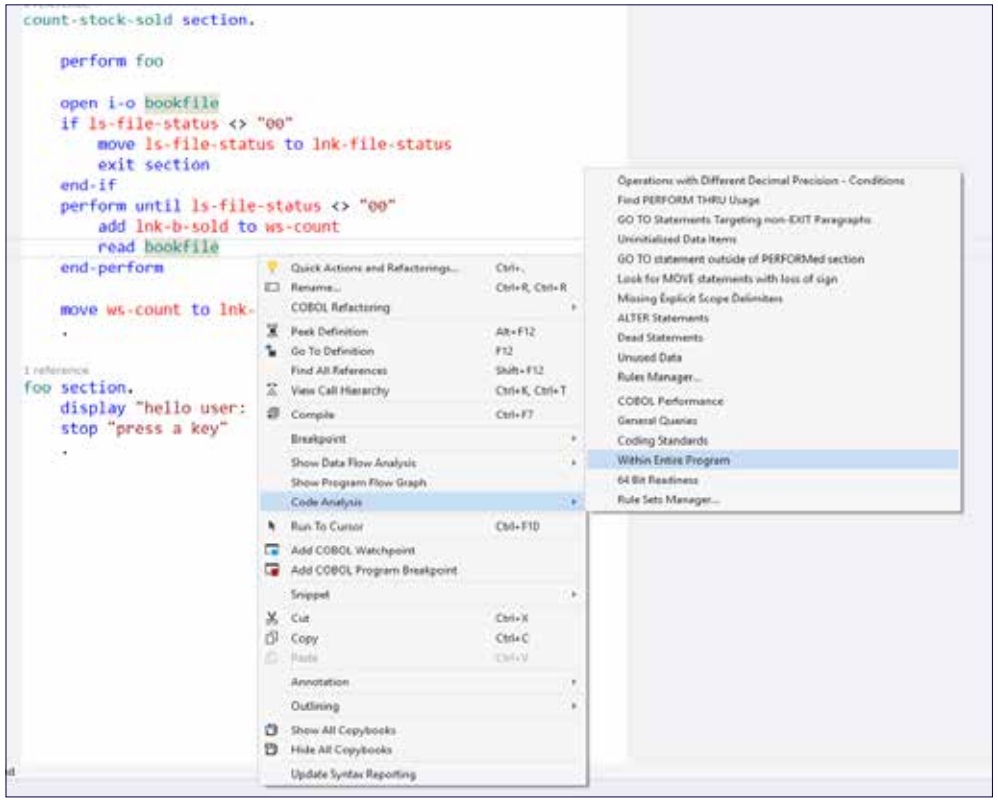


Fig. Running the code analysis tools

This will run a set of general rules against the code looking for a variety of things that could be of interest to the engineer, including dead code. The result of the tool is shown in a separate window and code is highlighted in the editor if there is something to look at more closely.

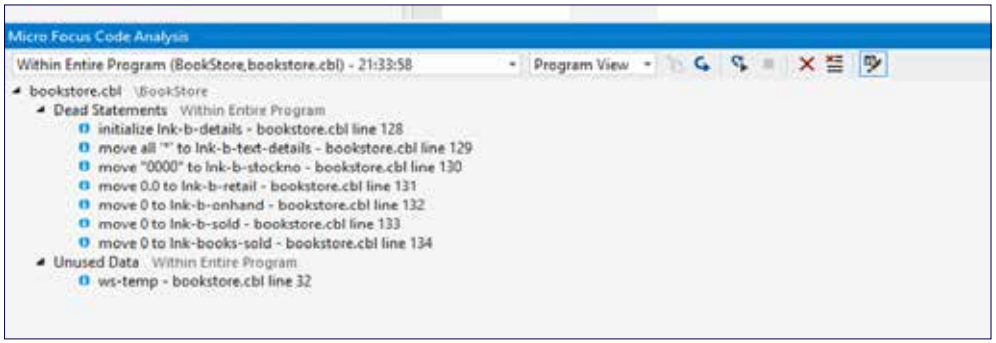


Fig. Results from the code analysis

You should see results that indicate both dead code and dead data, unreachable in the execution of the application -- this includes an entire section of code. You can safely delete this code and rerun the analysis.

```
           when other
*>------------No key specified - return unsuccessful read
                move "23" to ls-file-status

       end-evaluate.

0 references
init-book-details section.
       initialize lnk-b-details
       move all '*' to lnk-b-text-details
       move "0000" to lnk-b-stockno
       move 0.0 to lnk-b-retail
       move 0 to lnk-b-onhand
       move 0 to lnk-b-sold
       move 0 to lnk-books-sold.

1 reference
do-next-record section.
       open input bookfile
       if ls-file-status <> "00"
           initialize lnk-b-details
           move all '*' to lnk-b-text-details
           exit section
       end-if
```

Fig. Dead code found by the analysis tools

Creating an API from a Code Slice

Now that we've learned how to extract business logic from an application, let's do something new with it. In this step, we'll extract some code and use it as the basis of an API. The API will be implemented as a REST web service. The COBOL product you're using comes with an inbuilt web services framework, so you can design and host the API.

Let's start by extracting the code the will be the subject of the API – the next record function.

Open the bookstore.cbl file

Locate the do-next-record section

Invoke the COBOL refactoring menu and this time, select the Create Program from Section… option.

Remove the hyphens from the new program name, call it donextrecord.cbl

This new program will be stored in your existing BookStore project. We'll come back to it later.

For now we need to create a new project for the API itself.

Creating a Project for your API code

From the File menu, select Add then New Projects...

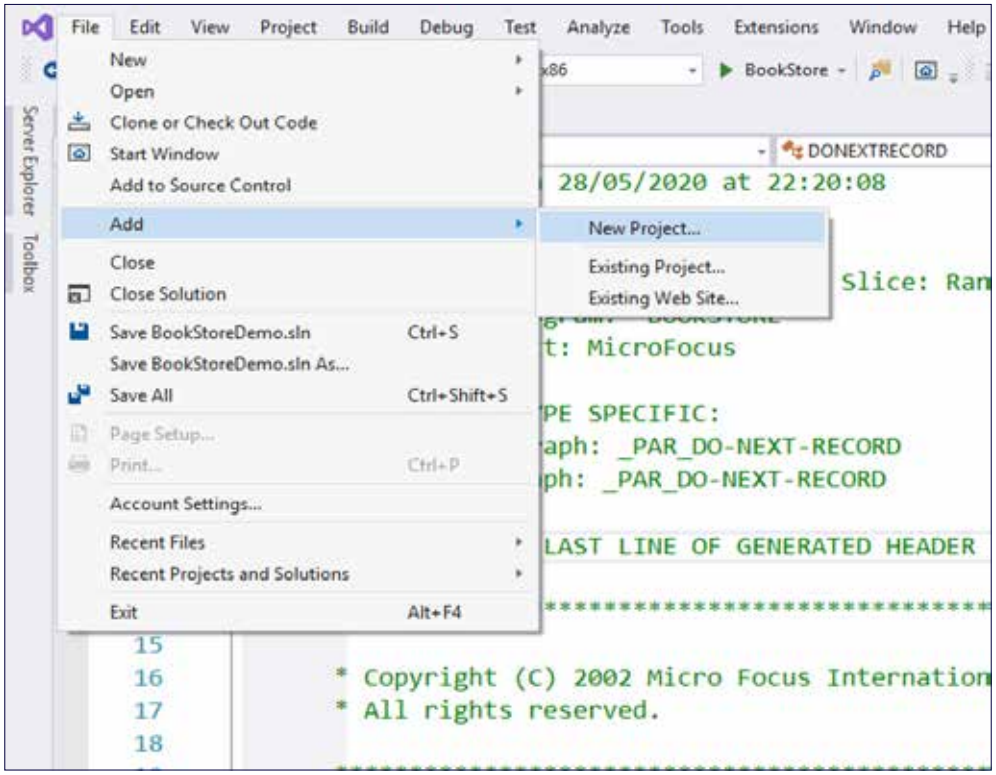
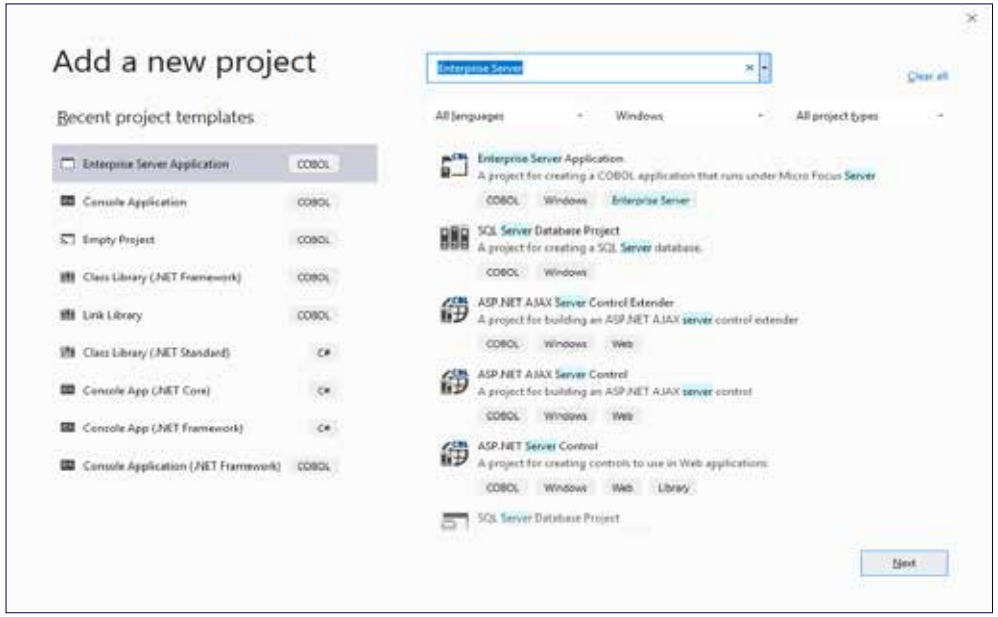


Fig. Creating a new project

From the Add a new project dialog, search for Enterprise Server in the list Select and click next



Name the new project BookAPI and place it in the same folder as your exiting COBOL project

Click create

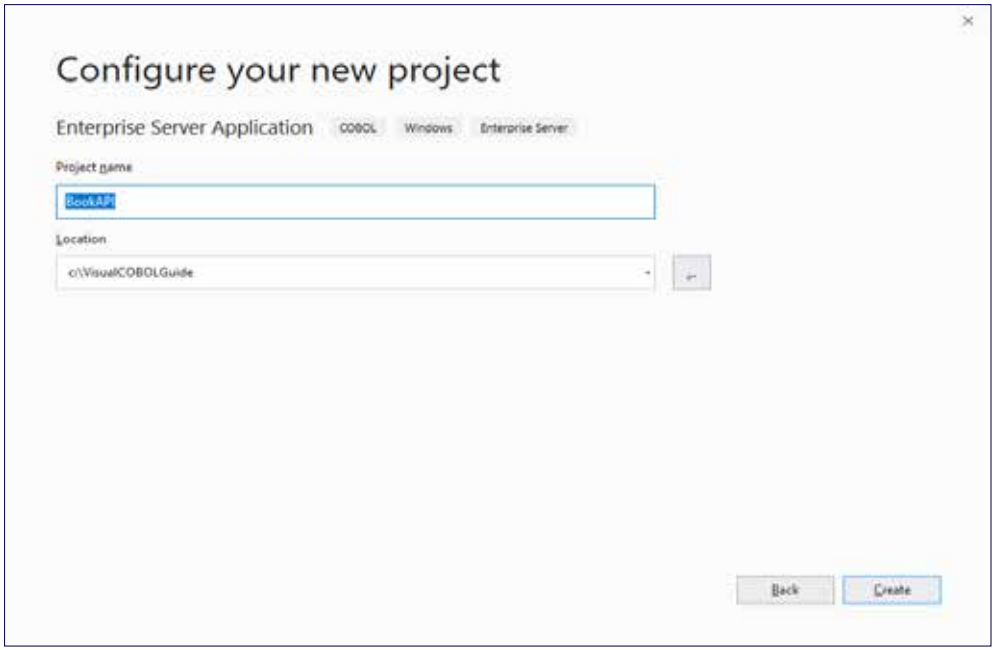


Fig. setting up your new project

The solution explorer should now look like similar to this, with a new BookAPI project, currently empty.

