

# Developer Course



## T200 Maintenance Forms 2024 R1

Revision: 5/8/2024

# Contents

<b>Copyright.....</b>	<b>5</b>
<b>How to Use This Course.....</b>	<b>6</b>
<b>Company Story and Customization Description.....</b>	<b>8</b>
<b>Getting Started.....</b>	<b>11</b>
Initial Configuration.....	11
Application Programming Overview.....	12
Querying of the Data.....	14
<b>Part 1: Creating a Form with the Customization Project Editor (Repair Services Form).....</b>	<b>16</b>
Maintenance Forms.....	16
Lesson 1.1: Prepare a Customization Project.....	16
Customization Projects.....	17
Step 1.1.1: Create the Customization Project.....	17
Step 1.1.2: Add a Database Table Schema.....	17
Lesson Summary.....	18
Lesson 1.2: Create a Form.....	18
Step 1.2.1: Use the New Screen Wizard to Create a Form Template.....	19
Step 1.2.2: Configure Access Rights for the Created Form.....	21
Step 1.2.3: Include Access Rights of the Created Form in the Customization Project.....	22
Analysis of the Generated Code of the Graph.....	24
Lesson Summary.....	25
Lesson 1.3: Make the New Form Visible in the UI.....	25
Step 1.3.1: Create a Workspace.....	25
Step 1.3.2: Add the Link to the Workspace.....	26
Step 1.3.3: Update the SiteMapNode Item.....	28
Lesson Summary.....	29
Lesson 1.4: Configure the Data Access Class .....	29
Definition of Data Access Classes .....	29
Step 1.4.1: Generate a DAC.....	30
Step 1.4.2: Configure the Attributes of the New DAC.....	31
Step 1.4.3: Configure a View.....	35
Lesson Summary.....	37
Lesson 1.5: Configure the Form.....	37
Step 1.5.1: Add Columns to the Grid.....	37
Step 1.5.2: Test the Form.....	40

Lesson Summary.....	42
Lesson 1.6: Add an Event Handler to the Walk-In Service Check Box.....	42
Step 1.6.1: Add an Event Handler in the Customization Project Editor.....	43
Step 1.6.2: Specify the CommitChanges Property.....	44
Step 1.6.3: Test the Event Handler.....	44
Lesson Summary.....	45
Lesson 1.7: Debug the Customization Code.....	45
Step 1.7.1: Debug the Customization Code.....	45
Lesson Summary.....	47
Lesson 1.8: Move the Customization Code to an Extension Library.....	47
Extension Libraries.....	47
Step 1.8.1: Create an Extension Library.....	48
Step 1.8.2: Move Code from the Customization Project to the Extension Library.....	49
Step 1.8.3: Open Solution in Visual Studio.....	51
Step 1.8.4: Build the Project in Visual Studio.....	52
Step 1.8.5: Include the Extension Library in the Customization Project.....	53
Lesson Summary.....	53
Lesson 1.9: Add an Event Handler In Visual Studio.....	54
Step 1.9.1: Add an Event Handler in Visual Studio.....	54
Step 1.9.2: Use Acuminator to Refactor the Event Handler Declaration.....	55
Step 1.9.3: Test the Event Handlers (Self-Guided Exercise).....	55
Lesson Summary.....	56
Part 1 Summary.....	56
<b>Part 2: Creating a Form with the Visual Studio (Serviced Devices Form).....</b>	<b>58</b>
Initial Steps.....	58
Lesson 2.1: Create a Graph and a DAC in Visual Studio.....	58
Step 2.1.1: Define the RSSVDeviceMaint Graph.....	58
Step 2.1.2: Create a DAC in Visual Studio.....	59
Step 2.1.3: Define DAC Fields in Visual Studio.....	61
Step 2.1.4: Configure the RSSVDeviceMaint Graph.....	64
Lesson Summary.....	64
Lesson 2.2: Create an ASPX Page in Visual Studio.....	64
Step 2.2.1: Create the RS202000.aspx Page.....	65
Step 2.2.2: Add ASPX and ASPX.CS Files to the Customization Project.....	66
Lesson Summary.....	67
Lesson 2.3: Configure a Form in Visual Studio.....	67

Step 2.3.1: Add Input Controls.....	67
Step 2.3.2: Configure the Layout.....	68
Step 2.3.3: Update the Files in the Customization Project.....	68
Lesson Summary.....	69
Lesson 2.4: Add the Form to the Site Map and Workspace.....	69
Step 2.4.1: Create a Site Map Item for the Form.....	70
Step 2.4.2: Add the Site Map Item to the Customization Project.....	71
Step 2.4.3: Add the Form to the Screen Editor.....	72
Step 2.4.4: Test the Form.....	74
Lesson Summary.....	75
Lesson 2.5: Create a Substitute Form.....	75
Step 2.5.1: Upload a Predefined Generic Inquiry.....	76
Step 2.5.2: Configure the Generic Inquiry as a Substitute Form.....	77
Step 2.5.3: Save the Generic Inquiry to the Customization Project.....	78
Step 2.5.4: Test the Substitute Form.....	79
Lesson Summary.....	79
Part 2 Summary.....	80
<b>Appendix: Customization Projects.....</b>	<b>81</b>

# Copyright

---

© 2024 Acumatica, Inc.

**ALL RIGHTS RESERVED.**

No part of this document may be reproduced, copied, or transmitted without the express prior consent of Acumatica, Inc.

3075 112th Avenue NE, Suite 200, Bellevue, WA 98004, USA

## 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: 2024 R1

Last Updated: 05/08/2024

# How to Use This Course

---

The Acumatica Cloud xRP Platform is the platform provided by Acumatica that is used to build the Acumatica ERP application itself, any customizations of Acumatica ERP, the mobile application for Acumatica ERP, and applications integrated with Acumatica ERP through the web services API.

Acumatica Framework provides the platform API and web controls for the development of the UI and business logic of an ERP application. The platform API is used for the development of Acumatica ERP and any embedded applications (that is, customizations of Acumatica ERP). Acumatica Framework can also be used to develop an ERP application from scratch.

Acumatica Customization Platform provides customization tools for the development of applications embedded in Acumatica ERP. Developers that work with Acumatica Customization Platform use the platform API provided by Acumatica Framework.

The *T200 Maintenance Forms* course introduces to you the main concepts of Acumatica Framework and Acumatica Customization Platform based on examples of the creation of simple Acumatica ERP forms.

The course is intended for application developers who are starting to learn how to customize Acumatica ERP.

The course is based on a set of examples that demonstrate the general approach to customizing Acumatica ERP. In the process of completing the examples, you will gain ideas about how to develop your own embedded applications by using the customization tools. As you go through the course, you will start to develop the customization for a cell phone repair shop, which you will continue in the further courses of the *T* series.

After you complete all the lessons of the course, you will be familiar with the basic programming techniques for the customization of Acumatica ERP.



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.

## What the Course Prerequisites Are

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

## What Is in a Part

The first part of the course explains how to create a custom Acumatica ERP form by using the Customization Project Editor and how to move the code to an extension library.

The second part of the course explains how to create a new form in Visual Studio and configure a substitute form.

Each part of the course consists of lessons you should complete.

## What Is in a Lesson

Each lesson is dedicated to a particular development scenario that you can implement by using Acumatica ERP customization tools and Acumatica Framework. Each lesson consists of a brief description of the scenario and an example of the implementation of this scenario.

The lesson may also include *Additional Information* topics, which are outside of the scope of this course but may be useful to some readers.

Each lesson ends with a *Lesson Summary* topic, which summarizes the development techniques used during the implementation of the scenario.

## Where the Source Code Is

You can find the source code of the customization described in this course and code snippets for the course in the `Customization\T200` folder of the [Help-and-Training-Examples](#) repository in Acumatica GitHub.

## What the Documentation Resources Are

The complete Acumatica ERP documentation is available on <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 this functionality, you have to activate the license your organization has purchased. Each particular feature may be subject to additional licensing; please consult the Acumatica ERP sales policy for details.

# Company Story and Customization Description

This topic describes the company story and explains what should be customized to meet the company's needs.

## 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, the Acumatica ERP instance of the Smart Fix company needs to be complemented with two maintenance forms: Repair Services and Serviced Devices. In this course, you will customize Acumatica ERP by developing these maintenance forms.

## Database Schema

For the customization task, two new tables are required: a table containing information about repair services, and a table containing information about the serviced devices, as described in the previous section. You will add these tables to the database when you complete [Initial Configuration](#).



The design of database tables is outside of the scope of this course. For details, see [Designing the Database Structure and DACs](#).

The table containing information about the provided services is called `RSSVRepairService` and contains the following custom columns:

- **ServiceID:** Serves as a primary key identifying a service.
- **ServiceCD:** Contains a service code.  
In Acumatica ERP, *CD* is used for natural keys (such as `ServiceCD`), which means keys that are human-readable and can have additional meaning. *ID* is used for surrogate keys (such as `ServiceID`), which are pure identifiers. For details, see [Naming Conventions for Tables \(DACs\) and Columns \(Fields\)](#).
- **Description:** Contains a description of a repair service.
- **Active:** Indicates whether a service is active at the moment.
- **WalkInService:** Indicates whether a service is provided immediately after a customer requested it.
- **PreliminaryCheck:** Indicates whether a service is provided after a preliminary diagnostic check.
- **Prepayment:** Indicates whether a service requires prepayment.

The table containing information about devices is called `RSSVDevice` and contains the following custom columns:

- **DeviceID:** Serves as a primary key identifying the device.
- **DeviceCD:** Contains the device code.
- **Description:** Contains a description of the device.
- **Active:** Indicates whether the device is being serviced at the moment.
- **AvgComplexityOfRepair:** Contains one of three possible values indicating the level of complexity of the repair: *Low*, *Medium*, or *High*.



## The Repair Services Form

The Repair Services form, which you will develop, will be used to view the list of services provided by the company. By clicking buttons on the form toolbar, users will be able to add a new service, edit an existing service, and delete a service. The following screenshot shows what this form should look like.

The screenshot shows a web form titled "Repair Services". At the top right, there are tabs for "CUSTOMIZATION" and "TOOLS" with a dropdown arrow. Below the title is a toolbar with icons for refresh, save, undo, add (+), delete (x), zoom in, and zoom out. The main content area contains a table with the following data:

		* Service ID	* Description	Active	Walk-In Service	Requires Prepaym	Requires Prelimina Check
>			BATTERYREPLACE	Battery Replacement	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
			LIQUIDDAMAGE	Liquid Damage	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
			SCREENREPAIR	Screen Repair	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>

At the bottom right of the form, there are navigation buttons: "<|", "<", ">", and ">|".

**Figure: Service list on the Repair Services form**

The Repair Services form will use the `RSSVRepairService` table.

## The Serviced Devices Form

You will also develop the Serviced Devices form, which will be used to view the list of devices that are serviced by the company. When a user brings up the form, the user will initially see a list of devices displayed in a grid. When the user selects a device in the grid, a detail view of the record will be displayed. (The following screenshots illustrate what these views look like.)

Service

Devices

CUSTOMIZATION

TOOLS

↺

↻

+

✎

↔

✕

Drag column header here to configure filter

🔍

💾

⋮

		Device Code	Description	Active	Complexity
>	🔗	<a href="#">IPHONE6</a>	iPhone 6	<input checked="" type="checkbox"/>	High
	🔗	<a href="#">MOTORRAZR</a>	Motorola RAZR V3	<input checked="" type="checkbox"/>	Low
	🔗	<a href="#">NOKIA3310</a>	Nokia 3310	<input checked="" type="checkbox"/>	Low
	🔗	<a href="#">SAMSUNGGS4</a>	Samsung Galaxy S4	<input checked="" type="checkbox"/>	Medium

Service

Devices

NOTES

FILES

CUSTOMIZATION

TOOLS

←

📄

💾

↺

+

🗑

📋

⏪

<

>

⏩

\* Device Code:

MOTORRAZR

🔍

Description:

Motorola RAZR V3

Complexity:

Low

▼

☒ Active

**Figure: List and detail views of the Serviced Devices form**

The Serviced Devices form will use the `RSSVDevice` table.

# Getting Started

---

In this part of the course, you will get an overview of application programming with Acumatica Framework.

## Initial Configuration

---

You need to perform the prerequisite actions before you start to complete the course.

### Step 1: Preparing the Environment

You should prepare the environment for the training course as follows:

1. Make sure that the environment that you are going to use conforms to the [System Requirements for Acumatica ERP 2024 R1](#).
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. Clone or download the customization project and the source code of the extension library from the [Help-and-Training-Examples](#) repository in Acumatica GitHub to a folder on your computer.
5. Install Acumatica ERP. On the Main Software Configuration page of the Acumatica ERP Setup Wizard, 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 the Needed Acumatica ERP Instance for the Training Course

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

1. Open the Acumatica ERP Configuration Wizard, and do the following:
  - a. Click **Deploy a New Acumatica ERP Instance for T-Series Developer Courses**.
  - b. On the **Instance Configuration** page, do the following:
    - a. In the **Training Course** box, select *T200 Maintenance Forms*.
    - b. In the **Local Path to the Instance** box, select a folder that is outside of the C:\Program Files (x86), C:\Program Files, and C:\Users folders. (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 customize the website.)
  - c. On the **Database Configuration** page, make sure the name of the database is SmartFix\_T200.

The system creates a new Acumatica ERP instance, adds a new tenant, and loads the 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**. The [User Profile](#) (SM203010) form opens. On the **General Info** tab, 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 your favorites, see [Favorites: General Information](#).

### Step 3: Creating the Database Tables

Add the `RSSVRepairService` and `RSSVDevice` tables to the instance database by executing the `Customization\T200\SourceFiles\DBScripts\T200_DatabaseTables.sql` script, which you have downloaded from Acumatica GitHub.

Before you can customize Acumatica ERP, tables for the instance database need to be designed and added to the database. For this course, the database scripts have been prepared in advance. This is why you needed to add them to the instance database.



The design of database tables is outside of the scope of this course. For details on designing database tables for Acumatica ERP, see [Designing the Database Structure and DACs](#).

## Application Programming Overview

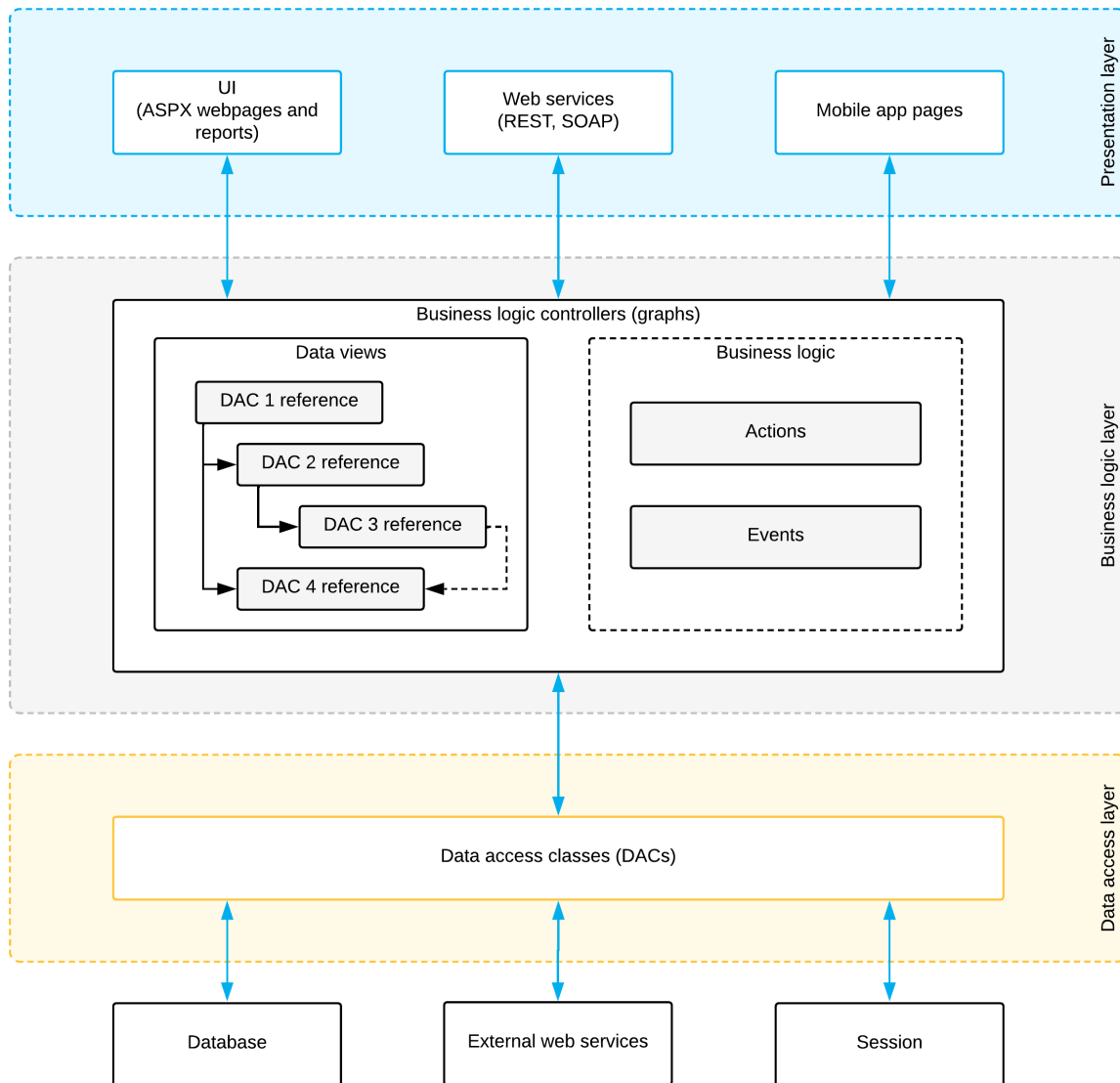
---

Acumatica Framework provides the platform and tools for developing cloud business applications. This topic explains the runtime structure of Acumatica Framework and introduces the main components of this platform.

