Developer Course

Customization

T280 Testing Business Logic with the Acumatica Unit Test Framework 2022 R2

Revision: 11/17/2022

Acumatica
The Cloud ERP

Contents

Copyright 3
Introduction 4
How to Use This Course 5
Course Prerequisites 6
Company Story and Mission Description 7
Preparing a Test Instance 8
Test Instance: General Information 8
Test Instance: To Deploy an Instance 8
Lesson 1: Creating a Test Project and a Test Class 10
Test Project and Test Class: General Information 10
Activity 1.1: To Create a Test Project 11
Activity 1.2: To Create a Test Class 14
Lesson 2: Creating a Test Method 15
Test Method: General Information 15
Activity 2.1: To Create a Test Method Without Parameters 18
Test Method: Test Management in Visual Studio 20
Activity 2.2: To Run a Test Method 20
Activity 2.3: To Debug a Test Method 21
Activity 2.4: To Create a Test Method with Parameters 22
Test Method: Registration of Services 24
Activity 2.5: To Register a Service 25
Lesson 3: Correctly Assigning Field Values in Tests 27
Correct Assignment of Field Values in Tests: General Information 27
Activity 3.1: To Test the Effect of Changing Field Values 27
Lesson 4: Testing the Display of Errors and Warnings 32
Testing of Errors and Warnings: General Information 32
Activity 4.1: To Test the Display of Errors and Warnings 33
Appendix: Reference Implementation 38
Appendix: Deploying the Needed Acumatica ERP Instance for the Training Course 39
Appendix: Publishing the Required Customization Project 40

Copyright

3933 Lake Washington Blvd NE, # 350, Kirkland, WA 98033

Restricted Rights

The product is provided with restricted rights. Use, duplication, or disclosure by the United States Government is subject to restrictions as set forth in the applicable License and Services Agreement and in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (c)(2) of the Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable.

Disclaimer

Acumatica, Inc. makes no representations or warranties with respect to the contents or use of this document, and specifically disclaims any express or implied warranties of merchantability or fitness for any particular purpose. Further, Acumatica, Inc. reserves the right to revise this document and make changes in its content at any time, without obligation to notify any person or entity of such revisions or changes.

Trademarks

Acumatica is a registered trademark of Acumatica, Inc. HubSpot is a registered trademark of HubSpot, Inc. Microsoft Exchange and Microsoft Exchange Server are registered trademarks of Microsoft Corporation. All other product names and services herein are trademarks or service marks of their respective companies.

Software Version: 2022 R2

Last Updated: 11/17/2022

Introduction

The *T280 Testing Business Logic with the Acumatica Unit Test Framework* training course teaches you how you can create unit tests for extension libraries that are used in customization projects.

This course is intended for application developers who customize Acumatica ERP. It explains how to complement the program code with unit tests that ensure the correctness of the program code even after it is changed or enhanced.

The course is based on a set of examples that demonstrate the general approach to creating unit tests for extension libraries developed for Acumatica ERP. The extension library used in this course is the one you developed in the training courses of the *T* series (which you should take before completing the current course). In this course, you will create one unit test for each graph or graph extension created for the *T200 Maintenance Forms*, *T210 Customized Forms and Master-Detail Relationship*, and *T220 Data Entry and Setup Forms* training courses.

After you complete all the lessons of the course, you will be familiar with some features of Acumatica Unit Test Framework.

We recommend that you complete the examples in the order in which they are provided in the course, because some examples use the results of previous ones.

How to Use This Course

To complete this course, you will complete the lessons of the course in the order in which they are presented and then pass the assessment test. More specifically, you will do the following:

1. Complete the *Course Prerequisites* and perform the *Preparing a Test Instance*.
2. Complete the lessons of the training guide.
3. In Partner University, take *T280 Certification Test: Unit Testing*.

After you pass the certification test, you will receive the Partner University certificate of course completion.

What Is in a Lesson

Each lesson consists of the concepts you are learning and the activities you are going to complete to learn these concepts.

What the Documentation Resources Are

The complete Acumatica ERP and Acumatica Framework documentation is available at *https://help.acumatica.com/* and is included in the Acumatica ERP instance. While viewing any form used in the course, you can click the **Open Help** button in the top pane of the Acumatica ERP screen to bring up a form-specific Help menu; you can use the links on this menu to quickly access form-related information and activities and to open a reference topic with detailed descriptions of the form elements.

Which License You Should Use

For the educational purposes of this course, you use Acumatica ERP under the trial license, which does not require activation and provides all available features. For the production use of the Acumatica ERP functionality, an administrator has to activate the license the organization has purchased. Each particular feature may be subject to additional licensing; please consult the Acumatica ERP sales policy for details.

Course Prerequisites

To complete this course, you should be familiar with the basic concepts of Acumatica Framework and Acumatica Customization Platform. We recommend that you complete the *T200 Maintenance Forms*, *T210 Customized Forms and Master-Detail Relationship*, and *T220 Data Entry and Setup Forms* training courses before you begin this course.

Required Knowledge and Background

To complete the course successfully, you should have the following required knowledge:

- Proficiency with C#, including but not limited to the following features of the language:
  - Class structure
  - OOP (inheritance, interfaces, and polymorphism)
  - Usage and creation of attributes
  - Generics
  - Delegates, anonymous methods, and lambda expressions
- Knowledge of the following main concepts of ASP.NET and web development:
  - Application states
  - The debugging of ASP.NET applications by using Visual Studio
  - The process of attaching to IIS by using Visual Studio debugging tools
  - Client- and server-side development
  - The structure of web forms
- Experience with SQL Server, including doing the following:
  - Writing and debugging complex SQL queries (WHERE clauses, aggregates, and subqueries)
  - Understanding the database structure (primary keys, data types, and denormalization)
- The following experience with IIS:
  - The configuration and deployment of ASP.NET websites
  - The configuration and securing of IIS

Company Story and Mission Description

This topic describes the company story and explains the scope of this course.

Company Story

The Smart Fix company specializes in repairing cell phones of several types. The company provides the following services:

- Battery replacement: This service is provided on customer request and does not require any preliminary diagnostic checks.
- Repair of liquid damage: This service requires a preliminary diagnostic check and a prepayment.
- Screen repair: This service is provided on customer request and does not require any preliminary diagnostic checks.

To manage the list of devices serviced by the company and the list of services the company provides, two custom maintenance forms, Repair Services (RS201000) and Serviced Devices (RS202000), have been added to the company's Acumatica ERP instance. Another custom maintenance form, Services and Prices (RS203000), gives users the ability to define and maintain the price for each provided repair service. Also, the *Stock Items* (IN202500) form of Acumatica ERP has been customized to give users the ability to mark particular stock items as repair items—that is, the items (such as replacement screens and batteries) that are supplied to the customer as part of the repair services. The Repair Work Orders (RS301000) custom data entry form is used to create and manage work orders for repairs. On the Repair Work Order Preferences (RS101000) custom setup form, an administrative user specifies the company's preferences for the repair work orders.

Mission Description

To keep the business logic consistent, you need to develop unit tests for the developed forms. Adding these unit tests to the customization library will ensure that the implemented behavior will persist even if changes are made later to the customization library.

Preparing a Test Instance

The topics of this chapter describe how to deploy an Acumatica ERP instance. You will use this instance for creating the unit tests that are described in this guide.

Test Instance: General Information

In this chapter, you will learn how to deploy an Acumatica ERP test instance that contains custom and customized forms. You can use this instance to complete the training course.

Learning Objectives

In this chapter, you will learn how to do the following:

- Prepare the environment
- Deploy an Acumatica ERP test instance

Applicable Scenarios

You deploy an Acumatica ERP test instance by using the instructions in this chapter, if you need to complete the training course.

Deploying a Test Instance

You can use the Acumatica ERP Configuration Wizard to deploy instances based on predefined datasets. It also makes it possible to deploy test instances that contain custom and customized forms.

In the Acumatica ERP Configuration Wizard, you click **Deploy New Application Instance for T-series Developer Courses** and select the training course in the list as specified in the Test Instance: To Deploy an Instance activity. The instance that you prepare contains the data that is required to complete the activities of this guide. The instance can also include the published customization project.

Test Instance: To Deploy an Instance

This activity will walk you through the process of preparing the environment and deploying an Acumatica ERP instance that you can use to perform the activities of this guide.

Process Overview

You will prepare the environment—that is, install the software that you need to use Acumatica ERP and to create customizations for it. Then you will deploy an Acumatica ERP instance with the data for the *T280 Testing Business Logic with the Acumatica Unit Test Framework* course.

Step 1: Preparing the Environment

Before you begin deploying the needed Acumatica ERP instance, do the following:

1. Make sure the environment that you are going to use conforms to the *System Requirements for Acumatica ERP 2022 R2*.

2. Make sure that the Web Server (IIS) features that are listed in *Configuring Web Server (IIS) Features* are turned on.
3. Install the Acuminator extension for Visual Studio.
4. Install Acumatica ERP. On the Main Software Configuration page of the installation program, select the **Install Acumatica ERP** and **Install Debugger Tools** check boxes.

> If you have already installed Acumatica ERP without debugger tools, you should remove Acumatica ERP and install it again with the **Install Debugger Tools** check box selected. The reinstallation of Acumatica ERP does not affect existing Acumatica ERP instances. For details, see *To Install the Acumatica ERP Tools*.

Step 2: Deploying an Acumatica ERP Instance

To deploy an Acumatica ERP instance and configure it, do the following:

1. Open the Acumatica ERP Configuration Wizard, and do the following:
   a. Click **Deploy New Application Instance for T-series Developer Courses**.
   b. On the **Database Configuration** page, make sure the name of the database is *PhoneRepairShop*.
   c. On the **Instance Configuration** page, do the following:
      a. In the **Local Path of the Instance** box, select a folder that is outside of the `C:\Program Files (x86)` and `C:\Program Files` folders. (By avoiding these folders, you can avoid an issue with permissions to work in them when you perform customization of the website.)
      b. In the **Training Course** box, select *T280 Testing Business Logic with the Acumatica Unit Test Framework*.