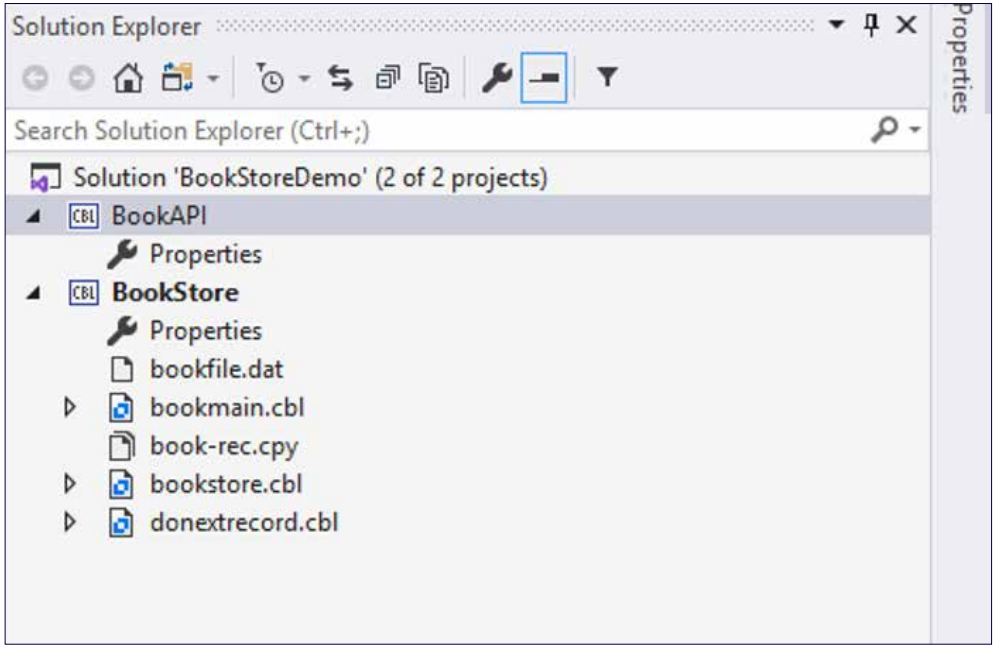


Fig. Solution explorer now showing the new BookAPI project

We now need to move our code slice from the BookStore project into the BookAPI project.

Select the donextrecord.cbl, right-click and select copy

Then select the BookAPI project, right click and select paste

Do the same for the bookfile.dat, and book-rec.cpy, paste these into the BookAPI project

You're free to delete the donextrecord.cbl file form the BookStore project, leaving only the instance of it in BookAPI

The result should look like this

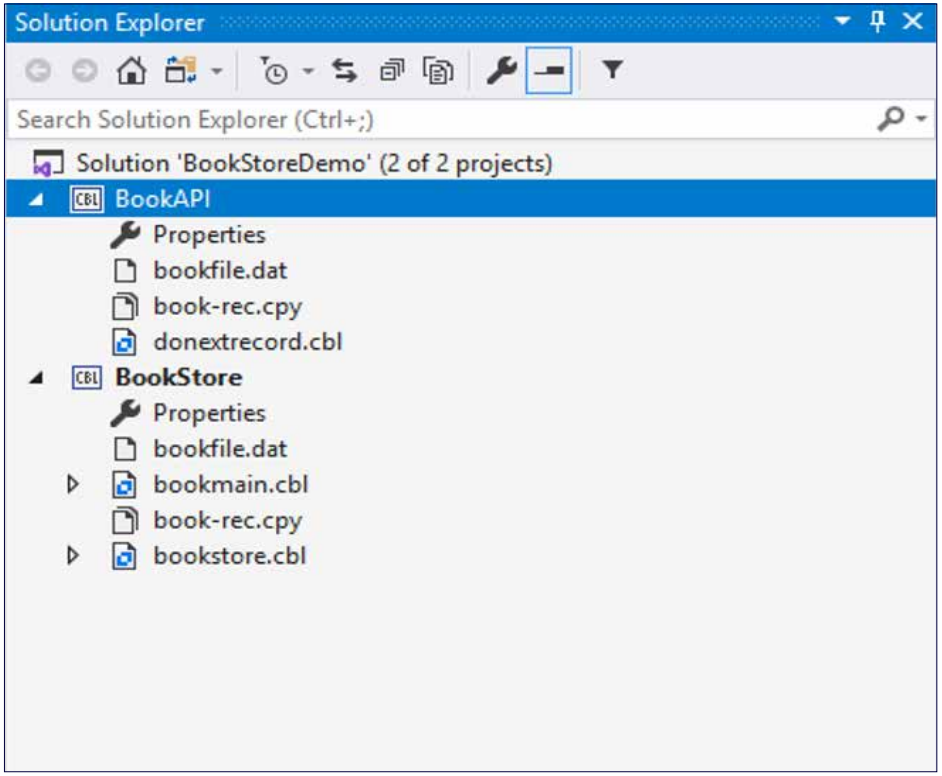


Fig. Results after moving files to the new project

Click Build Menu, click Build Solution… to make sure everything is still compiling

Let's talk testing

Before you set about creating a REST API for your donextrecord program, you're going to write a non-optional automated test. That's right, this isn't optional. Remember, one of the reasons why you had to refactor is because your application became too big, monolithic. That made it difficult to test and not being able to test the application means you lack confidence when making code changes… and things then start to slow down and before you know it, the boss is asking why things are taking so long.

Well, let's start out on the right foot and create some automated unit tests for what will be our API.

Open donextrecord.cbl in the BookAPI project

Scroll down to the procedure division, bring up the context menu and choose Create Unit Test.

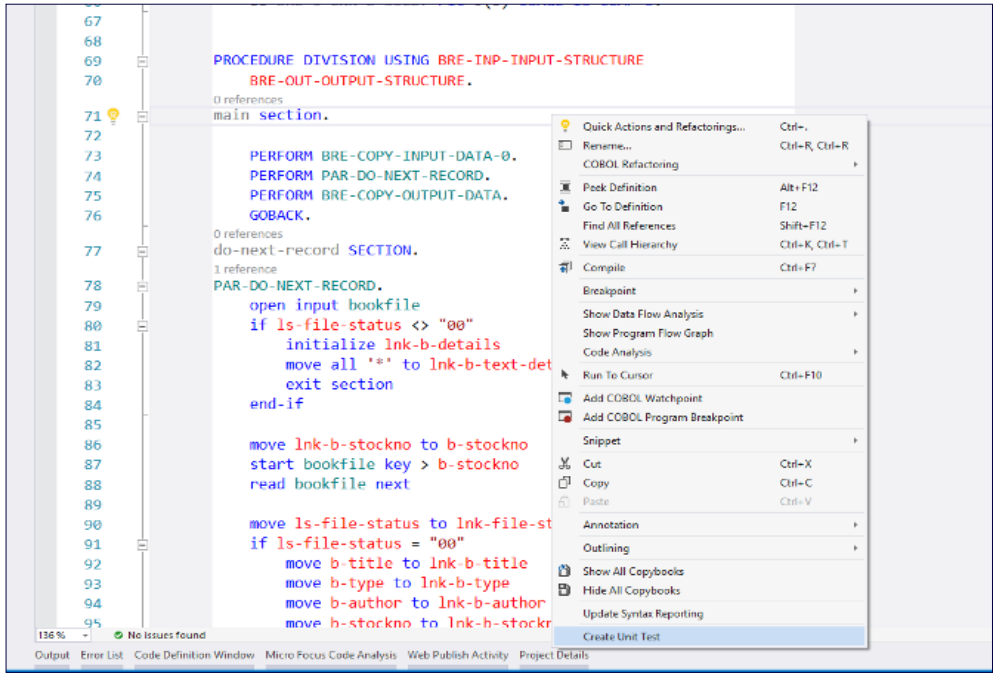


Fig. Creating a unit test

Accept the defaults in both of the next dialogs and click Finish to create a new project