### Runtime Structure and Components

An application written with Acumatica Framework has *n*-tier architecture with a clear separation of the presentation, business, and data access layers, as shown in the following diagram. You can find details about each layer in the sections below.

## Application architecture



### Data Access Layer

The data access layer of an application written using Acumatica Framework is implemented as a set of data access classes (DACs) that wrap data from database tables or data received through other external sources (such as Amazon Web Services).

The instances of data access classes are maintained by the business logic layer. Between requests, these instances are stored in the session. On a standalone Acumatica ERP server, session data is stored in the server memory. In a cluster of application servers, session data is serialized and stored in a high-performance remote server through a custom optimized serialization mechanism.

For details about data storage in a session, see [Session](#). For details on working with the data access layer, see [Accessing Data](#).

## Business Logic Layer

The business logic is implemented through the business logic controller (also called *graph*). Graphs are classes that you derive from the special API class (`PXGraph`) and that are tied to one or more data access classes.

Each graph conceptually consists of two parts:

- Data views, which include the references to the required data access classes, their relationships, and other meta information
- Business logic, which consists of actions and events associated with the modified data.

Each graph can be accessed from the presentation layer or from the application code that is implemented within another graph. When the graph receives an execution request, it extracts the data required for request execution from the data access classes included in the data views, triggers business logic execution, returns the result of the execution to the requesting party, and updates the data access classes instances with the modified data.

For details on working with the business logic layer, see [Implementing Business Logic](#).

## Presentation Layer

The presentation layer provides access to the application business logic through the UI, web services, and Acumatica mobile application. The presentation layer is completely declarative and contains no business logic.

The UI consists of ASPX webpages (which are based on the ASP.NET Web Forms technology) and reports created with Acumatica Report Designer. The ASPX webpages are bound to particular graphs.

When the user requests a new webpage, the presentation layer is responsible for processing this request. Webpages are used for generating static HTML page content and providing additional service information required for the dynamic configuration of the web controls. When the user receives the requested page and starts browsing or entering data, the presentation layer is responsible for handling asynchronous HTTP requests. During processing, the presentation layer submits a request to the business logic layer for execution. Once execution is completed, the business logic layer analyzes any changes in the graph state and generates the response that is sent back to the browser as an XML document.

For details on the configuration of ASPX webpages, see [Configuring ASPX Pages and Reports](#).

## Querying of the Data

Acumatica Framework provides a custom language called *BQL (business query language)* that developers can use for writing database queries. BQL is written in C# and based on generic class syntax, but is still very similar to SQL syntax.

Acumatica Framework provides two dialects of BQL: traditional BQL and fluent BQL. We recommend that you use fluent BQL because statements written in fluent BQL are simpler and shorter than the ones written with traditional BQL. Further in this topic, the examples are written in fluent BQL.



You can also use LINQ to select records from the database or to apply additional filtering to the data of a BQL query. For details on which approach to use, see [Comparison of Fluent BQL, Traditional BQL, and LINQ](#).

BQL has almost the same keywords as SQL does, and they are placed in the same order as they are in SQL, as shown in the following example of BQL.

```
SelectFrom<Product>.Where<Product.availQty.IsNotNull.  
    And<Product.availQty.IsGreater<Product.bookedQty>>>
```

If the database provider is Microsoft SQL Server, the framework translates this expression into the following SQL query.

```
SELECT * FROM Product
WHERE Product.AvailQty IS NOT NULL
AND Product.AvailQty > Product.BookedQty
```

BQL extends several benefits to the application developer. It does not depend on the specifics of the database provider, and it is object-oriented and extendable. Another important benefit of BQL is compile-time syntax validation, which helps to prevent SQL syntax errors.

Because BQL is implemented on top of generic classes, you need data types that represent database tables. In the context of Acumatica Framework, these types are called *data access classes (DACs)*. As an example of a DAC, you would define the `Product` data access class as shown in the following code fragment to execute the SQL query from the previous code example.

```
using System;
using PX.Data;

[PXCacheName("Product")]
public class Product : PX.Data.PXBqlTable, PX.Data.IBqlTable
{
    // The property holding the ProductID value in a record
    [PXDBIdentity(IsKey = true)]
    public virtual int? ProductID { get; set; }
    // The type used in BQL statements to reference the ProductID column
    public abstract class productID : PX.Data.BQL.BqlInt.Field<productID> { }

    // The property holding the AvailQty value in a record
    [PXDBDecimal(2)]
    public virtual decimal? AvailQty { get; set; }
    // The type used in BQL statements to reference the AvailQty column
    public abstract class availQty : PX.Data.BQL.BqlDecimal.Field<availQty> { }

    // The property holding the BookedQty value in a record
    [PXDBDecimal(2)]
    public virtual decimal? BookedQty { get; set; }
    // The type used in BQL statements to reference the BookedQty column
    public abstract class bookedQty : PX.Data.BQL.BqlDecimal.Field<bookedQty> { }
}
```

Each table field is declared in a data access class in two different ways, each for a different purpose:

- As a `public virtual` property (which is also referred to as a *property field*) to hold the table field data
- As a `public abstract class` (which is also referred to as a *class field* or *BQL field*) to reference a field in the BQL command

You will learn more about data access classes later in this course.

## Related Links

- [Querying Data in Acumatica Framework](#)

# Part 1: Creating a Form with the Customization Project Editor (Repair Services Form)

---

In this part of the course, you will start with creating the first simple form of the application. You will create a maintenance form, which is used to enter and maintain data that will be used on the main forms of the application: data entry and processing forms.

In the Smart Fix company, when a user enters an order, the particular repair services of the order need to be recorded. The user can select a particular repair service more quickly than type a description of it, and typed descriptions would not be usable in inquiry or processing forms. While the company currently offers only a small number of services (which is seldom added to, so a data entry form would not be useful), the set of repair services may change over time, as the company expands and devices evolve. Thus, adding a drop-down box for the repair service would not be a good option. Instead, the company needs a maintenance form where repair services can be entered, maintained, and deleted or added as needed.

In this part of the course, you will design the Repair Services maintenance form, which will hold a list of the services the repair shop provides and their basic settings.

## Maintenance Forms

---

*Maintenance forms* are forms on which data can be entered about particular types of entities, which are then available for selection on other forms. Compared with data entry forms, maintenance forms are generally used to define fewer entities and are used more rarely.

When entities of a particular type have been defined on a maintenance form, users can select rather than type them on a data entry form. However, unlike predefined options in a drop-down box, items defined on a maintenance form and selected on other forms can be added by any user and made immediately available for selection. The entities can also be selected on other types of forms, so that users can view (on an inquiry form or report) and process (on a processing form) data filtered or organized by particular entities of the type.

For instance, in Acumatica ERP, a data entry form is used to enter AR invoices. Some of the settings for an invoice can be defined on a maintenance form, such as credit terms used by customers to pay the company. These maintenance entities are entered less frequently and are fewer in number than AR invoices are.

## Lesson 1.1: Prepare a Customization Project

---

In this lesson, you will create a customization project, in which you will create maintenance forms as you complete this course. You will add the first item, a database script, to the customization project.

### Lesson Objectives

As you complete this lesson, you will learn the following:

- What customization project is
- How to create a customization project
- How to add a database script for a database table to the customization project



## Customization Projects

---

A *customization project* is a set of changes to the user interface, configuration data, and functionality of Acumatica ERP. The customization project holds the changes that have been made for a particular customization, which might include changes to the mobile site map, generic inquiries, and the properties of UI elements.

To apply the content of a customization project to an instance of Acumatica ERP, you have to publish the project. Before the project is published, the changes exist only in the project and are not yet applied to an instance.

For details on customization projects, see [Customization Project](#).

### Step 1.1.1: Create the Customization Project

---

The creation of a customization project is a first step in the customization of Acumatica ERP. To create the customization project you will use in this course, do the following:

1. In Acumatica ERP, open the [Customization Projects](#) (SM204505) form.
2. On the form toolbar, click **Add Row**.
3. In the **Project Name** column, enter the customization project name: *PhoneRepairShop*.
4. On the form toolbar, click **Save**.

You have created the customization project. In the next step, you will open the Customization Project Editor and begin the customization.

#### Related Links

- [To Create a New Project](#)

### Step 1.1.2: Add a Database Table Schema

---

In this step, you will add a table schema for the `RSSVRepairService` table, which you added to the instance database as part of the course prerequisite steps. When you publish a customization on a different instance of Acumatica ERP, the same table is created in the instance database based on the schema provided in the customization project.

For details on database script items, see [Database Scripts](#).



The design of database tables is outside of the scope of this course. For details on designing database tables for Acumatica ERP, see [Designing the Database Structure and DACs](#).

To add a table schema, do the following:

1. On the [Customization Projects](#) (SM204505) form, open the *PhoneRepairShop* customization project. The system opens the project in the Customization Project Editor.
2. In the navigation pane, click **Database Scripts**.
3. On the More menu of the Database Scripts page, click **Add Custom Table Schema**.



When you need to add a custom table to the instance database, we recommend adding the custom table schema to the customization project, not the custom table script, because a possible result of a custom SQL script is the loss of the integrity and consistency of the application data. For details, see [Changes in the Database Schema](#).

4. In the **Add Custom Table Schema** dialog box, which the system opens, start typing *RSSVRepairService* in the **Table** box, and select the *RSSVRepairService* option.
5. Click **OK**.

The script for the *RSSVRepairService* table has been added to the customization project.



In the project, the schema is kept in XML format. When the customization project is published, Acumatica Customization Platform will execute a procedure to create the table according to the schema, while meeting all the requirements of Acumatica ERP.

#### Related Links

- [To Add a Custom SQL Script to a Project](#)

## Lesson Summary

In this lesson, you have learned how to create customization projects and add database scripts to the project.

As you have completed the lesson, you have created the *PhoneRepairShop* customization project on the [Customization Projects](#) (SM204505) form. You have used the Customization Project Editor to add database scripts for custom tables to the customization project.

## Lesson 1.2: Create a Form

In this lesson, you will use the Customization Project Editor to create a simple form with a grid.

The Repair Services maintenance form will hold the list of services that the repair shop provides. The form will contain a toolbar and a grid. The columns of the grid are listed below, along with the data type and description of each.

Column Name	Data Type	Description
<b>Service ID</b>	String	The identifier of the service
<b>Description</b>	String	The description of the service
<b>Active</b>	Boolean	An indicator of whether a service is currently provided by the shop
<b>Walk-In Service</b>	Boolean	An indicator of whether this is a walk-in service
<b>Requires Preliminary Check</b>	Boolean	An indicator of whether the service requires diagnostic checks
<b>Requires Prepayment</b>	Boolean	An indicator of whether this service should be prepaid

## Lesson Objective

As you complete this lesson, you will learn how to create a form of the application by generating the needed items with the New Screen wizard.

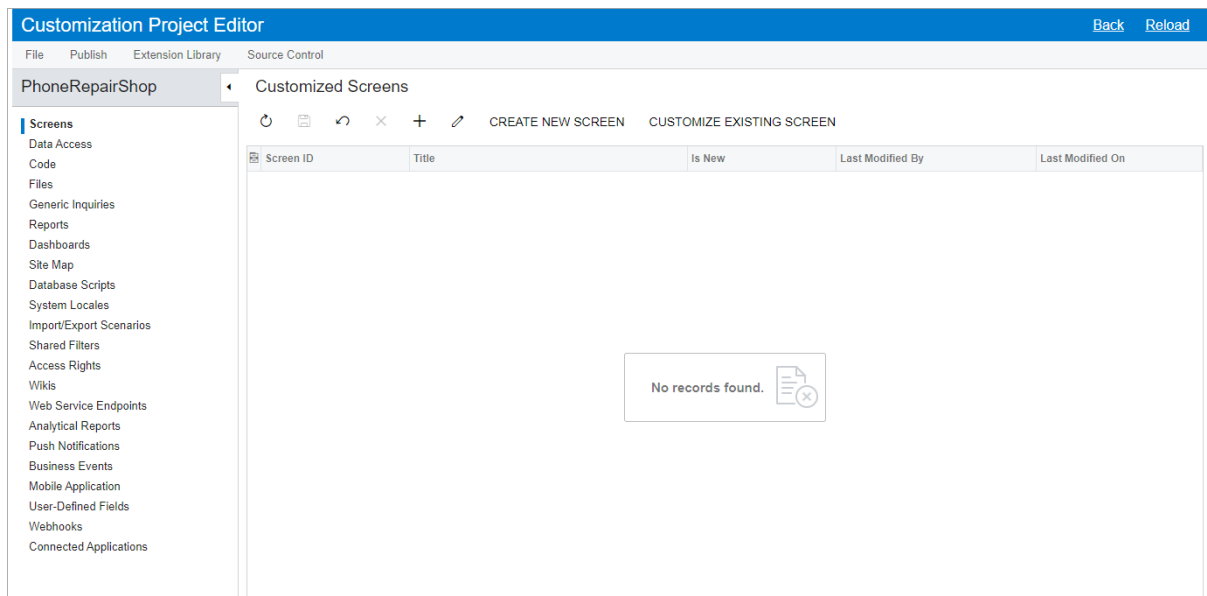
### Step 1.2.1: Use the New Screen Wizard to Create a Form Template

To simplify the process of creating a new form, the Acumatica Customization Platform provides the New Screen wizard, which creates a workable template for a new form. You open the New Screen wizard from the Customized Screen page of the Customization Project Editor.

To create a form template for the Repair Services form, do the following:

1. Open the **PhoneRepairShop** customization project in the Customization Project Editor: Click the *PhoneRepairShop* project name on the [Customization Projects](#) (SM204505) form.
2. On the navigation pane, click the **Screens** node.

The Customized Screen page opens with a blank table, as shown in the following screenshot.



**Figure: The Customized Screens page of the Customization Project Editor**

3. On the page toolbar, click **Create New Screen**.
4. In the **Create New Screen** dialog box, which Acumatica Customization Platform opens, specify the following values, as shown in the screenshot below:

- **Screen ID:** RS.20.10.00

The form ID complies with the following Acumatica Framework conventions: *RS* is a two-letter identifier indicating the part of the functional area (for Acumatica ERP) or subject area (which in this case is phone repair), *20* indicates a maintenance type of the form, and *10* is the number of the maintenance form in *RS*, which is generally sequential. For more information on naming conventions, see [Form and Report Numbering](#).

- **Graph Name:** RSSVRepairServiceMaint

Every page must be associated with a graph, and the graph's name should start with a prefix and end with a suffix. The prefix consists of the two-letter identifier indicating the part of the functional area (in

this case, *RS*) and a two-letter prefix of the application area (in this case, *SV* to indicate service). The suffix indicates the type of the form the graph is used for, in this case, *Maint*. For details, see [Graph Naming](#).

- **Graph Namespace:** PhoneRepairShop. This box is filled automatically.
- **Page Title:** Repair Services
- **Template:** Grid (GridView)

A form template determines which basic containers the form will have: a form with boxes, a grid, a tab, or a combination of these containers. For details, see [To Create a Custom Form Template](#).

**Figure: The Create New Screen dialog box**

- Click **OK** to create the form with these settings.

The Code Editor page opens with the generated code of the `RSSVRepairServiceMaint` class.

The wizard creates the form template and adds the following items to the customization project (all of which can be viewed in the navigation pane of the Customization Project Editor).

Item	Description
<i>RS201000</i>	This <i>Screen</i> item contains the content of the new form.
<i>RSSVRepairServiceMaint</i>	This <i>Code</i> item contains the code template of the graph for the new form. This item is saved in the database. When you publish the project, the platform creates a copy of the code in the <code>RSSVRepairServiceMaint.cs</code> file in the <code>App_RuntimeCode</code> folder of the Acumatica ERP application instance.
<i>Pages\RS\RS201000.aspx</i> <i>Pages\RS\RS201000.aspx.cs</i>	These <i>File</i> items contain ASPX page code for the new form. When you publish the customization project for the first time, the platform creates the files in the <code>Pages\RS</code> folder of the Acumatica ERP application instance, and the platform creates copies of these files in the <code>pages_RS</code> subfolder of the <code>CstPublished</code> folder of the instance.
<i>Repair Services</i>	This <i>SiteMapNode</i> item contains the site map object of the new form.

- Publish the customization project. To do that, on the main menu of the Customization Project Editor, select **Publish > Publish Current Project**.

The system opens the **Compilation** pane and displays the progress of publication in it. The publication is completed when you see the *Website updated.* message in the dialog box.

- Close the **Compilation** pane.

#### Related Links

- [To Create a Custom Form Template](#)

- [To Add a New Custom Form to a Project](#)
- [Graph](#)

## Step 1.2.2: Configure Access Rights for the Created Form

Whenever you create a new form in Acumatica ERP, you need to configure the necessary access rights for the appropriate roles before this form can be used.

To configure the access rights for the Repair Services (RS201000) form, which you created in the preceding step, do the following:

1. In Acumatica ERP, open the [Access Rights by Screen](#) (SM201020) form.
2. In the left pane, expand the **Hidden** node; find the **Repair Services** node in the expanded list of items, and click it.
3. With the **Repair Services** node selected in the left pane, in the table on the right pane, click the row with *Administrator* in the **Role** column, and change the value in the **Access Rights** column from *Revoked* to *Delete*.
4. Click the row with *Customizer* in the **Role** column, and change the value in the **Access Rights** column from *Revoked* to *Delete*.
5. On the form toolbar, click **Save**.



Refresh your browser window to ensure that the permission changes have taken effect.

The following screenshot shows the configured access rights for the Repair Services form on the Access Rights by Screen form.