   The system creates a new Acumatica ERP instance, adds a new tenant, loads the data to it, and publishes the customization project that is needed for this training course.
2. Make sure a Visual Studio solution is available in the `App_Data\Projects\PhoneRepairShop` folder of the Acumatica ERP instance folder. This is the solution of the extension library in which you will create unit tests.

Lesson 1: Creating a Test Project and a Test Class

Before you begin creating unit tests in the Unit Test Framework, you need to create a Visual Studio project, which you will set up to be a *test project*. In the test project, you create a test class for each graph or graph extension you want to test.

The topics of this chapter describe how to create and configure a Visual Studio project that is intended to contain unit tests for an Acumatica Framework-based application and how to create test classes.

Test Project and Test Class: General Information

As the first step in creating a set of unit tests for an extension library in a customization project, you create a Visual Studio project and configure it to serve as the test project. For each graph or graph extension to be tested, you create a separate test class in the test project.

Learning Objectives

In this chapter, you will learn how to do the following:

- Create a project in Visual Studio
- Configure the project to be a test project that uses the `xUnit.net` library and the Acumatica Unit Test Framework
- Create the test class that will contain unit tests

Applicable Scenarios

You create and configure a test project each time you need to create unit tests for graphs and graph extensions implemented in a customization project. You create a test class for each graph or graph extension for which you will create unit tests.

Requirements for Using the Acumatica Unit Test Framework

By design, the Acumatica Unit Test Framework does not require you to sign in or access a database on a disk. To perform unit tests, you do not need to deploy an Acumatica ERP instance or a database instance. All actions are performed in RAM.

The Acumatica Unit Test Framework is designed to be used with the xUnit.net library. You may configure your project for using another unit test library (for example, NUnit), but Acumatica does not guarantee compatibility with a unit test library other than xUnit.net.

For details about xUnit.net library, see *Getting Started with xUnit.net Using .NET Framework with Visual Studio*.

Naming Conventions for Test Projects, Namespaces, and Test Classes

By performing the usual process, for each project whose functionality is going to be tested, you create a test project in the same Visual Studio solution and name it by adding *.Tests* to the end of the name of the project being tested.

Similarly, the namespace used in the test project should be named the same as the namespace of the project being tested with *.Tests* appended to the name.

The test class should also be named the same as the class being tested with *Tests* appended to the name.

For example, to test the `InventoryItemMaint` class, which is in the `PhoneRepairShop_Code` namespace and contained in the `PhoneRepairShop_Code.csproj` project, you create

the `PhoneRepairShop_Code.Tests.csproj` project; in this project, you create the `InventoryItemMaintTests` class and place it in the `PhoneRepairShop_Code.Tests` namespace.

Test Class Definition

A test class is derived from the `TestBase` class, which is available in the `PX.Tests.Unit` namespace.

Activity 1.1: To Create a Test Project

The following activity will walk you through the process of creating and configuring a test project.

Process Overview

In Visual Studio, you will create a test project in the same solution with the extension library that implements the graphs and graph extensions being tested. You will add the `xunit` and `xunit.runner.visualstudio` NuGet packages to the test project. You will also add the `PX.Tests.Unit.dll` library and other libraries from the Acumatica ERP instance to the test project.

Step 1: Creating a Test Project

To create the `PhoneRepairShop_Code.Tests.csproj` project, proceed as follows:

1. Open the `PhoneRepairShop_Code.sln` file, which is located in the `\App_Data\Projects\PhoneRepairShop` subfolder of the *PhoneRepairShop* instance folder.
2. In Visual Studio, in the **Solution Explorer** panel, right-click the `PhoneRepairShop_Code` solution.
3. Select **Add > New Project**.
4. Select the *Class Library (.NET Framework)* project template.
5. Click **Next**.
6. Enter the `PhoneRepairShop_Code.Tests` project name.
7. Click **Create**.
8. Delete the `Class1.cs` item from the `PhoneRepairShop_Code.Tests.csproj` project. You will create the required C# classes later.

Step 2: Adding NuGet Packages

Do the following to add the necessary NuGet packages to the `PhoneRepairShop_Code.Tests.csproj` project:

1. In the **Solution Explorer** panel of Visual Studio, right-click the `PhoneRepairShop_Code.Tests` project, and select **Manage NuGet Packages**.
2. Select the **Browse** link or make sure that it is selected.
3. In the search box, type `xunit`.
4. In the search results, select the xunit package, and click **Install** (see the following screenshot).

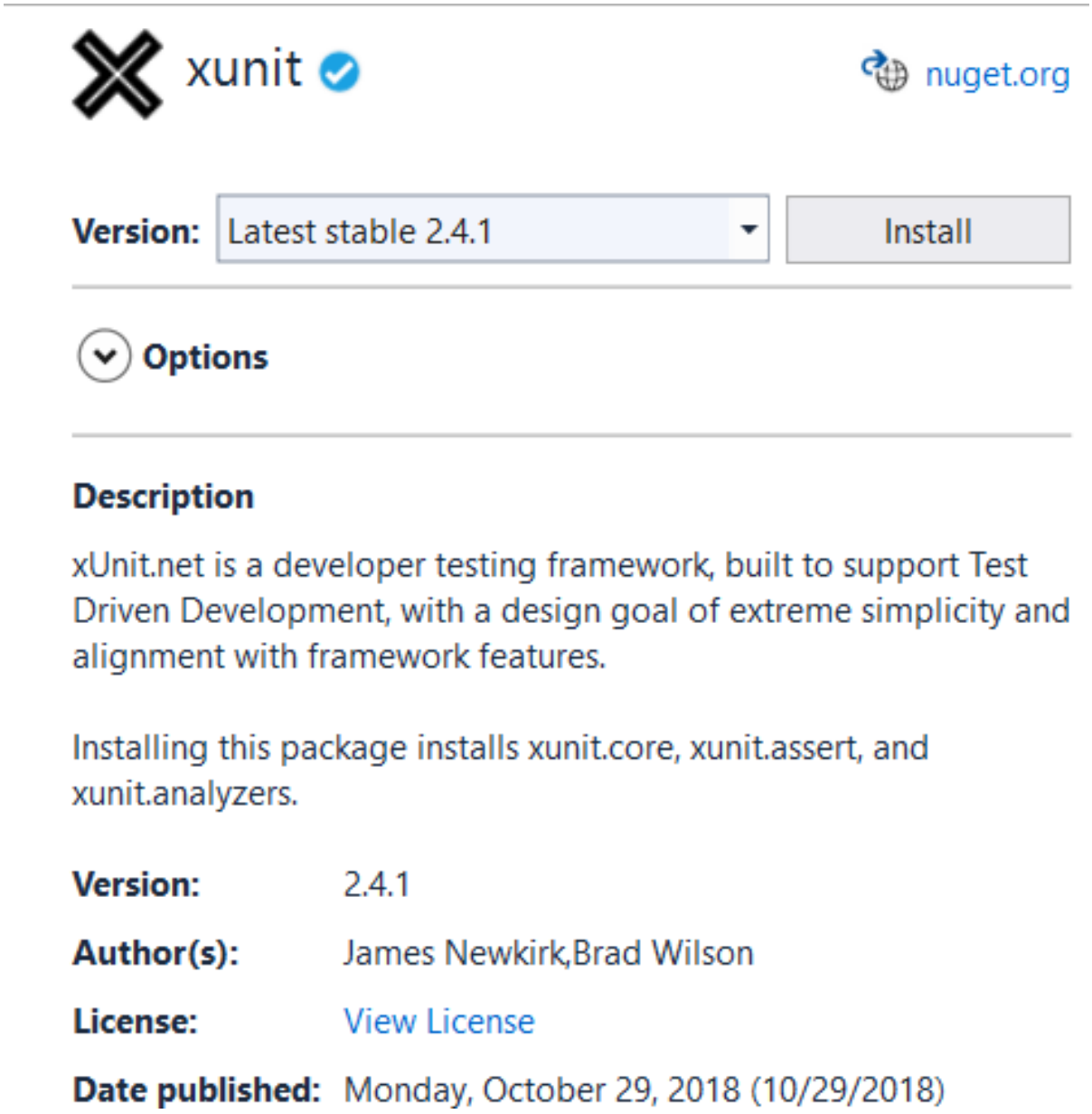

Figure: xunit package information

5. In the **Preview Changes** dialog box (shown in the following screenshot), click **OK**. This causes the xunit package to be installed.