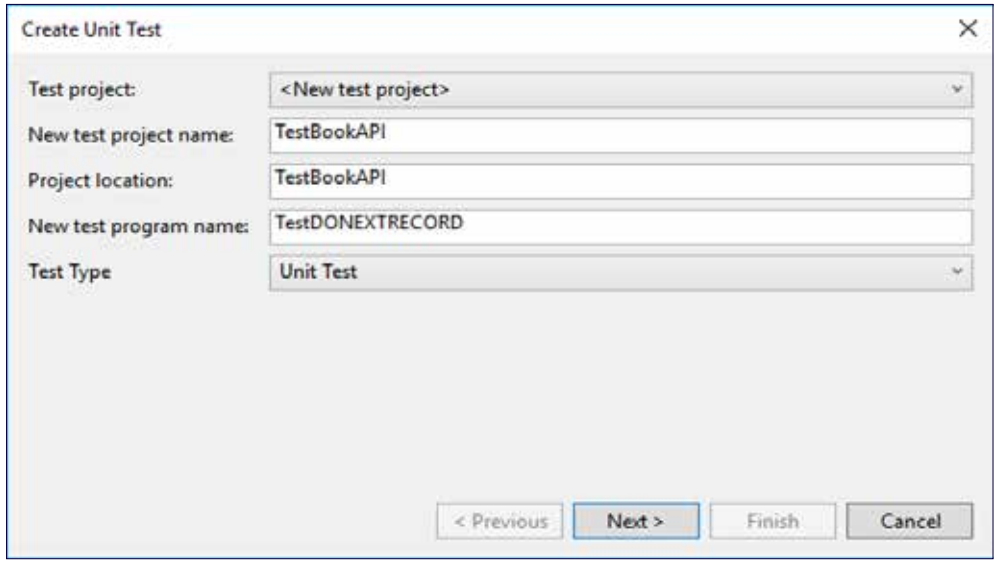


Fig. Configuring the unit test

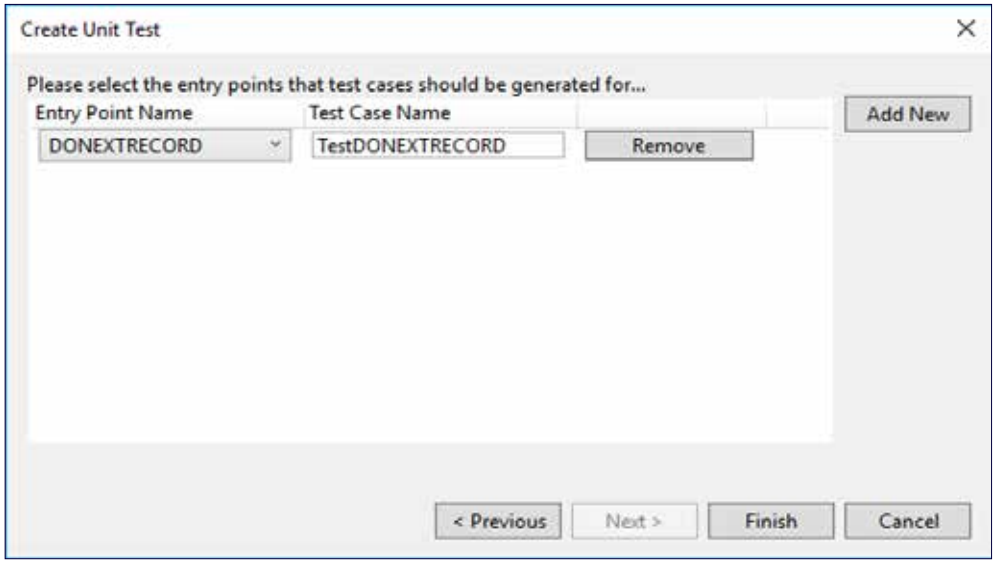


Fig. Finishing-up with the unit test configuration

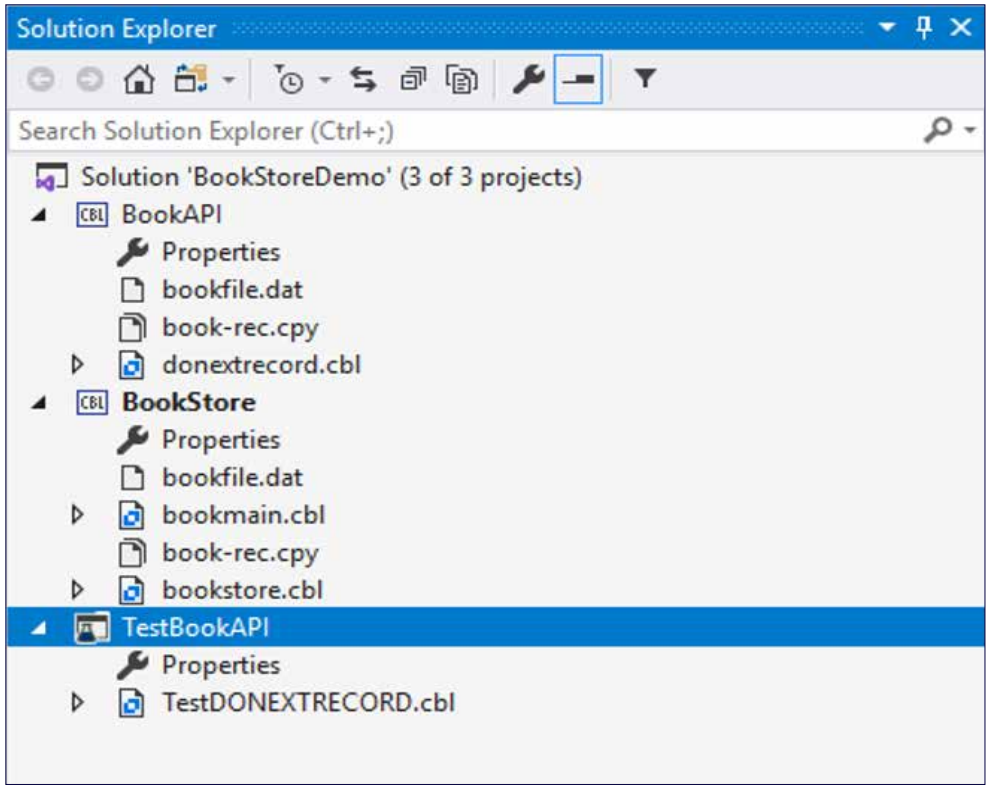


Fig. The new unit test project in the solution

You should now have a 3rd project which contains a single project called TestDONEXTRECORD.cbl. This is program has been automatically generated as a test harness for our donextrecord.cbl program. Open it and you'll see it includes working storage items needed to call the program we are going to test and 2 entry points.

The entry statement is a bit like having another PROGRAM-ID but within a single source file – you can CALL entry points just like you can CALL a program and like a program, each entry point needs a name. In this case, the entry point name is defined by constant values in the program. If you hover the mouse over the MFU-TC-PREFIX and TEST-TESTDONEXTRECORD you'll see that the actual name of the first entry point amounts to MFUT_testDONEXTRECORD. MFUT equates to Micro Focus Unit Test and the rest is the program you're testing.

This product includes a unit testing framework that allows you to write unit tests against your programs which you can then run inside Visual Studio but also as standalone test cases that can run as part of a continuous integration system. The whole point here is that you can write a set of test cases that will help identify a problem if another developer comes along and changes your application code at a future date.
Apparently, this happens – other developers break your hard work.

The first entry point in the code, is a single, automatically generated test case for donextrecord.cbl. At this point, it doesn't do a whole lot – just calls the program under test.

The second entry point is used to initialize anything you need before the test runs. As you'll see, you can include multiple tests within a single program and this is a way to configure your test cases.

```
entry MFU-TC-PREFIX & TEST-TESTDONEXTRECORD.

    call "DONEXTRECORD" using
                by reference BRE-INP-INPUT-STRUCTURE
                by reference BRE-OUT-OUTPUT-STRUCTURE

    *> Verify the outputs here
    goback returning MFU-PASS-RETURN-CODE
.

$region TestCase Configuration

entry MFU-TC-SETUP-PREFIX & TEST-TESTDONEXTRECORD.
perform InitializeLinkageData
    *> Add any other test setup code here
    goback returning 0
.

1 reference
InitializeLinkageData section.
    *> Load the library that is being tested
    set pp to entry "DONEXTRECORD"

    initialize BRE-INP-INPUT-STRUCTURE
    initialize BRE-OUT-OUTPUT-STRUCTURE
    exit section
.
```

Fig. The automatically generated code

Let's code up a test by modifying the current test case to do something meaningful. Code up the test case as you see in the next screenshot.

This test case ensures that a specific book is returned given an input stock value.

On return from the call, we check the file status is okay and the expected stock code is found. If either of these cases aren't true we tell the test framework there was an unexpected condition and we exit the test.

If you're not familiar with the exhibit verb in COBOL, it displays both the name of the field and its value, quite useful in this case.

Finally, we display a passing statement.

```
entry MFU-TC-PREFIX & TEST-TESTDONEXTRECORD.

    move "2222" to BRE-I-LNK-B-STOCKNO

    call "DONEXTRECORD" using
                by reference BRE-INP-INPUT-STRUCTURE
                by reference BRE-OUT-OUTPUT-STRUCTURE

    if BRE-O-LNK-FILE-STATU <> "00"
        CALL MFU-ASSERT-FAIL-Z using "File status not successful" & x"0"
        exhibit BRE-O-LNK-FILE-STATU
        goback returning MFU-FAIL-RETURN-CODE
    end-if

    if BRE-O-LNK-B-STOCKNO4 <> "3333"
        CALL MFU-ASSERT-FAIL-Z using "Unexpected book returned" & x"0"
        exhibit BRE-O-LNK-B-STOCKNO4
        goback returning MFU-FAIL-RETURN-CODE
    end-if

    display "DONEXTRECORD retrieved correct book code " BRE-O-LNK-B-STOCKNO4
    *> Verify the outputs here
    goback returning MFU-PASS-RETURN-CODE
.
```

Fig. A complete unit test

Run your test case

Let's run the test case within Visual Studio to check it's successful.

If it's not already visible, make sure the Micro Focus Unit Testing tool window is visible. If you can't see it, enable it from the View menu.

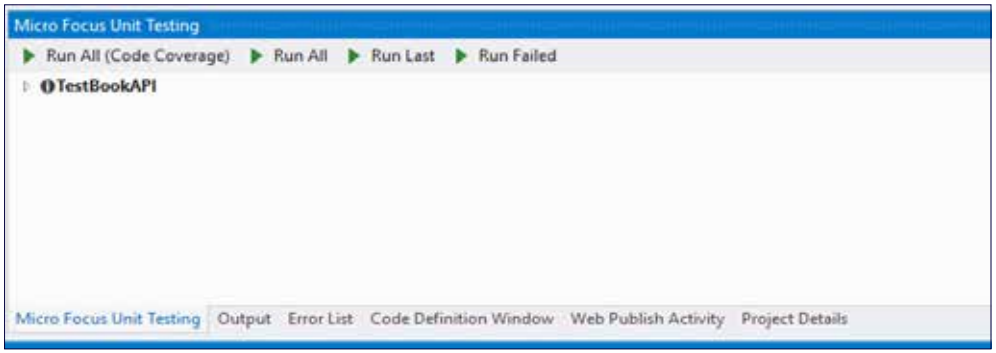


Fig. The Unit Testing window

Click the Run All button – not the Run all (Code Coverage), we'll come to this later.

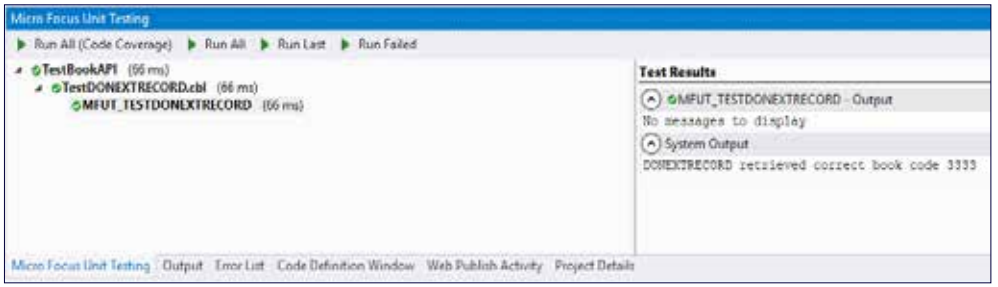


Fig. The output from running the test

When you click Run, the IDE will build your test project and execute the MFUT entry points. Everything should be green and passing. If you expand the test case in the Test window you can see the passing result and anything we displayed during the test is also shown.

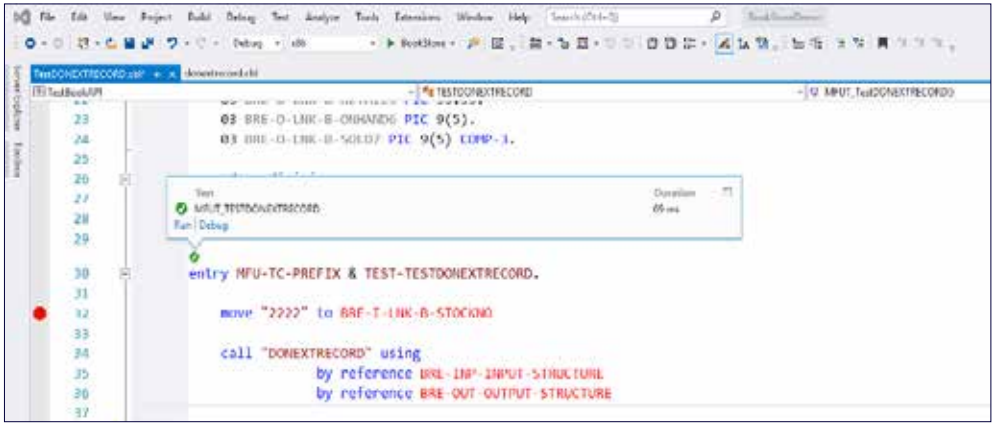


Fig. Running the test using the editor adornment

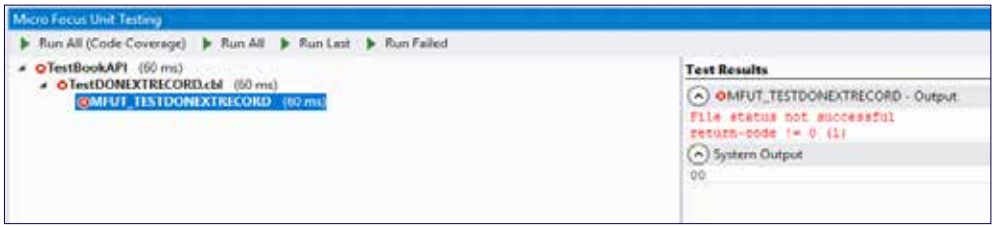


Fig. A failing test

To give you an idea of what a failing test will look like, let's make a quick change to the test.