Role	Description	Access Rights	Applied to Nested
AcumaticaSupport	Role for Acumatica Support. Access similar to ...	Revoked	<input checked="" type="checkbox"/>
Administrator	System Administrator	Delete	<input checked="" type="checkbox"/>
Anonymous	Anonymous	Revoked	<input checked="" type="checkbox"/>
AP Admin	Access to AP functions and settings	Revoked	<input checked="" type="checkbox"/>
AP Clerk	Access to AP functions	Revoked	<input checked="" type="checkbox"/>
AP Viewer	Read-only access to AP functions	Revoked	<input checked="" type="checkbox"/>
AR Admin	Access to AR functions and settings	Revoked	<input checked="" type="checkbox"/>
AR Clerk	Access to AR functions	Revoked	<input checked="" type="checkbox"/>
AR Viewer	Read-only access to AR functions	Revoked	<input checked="" type="checkbox"/>
Archivist	Archive Manager	Revoked	<input checked="" type="checkbox"/>
BI	Access to Business Intelligence Views	Revoked	<input checked="" type="checkbox"/>
CA Admin	Access to CA functions and settings	Revoked	<input checked="" type="checkbox"/>
CA Clerk	Access to CA functions	Revoked	<input checked="" type="checkbox"/>
CA Viewer	Read-only access to CA functions	Revoked	<input checked="" type="checkbox"/>
CM Admin	Access to CM functions and settings	Revoked	<input checked="" type="checkbox"/>
CM Viewer	Read-only access to CM functions	Revoked	<input checked="" type="checkbox"/>
CR Marketing Manager	Access to Marketing functions and settings	Revoked	<input checked="" type="checkbox"/>
CR Sales & Marketing Admin	Full access to Sales and Marketing functions a...	Revoked	<input checked="" type="checkbox"/>
CR Sales Representative	Access to Sales functions and settings	Revoked	<input checked="" type="checkbox"/>
CR Support Admin	Full access to Support functionality and settings	Revoked	<input checked="" type="checkbox"/>
CR Support Representative	Access to Support functionality and settings	Revoked	<input checked="" type="checkbox"/>
CR Viewer	Read-only access to marketing, sales and sup...	Revoked	<input checked="" type="checkbox"/>
CS Admin	Access to system functions and configuration l...	Revoked	<input checked="" type="checkbox"/>
Customer Data Manager	Full access to data entry forms related to custo...	Revoked	<input checked="" type="checkbox"/>
Customizer	Customizer	Delete	<input checked="" type="checkbox"/>

**Figure: The configured access rights for the Repair Services form**

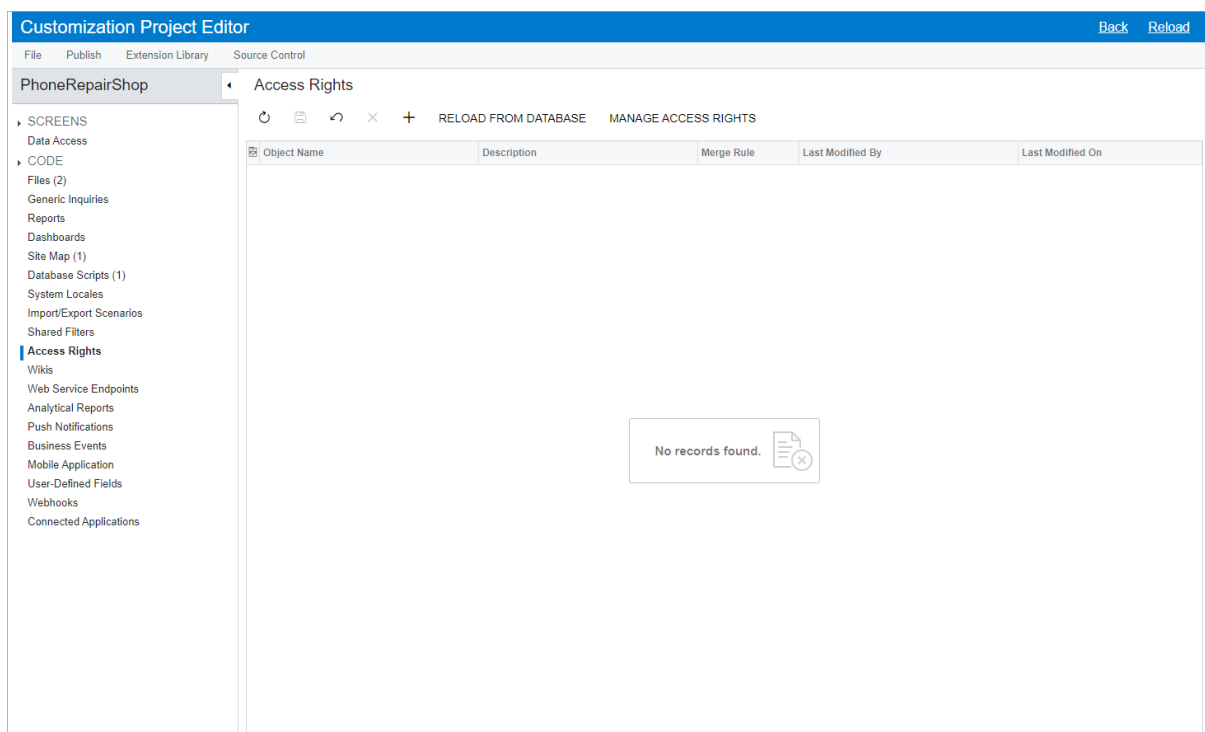
### Step 1.2.3: Include Access Rights of the Created Form in the Customization Project

After you configure the access rights for a new form on the [Access Rights by Screen](#) (SM201020) form, you should include these access rights in the customization project where you created this new form. This ensures that the access rights do not need to be configured for the same roles again when this customization project is published on another Acumatica ERP instance.

To include the access rights for the Repair Services (RS201000) form in the *PhoneRepairShop* customization project, do the following:

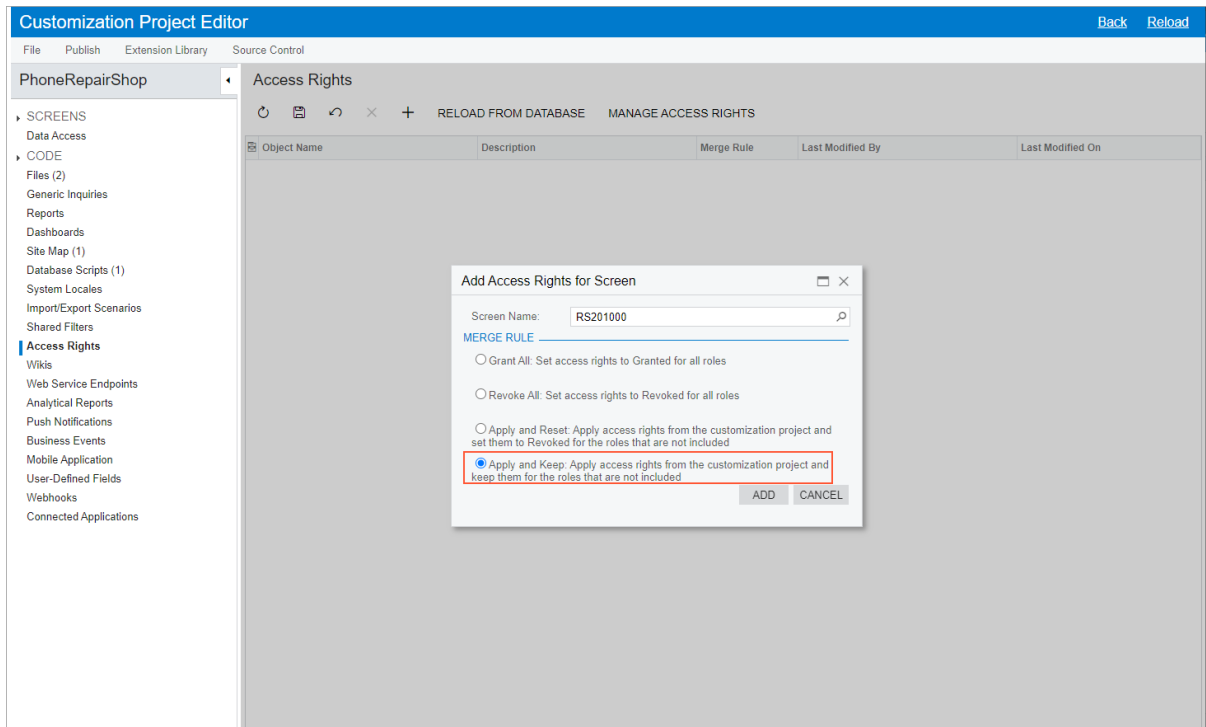
1. In the Customization Project Editor, open the *PhoneRepairShop* customization project.
2. On the navigation pane, click the **Access Rights** node.

The Access Rights page opens with a blank table, as shown in the following screenshot.



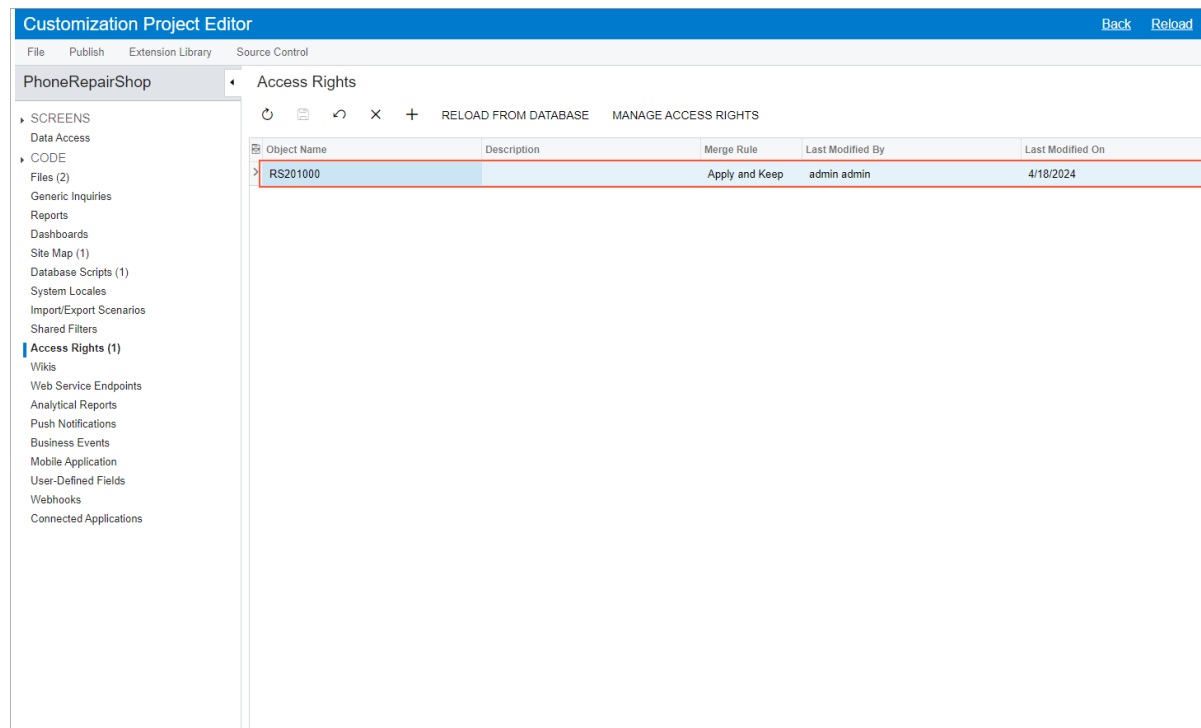
**Figure:** The Access Rights page of the Customization Project Editor

3. On the page toolbar, click **Add Row**.
4. In the **Add Access Rights for Screen** dialog box, which the system opens, select *Repair Services* in the **Screen Name** box.
5. In the **Merge Rule** section, select the **Apply and Keep** option button, as shown in the following screenshot; then click the **Add** button.



**Figure: The Merge Rule section of the dialog box**

The *RS201000* item is added to the table on the Access Rights screen, as shown in the following screenshot.



**Figure: The added record on the Access Rights page**

For more information about adding access rights to a customization project, see [To Add Access Rights to a Project](#).

## Analysis of the Generated Code of the Graph

As stated in [Application Programming Overview](#), graphs implement business logic in Acumatica ERP. A graph provides the interface for the presentation logic to operate with the business data and relies on data access layer components to store and retrieve the business data from the database.

A graph is derived from the `PXGraph` class with or without parameters. DACs are specified as parameters so that layout or background processing operations can be configured.

The following code was generated for the `RSSVRepairServiceMaint` graph.

```
using System;
using PX.Data;

namespace PhoneRepairShop
{
    public class RSSVRepairServiceMaint : PXGraph<RSSVRepairServiceMaint>
    {
        public PXSave<MasterTable> Save;
        public PXCancel<MasterTable> Cancel;

        public PXFilter<MasterTable> MasterView;
        public PXFilter<DetailsTable> DetailsView;

        [Serializable]
        public class MasterTable : PXBqlTable, IBqlTable
        {
        }

        [Serializable]
        public class DetailsTable : PXBqlTable, IBqlTable
        {
        }
    }
}
```

As you can see, the `RSSVRepairServiceMaint` graph is derived from the `PXGraph` class with itself as a parameter. For details, see [PXGraph<TGraph> Class](#).

In the graph, the following members are declared:

- The `Save` action, which commits the changes made to the data to the database and then reloads the committed data.
- The `Cancel` action, which discards all the changes made to the data and reloads it from the database.
- The `MasterView` and `DetailsView` views, which are used as data members for the form control and the grid control. You will add a custom view instead of these ones later in this course.
- The `MasterTable` and `DetailsTable` data access classes (DACs), which are used to work with data in the database. You will remove DAC declarations from this graph and declare custom DACs later in this course.

### Related Links

- [PXGraph Class](#)



## Lesson Summary

---

In this lesson, you have learned how to create a new form by using the New Screen wizard. From the example of the Repair Services form you created, you have learned about the following basic components of a form:

- The ASPX page
- The graph
- The site map node

You have also learned how to configure the access rights for a newly created form.

## Lesson 1.3: Make the New Form Visible in the UI

---

In Acumatica ERP, a workspace is a menu (which can be accessed from a link on the main menu of the product) that contains links to the forms and reports of a particular area of the product.

Now that you have created the project items required for a new form, you need to add the form to a workspace so that it appears in the Acumatica ERP UI. For the maintenance forms you are creating in this course, you will create a new workspace.

### Lesson Objectives

As you complete this lesson, you will learn how to do the following:

- Create a workspace
- Add a link to a custom form to the workspace
- Update the *SiteMapNode* item in the customization project

### Step 1.3.1: Create a Workspace

---

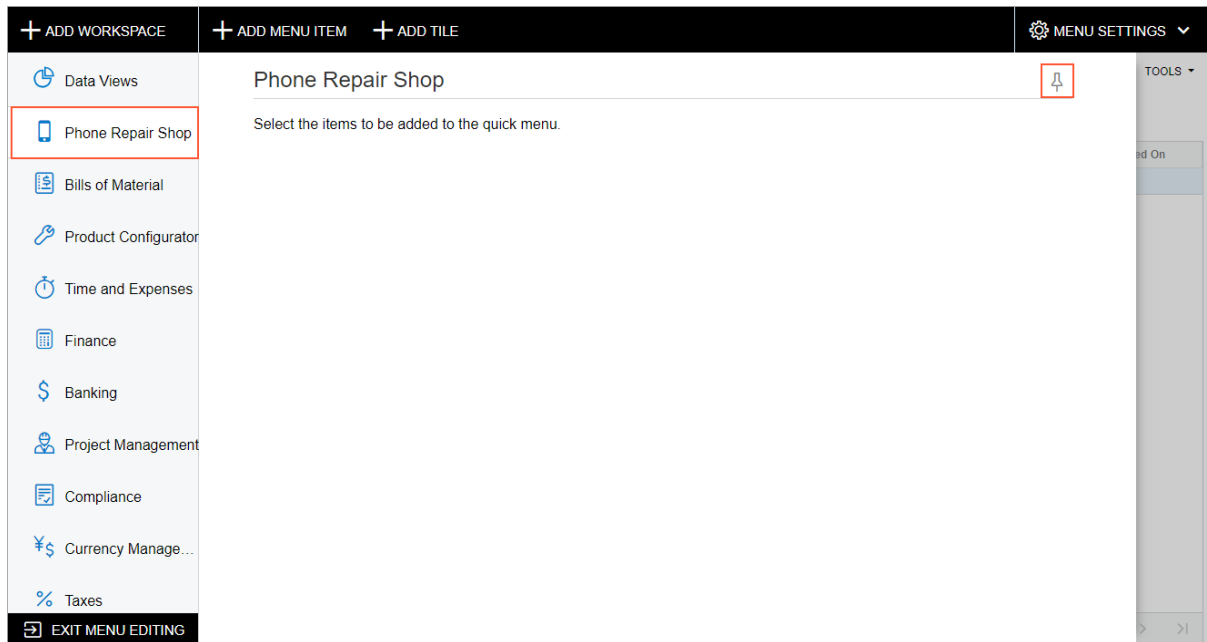
Before adding a form to the Acumatica ERP UI, you need to decide whether it will be organized in an existing workspace or a new one. In this case, for the Repair Services (RS201000) form, you should create a workspace, which will contain all forms related to the phone repair shop.

To create this workspace, do the following:

1. On the main menu of Acumatica ERP (in the lower left corner), click the configuration menu button (⚙️), and then click **Edit Menu** to switch to Menu Editing mode.
2. On the top toolbar (top left), click **Add Workspace**.
3. In the **Workspace Parameters** dialog box, specify the following settings:
  - **Icon:** *phone iphone*
  - **Area:** *Other*
  - **Title:** *Phone Repair Shop*
4. Click **OK** to save your changes and close the dialog box.
5. Pin the new workspace to the main menu panel by clicking the Pin button, which is shown in the following screenshot.