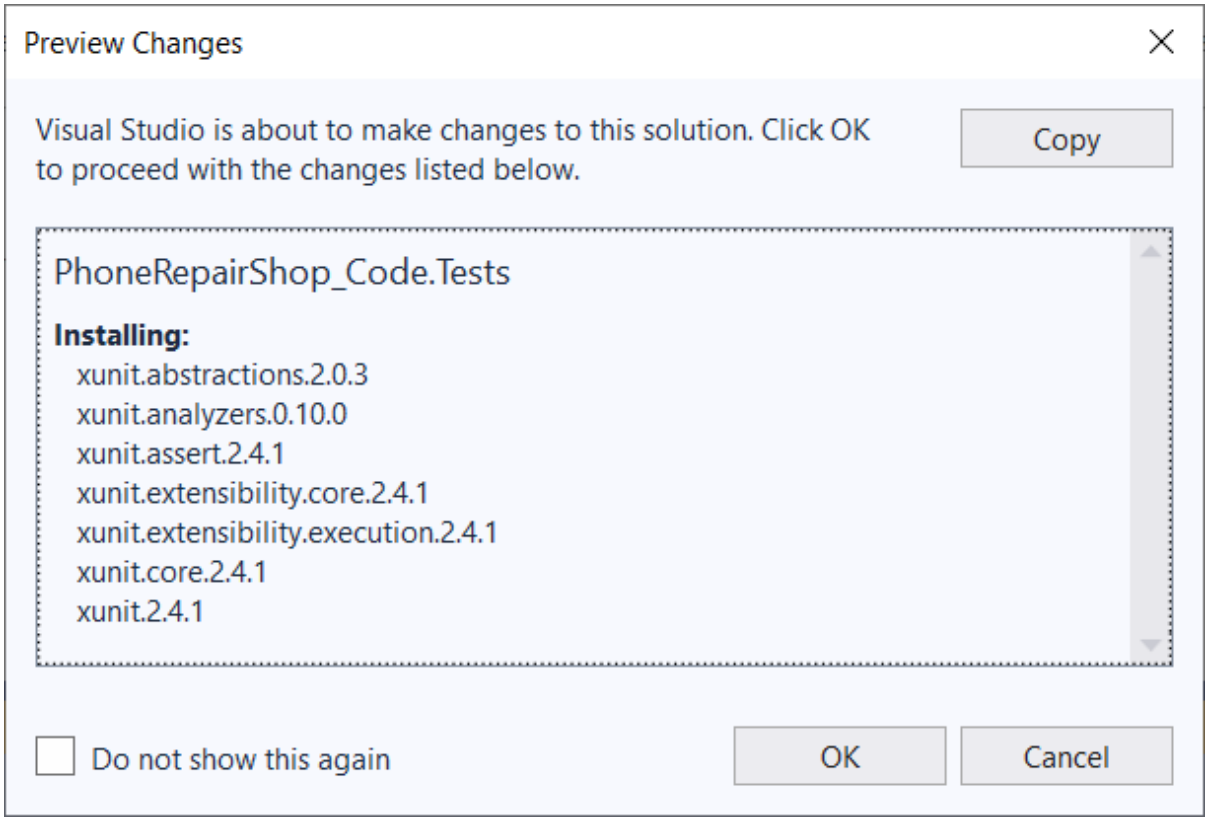

Figure: Preview Changes dialog box

6. Install the `xunit.runner.visualstudio` and `Microsoft.Bcl.AsyncInterfaces` packages by performing similar actions to those in Instructions 2–5.

Step 3: Adding References to the Customization Code and Acumatica ERP Libraries

To add references to the customization project and Acumatica ERP libraries to the `PhoneRepairShop_Code.Tests.csproj` project, proceed as follows:

1. Optional: Copy the `PX.Tests.Unit.dll` file from the `UnitTesting` subfolder of the Acumatica ERP installation folder to any folder you choose.
2. Right-click the **References** item of the `PhoneRepairShop_Code.Tests` project, and select **Add Reference**.
3. On the **Browse** tab, click **Browse**.
4. In the `PhoneRepairShop\Bin` folder, select the following files:
   - `PX.Common.dll`
   - `PX.Common.Std.dll` (this library makes it possible to use graph views in the code)
   - `PX.Data.BQL.Fluent.dll`
   - `PX.Data.dll`
   - `PX.Objects.dll` (this library contains the implementation of the `PX.Objects.IN.InventoryItem` class)
5. Select the `PX.Tests.Unit.dll` file in the folder where you have placed it.
6. Optionally, add references to other DLLs.
7. On the **Projects** tab, select the `PhoneRepairShop_Code` project.

8. Click **OK**.

Activity 1.2: To Create a Test Class

The following activity will walk you through the process of creating a test class in the test project that you have created.

Story

You need to create a class for testing the Repair Services (RS201000) custom form, whose business logic is implemented in the `RSSVRepairServiceMaint` class.

Process Overview

In the test project, you create a test class derived from the `PX.Tests.Unit.TestBase` class. This class is intended for containing test methods.

Step: Creating a Test Class

To create a test class for the Repair Services (RS201000) form, do the following:

1. In the **Solution Explorer** panel of Visual Studio, right-click the `PhoneRepairShop_Code.Tests` project, and select **Add > New Item**.
2. Select the *Visual C# item* | *Class* template, and type `RSSVRepairServiceMaintTests.cs` as the file name.
3. Click **Add**.
4. Specify the following `using` directives in the `RSSVRepairServiceMaintTests.cs` file.

```
using Xunit;
using PX.Data;
using PX.Tests.Unit;
using PhoneRepairShop;
```

5. Make the `RSSVRepairServiceMaintTests` class public and derived from `TestBase` as follows.

```
    public class RSSVRepairServiceMaintTests : TestBase
```

In this class, you will create a test method to test the logic of the `RSSVRepairServiceMaint` graph, as described in *Activity 2.1: To Create a Test Method Without Parameters*. For more information about creating test methods, see *Lesson 2: Creating a Test Method*.

Lesson 2: Creating a Test Method

While the purpose of a test project is to contain tests for a customization library, the purpose of a test class is to contain tests for a graph or graph extension, the purpose of a test method is to contain tests of some functionality implemented in the graph or graph extension being tested.

The topics of this chapter describe how to create a method that contains unit tests for a graph, how to register the necessary services, and how to manage a single test or a group of tests.

Test Method: General Information

In a test class, which is intended for testing a graph or graph extension, you will create test methods, each of which will test a particular functionality. Through the xUnit.net library, which you will use for creating unit tests in the activities of this guide, you can create test methods without parameters and test methods with parameters. In addition to the presence or absence of parameters, these kinds of methods differ in the attributes they have and the number of unit tests they produce.

Learning Objectives

In this chapter, you will learn how to do the following:

- Create test methods without parameters
- Create an instance of the tested graph
- Create and update objects in the graph cache
- Verify that the desired conditions are met for the values present in the method
- Run and debug test methods
- Create test methods with parameters
- Set values for the test parameters
- Create unit tests for graph extensions

Applicable Scenarios

You create a test method if you want to test some business logic of a graph or graph extension.

Development of a Test

A unit test is an automated test written by a software developer that tests some independent unit of code. A unit test usually launches some functionality of a software component and then compares the resulting values with the expected values. As you develop unit tests, the best practice is to create them according to the standard *AAA* pattern:

1. *Arrange*: You prepare the test's context and assign some values.
2. *Act*: You launch the functionality of a graph or graph extension.
3. *Assert*: You check the result.

With the Acumatica Unit Test Framework, you can easily create unit tests according to this process.

In the arrange part, you initialize the context until actions can be correctly executed. To use the Acumatica Unit Test Framework for this part, you may need to first enable some application features (by using the `Common.UnitTestAccessProvider.EnableFeature<FeatureType>` generic method) or initialize

the required setup DACs (by using the `Setup<PXGraph>` generic method). After that, you create an instance of a graph (by using the `PXGraph.CreateInstance<PXGraph>` generic method) and fill its caches.

Some automated tests, typically referred to as *integration tests*, validate the interaction of multiple independent components. With Acumatica Unit Test Framework, you can implement these tests and check the interaction of multiple components within a graph.

Creation of Test Methods Without Parameters

You create a test method without parameters as follows:

- You declare a method as public void.
- You assign the `[Fact]` attribute to the test method. This produces one test for the test method.

Creation of Test Methods with Parameters

When you create a test method with parameters, you declare it as `public void` and assign it the needed parameters. You assign the `[Theory]` attribute to the test method and add as many `[InlineData]` attributes as the method has parameters. This produces one test for each set of parameters provided in the `[InlineData]` attribute.

In each `[InlineData]` attribute, you specify constants separated by commas as the values for the method parameters. The number of the constants and their types must match the number and the types of the test method parameters.

You can provide values for the method parameters in the `[ClassData]` or `[MemberData]` attributes instead of the `[InlineData]` attributes.

Testing of the Business Logic of a Graph

In the test method, you create an instance of the tested graph by calling the `PXGraph.CreateInstance<>` generic method. In this generic method, you use the graph class as the type parameter.

You create data access class (DAC) records in the graph cache (`PXCache` objects). You can address a specific cache of a graph either by the DAC name (`<graph>.Caches[typeof(<DAC_name>)]`) or by the graph view that contains the DAC objects (`<graph>.<DACView>.Cache`).

Creation of a Test Method for a Graph Extension

In a test method for a graph extension that contains logic for a DAC extension, you get the DAC extension object with the `GetExtension<>` generic method. You alter the DAC extension object as you wish, and then you update the whole DAC object along with its extension in the cache by using the `PXCache.Update` method. This executes the logic implemented in the graph extension so that you can verify its correctness.

Assertions

You can use the methods of the `Xunit.Assert` class to make assertions in unit tests. For example, the `Assert.True` static method verifies that its argument is equal to *True*, and the `Assert.False` static method verifies that its argument is equal to *False*. The following table lists additional methods of the `Xunit.Assert` class that may be useful.

Table: Assert Class Methods

Method	Description
`False(bool condition)`	Verifies that `condition` is *false*. The method also throws a `FalseException` if `condition` is not *false*.
`True(bool condition)`	Verifies that `condition` is *true* and throws a `TrueException` if `condition` is not *true*.
`Contains<T>(T expected, IEnumerable<T> collection)`	Verifies that a collection contains a given object. The method also throws a `ContainsException` if the collection does not contain the object.
`DoesNotContain<T>(T expected, IEnumerable<T> collection)`	Verifies that a collection does not contain a given object. The method also throws a `DoesNotContainException` if the collection contains the object.
`Empty(IEnumerable collection)`	Verifies that `collection` is empty and throws an `EmptyException` if `collection` is not empty.
`NotEmpty(IEnumerable collection)`	Verifies that `collection` is not empty and throws a `NotEmptyException` if `collection` is empty.
`Single(IEnumerable collection)`	Verifies that `collection` contains a single element. The method also throws a `SingleException` if `collection` does not contain exactly one element.
`Null(object obj)`	Verifies that the object reference is *null* and throws a `NullException` if the object reference is not *null*.
`NotNull(object obj)`	Verifies that an object reference is not *null*, and throws a `NotNullException` if the object reference is *null*.
`Contains(string expectedSubstring, string actualString)`	Verifies that `actualString` contains `expectedSubstring` by using the current culture. The method also throws a `ContainsException` if `actualString` does not contain `expectedSubstring`.
`DoesNotContain(string expectedSubstring, string actualString)`	Verifies that `actualString` does not contain `expectedSubstring` by using the current culture. The method also throws a `DoesNotContainException` if `actualString` contains `expectedSubstring`.
`StartsWith(string expectedStartString, string actualString)`	Verifies that `actualString` starts with `expectedStartString` by using the current culture. The method also throws a `StartsWithException` if `actualString` does not start with `expectedStartString`.