Modify the IF statement that checks the file status so that it would fail, for example, test for "11" instead of "00"

To rerun the test, this time use the editor adornment above the test case. Hover over the green tick and click run.

If you have a whole bank of tests in red like this, you've definitely had a bad day in the office.

Restore the test case to a passing state.

One test isn't nearly enough

Our program contains only one test. Surely, we can do better than that. And we can, and we should and not just because it's easy to do.

Here's a new test case I created by copying the exiting test, pasting it into the same source file and renaming the entry point name. In this test, I'm checking to see that we get the right file status returned when we pass in a stock code that doesn't exist. Now one might argue a 2/3 is a better result than a 4/6 file status but this is the way the program works today. You are of course entirely free to modify the code in the program under test.

Once you've coded this up, run all the tests to ensure they pass. Here's my output.

```
entry MFU-TC-PREFIX & "TESTNotFound".

    move "5555" to BRE-I-LNK-B-STOCKNO

    call "DONEXTRECORD" using by reference BRE-INP-INPUT-STRUCTURE
                              by reference BRE-OUT-OUTPUT-STRUCTURE

    if BRE-O-LNK-FILE-STATU <> "46"
        CALL MFU-ASSERT-FAIL-Z using "File status not expected" & x"0"
        exhibit BRE-O-LNK-FILE-STATU
        goback returning MFU-FAIL-RETURN-CODE
    end-if

    if BRE-O-LNK-B-STOCKNO4 <> "    "
        CALL MFU-ASSERT-FAIL-Z using "Unexpected book returned" & x"0"
        exhibit BRE-O-LNK-B-STOCKNO4
        goback returning MFU-FAIL-RETURN-CODE
    end-if

    *> Verify the outputs here
    goback returning MFU-PASS-RETURN-CODE
```

Fig. Another test case

Fig. Passing tests, yay!

Code Coverage

Before we leave our testing, there's one more thing to do. Let's check how much of our program code we're actually testing.

This time, rerun the tests using the Run All (Code Coverage) button. You'll see this message box.

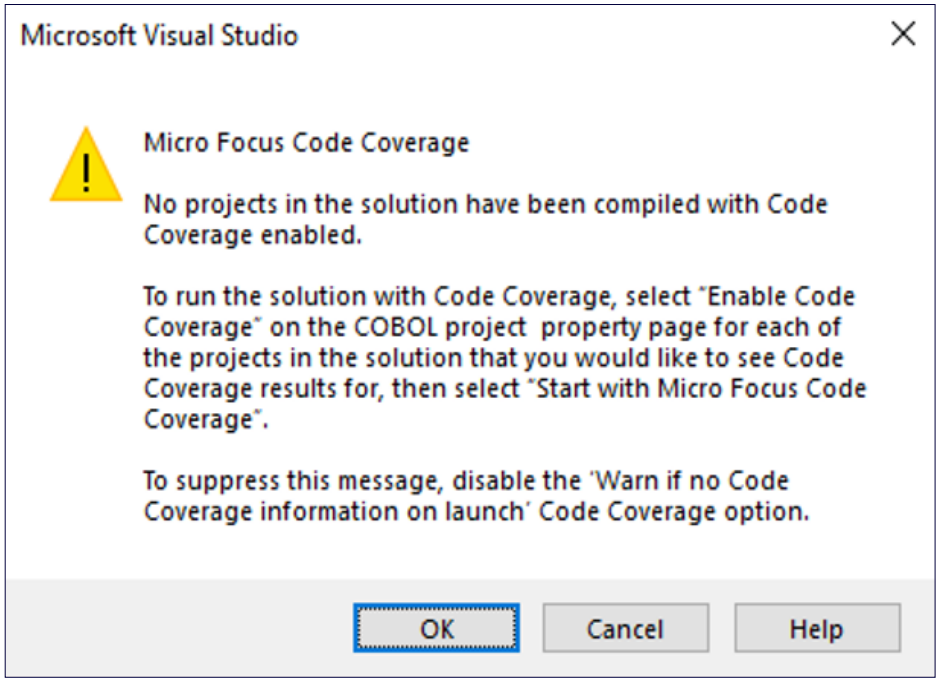


Fig. Code coverage didn't work!

Let's follow what it says.

Click cancel

In the BookAPI project in Solution Explorer, double click the Properties item to bring up the project properties pages

Make sure the COBOL tab is select in on the left and the check the Enable Code Coverage checkbox

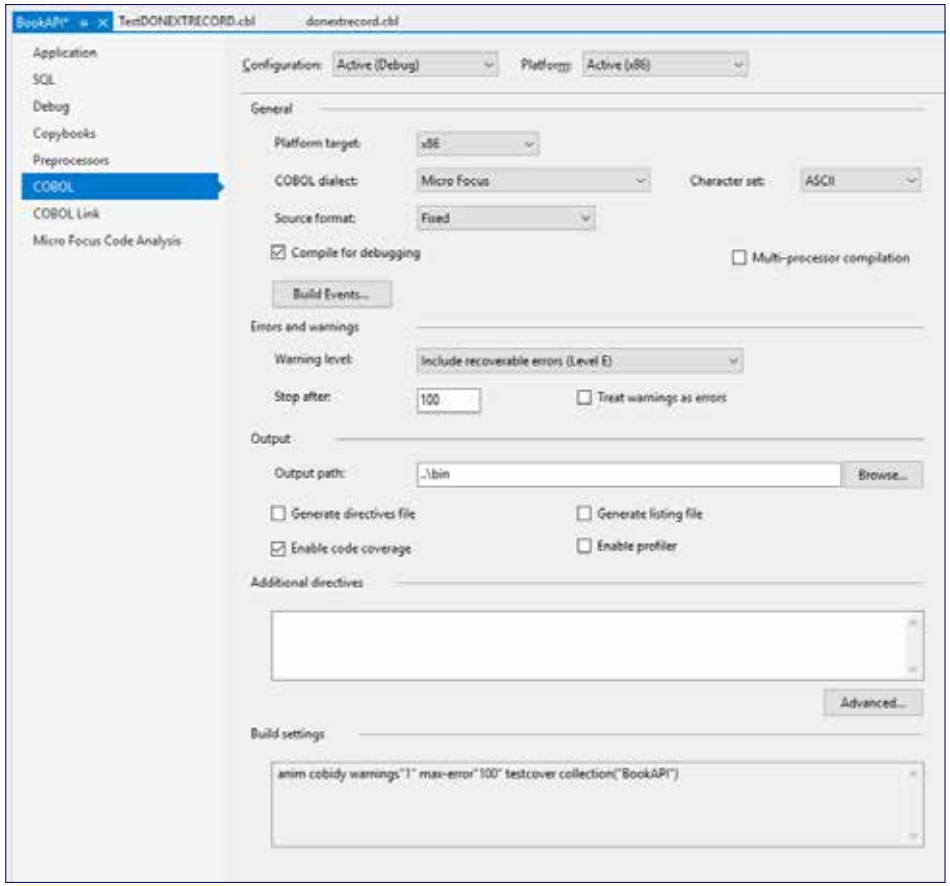


Fig. Enabling code coverage

Close the properties pages run the tests with Code Coverage again

This time, you should be successful and after the tests have been run, you should see the Code Coverage tool window.

Here are my results. This is telling us that around 78% of the code in the donextrecord program is being executed by the tests. That's pretty darn good. But hey, it's only about 100 lines of code, so let's not get too excited.

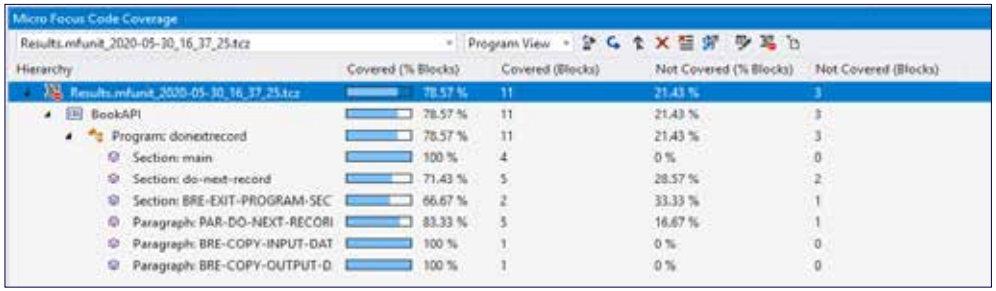


Fig. Code coverage results, good job!

Here we can see what code has and has not been executed at a glance.

Open the donextrecord.cbl file
In the Code Coverage tool window, enable the editor highlighting option

Every line of code that has been executed by a test is shown in blue. Anything in red, hasn't been executed and maybe something you want to write a test case for, or not, as the case may be.

By the way, if you struggle to differentiate these colors, you can configure these in the Visual Studio properties pages – just look for Fonts and *Colors*.

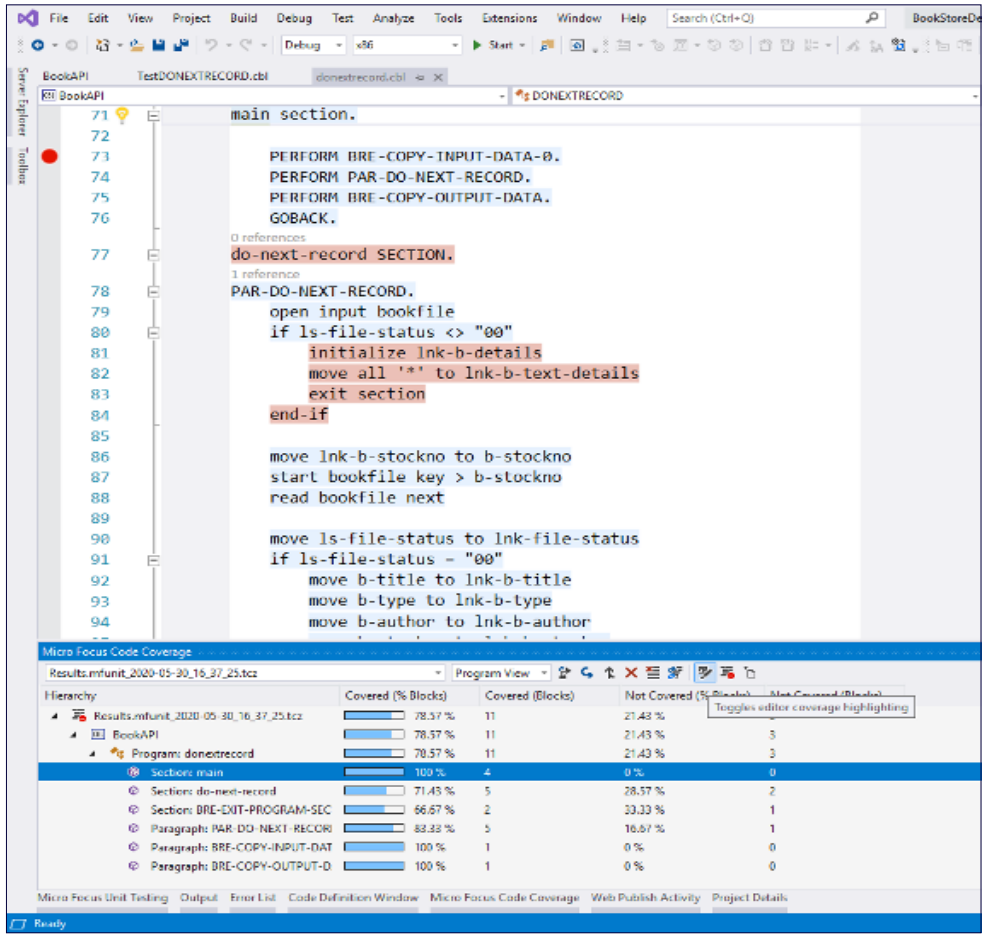


Fig. Editor show code executed by the unit test

Designing your REST API

Time to build ourselves a REST API. Should take, ooh, about 5 minutes perhaps.