6. Move the workspace in the main menu panel so that the workspace is located below the **Data Views** workspace.

The Menu Editing mode with the new workspace looks as shown in the following screenshot.



*Figure: The Phone Repair Shop workspace in Menu Editing mode*


#### Related Links

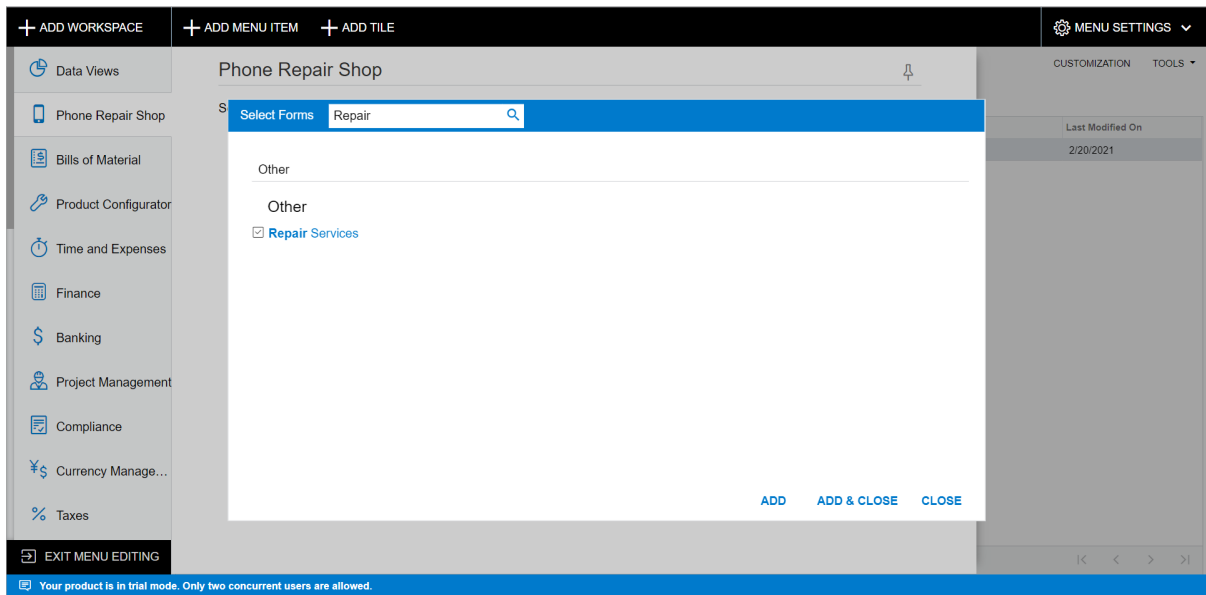
- [UI Navigation Options: To Configure a Workspace](#)

### Step 1.3.2: Add the Link to the Workspace

Now that you have created the **Phone Repair Shop** workspace, you can add to it a link to the Repair Services (RS201000) form.

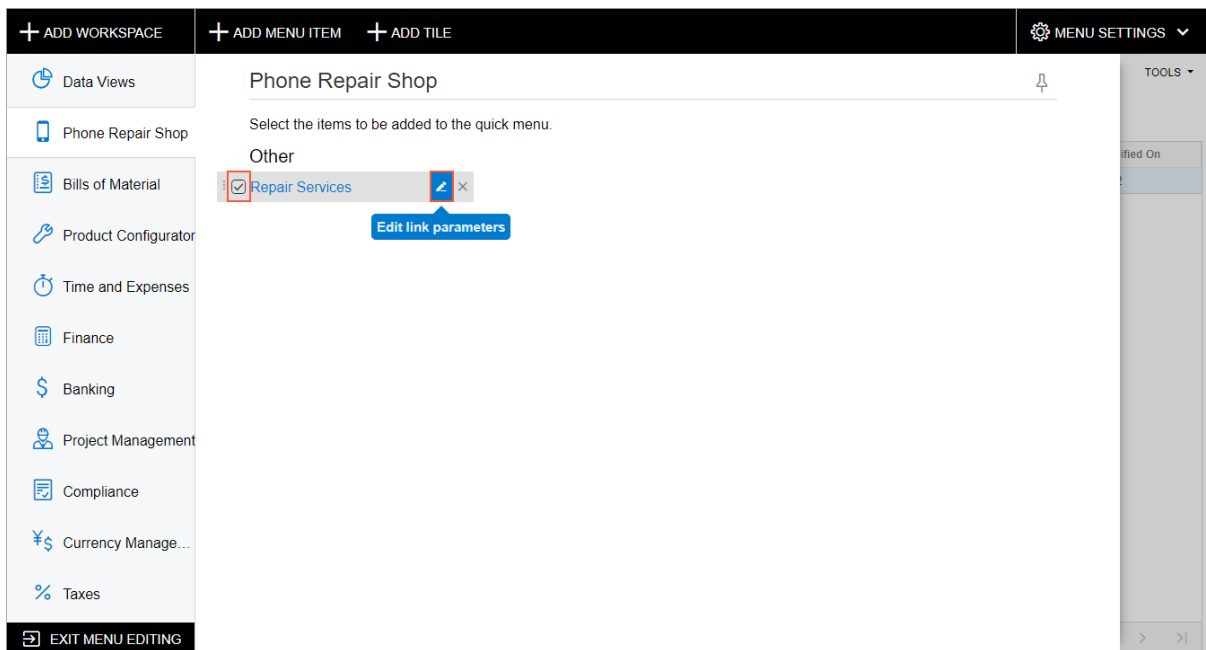
To add this link to the workspace, do the following:

1. If you are not still in Menu editing mode, on the main menu of Acumatica ERP (in the lower left corner), click the configuration menu button (  ), and then click **Edit Menu**.
2. On the main menu, click **Phone Repair Shop**.
3. On the top toolbar, click **Add Menu Item**.
4. In the **Select Forms** dialog box, which the system opens, select the check box left of **Repair Services**, as shown in the following screenshot.



*Figure: The Select Forms dialog box*

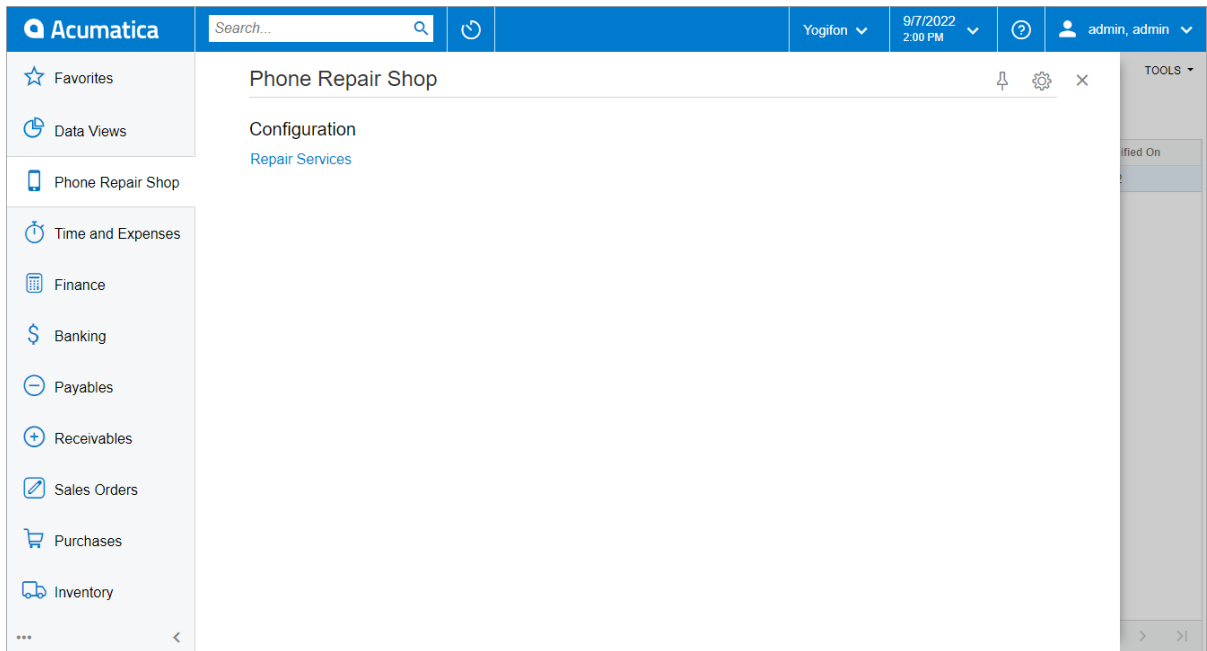
5. Click **Add & Close** to add the link and close the dialog box.
6. Select the check box to the left of **Repair Services** to make the item to be added to the quick menu and click **Edit link parameters**, as shown in the following screenshot.



*Figure: The Edit button*

7. In the **Item Parameters** dialog box, select the *Configuration* category and click **OK**.
8. In the bottom left corner of the screen, click **Exit Menu Editing** to save your changes and exit editing mode.
9. To make sure the link was added properly, on the main menu, click the **Phone Repair Shop** workspace menu item.

This workspace should look as shown in the following screenshot.



**Figure:** The Phone Repair Shop workspace

#### Related Links

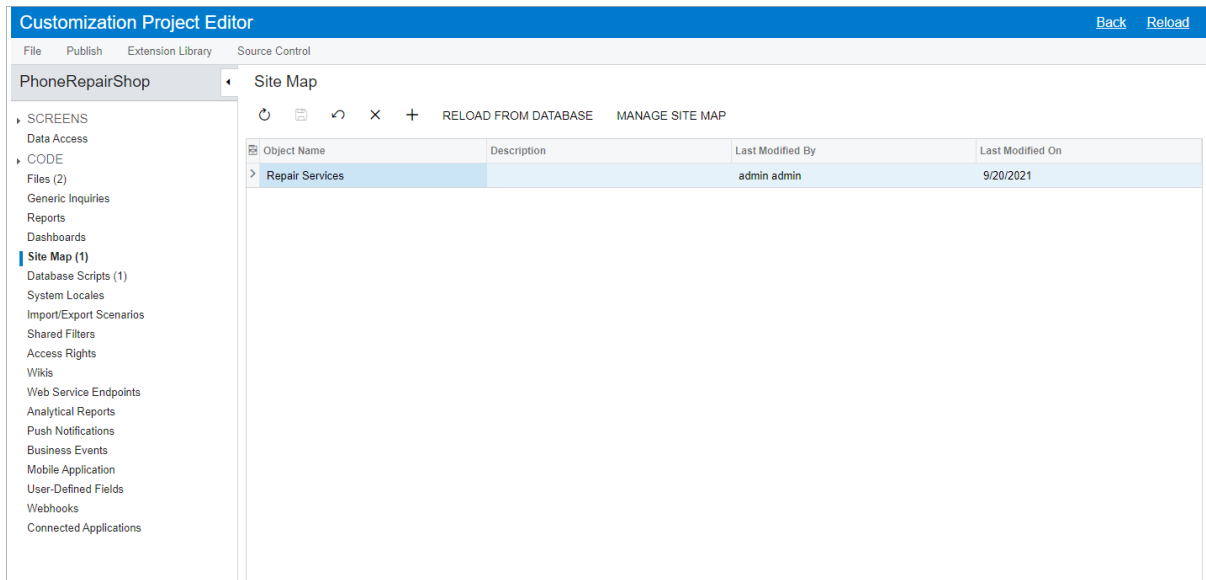
- [UI Navigation Options: To Configure a Workspace](#)

### Step 1.3.3: Update the SiteMapNode Item

When you configured the new form's location in Acumatica ERP in [Step 1.3.1: Create a Workspace](#) and [Step 1.3.2: Add the Link to the Workspace](#), the changes you made were saved to the database but not to the customization project. To save your changes to the site map in the customization project, do the following:

1. In the Customization Project Editor, open the *PhoneRepairShop* customization project.
2. On the navigation pane, click **Site Map**.

The Site Map page opens, as shown in the following screenshot.



**Figure: The Site Map page**

- On the page toolbar, click **Reload from Database**.
- Publish the customization project.

#### Related Links

- [To Update a Site Map Node in a Project](#)

## Lesson Summary

In this lesson, you have learned how to add a new workspace to the Acumatica ERP UI, add links to a workspace, and update the *SiteMapNode* item of the customization project.

## Lesson 1.4: Configure the Data Access Class

In this lesson, you will configure the data access class (DAC) generated for the Repair Services form. You need this class to access data from the database.

### Lesson Objectives

As you complete this lesson, you will learn how to do the following:

- Generate and configure a DAC
- Configure a view in a graph

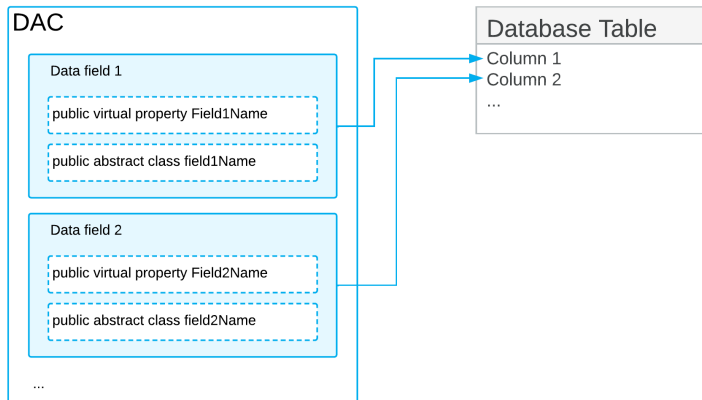
## Definition of Data Access Classes

*Data access classes (DACs)* are types that represent database tables in the application. A data access class consists of data fields. A data field definition consists of two members of the class, which have the same name except that it differs by the case of the first letter:

- A public abstract class (which is also referred to as a *class field* or *BQL field*) that represents the data field in BQL statements, such as `companyType`.
- A public virtual property (which is also referred to as a *property field*) that holds the field value, such as `CompanyType`.

The following diagram shows the connection between columns of a database table and DAC fields.

The connection between DAC fields and database columns



You can define a data access class by manually typing the code, or by using the **Create Code File** dialog box of the Customization Project Editor. By using the dialog box, you can define the initial code of a data access class based on the schema of the database table.

When you define a data access class, consider the following requirements:

- The class must have either the `PXCacheName` attribute or the `PXHidden` attribute.  
The `PXCacheName` attribute specifies a user-friendly DAC name. This name can be used in generic inquiries, reports, and the error message that is displayed when no setup data records exist. Without the `PXCacheName` attribute, the error message would use the DAC name for the link.  
The `PXHidden` attribute hides the DAC from generic inquiries, reports, and web services API clients.
- The class must be declared as extending the `PX.Data.PXBqlTable` class and implementing the `PX.Data.IBqlTable` interface.
- Abstract classes of data fields must be defined as implementing interfaces of the `PX.Data.BQL` namespace.
- A DAC property field must have a nullable type (such as `decimal?` or `DateTime?`).

It is important to pay attention to the order in which fields are declared in a DAC: Every roundtrip Acumatica Framework applies changes to DAC instances in the same order as their fields are declared. All field-level event handlers are always raised in the same order as fields are declared in the DAC.

#### Related Links

- [Data Access Class](#)
- [Designing the Database Structure and DACs](#)

## Step 1.4.1: Generate a DAC

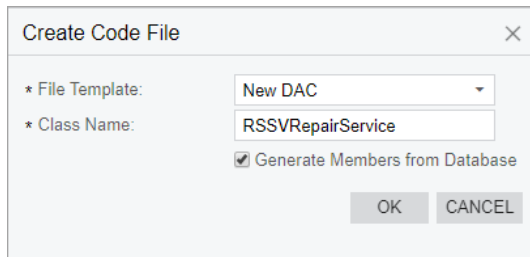
Generate the DAC code and configure the generated code by doing the following:

1. Open the *PhoneRepairShop* customization project in the Customization Project Editor.

- In the navigation pane, click **Code**.

The Code page opens. It already contains the record about the `RSSVRepairServiceMaint` graph created in [Step 1.2.1: Use the New Screen Wizard to Create a Form Template](#).

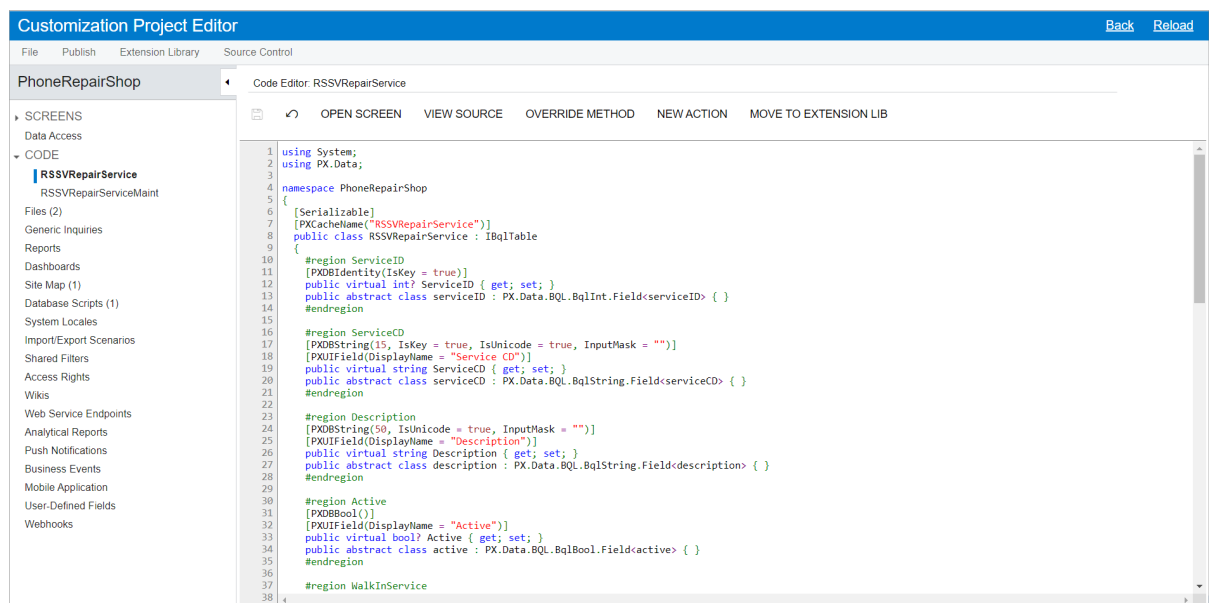
- On the Code page toolbar, click **Add New Record**.
- In the **Create Code File** dialog box, which opens, specify the following values:
  - File Template:** *New DAC*
  - Class Name:** `RSSVRepairService`
  - Generate Members from Database:** Selected



*Figure: The Create Code File dialog box*

- Click **OK** to close the dialog box.