Method	Description
`EndsWith(string expectedEndString, string actualString)`	Verifies that `actualString` ends with `expectedEndString` by using the current culture, and throws an `EndsWithException` if `actualString` does not end with `expectedEndString`.
`Equal(string expected, string actual)`	Verifies that the `actual` and `expected` strings are equivalent. The method also throws an `EqualException` if the `actual` and `expected` strings are not equivalent.
`Equal<T>(T expected, T actual)`	Verifies that two objects are equal by using a default comparer, and throws an `EqualException` if the objects are not equal.
`Equal<T>(T[] expected, T[] actual)`	Verifies that two arrays of the unmanaged type `T` are equal. The method also throws an `EqualException` if the arrays are not equal.

You can use some assertion library instead of the `Xunit.Assert` class (for example, `FluentAssertions`).

Running and Debugging of Test Methods

Visual Studio includes the Test Explorer tool, which you can use to run a test, a test method, a group of test methods, or all available test methods. You can also debug a single test or multiple tests.

Activity 2.1: To Create a Test Method Without Parameters

The following activity will walk you through the process of creating a test method without parameters.

Story

Suppose that you want to make sure that the following behavior of the Repair Services (RS201000) custom form has not changed: The selection of the **Walk-In Service** check box and the selection of the **Requires Preliminary Check** check box are mutually exclusive. You need to create a test method that changes the state of one check box and checks the state of the other check box.

Process Overview

To create the test method, you will create a public void method and assign it the `[Fact]` attribute. In the method, you will create an instance of the tested graph. In the cache of the graph, you will create a record. In this record, you will change the values of the fields that indicate the states of the check boxes, update the cache of the graph, and check that the values of dependent fields have been changed according to the tested logic of the graph.

Step: Creating a Test Method Without Parameters

To create a method for checking the states of the check boxes of the Repair Services (RS201000) form, do the following:

1. In the `RSSVRepairServiceMaintTests` class, use the following code to create a public void method and name it *PreliminaryCheckAndWalkInServiceFlags_AreOpposite*.

```
public void PreliminaryCheckAndWalkInServiceFlags_AreOpposite()
{
}
```

2. By using the following code, assign the `[Fact]` attribute to the method to specify that this unit test is called without parameters.

```
[Fact]
public void PreliminaryCheckAndWalkInServiceFlags_AreOpposite()
```

3. In the `PreliminaryCheckAndWalkInServiceFlags_AreOpposite` method, create an instance of the `RSSVRepairServiceMaint` graph as follows.

```
var graph = PXGraph.CreateInstance<RSSVRepairServiceMaint>();
```

4. Now that the `RSSVRepairServiceMaint` graph is initialized, use the following code to create an `RSSVRepairService` object that represents a record in the table of the Repair Services form.

```
RSSVRepairService repairService =
    graph.Caches[typeof(RSSVRepairService)].
    Insert(new RSSVRepairService
    {
        ServiceCD = "Service1",
        Description = "Service 1",
        WalkInService = true,
        PreliminaryCheck = false
    }) as RSSVRepairService;
```

Objects that represent table records are created in the graph cache (`PXCache` objects). You can address a specific cache of a graph either by the DAC name (`graph.Caches[typeof(<DAC_name>)]`) or by the graph view that contains the DAC objects (`graph.<DACView>.Cache`).

5. After a record is created, determine whether a change of the **Walk-In Service** check box state leads to a change of the **Requires Preliminary Check** check box state. Also, determine whether a change of the **Requires Preliminary Check** check box state leads to a change of the **Walk-In Service** check box. Perform this checking by adding the following code.

```
repairService.WalkInService = false;
graph.Caches[typeof(RSSVRepairService)].Update(repairService);
Assert.True(repairService.PreliminaryCheck);

repairService.WalkInService = true;
graph.Caches[typeof(RSSVRepairService)].Update(repairService);
Assert.False(repairService.PreliminaryCheck);

repairService.PreliminaryCheck = false;
graph.Caches[typeof(RSSVRepairService)].Update(repairService);
Assert.True(repairService.WalkInService);

repairService.PreliminaryCheck = true;
graph.Caches[typeof(RSSVRepairService)].Update(repairService);
Assert.False(repairService.WalkInService);
```

Test Method: Test Management in Visual Studio

The test methods that you have created appear in the **Test Explorer** window of Visual Studio, which is shown in the following screenshot.

For each test method without parameters (see *Activity 2.1: To Create a Test Method Without Parameters*), a test has been added. For parameterized test methods (see *Activity 2.4: To Create a Test Method with Parameters*), a test has been added for each set of parameter values that you have specified for the method.

In the **Test Explorer** window of Visual Studio, you can select the desired test, test method, or group of test methods, and you can run (see *Activity 2.2: To Run a Test Method*) or debug (see *Activity 2.3: To Debug a Test Method*) them. You can also get information about the tests' outcomes (whether they have succeeded or not) and execution time.

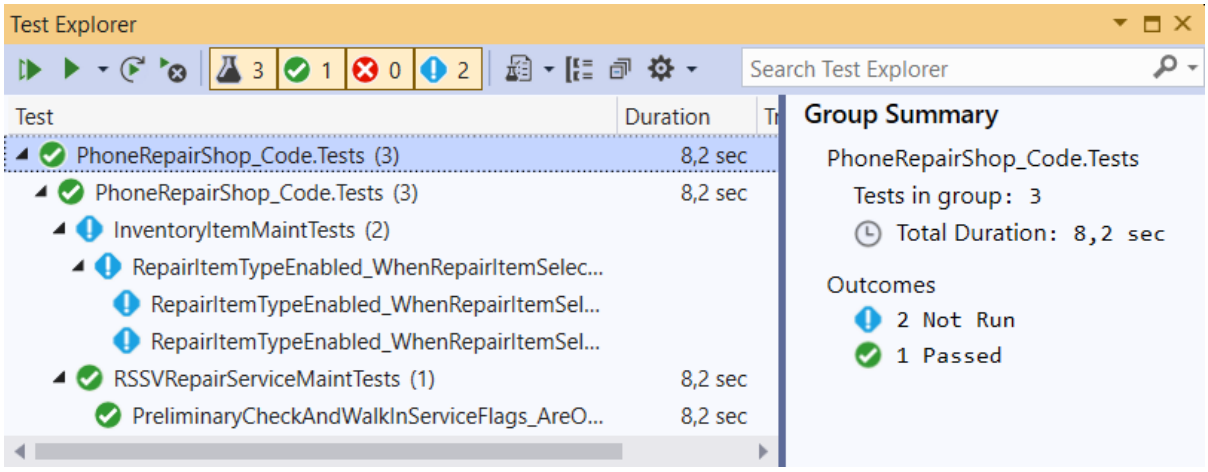

Figure: The Test Explorer window with added tests

Activity 2.2: To Run a Test Method

The following activity will walk you through the process of running a single test method.

Process Overview

In the **Test Explorer** window of Visual Studio, you will select a method and run it.

Step: Running a Test Method

> If your code contains test methods other than the one created in *Activity 2.1: To Create a Test Method Without Parameters*, or if you have already run the existing test method before, the number of items mentioned in the following instruction may differ.

Do the following to run the method created in *Activity 2.1: To Create a Test Method Without Parameters*:

1. In Visual Studio, select the **Test > Test Explorer** menu item. The **Test Explorer** window opens, which currently displays no tests.

2. Click **Run All Tests In View**. The solution is built, and one test for the `PreliminaryCheckAndWalkInServiceFlags_AreOpposite` test method appears in the **Test Explorer** window (shown in the following screenshot) and is run.

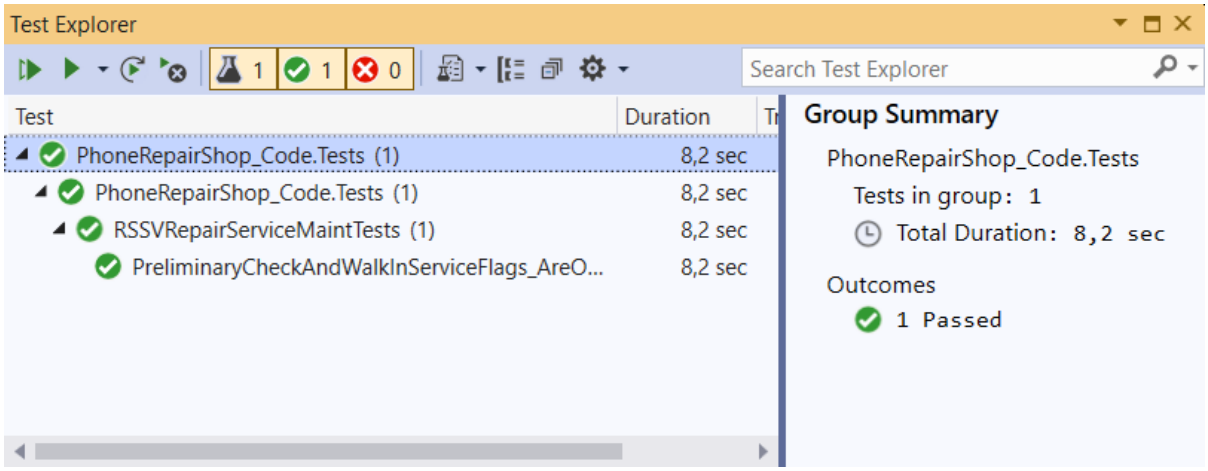

Figure: The Test Explorer window with one test

Activity 2.3: To Debug a Test Method

The following activity will walk you through the process of debugging a test method.