A REST API is a web service that's typically called by sending some JSON data to an end point URL which implements the API.

JSON is a human readable data format, a bit like XML but different. It's used a lot by Javascript and web applications to exchange data. You really don't need to worry about it at all. That's because we're going to deploy our REST API using the Micro Focus REST API framework which is called Enterprise Server. It will take care of receiving the JSON traffic and calling your COBOL REST service which is in fact, just a COBOL program. You just need to tell Enterprise Service about your API. Let's do that now.

Right-click the BookAPI project and on the context menu, click Add New Item.

In the Add New Item dialog

- ***Click Native***
- **Select Service Interface**
- **And in the name field, enter BookAPI**
- ***Click Add***

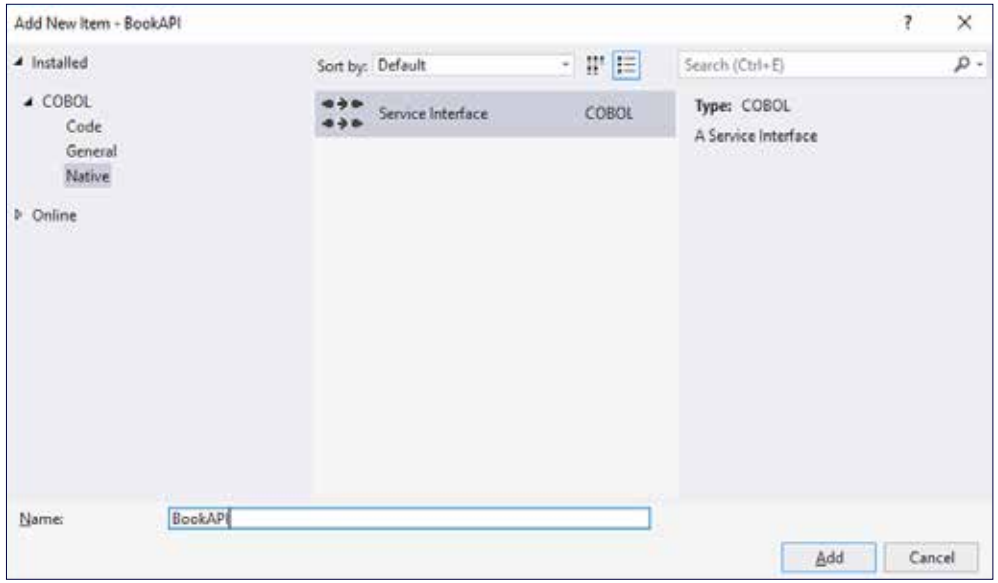


Fig. Adding a new API to your project

In the next dialog, select JSON (RESTful)

Click OK

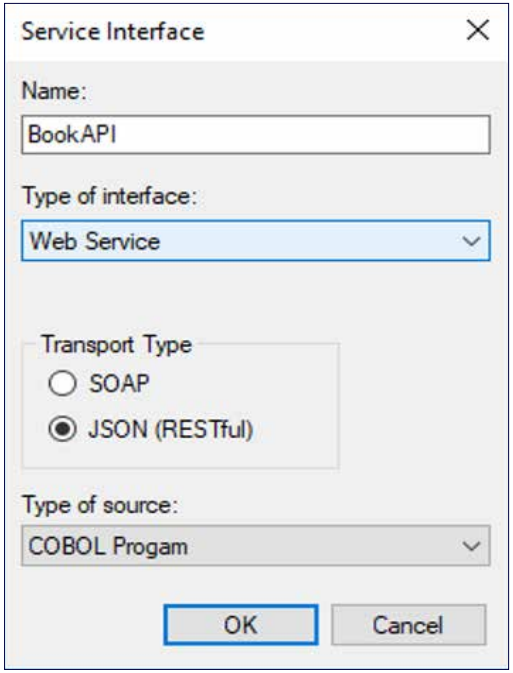


Fig. Editor show code executed by the unit test

You are now presented with a new tool called the IMTK for short – Interface Mapping Toolkit, to give its full title.

The left hand window is where you will load the program you want to create as an API. This window will contain the linkage parameters for the program.

The top right hand pane is where you will define the API input/output parameters. These will map on to linkage parameters.

The other two windows you can ignore.

Notice you also have a new file in your project, BookAPI.svi. If you need to open the IMTK window again, just double-click this file in the project.

First task is to load the program we want to create an API for.

From the extensions menu, choose Operation then New…

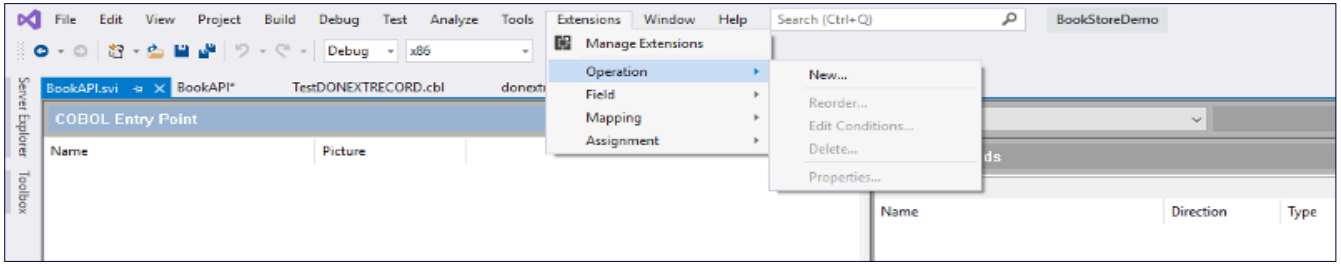


Fig. Adding a new operation to your API

In the first page:

- Name your API NextBook
- Select the donextrecord as the program
- And DONEXTRECORD as the entry point
- *Then click the Path/HTTP properties page*
- Select GET as the method
- *Click OK*

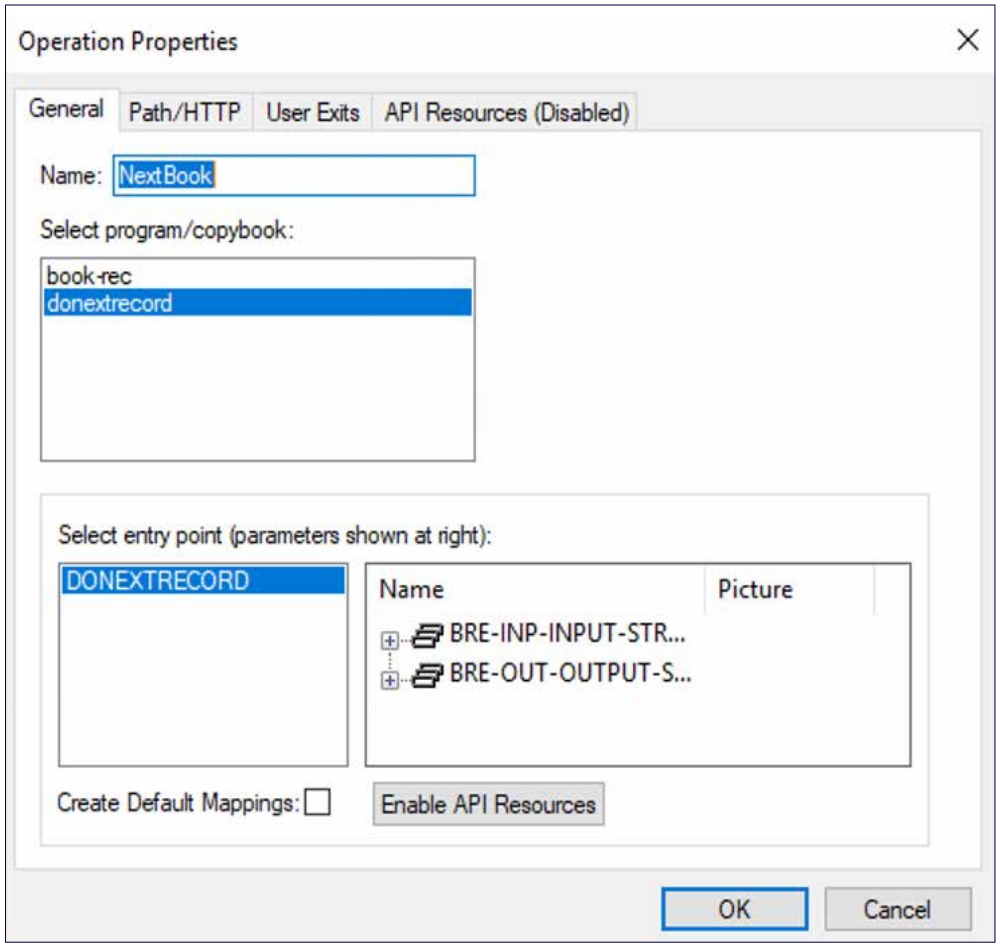


Fig. Setting up the COBOL program will implement the API

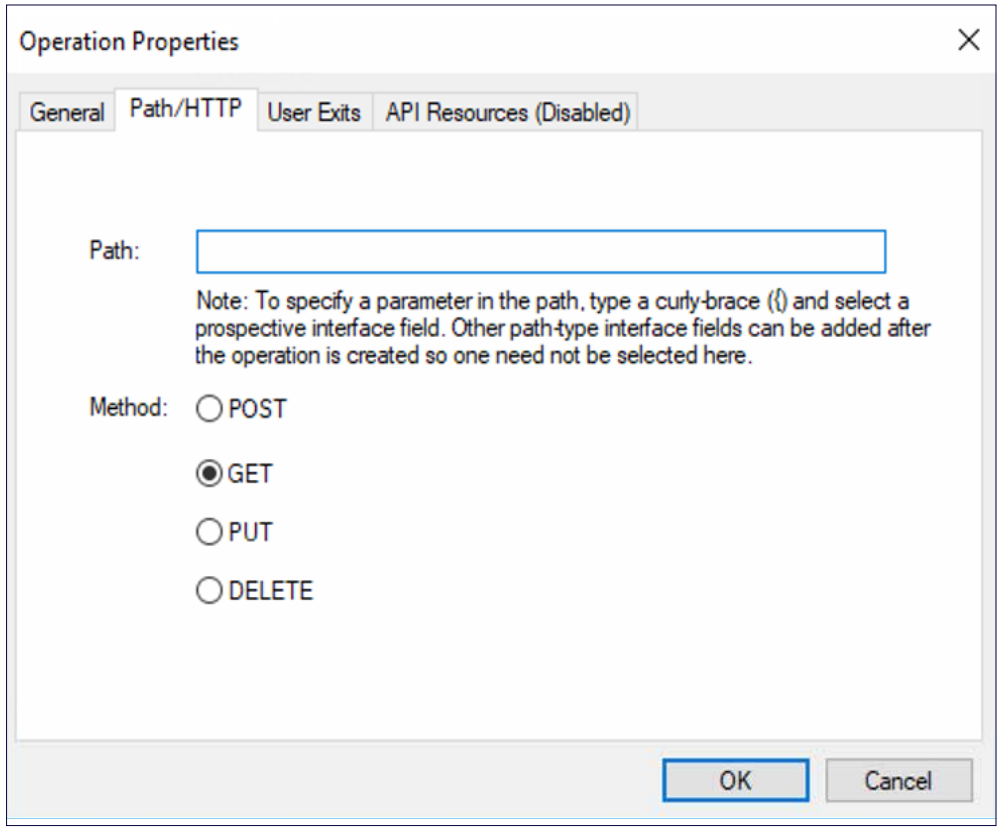


Fig. Make sure the GET option is set

You'll now see the linkage section of the program loaded and displayed in the left hand pane

It's now time to create the API defintion for this linkage section.