The new DAC code is opened in the Code Editor, as shown in the following screenshot.



*Figure: The RSSVRepairService DAC code*

## Related Links

- [To Create a New DAC](#)

## Step 1.4.2: Configure the Attributes of the New DAC

After the code of the `RSSVRepairService` DAC has been generated, you should configure attributes for each field of the DAC.

## Background of Attributes

In Acumatica Framework, you use *attributes* to add common business logic to the application components. An attribute may be placed on a declaration of a class or a class member, with or without parameters. The possible parameters for an attribute depend on the constructor parameters and the properties defined in the attribute. The parameters of a constructor are placed first without names, and the named property settings follow them, as shown in the following example.

```
[PXDefault(false, PersistingCheck = PXPersistingCheck.Nothing)]
public virtual Boolean? Released { get; set; }
```

Here the `PXDefault` attribute is created with a constructor that has a Boolean-type parameter (which is set to `false`). That means the default value of the `Released` property is set to `false`. Additionally, the `PersistingCheck` property is specified.

Another typical example of an attribute is `PXUIField` (used in the following example). It is used to configure the input control for the column in the user interface, so that the column can have the same visual representation on all application forms (unless it is redefined for a particular form).

```
[PXUIField(DisplayName = "Available Qty", Enabled = false)]
public virtual string AvailQty { get; set; }
```

Here, in the `PXUIField` constructor, the `DisplayName` and `Enabled` properties of the `AvailQty` field are specified. The `PXUIField` attribute is required for all fields you want to be displayed on the form. To visually identify a field that requires a value, you can specify the `Required = true` property of that field in the `PXUIField` constructor, which places an asterisk by the UI element corresponding to the field. For details, see [Mandatory Attributes](#) and [UI Field Configuration](#).



The fields of a DAC are bound to the database by data mapping attributes (such as `PXDBIdentity` and `PXDBString`). The fields that are bound to the database must have the same name as the fields in a database table. For details, see [Bound Field Data Types](#).

For details on predefined attributes and their constructors, see [API Reference](#).

## Configuration of the Attributes of the RSSVRepairService DAC

Do the following to configure the attributes of the `RSSVRepairService` DAC:

1. In the Code Editor of the Customization Project Editor, open the `RSSVRepairService` code item.
2. Remove the `[Serializable]` attribute before the `RSSVRepairService` DAC declaration and adjust the `[PXCacheName("Repair Service")]` attribute. The attribute gives the DAC a user-friendly name. For details, see [PXCacheNameAttribute Class](#).
3. In the DAC code that is generated, replace the generated attributes with the following attributes:
  - For the `ServiceID` field, remove `IsKey=true` for the `PXDBIdentity` attribute as follows.

```
#region ServiceID
[PXDBIdentity]
public virtual int? ServiceID { get; set; }
public abstract class serviceID : PX.Data.BQL.BqlInt.Field<serviceID> { }
#endregion
```

The `PXDBIdentity` attribute is intended to identify the identity column in the database. For details, see [PXDBIdentity Attribute](#).

- For the `ServiceCD` field, add the `PXDefault` attribute, and correct the display name as follows.



```
#region ServiceCD
[PXDBString(15, IsUnicode = true, IsKey = true,
    InputMask = ">aaaaaaaaaaaaaa")]
[PXDefault]
[PXUIField(DisplayName = "Service ID")]
public virtual string ServiceCD { get; set; }
public abstract class serviceCD :
    PX.Data.BQL.BqlString.Field<serviceCD> { }
#endregion
```

The `PXDBString` attribute generated for the field maps a DAC field of the string type to the database column and determines the string field properties. For details, see [PXDBString Attribute](#).

The `IsKey` property marks the field that must uniquely identify a data record. The key fields defined in the DAC should not necessarily be the same as the keys in the database.

Normally, all letters in record identifiers in Acumatica ERP are in uppercase. To stay consistent with the core product, you should define an input mask for the `ServiceCD` field by adding the `InputMask = ">aaaaaaaaaaaaaa"` parameter to the `PXDBString` attribute. For more information about input masks, see [To Configure an Input Mask and a Display Mask for a Field](#).

The `PXDefault` attribute makes the field required.

- For the `Description` field, add the `PXDefault` attribute and remove `InputMask` from the `PXDBString` attribute as follows.

```
#region Description
[PXDBString(50, IsUnicode = true)]
[PXDefault]
[PXUIField(DisplayName = "Description")]
public virtual string Description { get; set; }
public abstract class description :
    PX.Data.BQL.BqlString.Field<description> { }
#endregion
```

You add the `PXDefault` attribute for the `Description` field to make the field required. For details, see [Default Values](#).

- For the `Active` field, add the `[PXDefault(true)]` attribute as follows.

```
#region Active
[PXDBBool()]
[PXDefault(true)]
[PXUIField(DisplayName = "Active")]
public virtual bool? Active { get; set; }
public abstract class active : PX.Data.BQL.BqlBool.Field<active> { }
#endregion
```

The `PXDBBool` attribute generated for the field maps a DAC field of the `bool?` type to the database column. For details, see [PXDBBool Attribute](#).

By adding `[PXDefault(true)]`, you specify that the field is required and the default value to be inserted in the database is `true`.

- For the `WalkInService`, add the `[PXDefault(false)]` attribute and change the `DisplayName` parameter of the `PXUIField` attribute to *Walk-In Service* as follows.

```
#region WalkInService
[PXDBBool()]
[PXDefault(false)]
[PXUIField(DisplayName = "Walk-In Service")]
public virtual bool? WalkInService { get; set; }
public abstract class walkInService :
```

```
PX.Data.BQL.BqlBool.Field<walkInService> { }
#endregion
```

By adding `[PXDefault(false)]`, you specify that the field is required and the default value to be inserted in the database is *false*. By modifying the `DisplayName` parameter value of the `PXUIField` attribute, you change the label for the corresponding check box that is displayed in the UI.

- For the `PreliminaryCheck` field, add the `[PXDefault(false)]` attribute and change the `DisplayName` parameter of the `PXUIField` attribute to *Requires Preliminary Check* as follows.

```
#region PreliminaryCheck
[PXDBBool()]
[PXDefault(false)]
[PXUIField(DisplayName = "Requires Preliminary Check")]
public virtual bool? PreliminaryCheck { get; set; }
public abstract class preliminaryCheck :
    PX.Data.BQL.BqlBool.Field<preliminaryCheck> { }
#endregion
```

- For the `Prepayment` field, add the `[PXDefault(false)]` attribute and change the `DisplayName` parameter of the `PXUIField` attribute to *Requires Prepayment* as follows.

```
#region Prepayment
[PXDBBool()]
[PXDefault(false)]
[PXUIField(DisplayName = "Requires Prepayment")]
public virtual bool? Prepayment { get; set; }
public abstract class prepayment :
    PX.Data.BQL.BqlBool.Field<prepayment> { }
#endregion
```

- For the system fields, make sure that the following attributes are specified. For details, see [Audit Fields](#).

Field	Attribute
CreatedDateTime	<code>[PXDBCreatedDateTime()]</code>
CreatedByID	<code>[PXDBCreatedByID()]</code>
CreatedByScreenID	<code>[PXDBCreatedByScreenID()]</code>
LastModifiedDateTime	<code>[PXDBLastModifiedDateTime()]</code>
LastModifiedByID	<code>[PXDBLastModifiedByID()]</code>
LastModifiedByScreenID	<code>[PXDBLastModifiedByScreenID()]</code>
Tstamp	<code>[PXDBTimestamp()]</code>
Noteid	<code>[PXNote()]</code>

- Remove the `PXUIField` attribute from the `Tstamp` field.
- Update the name of the `NoteID` property: The name of the property should be `NoteID`, the name of the class should be `noteID` as the following code shows.

```
#region NoteID
[PXNote()]
```

```
public virtual Guid? NoteID { get; set; }
public abstract class noteID : PX.Data.BQL.BqlGuid.Field<noteID> { }
#endregion
```

4. Save your changes.
5. Publish the customization project.



If you want to publish a project after it has already been published and the **Compilation** pane is displayed, close the **Compilation** pane first.

#### Related Links

- [Working with Attributes](#)

### Step 1.4.3: Configure a View

In the previous step, you have defined the needed DAC. In this step, you need to configure the view that will retrieve data from the database by using the defined DAC as follows:

1. Open the `RSSVRepairServiceMaint` graph on the [Code Editor](#) page.  
To do this, in the navigation pane of the Customization Project Editor, click **RSSVRepairServiceMaint** under the **Code** node.

2. Add the following `using` directive.

```
using PX.Data.BQL.Fluent;
```

3. In the `RSSVRepairServiceMaint` graph, add the definition of the `RepairService` view as follows. You will later specify it as a property value in ASPX.

```
public SelectFrom<RSSVRepairService>.View RepairService;
```



The view here is declared through the use of a fluent BQL query. For details, see [Search and Select Commands and Data Views in Fluent BQL](#).

4. Replace the class name for the `Save` and `Cancel` actions, as shown in the following code.

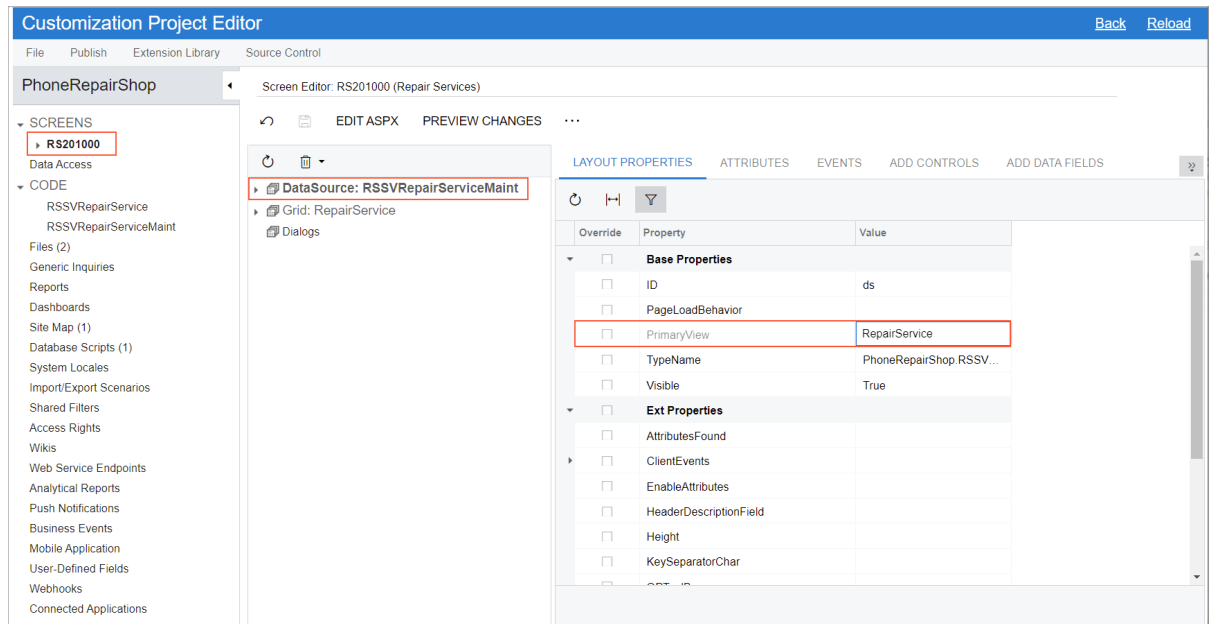
```
public PXSave<RSSVRepairService> Save;
public PXCancel<RSSVRepairService> Cancel;
```



For details about declaration of standard actions, see [Standard Buttons of the Form Toolbar](#).

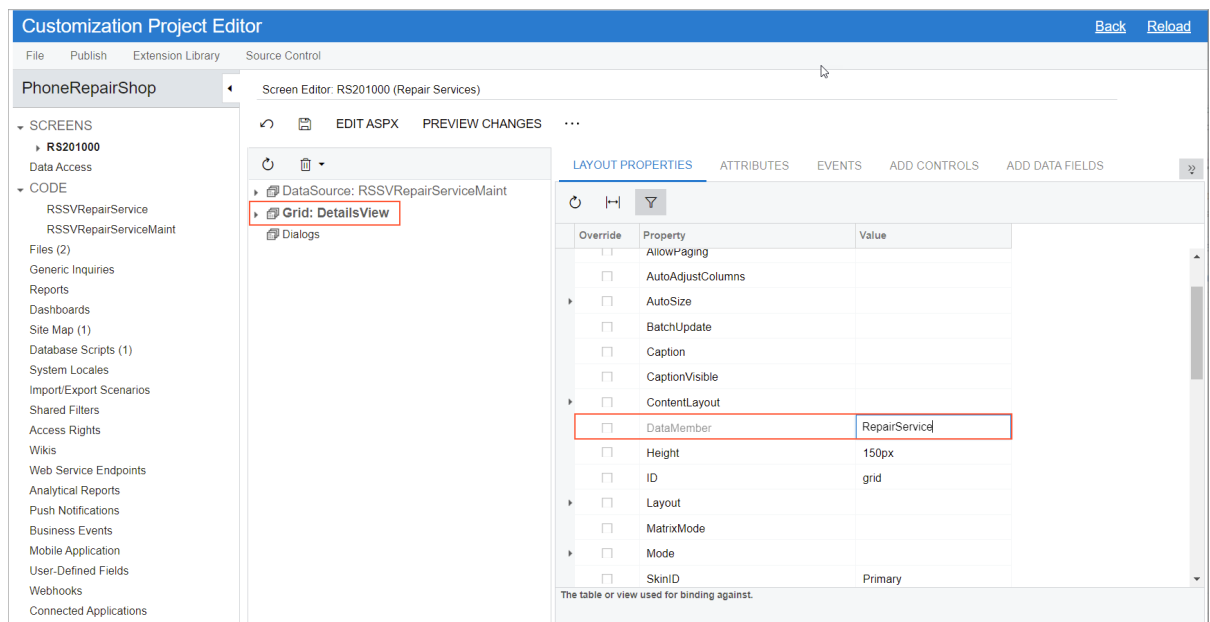
5. Save your changes.
6. Publish the customization project.
7. Specify `RepairService` as the data member for the Repair Services (RS201000) form by doing the following:
  - a. In the **Screens** node on the navigation pane, click **RS201000**.  
The [Screen Editor](#) page opens for the Repair Services form.
  - b. In the control tree, click the **DataSource** node.

- c. In the **Layout Properties** tab of the right pane, type the `RepairService` value for the **PrimaryView** property, as shown in the following screenshot.



*Figure: Specifying the PrimaryView property*

- d. Save your changes.
- e. In the control tree, click the **Grid: DetailsView** node.
- f. In the **Layout Properties** tab of the right pane, type the `RepairService` value for the **DataMember** property, as shown in the following screenshot.



*Figure: Specifying the DataMember property*

8. Save your changes.
9. Publish your customization project.



After you have configured the form and published the customization project, you can remove the definitions of the `MasterView`, `DetailsView`, `MasterTable`, and `DetailsTable` members from the `RSSVRepairServiceMaint` graph because they will not be used further in the course.

#### Related Links

- [Data View](#)

## Lesson Summary

---

In this lesson, you have learned the concept of a data access class (DAC), and have generated and configured a DAC. You have learned about the attributes that are required for DAC fields and have configured a view by using a fluent BQL query.

## Lesson 1.5: Configure the Form

---

In this lesson, you will configure the Repair Services (RS201000) form that you created earlier in this part. You will add columns to the grid and test the form.