Process Overview

In Visual Studio, you will set breakpoints both in the code being tested and in the test method, launch the debugging for the test method from the **Test Explorer** window, and check that the values of the fields are as expected.

Step: Debugging a Test Method

To debug the `PreliminaryCheckAndWalkInServiceFlags_AreOpposite` method, do the following:

1. In Visual Studio, open the `RSSVRepairServiceMaintTests.cs` file.
2. Move the caret to the following line.

```
repairService.WalkInService = false;
```

3. Press F9 to set a breakpoint on the line.
4. Open the `RSSVRepairServiceMaint.cs` file.
5. In the `_(Events.FieldUpdated<RSSVRepairService, RSSVRepairService.walkInService> e)` method, move the caret to the following line.

```
row.PreliminaryCheck = true;
```

6. Press F9 to set a breakpoint on the line.
7. Select the **Test > Test Explorer** menu command to open the **Test Explorer** window.
8. Right-click the `PreliminaryCheckAndWalkInServiceFlags_AreOpposite` method, and select **Debug**. After the debugging process starts, the execution pauses at the

breakpoint in the `RSSVRepairServiceMaintTests.cs` file. You can verify that the value of `repairService.WalkInService` is *true*.

9. Step over the statement in the current line (you can use F10 for this). The value of `repairService.WalkInService` becomes *false*.
10. Step over the statement in the current line. This updates the `repairService` object in the cache. During the update, the `FieldUpdated` event is generated for the **Walk-In Service** check box, and the `_(Events.FieldUpdated<RSSVRepairService, RSSVRepairService.walkInService> e)` method of the `RSSVRepairServiceMaint` class is launched. The execution pauses at the breakpoint in the `RSSVRepairServiceMaint.cs` file, which signifies that the extension library and unit test match each other.
11. Continue debugging (you can press F5 for this). The debugging process continues until it completes successfully.

Activity 2.4: To Create a Test Method with Parameters

The following activity will walk you through the process of creating a test method with parameters.

Story

Suppose that you want to make sure that the following behavior of the customized *Stock Items* (IN202500) form has not changed: The selection of the **Repair Item** check box causes the **Repair Item Type** drop-down list to be available, and the clearing of the **Repair Item** check box causes the **Repair Item Type** drop-down list to be unavailable. You need to create a test method that selects or clears the **Repair Item** check box and checks whether the **Repair Item Type** drop-down list is available or unavailable, respectively.

Process Overview

To create the test method, you will create a public void method with one Boolean parameter. You will assign the method the `[Theory]` attribute and two `[InlineData]` attributes, each of which has the possible values of the method parameter (*true* and *false*). In the method, you will create an instance of the tested graph. In the cache of the graph, you will create a record. For this record, by using the `GetExtension` generic method, you will retrieve the extension, which contains custom fields. Depending on the method parameter, you will change the state of the check box, update the cache of the graph, and make sure that the availability of the drop-down list has been changed according to the tested logic of the graph.

System Preparation

Before you begin creating the test method, create the `InventoryItemMaintTests` test class. For an example that shows how to create a test class, see *Activity 1.2: To Create a Test Class*.

Step 1: Creating and Configuring a Test Method with Parameters

To create and configure a method for testing the business logic of the `InventoryItemMaint` class, do the following:

1. In the `InventoryItemMaintTests` class, create a public void method, and name it *RepairItemTypeEnabled_WhenRepairItemSelected*.
2. To specify that this unit test is called with parameters, assign the `[Theory]` attribute to the method as follows.

```
        [Theory]
```

```
        public void RepairItemTypeEnabled_WhenRepairItemSelected
```

3. Add the Boolean `enabled` parameter to the `RepairItemTypeEnabled_WhenRepairItemSelected` method as follows.

```
        public void RepairItemTypeEnabled_WhenRepairItemSelected
            (bool enabled)
```

4. Add two `[InlineData]` attributes, which provide two values for the `enabled` parameter, as follows.

```
        [Theory]
        [InlineData(true)]
        [InlineData(false)]
        public void RepairItemTypeEnabled_WhenRepairItemSelected
            (bool enabled)
```

   The `RepairItemTypeEnabled_WhenRepairItemSelected` method will be called twice: The first call will pass *true* as its argument, and the second will pass *false*.

Step 2: Implementing a Method for Testing the InventoryItemMaint Class

To test the business logic of the `InventoryItemMaint` class, do the following:

1. In the `RepairItemTypeEnabled_WhenRepairItemSelected` method, create an instance of the `InventoryItemMaint` graph by adding the following code.

```
            var graph = PXGraph.CreateInstance<InventoryItemMaint>();
```

2. After the `InventoryItemMaint` graph is initialized, create an `InventoryItem` object as follows.

```
            InventoryItem item =
                (InventoryItem)graph.Caches[typeof(InventoryItem)].Insert(
                    new InventoryItem
                    {
                        InventoryCD = "Item1",
                        Descr = "Item 1"
                    });
```

3. Get the `InventoryItemExt` extension of the `InventoryItem` object as follows.

```
            InventoryItemExt itemExt = item.GetExtension<InventoryItemExt>();
```

4. Select (or clear) the **Repair Item** check box by using the needed value of the `enabled` parameter, and make sure that the **Repair Item Type** drop-down list is available (or, respectively, unavailable) by adding the following code.

```
            itemExt.UsrRepairItem = enabled;
            graph.Caches[typeof(InventoryItem)].Update(item);
            PXFieldState fieldState =
                ((PXFieldState)graph.Caches[typeof(InventoryItem)].GetStateExt<
                InventoryItemExt.usrRepairItemType>(item));
            Assert.True(enabled == fieldState.Enabled);
```

A call of the `PXCache.Update` method for the `InventoryItem` object updates both the object and its extension.

As a result of the actions you have performed in this activity, the `InventoryItemMaintTests.cs` file contains the following code.

```
using Xunit;
using PX.Data;
using PX.Tests.Unit;
using PX.Objects.IN;

namespace PhoneRepairShop_Code.Tests
{
    public class InventoryItemMaintTests : TestBase
    {
        [Theory]
        [InlineData(true)]
        [InlineData(false)]
        public void RepairItemTypeEnabled_WhenRepairItemSelected
            (bool enabled)
        {
            var graph = PXGraph.CreateInstance<InventoryItemMaint>();

            InventoryItem item =
                (InventoryItem)graph.Caches[typeof(InventoryItem)].Insert(
                    new InventoryItem
                    {
                        InventoryCD = "Item1",
                        Descr = "Item 1"
                    });

            InventoryItemExt itemExt = item.GetExtension<InventoryItemExt>();

            itemExt.UsrRepairItem = enabled;
            graph.Caches[typeof(InventoryItem)].Update(item);
            PXFieldState fieldState =
                ((PXFieldState)graph.Caches[typeof(InventoryItem)].GetStateExt<
                InventoryItemExt.usrRepairItemType>(item));
            Assert.True(enabled == fieldState.Enabled);
        }
    }
}
```

This test uses the *arrange-act-assert* pattern.

In some circumstances, running this test method may fail. To learn the actions that you must perform to make this test method successful, see *Activity 2.5: To Register a Service*.

Test Method: Registration of Services

Acumatica uses the *dependency injection* technique to design software. With this technique, an object (a *client*) receives other objects (*services*) that it depends on; in this context, these services are called *dependencies*. Each of these services must be registered: For each needed interface, you must specify the service that implements this interface.

This software design pattern is used to decouple code components and make them easier to extend and test. For more information about dependency injection, see *https://docs.microsoft.com/en-us/dotnet/core/extensions/dependency-injection*.

Implementation of the Dependency Injection in Acumatica ERP and Unit Tests

In Acumatica ERP, the `Autofac` library is used to implement the dependency injection. For more information, see *Dependency Injection*.

In unit tests, mock services are used instead of real complex objects. These mock services are simplified services that mimic real ones.

To register a service, you need to override the `TestBase.RegisterServices` method and make calls to the `Autofac.ContainerBuilder.RegisterType` generic method.

Table: Frequently Used Mock Services

Mocked Interface	Mock Service
`IFinPeriodRepository` (defined in the `PX.Objects.GL.FinPeriods` namespace)	`FinPeriodServiceMock` (defined in the `PX.Objects.Unit` namespace)
`IPXCurrencyService` (defined in the `PX.Objects.CM.Extensions` namespace)	`CurrencyServiceMock` (defined in the `PX.Objects.Unit` namespace)

You can also define your own service, register it, and use it in a `PX.Tests.Unit.TestBase`-derived class as described in *Dependency Injection*.

Activity 2.5: To Register a Service

The following activity will walk you through the process of registering a mock service in a test class.

Story

Suppose that running a test method leads to the generation of the `Autofac.Core.Registration.ComponentNotRegisteredException` exception. When this exception occurs, the error message specifies which component (interface) is not registered. You need to register the service that implements the specified interface.

Process Overview

To get rid of the exception, you will add the use of the Autofac library and override the `TestBase.RegisterServices` method in the test class. In the overridden `RegisterServices` method, you will register the needed service that implements the interface specified in the exception message.

Step: Registering a Service

When the test method created in *Activity 2.4: To Create a Test Method with Parameters* is run, the following error message is generated if a proper service is not registered (only part of the error message is quoted).

Autofac.Core.Registration.ComponentNotRegisteredException : The requested service 'System.Func`2[[PX.Data.PXGraph, PX.Data, Version=1.0.0.0, Culture=neutral, PublicKeyToken=3b136cac2f602b8e], [PX.Objects.CM.Extensions.IPXCurrencyService, PX.Objects, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null]]' has not been registered. To avoid this exception, either register a component to provide

the service, check for service registration using IsRegistered(), or use the ResolveOptional() method to resolve an optional dependency.