- **Drag the 2nd item in the list, BRE-I-LNK-B-STOCKNO, over to the righthand Interface fields window**
- ***Double-click the item to bring up the properties:***
- **Make the name BookStockNo**
- **Set the Direction to Input**
- **Set Location to Query**
- ***Click OK***

Now drag the BRE-OUT-OUTPUT-STRUCTURE field from the linkage parameters over to Interface fields window. You can leave it with it's default values.

Your Interface fields should now look like this:

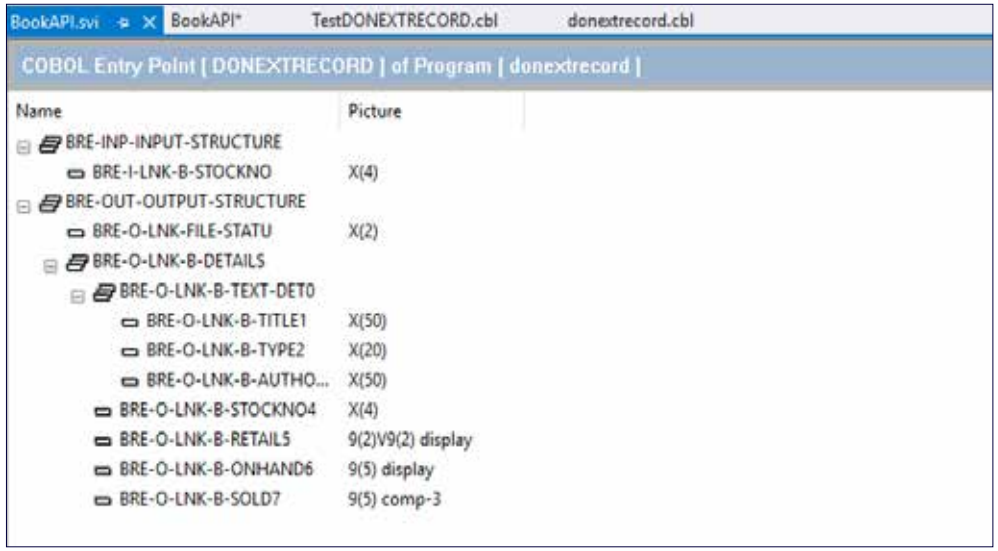


Fig. The COBOL program linkage section loaded into the left hand IMTK window

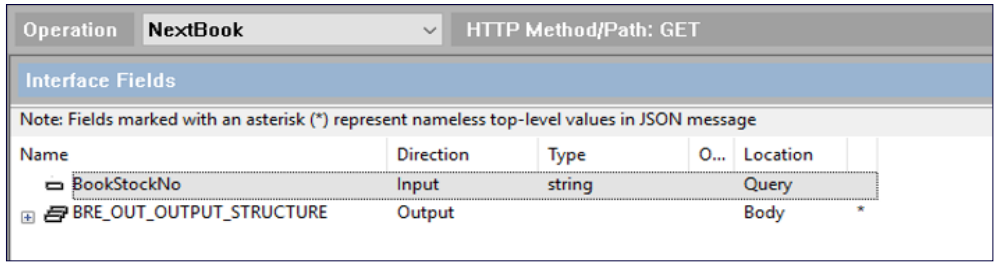


Fig. The definition of our API

We renamed the stock number field to be a little more user friendly outside of COBOL. You can come back later and make adjustments to the output fields too but for now, let's leave this as-is.

It's now time to deploy your API into its home so it can be called by a client, Micro Focus Enterprise Server, is that home.

Step 1. Fire up Enterprise Server

While it is possible to start and stop Enterprise Server from within Visual Studio, you could encounter some permissions issues. So we'll bypass those gotchas and do this using a browser instead.

You can access the Enterprise Server dashboard at the following address in a web browser

http://localhost:10004/

You should see a welcome screen similar to this:

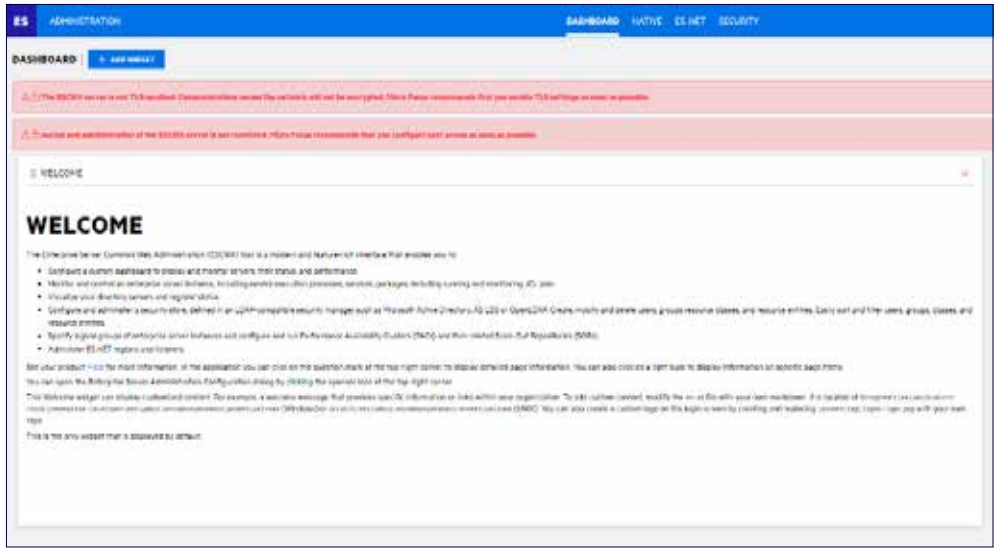


Fig. Enterprise Server web administration

If you don't already see a list of Enterprise Servers, use the Add Widget button to add a Native Region list Widget.

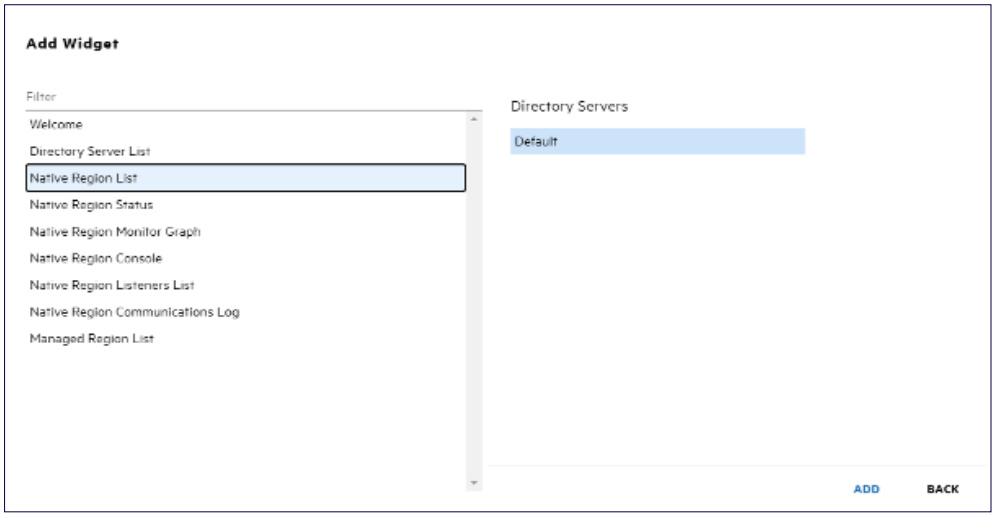


Fig. Adding a Region list to the dashboard

You should then see the following list window in the dashboard which contains our demonstration Enterprise Server called ESDEMO.

Fig. The ESDEMO server for our API

Hover over the ESDEMO item and a set of controls will appear on the righthand side—click the cog icon.

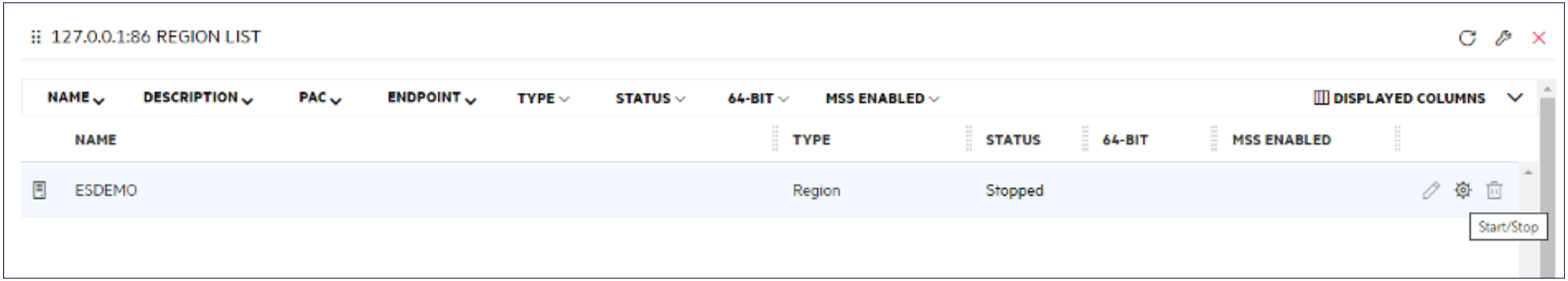


Fig. Starting the ESDEMO server

In the next window, click the start button and click the start button again on the next page you see – we like to be sure you mean it, so we ask you twice.