### Lesson Objectives

As you complete this lesson, you will learn how to do the following:

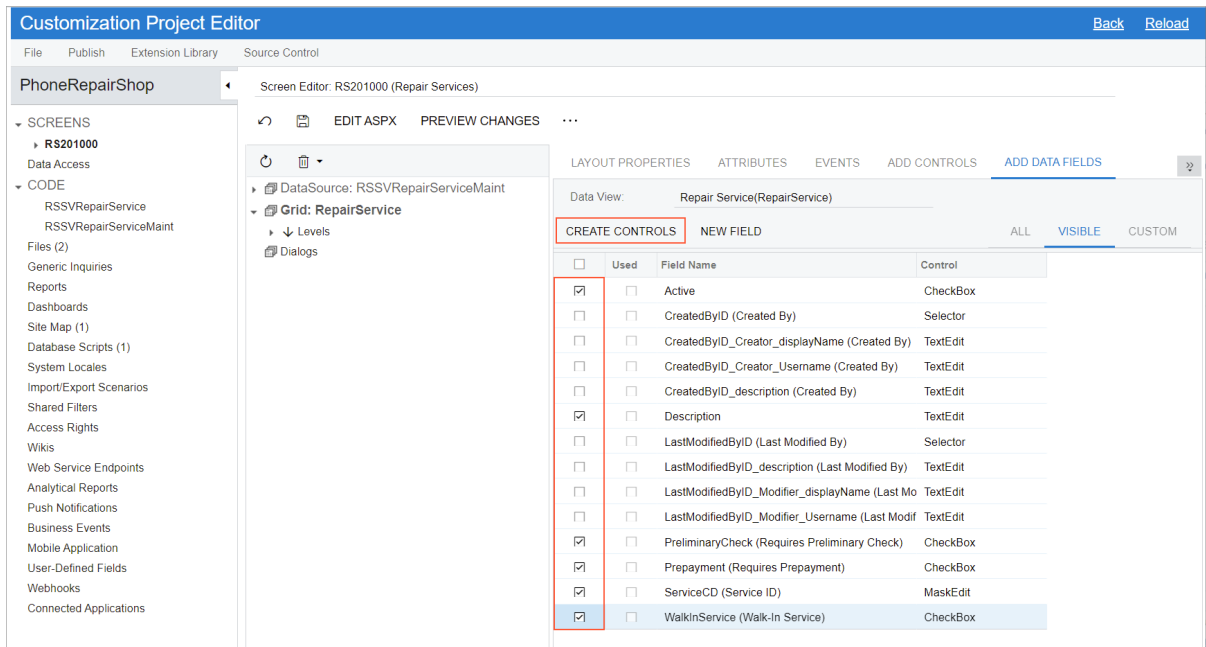
- Add columns to a form grid
- Configure the appearance of the columns in the grid

### Step 1.5.1: Add Columns to the Grid

---

In the previous lesson, you defined the fields of the `RSSVRepairService` DAC. Now you can add columns corresponding to the DAC fields to the grid on the Repair Services (RS201000) form. Do the following:

1. Open the *PhoneRepairShop* customization project in the Customization Project Editor.
2. In the **Screen** node of the navigation pane, select **RS201000**.  
The Screen Editor for the Repair Services form opens.
3. In the control tree, select **Grid: RepairService**.
4. On the **Add Data Fields** tab in the right pane, notice that the `RSSVRepairService` DAC fields are displayed. Select the check boxes in the rows of all fields except system fields (system fields are listed in [Step 1.4.2: Configure the Attributes of the New DAC](#)), as shown in the following screenshot, and click **Create Controls** on the table toolbar.



**Figure: Columns to be added**

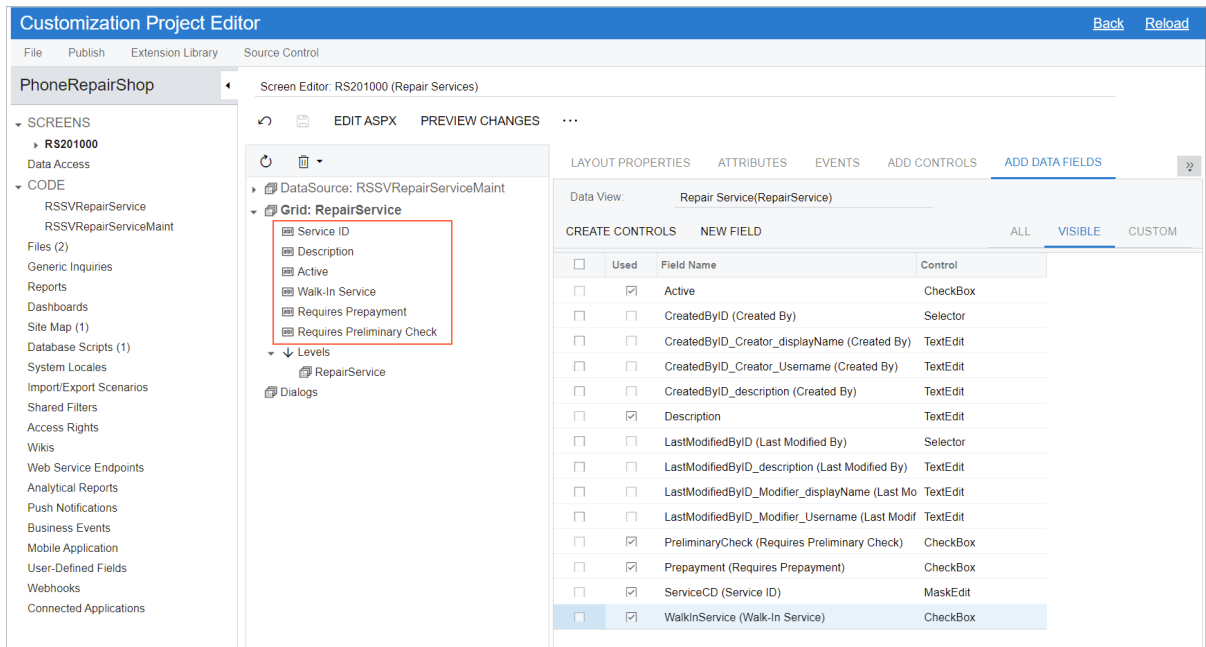


You can generate all columns automatically by setting the **AutoGenerateColumns** value on the **Layout Properties** tab to *Append*.

The columns appear in the control tree.

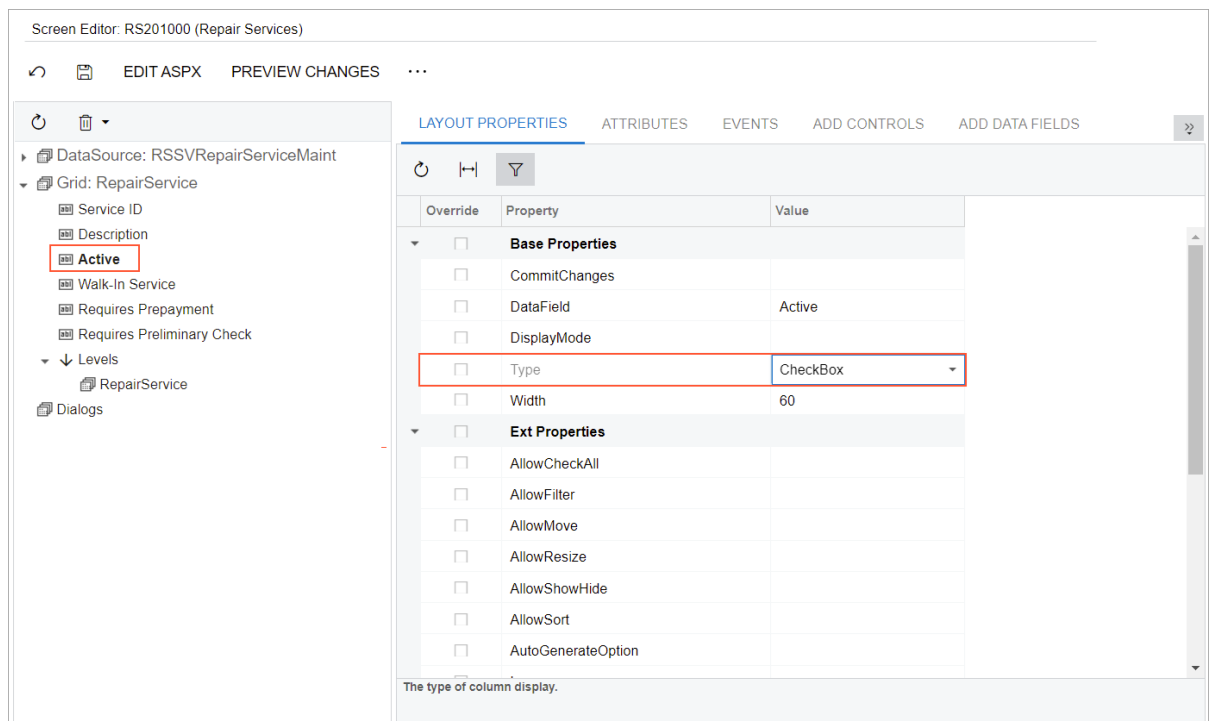
5. Change the order of columns in the control tree to the following one by dragging the controls in the control tree:
  - a. Service ID
  - b. Description
  - c. Active
  - d. Walk-In Service
  - e. Requires Prepayment
  - f. Requires Preliminary Check

The resulting control tree is shown in the screenshot below.



**Figure: Control tree with new columns**

6. Save your changes.
7. Configure the appearance of the elements that should look like check boxes as follows:
  - a. In the control tree of the Screen Editor, select the **Active** element.
  - b. On the **Layout Properties** tab of the right pane, set the **Type** property value to *CheckBox*, as shown in the following screenshot.



**Figure: Property to define the Active column as a check box**

- c. Save your changes.





**Figure: The Repair Services form**



If the form is already open, refresh the page.

2. On the form toolbar, click **Add Row**.
3. Enter the following values in the columns of the table:
  - **Service ID:** TestID
  - **Description:** Test Description
4. On the form toolbar, click **Save**.

The new row has been added, as shown in the following screenshot.

**Figure: The new row on the Repair Services form**

5. Add the following rows to the table.

Service ID	Description	Active	Walk-In Service	Requires Pre-payment	Requires Preliminary Check
BatteryReplace	Battery Replace-ment	Selected	Selected	Cleared	Cleared
LiquidDamage	Liquid Damage	Selected	Cleared	Selected	Selected

Service ID	Description	Active	Walk-In Service	Requires Pre-payment	Requires Preliminary Check
ScreenRepair	Screen Re-pair	Selected	Selected	Cleared	Cleared

6. On the form toolbar, click **Save**.

The system automatically sorts the records by the **Screen ID** value.

7. Click the *TestID* row in the grid.

8. On the form toolbar, click **Delete Row**.

The row has been deleted.

9. On the form toolbar, click **Save**.

The form should look like the one shown in the following screenshot.

The screenshot shows the 'Repair Services' form interface. At the top, there are tabs for 'CUSTOMIZATION' and 'TOOLS'. Below the tabs is a toolbar with icons for refresh, save, undo, redo, add, delete, and search. The main area contains a table with the following data:

* Service ID	* Description	Active	Walk-In Service	Requires Prepayment	Requires Preliminary Check
BATTERYREPLACE	Battery Replacement	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
LIQUIDDAMAGE	Liquid Damage	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
SCREENREPAIR	Screen Repair	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

*Figure: New rows on the Repair Services form*

## Lesson Summary

In this lesson, you have learned how to configure the elements on a form by using the Screen Editor; you have also tested the created form.

## Lesson 1.6: Add an Event Handler to the Walk-In Service Check Box

In Smart Fix, depending on the type of work to be done, a repair service can be provided right away (which is indicated by the **Walk-In Service** check box on the Repair Services (RS201000) form) or after a preliminary check (which is indicated by the **Requires Preliminary Check** check box). This means that the check boxes on the completed Repair Services form must be mutually exclusive: If one is selected, the other must be cleared.

To implement this logic, you need to define event handlers for the **Walk-In Service** and **Requires Preliminary Check** check boxes.

In this lesson, you will add an event handler for the **Walk-In Service** check box. Adding the event handler for the **Requires Preliminary Check** check box will be covered in [Lesson 1.9: Add an Event Handler In Visual Studio](#).

## Lesson Objectives

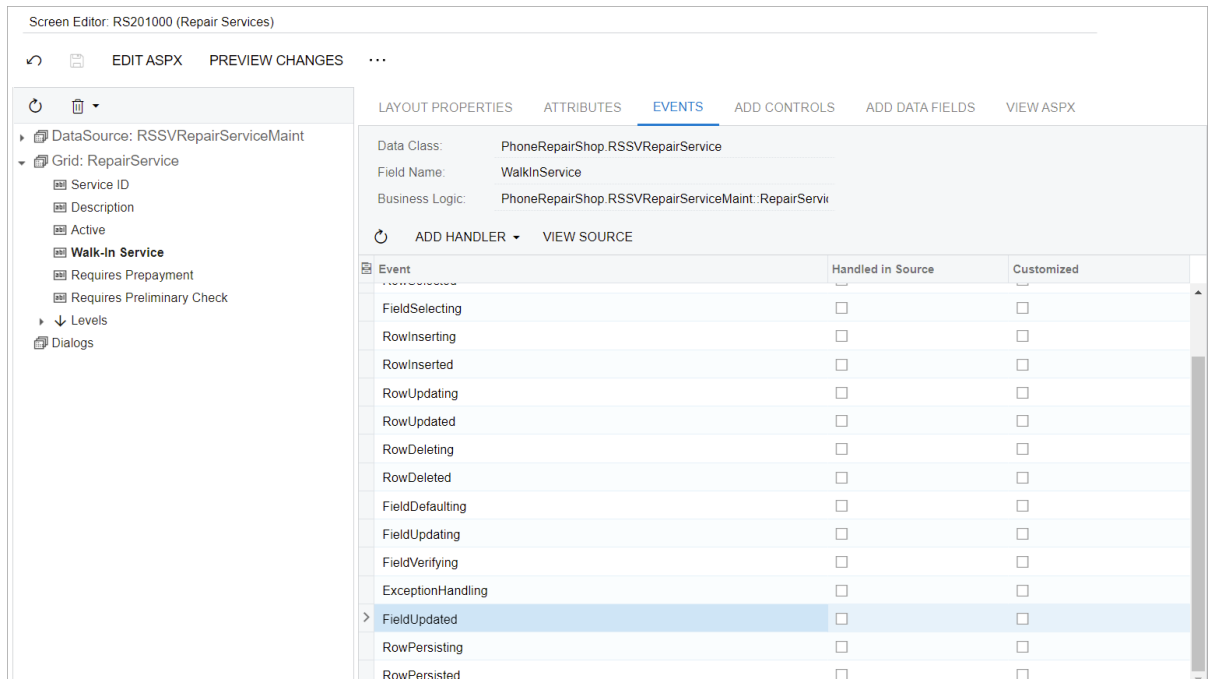
As you complete this lesson, you will learn how to add an event handler for a check box.

### Step 1.6.1: Add an Event Handler in the Customization Project Editor

A number of events can occur with a field of a record. The `FieldUpdated` event is raised for each field of a record that is currently updated or inserted. This event type is intended for modification of other fields of the same data record. For details, see [Sequence of Events: Update of a Data Record](#).

For the form you are designing, you will define the handler for the event of updating the value of the **Walk-In Service** column as follows.

1. Open the Repair Services (RS201000) form in the Screen Editor.
2. In the control tree, select **Grid: RepairService > Walk-In Service**.
3. On the **Events** tab of the right pane, click the row with the `FieldUpdated` event, as shown in the following screenshot.



*Figure: The FieldUpdated event*

4. On the **Events** tab toolbar, click **Add Handler > Keep Base Method**.

The `RSSVRepairServiceMaint` graph opens in the [Code Editor](#). The following handler code is generated.

```
protected void RSSVRepairService_WalkInService_FieldUpdated(
    PXCache cache, PXFieldUpdatedEventArgs e)
{
    var row = (RSSVRepairService)e.Row;
}
```

5. Insert the following code in the handler after the declaration of the `row` variable.

```
row.PreliminaryCheck = !(row.WalkInService == true);
```

6. Save your changes.

#### Related Links

- [Working with Events](#)
- [To Add an Event Handler](#)

## Step 1.6.2: Specify the CommitChanges Property

To enable a callback for a column, you should specify the `CommitChanges` property of the field. When the `CommitChanges` property is set to `true`, the event is triggered every time the user changes the value within the column and moves focus out of it.

To specify the `CommitChanges` property of the column, do the following:

1. Open the Repair Services (RS201000) form in the Screen Editor.
2. In the control tree, select **Grid: RepairService > Walk-In Service**.
3. On the **Layout Properties** tab of the right pane, set the **CommitChanges** property value to `True`.
4. Save your changes.
5. Publish the customization project.

#### Related Links

- [Use of the CommitChanges Property of Boxes](#)

## Step 1.6.3: Test the Event Handler

Now you will test the event handler you have added by doing the following:

1. In Acumatica ERP, open the Repair Services (RS201000) form.
2. In the *ScreenRepair* row, clear the **Walk-In Service** check box, as shown on the following screenshot.

Repair Services				CUSTOMIZATION		TOOLS ▾	
↺	↻	↶	+	×	⏮	⏭	⏹
		* Service ID	* Description	Active	Walk-In Service	Requires Prepayme	Requires Prelimina Check
🔗	📄	BATTERYREPLACE	Battery Replacement	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
🔗	📄	LIQUIDDAMAGE	Liquid Damage	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
>	🔗	SCREENREPAIR	Screen Repair	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>

*Figure: Clearing the Walk-In Service check box*

3. Notice that the **Requires Preliminary Check** check box has been selected automatically.
4. Select the **Walk-In Service** check box in the *ScreenRepair* row.
5. Notice that the **Requires Preliminary Check** check box has been cleared automatically.

6. Save your changes.

## Lesson Summary

---

In this lesson, you have learned how to add and enable an event handler by using the Customization Project Editor. You have added code to the generated event handler and tested it on the Repair Services (RS201000) form to be sure it works as intended.

To add and enable an event handler, you have done the following:

1. In the Screen Editor, specified the `CommitChanges` property of the control for which an event handler should work.
2. Generated code for the event handler by using the Screen Editor. You have added the `FieldUpdated` event, which is raised for each field of a record that is currently updated or inserted.
3. Added code to the event handler by using the Code Editor.

## Lesson 1.7: Debug the Customization Code

---

After you have added some code to your customization, you can debug the code, if necessary. The only way to debug customization code is to use Visual Studio.



You can use Visual Studio also to develop customization code. This topic is covered in Part 2 of this training course. If you develop customization code in Visual Studio, you can debug the code there.

## Lesson Objectives

In this lesson, you will learn how to debug the code of a customization project by using Visual Studio.