To register this service, do the following:

1. In the `PhoneRepairShop_Code.Tests.csproj` project, add a reference to `Autofac.dll`, or install the Autofac NuGet package. Both procedures are described in *Activity 1.1: To Create a Test Project*.
2. Add the following `using` directives to the `InventoryItemMaintTests.cs` file.

```
using Autofac;
using PX.Objects.CM.Extensions;
```

3. In the `InventoryItemMaintTests` class, override the `TestBase.RegisterServices` method as follows.

```
        protected override void RegisterServices(ContainerBuilder builder)
        {
            base.RegisterServices(builder);
            builder.RegisterType<PX.Objects.Unit.CurrencyServiceMock>().
                As<IPXCurrencyService>();
        }
```

After the service is registered, run the tests for the `RepairItemTypeEnabled_WhenRepairItemSelected` test method, which had failed previously. Make sure that the tests succeed.

Lesson 3: Correctly Assigning Field Values in Tests

When you develop unit tests for Acumatica Framework-based applications, make sure that you assign field values in the appropriate places in the code. The topics of this chapter describe how to assign values to key and non-key fields and use default field values in unit tests.

Correct Assignment of Field Values in Tests: General Information

When you create unit tests for graphs or graph extensions, you may encounter various difficulties related to assigning field values: If you assign field values in the wrong places, the results of the code execution may be unpredictable.

Learning Objectives

In this chapter, you will learn how to do the following:

- Specify the values of key and non-key fields when an object is created or updated
- Use the default field values when an object is created

Applicable Scenarios

In unit tests, you specify field values every time you create or update DAC objects in the cache.

Assignment of Values to Fields

When you are assigning values to DAC fields in unit tests, you need to handle key fields and non-key fields differently.

When you create a DAC object in the cache, you use the `PXCache.Insert` method. As a result, some values (either default values or values you specify) are assigned to the key fields of the DAC object. After a DAC object is created, you alter its fields by using the `PXCache.Update` method. If you alter the key fields in this way, the old DAC object remains unchanged in the cache, but another DAC object with the specified values of the key fields is created or updated in the cache.

So to avoid mistakes, you should assign values to the key fields of a DAC object at the stage of creation and should not change them afterward.

If a DAC object you are going to create has a key field that is defined with the `PXSelector` or `PXDBDefault` attribute, the default value of this key field is taken from the `Current` property of the DAC specified in the attribute.

Activity 3.1: To Test the Effect of Changing Field Values

The following activity will walk you through the process of creating a method that tests complicated business logic involving the assignment of field values.

Story

Suppose that you want to make sure that the following behavior of the Services and Prices (RS203000) form has not changed: If changes are made to the state of the **Required** or **Default** check box (or both check boxes) of a row,

these changes affect the states of these check boxes in other rows. Also, you want to make sure that the prices are calculated correctly.

You need to create a test method. In this method, you need to create at least three repair items of different types and a labor item, configure them, and make sure that the fields that correspond to the **Required** and **Default** check boxes have the correct states and that the boxes with prices have the correct values.

Process Overview

You will create a test method without parameters for the Services and Prices (RS203000) form. In this method, you will create an instance of the tested graph, `RSSVRepairPriceMaint`. In the cache of the graph, you will create the necessary objects by using the `PXCache.Insert` method, and configure them by using the `PXCache.Update` method. Then you will use the methods of the `Assert` class to verify that the necessary conditions for the values of the objects' fields are met.

System Preparation

Before you begin creating the test method, create the `RSSVRepairPriceMaintTests` test class. For an example that shows how to create a test class, see *Activity 1.2: To Create a Test Class*.

Step 1: Creating a Test Method for the Services and Prices Form

In this step, you will create a test method for the Services and Prices (RS203000) form. (For basic information on the creation of a test method without parameters, see *Test Method: General Information*.) Do the following:

1. In the `RSSVRepairPriceMaintTests` class, create a public void method, and name it *TestServicesAndPricesForm*.
2. Specify the `[Fact]` attribute for the method to indicate that this unit test is called without parameters.

Step 2: Creating Necessary Objects

To create the objects that are necessary for testing the business logic, do the following:

1. Create an instance of the `RSSVRepairPriceMaint` graph by adding the following code.

```
var graph = PXGraph.CreateInstance<RSSVRepairPriceMaint>();
```

2. Create a device to repair (an `RSSVDevice` object) and a service for repairing the device (an `RSSVRepairService` object) by adding the following code. You will use the IDs of the device and the service to create an `RSSVRepairPrice` object.

```
graph.Caches[typeof(RSSVDevice)].Insert(new RSSVDevice
{
    DeviceCD = "Device1"
});
graph.Caches[typeof(RSSVRepairService)].Insert(new
 RSSVRepairService
{
    ServiceCD = "Service1"
});
```

When you are creating an object, you need to specify values for its key fields. If you do not specify a value for an object's key field during the creation of the object, you cannot change the field value later.

3. Create an `RSSVRepairPrice` object that will contain details about the repair service as follows.

```
RSSVRepairPrice repairPrice =
    (RSSVRepairPrice)graph.Caches[typeof(RSSVRepairPrice)].
    Insert(new RSSVRepairPrice());
```

You could have specified the values for the key fields, `DeviceID` and `ServiceID`, but you did not because the `DeviceID` and `ServiceID` fields are assigned their values from the `Current` properties of the `RSSVDevice` and `RSSVRepairService` types.

4. Create three repair items—two of the `Battery` type, and one of the `BackCover` type—by using the following code.

```
InventoryItem battery1 = (InventoryItem)graph.
    Caches[typeof(InventoryItem)].Insert(new
    InventoryItem
    {
        InventoryCD = "Battery1"
    });
graph.Caches[typeof(InventoryItemCurySettings)].Insert(new
    InventoryItemCurySettings
    {
        InventoryID = battery1.InventoryID,
        CuryID = "USD"
    });
InventoryItemExt batteryExt1 =
    battery1.GetExtension<InventoryItemExt>();
batteryExt1.UsrRepairItem = true;
batteryExt1.UsrRepairItemType = RepairItemTypeConstants.Battery;
graph.Caches[typeof(InventoryItem)].Update(battery1);

InventoryItem battery2 =
 (InventoryItem)graph.Caches[typeof(InventoryItem)].Insert(new
    InventoryItem
 {
     InventoryCD = "Battery2"
 });
graph.Caches[typeof(InventoryItemCurySettings)].Insert(new
    InventoryItemCurySettings
    {
        InventoryID = battery2.InventoryID,
        CuryID = "USD"
    });
InventoryItemExt batteryExt2 =
    battery2.GetExtension<InventoryItemExt>();
batteryExt2.UsrRepairItem = true;
batteryExt2.UsrRepairItemType = RepairItemTypeConstants.Battery;
graph.Caches[typeof(InventoryItem)].Update(battery2);

InventoryItem backCover1 =
 (InventoryItem)graph.Caches[typeof(InventoryItem)].Insert(new
    InventoryItem
 {
     InventoryCD = "BackCover1"
 });
graph.Caches[typeof(InventoryItemCurySettings)].Insert(new
    InventoryItemCurySettings
    {
```

```
            InventoryID = backCover1.InventoryID,
            CuryID = "USD"
        });
    InventoryItemExt backCoverExt1 =
     backCover1.GetExtension<InventoryItemExt>();
    backCoverExt1.UsrRepairItem = true;
    backCoverExt1.UsrRepairItemType = RepairItemTypeConstants.BackCover;
    graph.Caches[typeof(InventoryItem)].Update(backCover1);
```

This code adds an `InventoryItemCurySettings` object to the cache for each created repair item to give users the ability to specify the price of the repair items.

This code also demonstrates the right order of instructions when you are implementing unit tests: You insert a new object into the cache by calling the `PXCache.Insert` method, and then you change non-key fields of the object and call the `PXCache.Update` method.

5. Create a non-stock item that represents the work to be done by adding the following code.

```
    InventoryItem work1 = (InventoryItem)graph.
        Caches[typeof(InventoryItem)].Insert(new InventoryItem
        {
            InventoryCD = "Work1",
            StkItem = false
        });
```

6. Create repair items based on the stock items, and configure them by adding the following code.

```
    // Configure the back cover repair item
    RSSVRepairItem repairItemBackCover1 =
        (RSSVRepairItem)graph.Caches[typeof(RSSVRepairItem)].Insert(
            new RSSVRepairItem
            {
                InventoryID = backCover1.InventoryID,
                Required = true,
                BasePrice = 10,
                IsDefault = true
            });

    // Configure the first battery repair item
    RSSVRepairItem repairItemBattery1 =
        (RSSVRepairItem)graph.Caches[typeof(RSSVRepairItem)].Insert(
            new RSSVRepairItem
            {
                InventoryID = battery1.InventoryID
            });
    repairItemBattery1.Required = true;
    repairItemBattery1.BasePrice = 20;
    repairItemBattery1.IsDefault = true;
    graph.Caches[typeof(RSSVRepairItem)].Update(repairItemBattery1);

    // Configure the second battery repair item
    RSSVRepairItem repairItemBattery2 =
        (RSSVRepairItem)graph.Caches[typeof(RSSVRepairItem)].Insert(
            new RSSVRepairItem
            { InventoryID = battery2.InventoryID });
    repairItemBattery2.Required = false;
    repairItemBattery2.BasePrice = 30;
    repairItemBattery2.IsDefault = true;
```

```
graph.Caches[typeof(RSSVRepairItem)].Update(repairItemBattery2);
```

This code creates three repair items: one back cover, and two batteries.