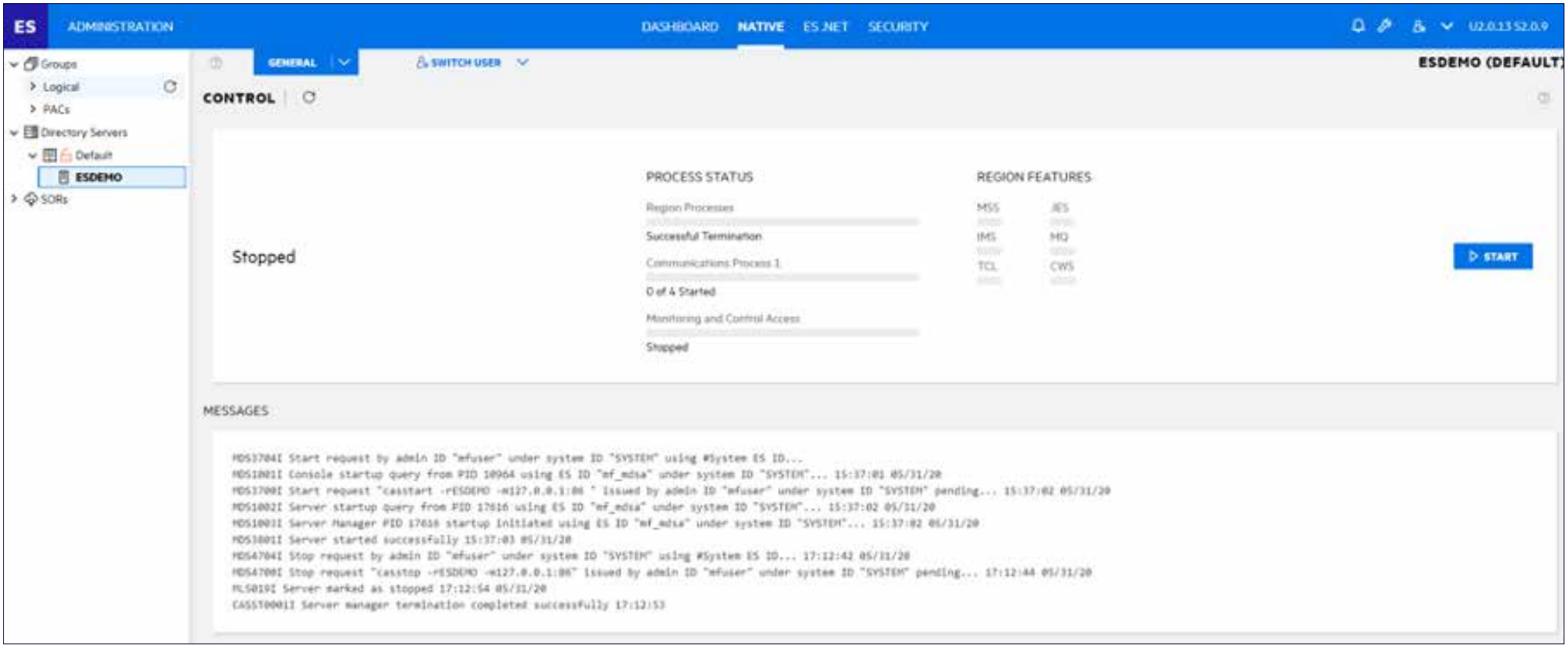


Fig. Output messages as ESDEMO fires up

Step 2. Connect your BookAPI project to ESDEMO

In Visual Studio, make sure the Server Explorer Window is visible. Enable it from the View menu.

Expand the Micro Focus Servers and locate ESDEMO. If it is not showing as started, use the context menu to Refresh

Bring up the context menu on ESDEMO and use the Associate with Project option to connect it with the BookAPI project

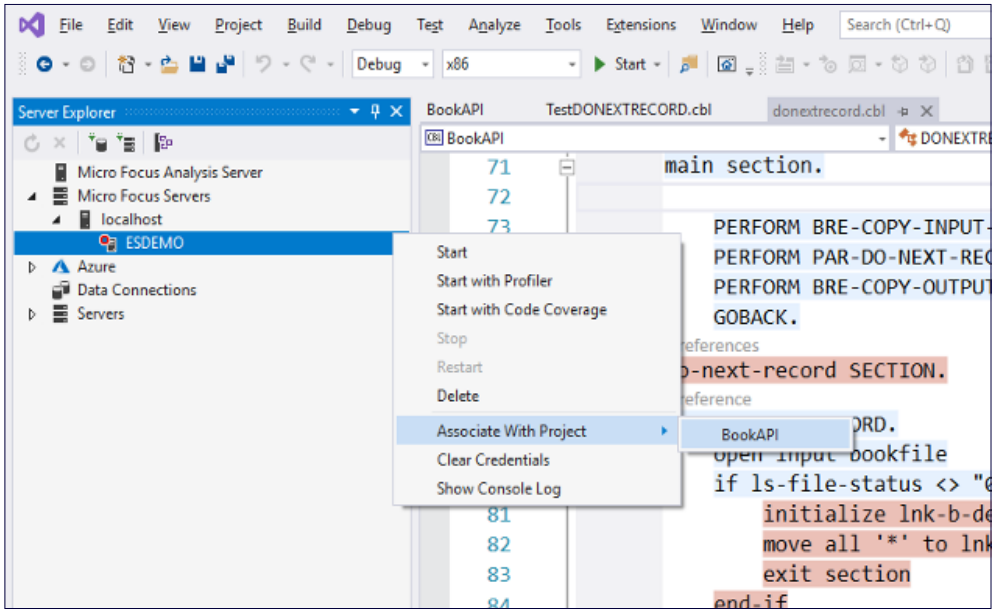


Fig. Deploying the API from Visual Studio

Step 3 . Deploy your API

Bring up the context menu on the BookAPI.svi file your BookAPI project

Click Deploy

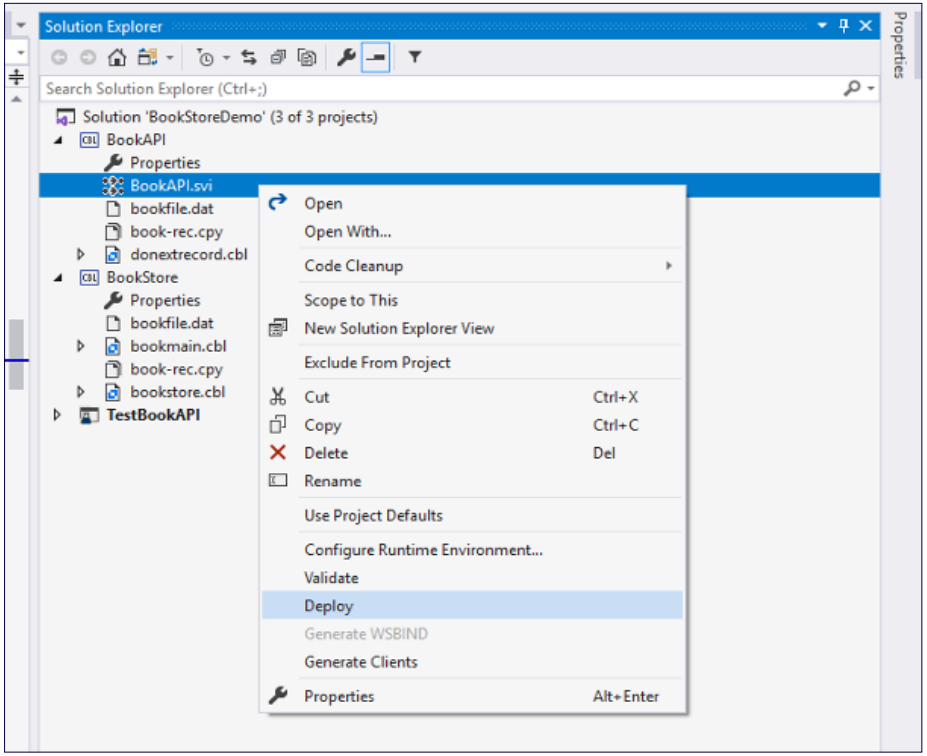


Fig. Deploying the API from Visual Studio

All being well, you should see the following messages in the Output window within Visual Studio.

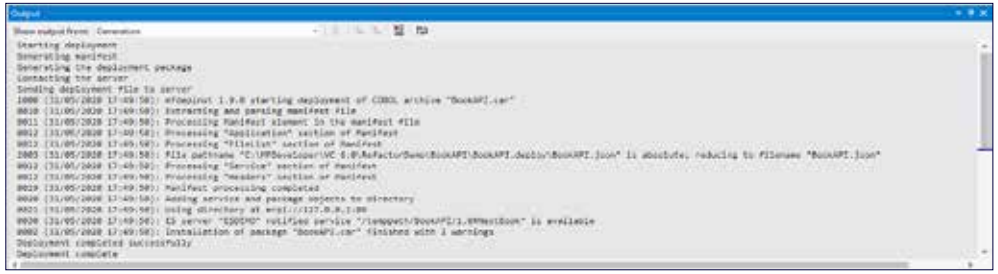


Fig. Deploying the API from Visual Studio

Step 4. Test your API using a web browser

Because we created a REST API that supported HTTP GET messages, we can call our API using a browser.

Use this link to call your API:

http://localhost:9003/temppath/BookAPI/1.0/NextBook

Here are my results using the Chrome web browser:

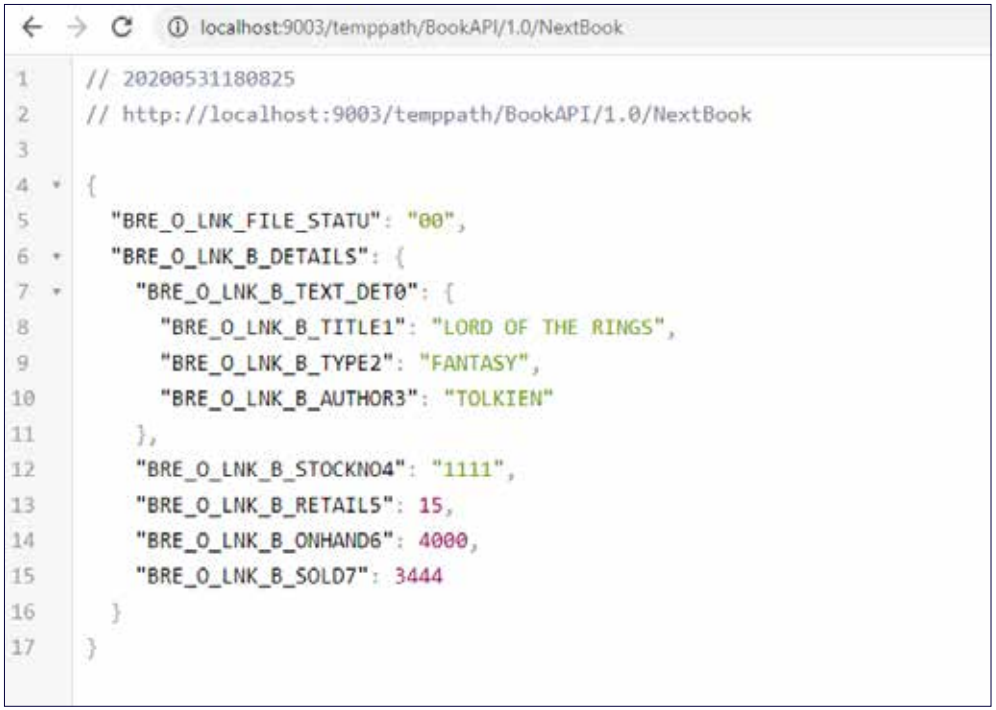


Fig. JSON output from calling the API in a browser

I'm using a handy JSON extension in Chrome which formats the results nicely. Depending on your choice of browser, what you see will look different.

If you remember our COBOL program, it takes in a Stock Code value and locates the next book in the data file. It then returns that information in the linkage section. In this case, the linkage section is formatted as JSON data and sent back to the browser.

We didn't supply a Stock Number so the program the program found the first record in the file.

Let's run again, this time specifying a Stock Number in the URL. The stock number parameter name needs to match the name you used in the IMTK – that should be BookStockNo if you followed the instructions before. If you decided not to rename it or call it something else, go back to your IMTK service and check what you called it. Then substitute it in the URL below.