### Step 1.7.1: Debug the Customization Code

---

To start debugging the customization code, do the following:

1. In the file system, open in a text editor the `Web.config` file that is located in the root folder of the SmartFix\_T200 instance.
2. In the `<system.web>` tag of the file, locate the `<compilation>` element.
3. Set the debug attribute of the element to `True`, as shown in the following code.

```
<system.web>
  <compilation debug="True" ...>
```

Save your changes.

4. Make sure the SmartFix\_T200 instance of Acumatica ERP is running by opening any page of the instance in a browser.
5. Launch Visual Studio as administrator.
6. On the main menu of Visual Studio, click **File > Open > Web Site**.
7. In the **Open Web Site** dialog box, click **File System** and select the `SmartFix_T200` instance folder.

8. Click **Open**.
9. In the Solution Explorer of Visual Studio, open the `App_RuntimeCode` folder and double-click the `RSSVRepairServiceMaint.cs` file.



The `App_RuntimeCode` folder of a Acumatica ERP instance contains copies of files with customization code. The platform creates these files during publication of a customization project.

10. In the `RSSVRepairService_WalkInService_FieldUpdated` event handler, set a breakpoint on the following line.

```
var row = (RSSVRepairService)e.Row;
```

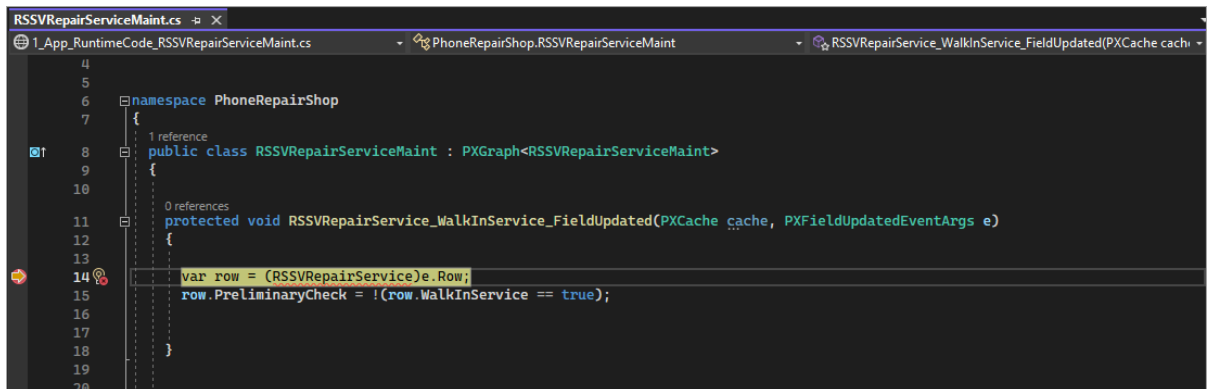
11. On the main menu, click **Debug > Attach to Process**.
12. In the **Attach to Process** dialog box, which opens, select the `w3wp.exe` process in the **Available Processes** list and click **Attach**.



If the `w3wp.exe` file is not displayed in the list, try selecting the **Show processes from all users** check box.

13. In the **Attach Security Warning** dialog box, which opens, click **Attach**.
14. In a browser, open the Repair Services (RS201000) form.
15. In the *ScreenRepair* row, clear the **Walk-In Service** check box.

The Visual Studio window opens with the breakpoint highlighted, as shown in the following screenshot.



*Figure: Breakpoint hit*

16. Press F10 (or click **Debug > Step Over** on the main menu) until you reach the end of the handler.
17. Press F5 (or click **Debug > Continue** in the main menu) to return to the Repair Services form.
18. In Visual Studio, remove the breakpoint at the `var row = (RSSVRepairService)e.Row;` line.
19. On the Repair Services form, select the **Walk-In Service** check box for the *ScreenRepair* row to restore the record settings. Note that Visual Studio window is not opened.

## Related Links

- [To Debug the Customization Code](#)

## Lesson Summary

---

In this lesson, you have learned how to debug customization code by using Visual Studio.

To debug customization code, you have completed the following steps:

1. Configured the `web.config` file of the Acumatica ERP instance.
2. In Visual Studio, added a breakpoint.
3. Attached the Visual Studio debugger to the `w3wp.exe` process.
4. Performed the debugging of the instance in Visual Studio.

## Lesson 1.8: Move the Customization Code to an Extension Library

---

In the previous lessons, you have learned how to work with items of a customization project in the Customization Project Editor. In this lesson, you will learn how to integrate a customization project with Microsoft Visual Studio to use the capabilities of Visual Studio to develop the customization code. You will create an extension library that includes all the code of the *PhoneRepairShop* customization project, compile the source code using Visual Studio, and add the binary file of the library to the *PhoneRepairShop* project.

### Lesson Objectives

As you complete this lesson, you will learn how to do the following:

- Create an extension library
- Open the created extension library in Visual Studio
- Build the created extension library in Visual Studio

## Extension Libraries

---

To develop the customization code in Visual Studio, you need to use an *extension library*. An *extension library* is a Visual Studio project that contains customization code and can be individually developed and tested.

An extension library `.dll` file must be located in the `Bin` folder of the website. At run time during the website initialization, all the `.dll` files of the folder are loaded into the server memory for use by the Acumatica ERP application. Therefore, all the code extensions included in a library are accessible from the application.

During the first initialization of a base class, the Acumatica Customization Platform automatically discovers an extension for the class in the memory and applies the customization by replacing the base class with the merged result of the base class and the discovered extension.

If you need to deploy the customization code of an extension library to another system, you have to add the library to a customization project as a *File* item to include it in a customization package.

For details on extension libraries, see [Extension Library](#).



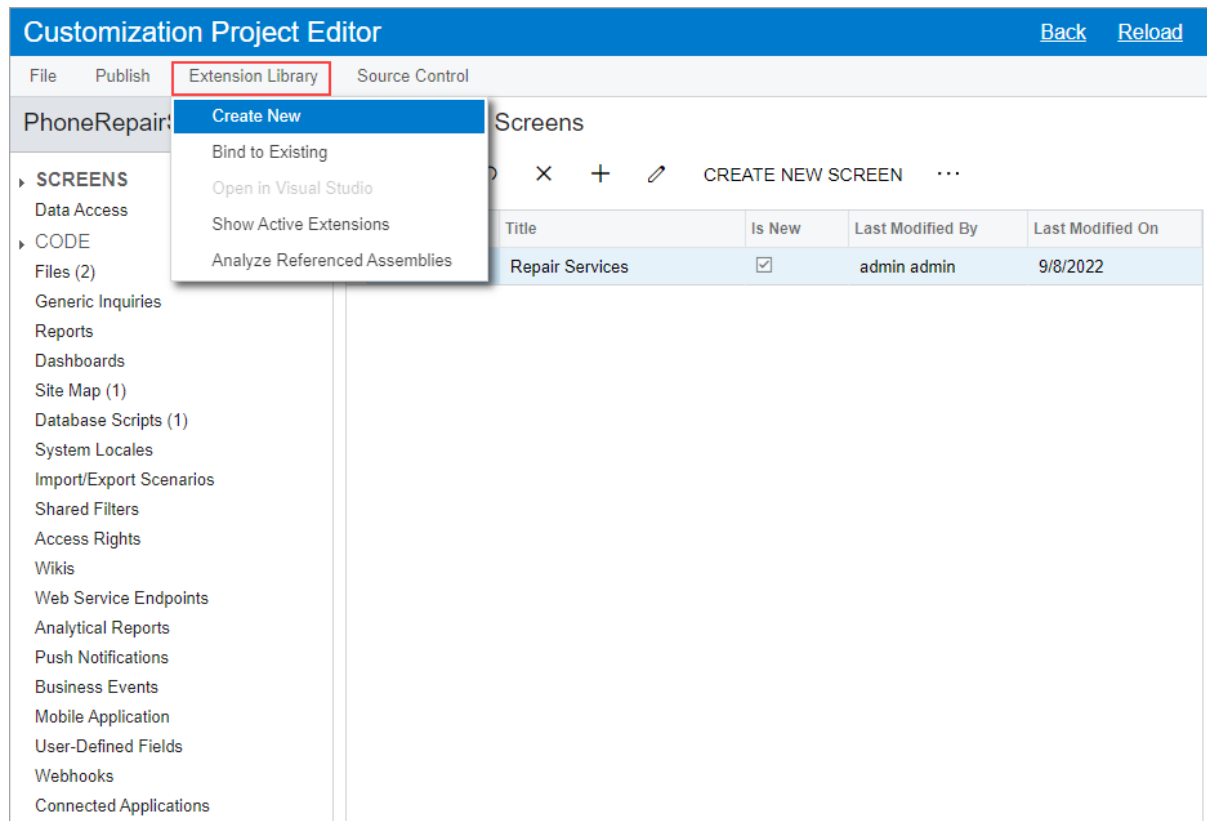
See [Extension Library \(DLL\) Versus Code in a Customization Project](#) in the Acumatica ERP Customization Guide for our recommendations about where you should keep your customization code.

## Step 1.8.1: Create an Extension Library

Now you will create the `PhoneRepairShop_Code` extension library, which includes all the code of the `PhoneRepairShop` customization project. To do this, perform the following actions:

1. Open the `PhoneRepairShop` project in the Customization Project Editor.
2. On the main menu of the Customization Project Editor, click **Extension Library > Create New**, as the following screenshot shows.

**Figure: Creation of the `PhoneRepairShop_Code` extension library**



3. In the **Project Name** box of the **Create Extension Library** dialog box, which opens, enter `PhoneRepairShop_Code`.

By default, the platform uses the `App_Data\Projects` folder of the website as the parent folder for the solution project. If the website folder is outside of the `C:\Program Files (x86)`, `C:\Program Files`, and `C:\Users` folders, we recommend that you not change it. Otherwise, we recommend that you specify a parent folder outside these folders to avoid an issue with permission to work in the `C:\Program Files (x86)`, `C:\Program Files`, and `C:\Users` folders. For example, specify the following folder: `C:\AcumaticaSites\T200`.

4. Click **OK** to close the dialog box and start the process of creating the library.

During the process, Acumatica Customization Platform creates the following files in the folder specified in the dialog box.



File	Description
PhoneRepairShop_Code.sln	The Microsoft Visual Studio Solution file
Solution.bat	The Windows batch file to open the website solution in Microsoft Visual Studio
Solution.lnk	The shortcut file to the project to open the website solution in Microsoft Visual Studio
folder.lnk	The shortcut file to the website folder
PhoneRepairShop_Code\PhoneRepairShop_Code.csproj	The Visual C# project file
PhoneRepairShop_Code\Examples.cs	The Visual C# source file that contains examples of source code to customize data access classes and business logic controllers
PhoneRepairShop_Code\Properties\AssemblyInfo.cs	The Visual C# Source file that contains general information about an assembly

The platform also creates the `OpenSolution.bat` batch file, which is a copy of the `Solution.bat` file created in the solution project folder. Depending on the settings of your browser, the `OpenSolution.bat` file is saved either in the `Downloads` folder or in another location.

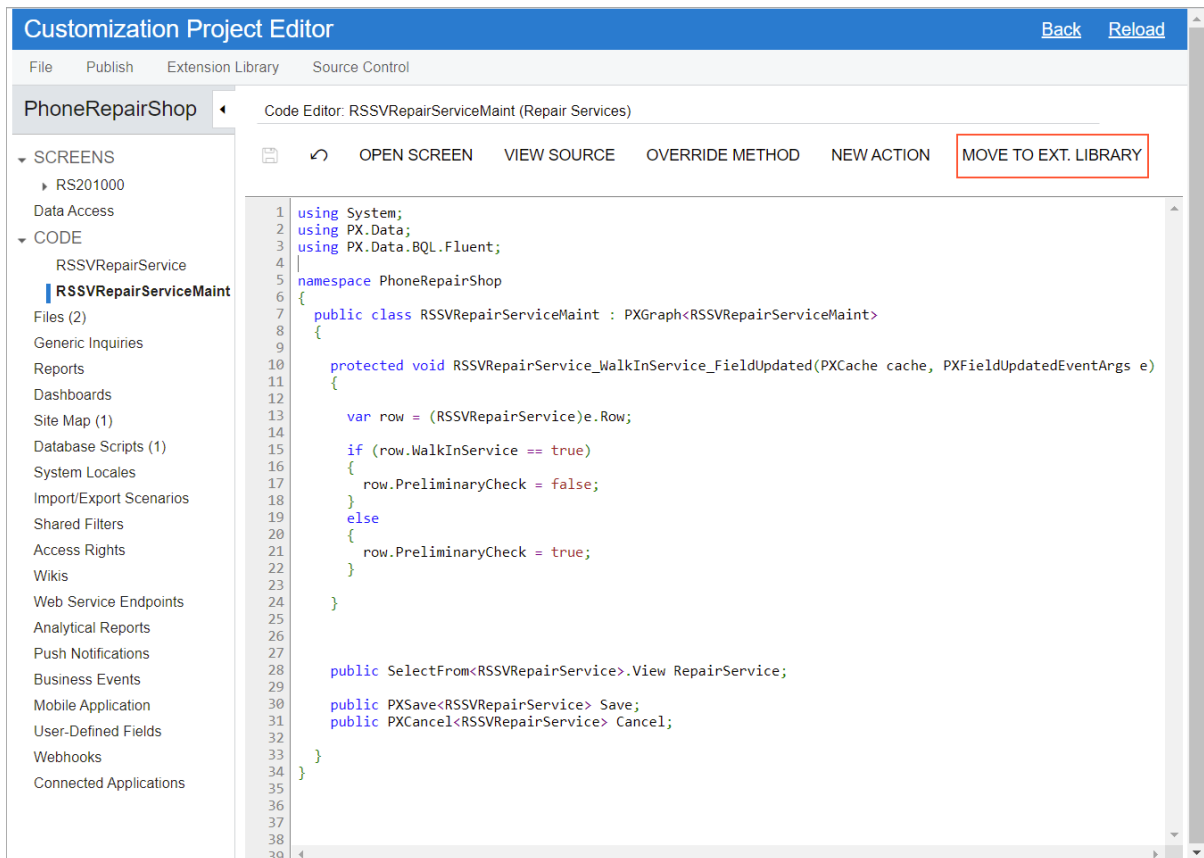
#### Related Links

- [To Create an Extension Library](#)

## Step 1.8.2: Move Code from the Customization Project to the Extension Library

Now that you have created an extension library, you can move code you have created in previous lessons to the extension library. To do this, perform the following actions:

1. Open the *PhoneRepairShop* project in the Customization Project Editor.
2. In the **Code** node of the navigation pane, select **RSSVRepairServiceMaint**.  
The Code Editor page opens.
3. On the page toolbar, click **Move to Ext. Library**, as shown in the following screenshot.

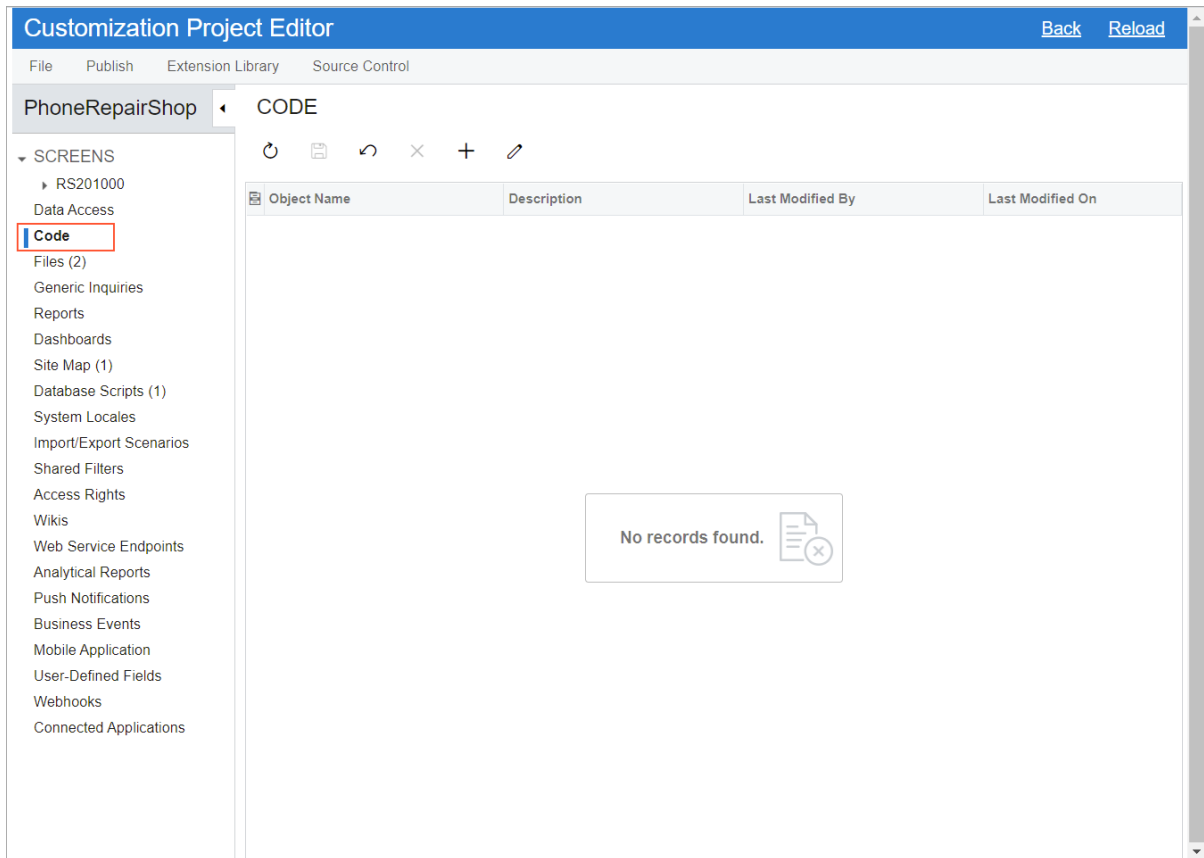
**Figure: Button to move the Code item to the extension library**

While it moves the item to the extension library, the platform does the following:

- In the `App_Data\Projects\PhoneRepairShop_Code\PhoneRepairShop_Code` folder of the website, creates the `RSSVRepairServiceMaint.cs` file, which contains the corresponding customization code.
- Includes the `RSSVRepairServiceMaint.cs` file in the Visual C# project file for the website solution as follows.

```
<Compile Include="RSSVRepairServiceMaint.cs" />
```

- Deletes the `RSSVRepairServiceMaint` item from the customization project.
4. Move the `RSSVRepairService` DAC to the extension library as described in the previous instruction.
  5. By viewing the Code page, ensure that the customization project no longer contains any *Code* items (see the following screenshot).