7. Add the following code to check the values of the `Required` and `IsDefault` fields.

```
// 2nd battery is not required -> 1st battery is also not required
Assert.False(repairItemBattery1.Required);
// 2nd batt is used by default -> 1st batt is not used by default
Assert.False(repairItemBattery1.IsDefault);
// The back cover's Required and Default fields are not affected
Assert.True(repairItemBackCover1.Required);
Assert.True(repairItemBackCover1.IsDefault);
```

Step 3: Testing the Calculated Values

To test the calculated values, do the following:

1. Add the following code to check the values of the `Required` and `IsDefault` fields.

```
// 2nd battery is not required -> 1st battery is also not required
Assert.False(repairItemBattery1.Required);
// 2nd batt is used by default -> 1st batt is not used by default
Assert.False(repairItemBattery1.IsDefault);
// The back cover's Required and Default fields are not affected
Assert.True(repairItemBackCover1.Required);
Assert.True(repairItemBackCover1.IsDefault);
```

2. Add the following code to ensure that the prices are calculated correctly.

```
RSSVLabor labor = (RSSVLabor)graph.Caches[typeof(RSSVLabor)].
    Insert(new RSSVLabor
    {
        InventoryID = work1.InventoryID,
        DefaultPrice = 2,
        Quantity = 3
    });
Assert.Equal(6, labor.ExtPrice);
Assert.Equal(66, repairPrice.Price);
```

3. Run the created test, and make sure that it succeeds.

Lesson 4: Testing the Display of Errors and Warnings

The topics of this chapter describe how to test whether error or warning messages are displayed properly on the UI when users perform particular actions.

Testing of Errors and Warnings: General Information

The logic of a graph can include the displaying of errors and warnings on the UI. In a test method, you can verify that certain actions cause an error or warning to be displayed.

Learning Objectives

In this chapter, you will learn how to do the following:

- Test the logic of a graph that has a setup DAC
- Test whether a proper error is attached to a field
- Test whether a proper warning is attached to a field
- Clear the cache of errors
- Find the proper DAC object in the cache

Applicable Scenarios

In unit tests, you can check whether an appropriate warning or error is attached to a box when the entered value does not fit the logic of the form.

Creation of an Instance of a Graph That Has a Setup DAC

If a tested graph has setup DACs, you must call the `Setup<Graph>(params IBqlTable[] items)` generic method to initialize and bind the setup DACs. In this case, you specify the tested graph as a type parameter and the objects of the setup DACs as the parameters. After that, you create an instance of the tested graph. If you do not initialize and bind the setup DACs for the test, they will not be used by the graph.

The `Setup<Graph>(params IBqlTable[] items)` method mocks the Acumatica ERP setup preferences. The `Setup` method should not be used to initialize non-setup DACs.

Testing of the Displayed Errors and Warnings

The tested graph may contain the logic that attaches an error or warning to a DAC field in some circumstances. The use of the `PXCache.GetStateExt<>` generic method is the recommended way to get the state of a field to test whether an error or warning is attached to the field. You can also use the `PXCache.GetError`, `PXCache.GetErrors`, and `PXCache.GetWarning` method for it.

You pass the class that corresponds to the tested field as the type parameter of the `PXCache.GetStateExt<>` generic method, and you pass the tested DAC object as the parameter of this method. The method returns a `PXFieldState` object for which the following are true:

- The object's `Error` property contains the error message.
- The `Warning` property of the object contains the warning message.
- The object's `ErrorLevel` property contains the level of the warning or error attached. (The value of this property is `PXErrorLevel.Error` if there is an error attached, and `PXErrorLevel.Warning` if there is a warning attached.)

In your code, after you have tested the displaying of an error or warning, you clear the cache of errors and warnings with the `PXCache.Clear` method.

The attaching of an error or warning to a field may be the result of an action that generates an exception or cancels the assignment of a value. In this case, the cache may contain different DAC objects before and after performing such an action. To pick the proper DAC object for testing, you call the `PXCache.Locate` method and pass an object of that DAC type having the correct values of the key fields.

Activity 4.1: To Test the Display of Errors and Warnings

The following activity will walk you through the process of creating a method that tests the display of warning and error messages.

Story

Suppose that you want to make sure that the following behavior of the Repair Work Orders (RS301000) form has not changed:

- When a new work order is initialized, the system copies to it the repair items and labor items of the `RSSVRepairPrice` object that has the same `DeviceID` and `ServiceID` values as the work order being initialized.
- Negative values of labor quantities are prohibited.
- The values of labor quantities are not permitted to be less than the minimum values.
- In work orders for which the priority is *Low*, repair services that are marked as requiring preliminary checks are prohibited.

You need to create a test method. In this method, you need to initialize the settings of the graph.

Process Overview

You will create a test method without parameters for the Repair Work Orders (RS301000) form. In this method, you will initialize the settings of the tested graph `RSSVWorkOrderEntry` by calling the `Setup<>` generic method. Then you will create an instance of the `RSSVWorkOrderEntry` graph. In the cache of the graph, you will create the necessary objects and configure them. You will assign to the fields values that are not allowed by the business logic of the graph and ensure that the appropriate warnings or errors appear in the UI; you will do this by calling the `PXCache.GetStateExt` method.

System Preparation

Before you begin creating the test method, create the `RSSVWorkOrderEntryTests` test class. For an example that shows how to create a test class, see *Activity 1.2: To Create a Test Class*.

Step 1: Creating a Test Method for the Repair Work Orders Form

In this step, you will create a test method for the Repair Work Orders (RS301000) form. (For basic information on the creation of a test method without parameters, see *Test Method: General Information*.) Do the following:

1. In the `RSSVWorkOrderEntryTests` class, create a public void method, and name it *TestRepairWorkOrdersForm*.
2. Specify the `[Fact]` attribute for the method to indicate that this unit test is called without parameters.
3. Initialize the settings for the `RSSVWorkOrderEntry` graph by adding the following code.

```
Setup<RSSVWorkOrderEntry>(new RSSVSetup());
```

The RSSVSetup DAC contains settings for the RSSVWorkOrderEntry graph. The call of the Setup<> generic method is mandatory because it initializes an RSSVSetup object.

4. Create an instance of the RSSVWorkOrderEntry graph as follows.

```
var graph = PXGraph.CreateInstance<RSSVWorkOrderEntry>();
```

Step 2: Creating the Necessary Objects

In the TestRepairWorkOrdersForm method, create the objects that are needed to test the displaying of warnings and errors as follows:

1. Create an RSSVDevice object and two RSSVRepairService objects (one of these is the main object, and the other is auxiliary) by adding the following code.

```
RSSVDevice device = (RSSVDevice)graph.Caches[typeof(RSSVDevice)].
    Insert(new RSSVDevice { DeviceCD = "Device1" });

RSSVRepairService repairService =
    (RSSVRepairService)graph.Caches[typeof(RSSVRepairService)].
    Insert(new RSSVRepairService
    {
        ServiceCD = "Service1"
    });
RSSVRepairService repairService2 =
    (RSSVRepairService)graph.Caches[typeof(RSSVRepairService)].
    Insert(new RSSVRepairService
    {
        ServiceCD = "Service2"
    });
```

2. Create an RSSVRepairPrice object, and initialize it with the RSSVDevice object and the main RSSVRepairService object. You do this by adding the following code.

```
graph.Caches[typeof(RSSVRepairPrice)].Insert(new RSSVRepairPrice
{
    DeviceID = device.DeviceID,
    ServiceID = repairService.ServiceID
});
```

3. Add the following code to create two stock items and one non-stock item, and to make the two stock items repair items.

```
InventoryItem battery1 =
    (InventoryItem)graph.Caches[typeof(InventoryItem)].Insert(new
    InventoryItem
    {
        InventoryCD = "Battery1"
    });
InventoryItemExt batteryExt1 =
    battery1.GetExtension<InventoryItemExt>();
batteryExt1.UsrRepairItem = true;
batteryExt1.UsrRepairItemType = RepairItemTypeConstants.Battery;
graph.Caches[typeof(InventoryItem)].Update(battery1);

InventoryItem backCover1 =
    (InventoryItem)graph.Caches[typeof(InventoryItem)].Insert(new
```

```
 InventoryItem
    {
        InventoryCD = "BackCover1"
    });
InventoryItemExt backCoverExt1 =
 backCover1.GetExtension<InventoryItemExt>();
backCoverExt1.UsrRepairItem = true;
backCoverExt1.UsrRepairItemType =
    RepairItemTypeConstants.BackCover;
graph.Caches[typeof(InventoryItem)].Update(backCover1);

InventoryItem work1 =
    (InventoryItem)graph.Caches[typeof(InventoryItem)].Insert(new
 InventoryItem
    {
        InventoryCD = "Work1",
        StkItem = false
    });
```

4. Create two repair items based on the two stock items as follows.

```
RSSVRepairItem repairItemBackCover1 =
    (RSSVRepairItem)graph.Caches[typeof(RSSVRepairItem)].Insert(
        new RSSVRepairItem
        {
            DeviceID = device.DeviceID,
            ServiceID = repairService.ServiceID
        });
repairItemBackCover1.InventoryID = backCover1.InventoryID;
repairItemBackCover1.Required = true;
repairItemBackCover1.BasePrice = 10;
repairItemBackCover1.IsDefault = true;
repairItemBackCover1.RepairItemType =
    backCoverExt1.UsrRepairItemType;
graph.Caches[typeof(RSSVRepairItem)].Update(repairItemBackCover1);

RSSVRepairItem repairItemBattery1 =
    (RSSVRepairItem)graph.Caches[typeof(RSSVRepairItem)].Insert(
        new RSSVRepairItem
        {
            DeviceID = device.DeviceID,
            ServiceID = repairService.ServiceID
        });
repairItemBattery1.InventoryID = battery1.InventoryID;
repairItemBattery1.Required = true;
repairItemBattery1.BasePrice = 20;
repairItemBattery1.IsDefault = true;
repairItemBattery1.RepairItemType = batteryExt1.UsrRepairItemType;
graph.Caches[typeof(RSSVRepairItem)].Update(repairItemBattery1);
```