Here's URL that gets the next book:

http://localhost:9003/temppath/BookAPI/1.0/NextBook?BookStockNo=1111

And my results look like this:

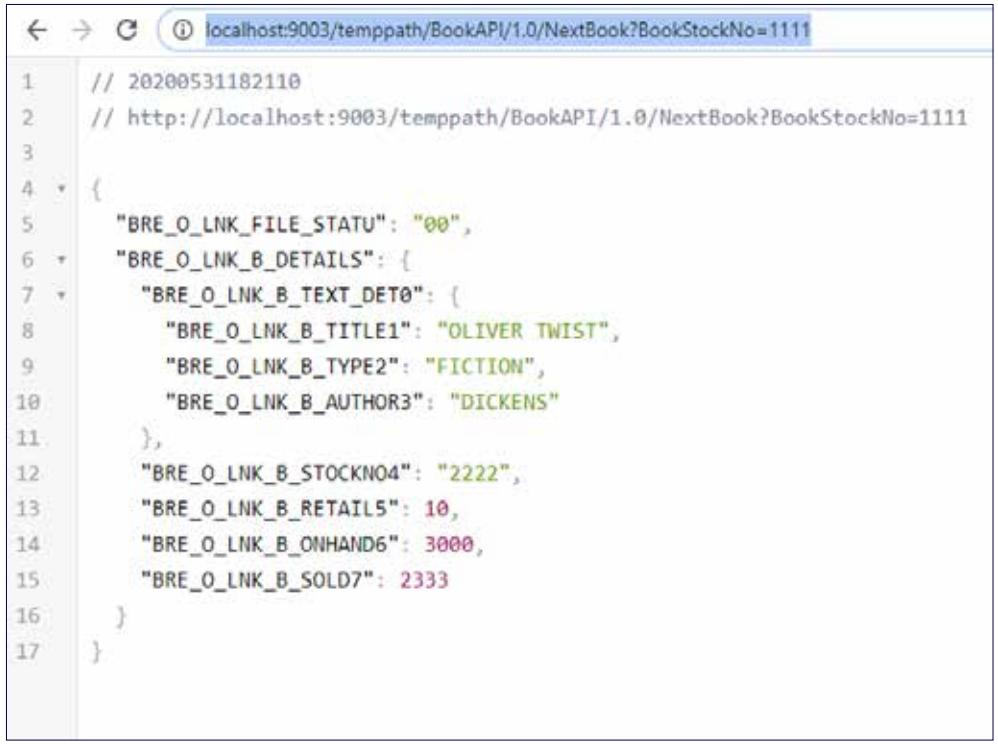


Fig. Calling the API passing in a parameter

Debugging your service

To enable debugging, you need to check *Allow Dynamic Debugging* in the Enterprise Server Dashboard. You'll find this under the General properties section.

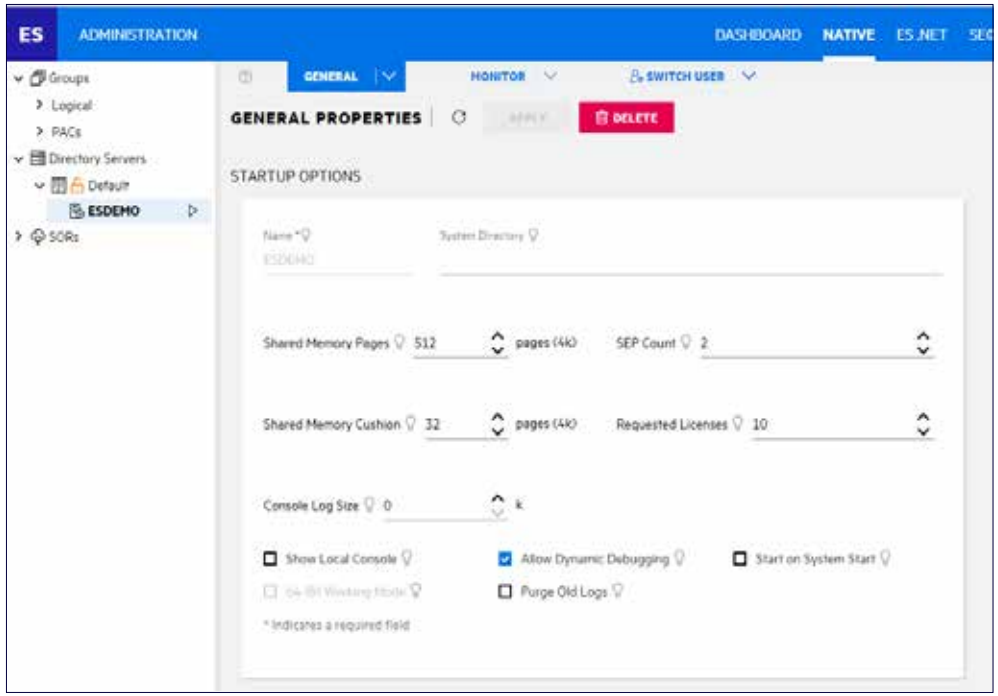


Fig. Enabling debugging

You'll need to stop and start ESDEMO for the setting to take effect.

Now go back to Visual Studio

Set the Book API project to be the startup project using the context menu on the project name

Set a breakpoint on the first line of the procedure division in donextrecord.cbl in the Book API project.

Then hit F5 to run the API

The Visual Studio debugging will sit in a wait state until you invoke the API.

Go back to the browser and resent a request using this URL.

As soon as you do, the Visual Studio debugging should attach and you can step through the code

http://localhost:9003/temppath/BookAPI/1.0/NextBook

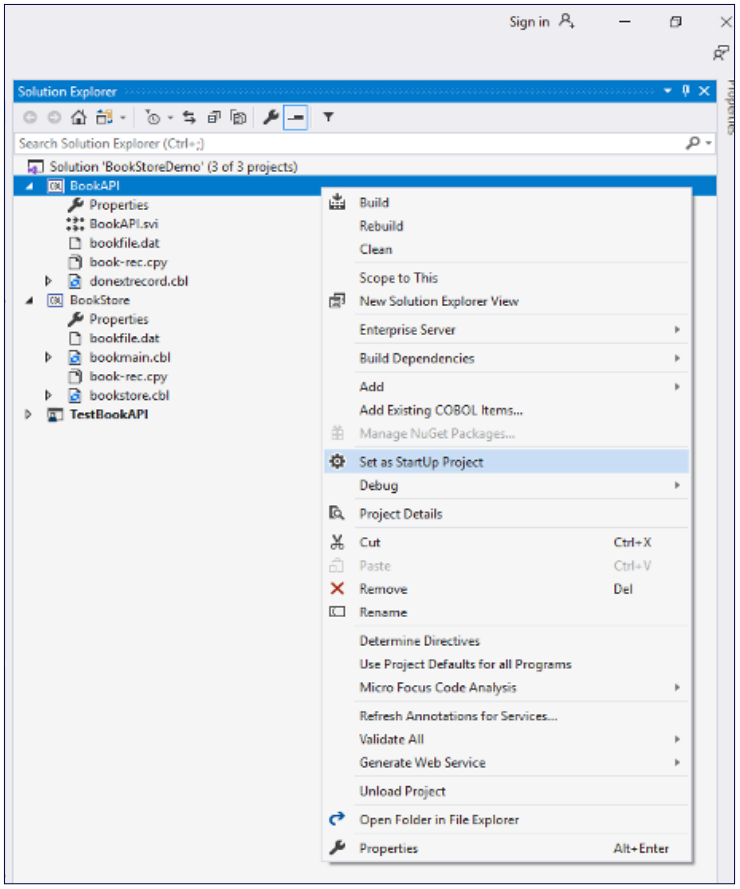


Fig. Making the API project the start up project

How to delete your API…

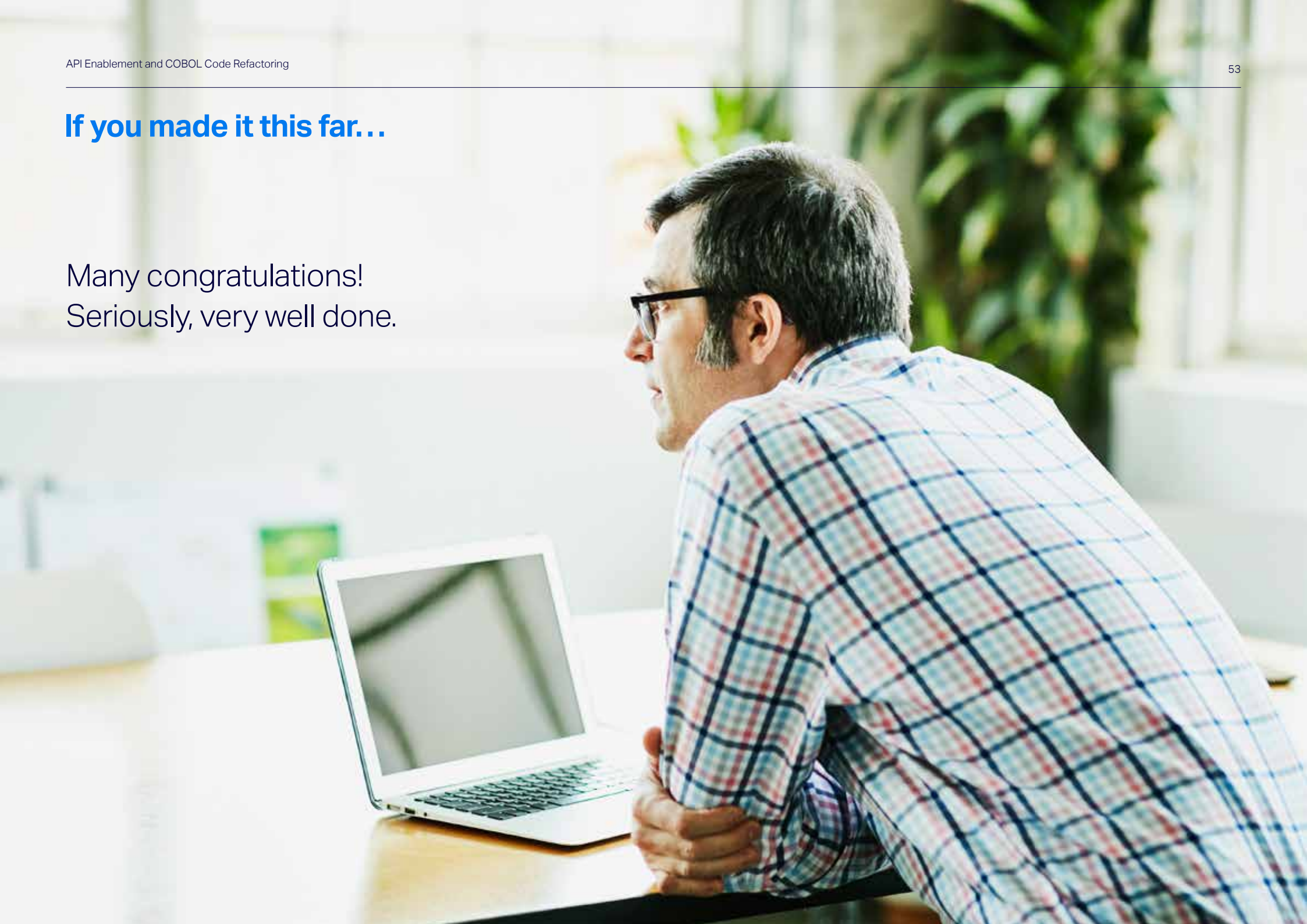
If you ever need to redeploy your service to Enterprise Server, you'll need to delete the existing API first.

- **Load up the Enterprise Server dashboard**
- **Go to the Services page of ESDEMO**
- **Delete your NextBook service**
- ***In the same page, click the Handlers and Packages tab***
- **Scroll down to find the Packages**
- **Delete your NextBook page**

You will now be able to redeploy from Visual Studio

If you made it this far...

Many congratulations!
Seriously, very well done.



Now what...

Well, first of all, pat yourself on the back.

Now, if you're feeling brave, you could hook up a basic webpage and use Javascript to call your API.

And why not try using the refactoring tools on your own application code.

© 2020 Micro Focus