5. Create an `RSSVLabor` object based on the non-stock item by using the following code.

```
RSSVLabor labor = (RSSVLabor)graph.Caches[typeof(RSSVLabor)].
    Insert(new RSSVLabor
    {
        InventoryID = work1.InventoryID,
        DeviceID = device.DeviceID,
        ServiceID = repairService.ServiceID
```

```
        });
labor.DefaultPrice = 2;
labor.Quantity = 3;
graph.Caches[typeof(RSSVLabor)].Update(labor);
```

6. Create a work order with the same `DeviceID` and `ServiceID` values as those specified for the `RSSVRepairPrice` object, which was created in Instruction 2. To do this, add the following code.

```
RSSVWorkOrder workOrder = (RSSVWorkOrder)graph.
    Caches[typeof(RSSVWorkOrder)].Insert(new RSSVWorkOrder());
workOrder.DeviceID = device.DeviceID;
workOrder.ServiceID = repairService.ServiceID;
graph.Caches[typeof(RSSVWorkOrder)].Update(workOrder);
```

Step 3: Testing the Display of Errors and Warnings

Now that the necessary objects have been created, perform the following instructions to test the display of errors and warnings:

1. As the first test, add the following code to make sure that there are two `RSSVWorkOrderItem` objects, which have been created based on the two `RSSVRepairItem` objects, and an `RSSVWorkOrderLabor` object, which has been created based on the `RSSVLabor` object.

```
Assert.Equal(2, graph.RepairItems.Select().Count);
Assert.Single(graph.Labor.Select());
```

2. As the second test, add the following code to ensure that the changing of the `ServiceID` field of the work order does not affect the `RepairItems` and `Labor` views.

```
workOrder.ServiceID = repairService2.ServiceID;
graph.Caches[typeof(RSSVWorkOrder)].Update(workOrder);
Assert.Equal(2, graph.RepairItems.Select().Count);
Assert.Single(graph.Labor.Select());
```

3. Restore the `ServiceID` value as follows.

```
workOrder.ServiceID = repairService.ServiceID;
graph.Caches[typeof(RSSVWorkOrder)].Update(workOrder);
```

4. Obtain the `RSSVWorkOrderLabor` object (the value of its `Quantity` field must be *3*), and try to assign a negative value to its `Quantity` field by using the following code. An error message must be attached to the `Quantity` field instead.

```
RSSVWorkOrderLabor woLabor = graph.Labor.SelectSingle();
Assert.Equal(3, woLabor.Quantity);
woLabor.Quantity = -1;
graph.Caches[typeof(RSSVWorkOrderLabor)].Update(woLabor);
PXFieldState fieldState =
    (PXFieldState)graph.Caches[typeof(RSSVWorkOrderLabor)].
    GetStateExt<RSSVWorkOrderLabor.quantity>(woLabor);
Assert.Equal(PhoneRepairShop.Messages.QuantityCannotBeNegative,
    fieldState.Error);
Assert.Equal(PXErrorLevel.Error, fieldState.ErrorLevel);
graph.Labor.Cache.Clear();
```

In the code, the `PXCache.Clear` method is called to clear the cache of the generated error.

5. By using the following code, assign to the `Quantity` field of the `RSSVWorkOrderLabor` object a value that is less than the value of the corresponding `RSSVLabor.Quantity` field. Then make sure that the following result is achieved: A warning message is attached to the `RSSVWorkOrderLabor.Quantity` field, and the field is assigned the value of the corresponding `RSSVLabor.Quantity` field.

```
woLabor.Quantity = 1;
graph.Caches[typeof(RSSVWorkOrderLabor)].Update(woLabor);
woLabor = (RSSVWorkOrderLabor)graph.
    Caches[typeof(RSSVWorkOrderLabor)].Locate(woLabor);
fieldState = (PXFieldState)graph.
    Caches[typeof(RSSVWorkOrderLabor)].
    GetStateExt<RSSVWorkOrderLabor.quantity>(woLabor);
Assert.Equal(PhoneRepairShop.Messages.QuantityToSmall,
    fieldState.Error);
Assert.Equal(PXErrorLevel.Warning, fieldState.ErrorLevel);
Assert.Equal(3, woLabor.Quantity);
```

   In the code, you use the `PXCache.Locate` method to obtain the proper `RSSVWorkOrderLabor` object from the cache to check the presence and text of the warning.

6. By using the following code, configure the second repair service `repairService2` to require a preliminary check, and assign it to the work order. The code also sets the priority of the work order to *Low* and makes sure that an error message is attached to the `Priority` field of the work order; this error message informs the user that the priority of the work order is too low.

```
repairService2.PreliminaryCheck = true;
graph.Caches[typeof(RSSVRepairService)].Update(repairService2);
workOrder.ServiceID = repairService2.ServiceID;
workOrder.Priority = WorkOrderPriorityConstants.Low;
graph.Caches[typeof(RSSVWorkOrder)].Update(workOrder);
workOrder = (RSSVWorkOrder)graph.Caches[typeof(RSSVWorkOrder)].
    Locate(workOrder);
fieldState = (PXFieldState)graph.Caches[typeof(RSSVWorkOrder)].
    GetStateExt<RSSVWorkOrder.priority>(workOrder);
Assert.Equal(PhoneRepairShop.Messages.PriorityTooLow, fieldState.Error);
Assert.Equal(PXErrorLevel.Error, fieldState.ErrorLevel);
```

7. Run the test method you have created, and make sure that it succeeds.

Appendix: Reference Implementation

You can find the reference implementation of the customization described in this course in the `Customization\T280` folder of the *Help-and-Training-Examples* repository in Acumatica GitHub.

Appendix: Deploying the Needed Acumatica ERP Instance for the Training Course

If for some reason you cannot complete the instructions in *Test Instance: To Deploy an Instance*, you can create an Acumatica ERP instance as described in this topic and manually publish the needed customization project as described in *Appendix: Publishing the Required Customization Project*.

You deploy an Acumatica ERP instance and configure it as follows:

1. To deploy a new application instance, open the Acumatica ERP Configuration Wizard, and do the following:
   a. On the Database Configuration page, type the name of the database: `PhoneRepairShop`.
   b. On the Tenant Setup page, set up a tenant with the *T100* data inserted by specifying the following settings:
      - **Login Tenant Name**: `MyTenant`
      - **New**: Selected
      - **Insert Data**: *T100*
      - **Parent Tenant ID**: *1*
      - **Visible**: Selected
   c. On the **Instance Configuration** page, in the **Local Path of the Instance** box, select a folder that is outside of the `C:\Program Files (x86)`, `C:\Program Files`, and `C:\Users` folder. We recommend that you store the website folder outside of these folders to avoid an issue with permission to work in these folders when you perform customization of the website.

   The system creates a new Acumatica ERP instance, adds a new tenant, and loads the selected data to it.
2. Sign in to the new tenant by using the following credentials:
   - Username: `admin`
   - Password: `setup`

   Change the password when the system prompts you to do so.
3. In the top right corner of the Acumatica ERP screen, click the username and then click **My Profile**. On the **General Info** tab of the *User Profile* (SM203010) form, which the system has opened, select *YOGIFON* in the **Default Branch** box; then click **Save** on the form toolbar.

   In subsequent sign-ins to this account, you will be signed in to this branch.
4. Optional: Add the *Customization Projects* (SM204505) and *Generic Inquiry* (SM208000) forms to your favorites. For details about how to add a form to favorites, see *Managing Favorites: General Information*.

Appendix: Publishing the Required Customization Project

> If for some reason you cannot complete the instructions in *Test Instance: To Deploy an Instance*, you can create an Acumatica ERP instance as described in *Appendix: Deploying the Needed Acumatica ERP Instance for the Training Course* and manually publish the needed customization project as described in this topic.

You load the customization project with the results of the *T220 Data Entry and Setup Forms* training course and publish this project as follows:

1. On the *Customization Projects* (SM204505) form, create a project with the name *PhoneRepairShop*, and open it.
2. In the menu of the Customization Project Editor, click **Source Control > Open Project from Folder**.
3. In the dialog box that opens, specify the path to the `Customization\T220\PhoneRepairShop` folder, which you have downloaded from Acumatica GitHub, and click **OK**.
4. Bind the customization project to the source code of the extension library as follows:
   a. Copy the `Customization\T220\PhoneRepairShop_Code` folder to the `App_Data\Projects` folder of the website.

      > By default, the system uses the `App_Data\Projects` folder of the website as the parent folder for the solution projects of extension libraries.
      >
      > If the website folder is outside of the `C:\Program Files (x86)`, `C:\Program Files`, and `C:\Users` folders, we recommend that you use the `App_Data\Projects` folder for the project of the extension library.
      >
      > If the website folder is in the `C:\Program Files (x86)`, `C:\Program Files`, or `C:\Users` folder, we recommend that you store the project outside of these folders to avoid an issue with permission to work in these folders. In this case, you need to update the links to the website and library references in the project.

   b. Open the solution, and build the `PhoneRepairShop_Code` project.
   c. Reload the Customization Project Editor.
   d. In the menu of the Customization Project Editor, click **Extension Library > Bind to Existing**.
   e. In the dialog box that opens, specify the path to the `App_Data\Projects\PhoneRepairShop_Code` folder, and click **OK**.
5. In the menu of the Customization Project Editor, click **Publish > Publish Current Project**.

   > The **Modified Files Detected** dialog box opens before publication because you have rebuilt the extension library in the `PhoneRepairShop_Code` Visual Studio project. The `Bin\PhoneRepairShop_Code.dll` file has been modified, and you need to update it in the project before the publication.

The published customization project contains all changes to the Acumatica ERP website and database that have been performed in the *T200 Maintenance Forms*, *T210 Customized Forms and Master-Detail Relationship*, and *T220 Data Entry and Setup Forms* training courses. This project also contains the customization plug-ins that fill in the tables created in the *T200 Maintenance Forms*, *T210 Customized Forms and Master-Detail Relationship*, and *T220 Data Entry and Setup Forms* training courses with the custom data entered in these training courses.

For details about the customization plug-ins, see *To Add a Customization Plug-In to a Project*. The creation of customization plug-ins is outside of the scope of this course.