*Figure: The empty list of the Code page*

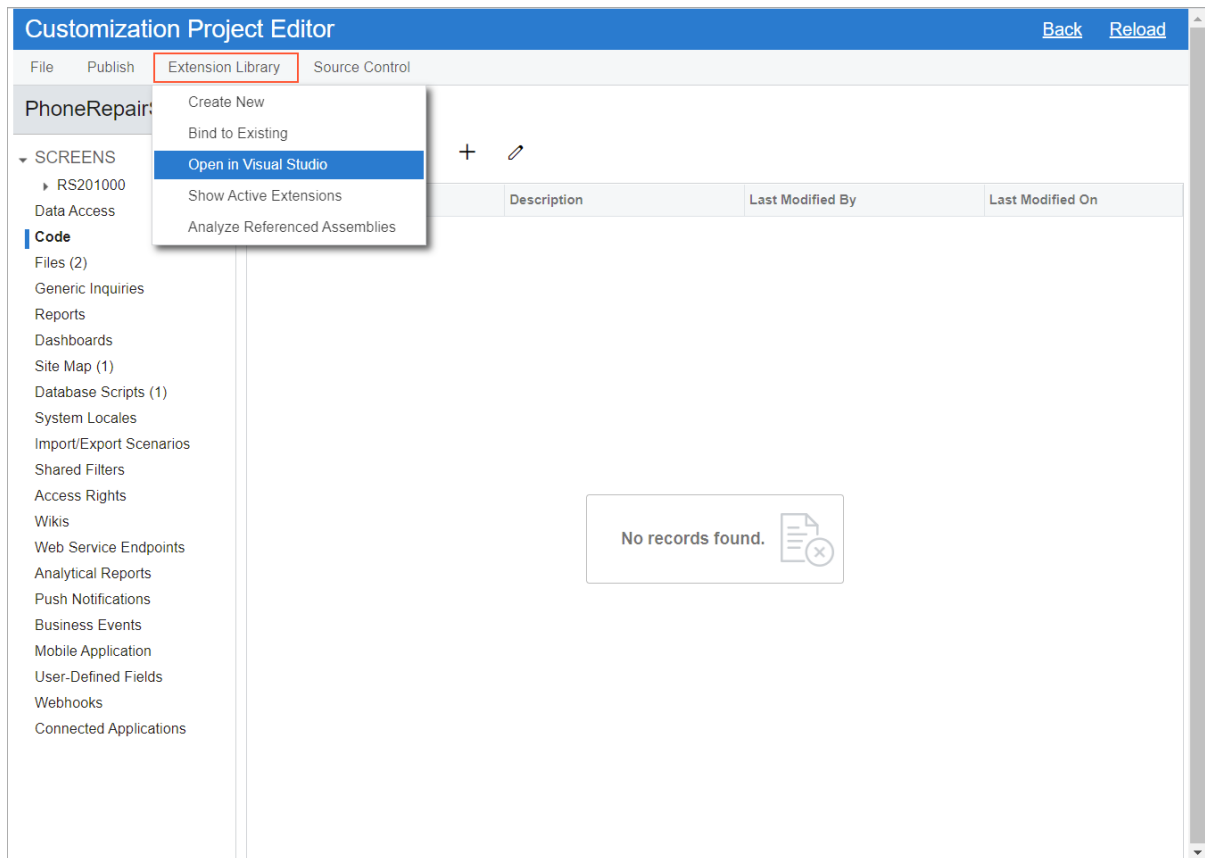
#### Related Links

- [To Move a Code Item to the Extension Library](#)

### Step 1.8.3: Open Solution in Visual Studio

Now you will open the solution in Microsoft Visual Studio. Do the following:

1. Open the *PhoneRepairShop* project in the Customization Project Editor.
2. In the main menu of the Customization Project Editor, click **Extension Library > Open in Visual Studio**, as the following screenshot shows.

**Figure: Opening of the solution from the Customization Project Editor**

The platform downloads the `OpenSolution.bat` batch file on your computer.

3. Run the `OpenSolution.bat` batch file.

Microsoft Visual Studio opens the `PhoneRepairShop_Code` solution.

4. In Solution Explorer of Visual Studio, expand the `PhoneRepairShop_Code` project to view the files included in the project.
5. Ensure that the project includes the `RSSVRepairServiceMaint` and `RSSVRepairService` code items that you moved to extension library in the previous step.
6. To organize the files in the `PhoneRepairShop_Code` project, add a new folder named `DAC` to the `PhoneRepairShop_Code` project, and move the `RSSVRepairService.cs` file to the `DAC` folder. You will create other DACs in this folder.
7. Remove the `Examples.cs` file from the project.

## Step 1.8.4: Build the Project in Visual Studio

Now you will build the project in Visual Studio, which will create the binary file of the extension library. To do this, perform the following actions:

1. In the `PhoneRepairShop_Code` project, add assembly references (if they have not been added yet) for the `PX.Common.Std.dll`, `PX.Data.dll`, and `PX.Data.BQL.Fluent.dll` files, which are located in the `Bin` folder of the `SmartFix_T200` instance folder.
2. Build the `PhoneRepairShop_Code` project in Visual Studio.



To apply changes in the extension library, you do not need to build the entire solution or the `SmartFix_T200` website. You need to build only the `PhoneRepairShop_Code` project.

To ensure that the `PhoneRepairShop_Code.dll` file of the extension library has been created in the `Bin` folder of the website, perform the following actions:

1. Expand the website project in Solution Explorer of Microsoft Visual Studio.
2. In Solution Explorer, expand the **Bin** node of the website project.
3. Scroll down the content of the folder to display the `PhoneRepairShop_Code.dll` binary file.

## Step 1.8.5: Include the Extension Library in the Customization Project

---

Now you will include the extension library in the customization project and test the updated project. Do the following:

1. Open the *PhoneRepairShop* project in the Customization Project Editor.
2. In the navigation pane, click **Files**.  
The Custom Files page opens.
3. On the page toolbar, click **Add New Record**.
4. In the **Add Files** dialog box, which opens, select the **Selected** check box for the *Bin\PhoneRepairShop\_Code.dll* item and click **Save**.
5. Publish the customization project.
6. In Acumatica ERP, open the Repair Services (RS201000) form.
7. In the *ScreenRepair* row, clear the **Walk-In Service** check box. Make sure the **Requires Preliminary Check** check box becomes selected.
8. Select the **Walk-In Service** check box again. The **Requires Preliminary Check** check box becomes cleared.
9. Save your changes.

### Related Links

- [To Add a Custom File to a Project](#)

## Lesson Summary

---

In this lesson, you have learned how to create an extension library, move customization code to the extension library, and open the code items of a customization project in Visual Studio in order to start developing customization code in Visual Studio.

You have completed the following steps:

1. Created an extension library
2. Moved all code items from the customization project to the Visual Studio project
3. Opened the created project in Visual Studio
4. Built the project in Visual Studio

You have also included the extension library in the customization project as a *File* item.

## Lesson 1.9: Add an Event Handler In Visual Studio

In this lesson, you will add an event handler for the **Requires Preliminary Check** check box by using Visual Studio.

### Lesson Objectives

You will learn how to add an event handler in Visual Studio and use Acuminator to refactor existing code.

### Step 1.9.1: Add an Event Handler in Visual Studio

To add an event handler in Visual Studio, do the following:

1. Open the `PhoneRepairShop_Code` solution in Visual Studio.
2. In Solution Explorer, open the `RSSVRepairServiceMaint.cs` file.
3. In the `RSSVRepairServiceMaint` class, add the following event handler.

```
protected void _(Events.FieldUpdated<RSSVRepairService,
    RSSVRepairService.preliminaryCheck> e)
{
    var row = e.Row;
    row.WalkInService = !(row.PreliminaryCheck == true);
}
```

4. In the `PhoneRepairShop_Code` project, add an assembly reference (if it has not been added yet) for the `PX.Common.dll` file, which is located in the `Bin` folder of the `SmartFix_T200` instance folder.
5. Rebuild the project.
6. In the Customization Project Editor, open the `RS201000` screen in the Screen Editor.
7. In the control tree, click **Grid: RepairService > Requires Preliminary Check**.
8. On the **Layout Properties** tab, set the **CommitChanges** property of **Requires Preliminary Check** field to `True`.
9. On the page toolbar, click **Save**.
10. Publish the customization project.



Because you have rebuilt the `PhoneRepairShop.dll` file, which is included in the customization project, you need to update information about this file in the customization project before publication in the **Modified Files Detected** dialog box.

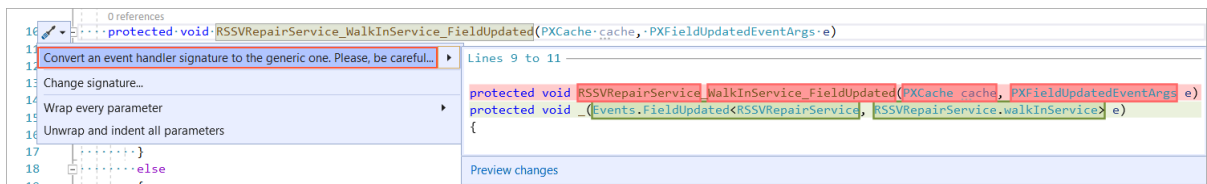
11. Test the added handler as follows:
  - a. In Acumatica ERP, open the Repair Services (`RS201000`) form.
  - b. In the *ScreenRepair* row, select the **Requires Preliminary Check** check box. The **Walk-In Service** check box should be cleared automatically.
  - c. Do not save your changes.

## Step 1.9.2: Use Acuminator to Refactor the Event Handler Declaration

After you added the second event handler, you may have noticed that the definitions of the two event handlers differ. This is because the event handler for the **Walk-In Service** column, which was generated in the Screen Editor, is a traditional one, and the event handler for the **Preliminary Check** column is a generic one. We recommend that you use generic event handlers. For details about these types of event handlers, see [Types of Graph Event Handlers](#).

You can replace the definition of the traditional event handler by using the refactoring feature of Acuminator. Do the following:

1. In Visual Studio, open the `RSSVRepairServiceMaint` class.
2. Right-click the definition of the `RSSVRepairService_WalkInService_FieldUpdated` event handler.
3. In the menu, select **Quick Actions and Refactorings**.
4. In the pop-up menu, select **Convert an event handler signature to the generic one.**, as shown in the following screenshot.



**Figure: Refactoring of the event handler**

The event handler definition changes to the one shown in the following code.

```
protected void _(Events.FieldUpdated<RSSVRepairService,
    RSSVRepairService.WalkInService> e)
```

5. Remove conversion of `e.Row` to the `RSSVRepairService` type, which is not necessary in a generic event handler, as shown in the following code.

```
var row = e.Row;
```

6. Save your changes.
7. Rebuild the project.

### Related Links

- [Types of Graph Event Handlers](#)

## Step 1.9.3: Test the Event Handlers (Self-Guided Exercise)

Now that you have added an event handler for the `FieldUpdated` event of the **Preliminary Check** check box and refactored the event handler for the `FieldUpdated` event of the **Walk-In Service** check box, you should test the behavior in the same way as you did in [Step 1.6.3: Test the Event Handler](#).

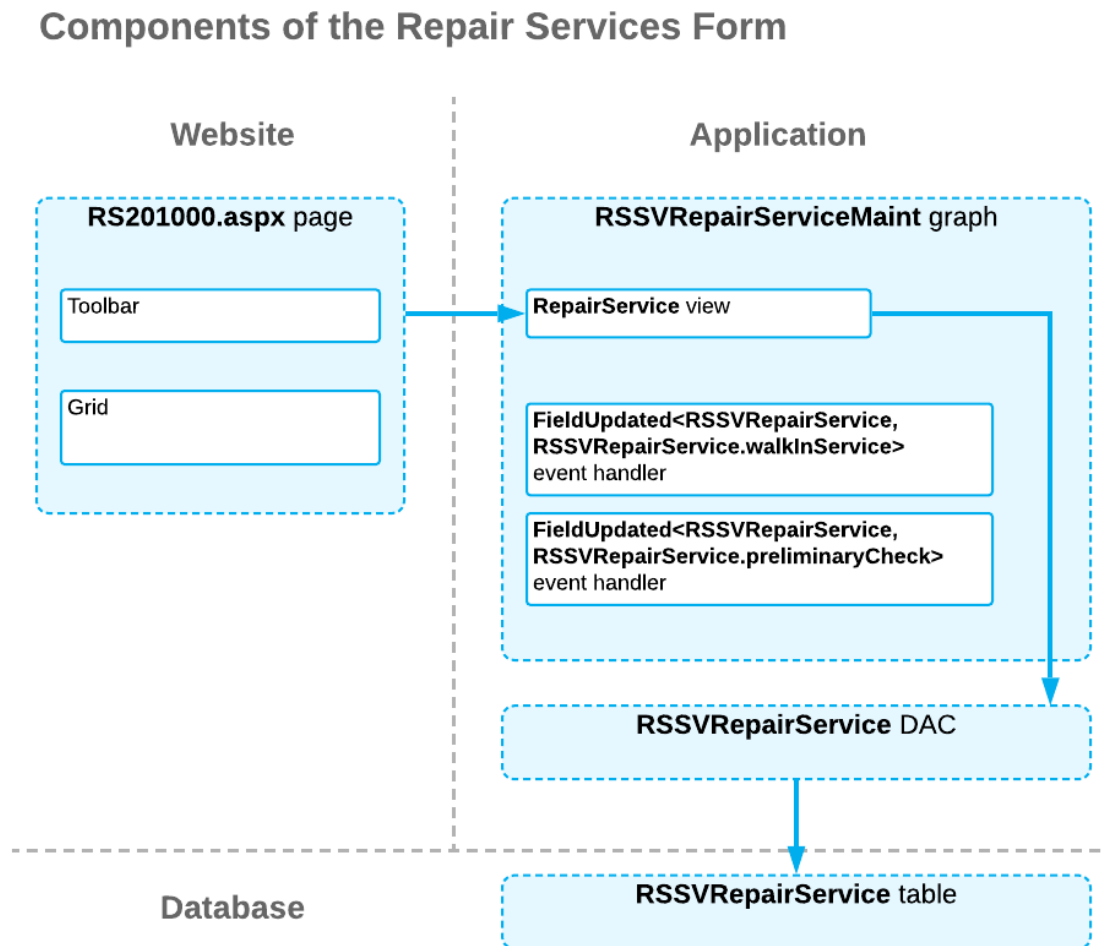
When you are testing the event handlers, make sure the check boxes are selected and cleared automatically in the needed ways for both the **Walk-In Service** and **Preliminary Check** check boxes.

## Lesson Summary

In this lesson, you have added an event handler in Visual Studio and learned how to use Acuminator to refactor the existing code.

## Part 1 Summary

While completing all of the lessons of Part 1, you learned how to create a new form by using the Customization Project Editor. The components that were needed for the new Repair Services (RS201000) maintenance form are shown in the following diagram.



The Repair Services form consists of the following components: the RS201000 ASPX page, the RSSVRepairServiceMaint graph, the RSSVRepairService DAC, and the RSSVRepairService table in the instance database. The RS201000 ASPX page uses the RepairService view as a data member and the RSSVRepairServiceMaint graph as a data source. The RepairService data view of the graph uses the RSSVRepairService DAC to select records from the RSSVRepairService database table.



As you have completed the part, you have debugged the code of the `RSSVRepairServiceMaint` graph, and moved the customization code to an extension library. As a result, you should have the project items in the *PhoneRepairShop* customization project that are shown in the following screenshot.

The screenshot shows the Customization Project Editor interface. The left pane displays a tree view of project items for 'PhoneRepairShop'. The right pane shows a table of custom files.

**Left Pane (PhoneRepairShop):**

- SCREENS
  - RS201000
  - Data Access
  - Code
  - Files (3)
  - Generic Inquiries
  - Reports
  - Dashboards
  - Site Map (1)
  - Database Scripts (1)
  - System Locales
  - Import/Export Scenarios
  - Shared Filters
  - Access Rights
  - Wikis
  - Web Service Endpoints
  - Analytical Reports
  - Push Notifications
  - Business Events
  - Mobile Application
  - User-Defined Fields
  - Webhooks
  - Connected Applications

**Right Pane (Custom Files):**

Object Name	Third Party Assembly	Description	Last Modified By	Last Modified On
Bin\PhoneRepairShop_Code.dll	<input type="checkbox"/>		admin admin	9/9/2022
Pages\RS\RS201000.aspx	<input type="checkbox"/>		admin admin	9/7/2022
Pages\RS\RS201000.aspx.cs	<input type="checkbox"/>		admin admin	9/7/2022

**Figure: Customization project items**

## Part 2: Creating a Form with the Visual Studio (Serviced Devices Form)

---

In this part of the course, you will create the second form of the application, which is the Serviced Devices maintenance form. This form will provide information about devices that can be repaired in the Smart Fix company. You will create the form by using Microsoft Visual Studio.

### Initial Steps

---

Before developing a form in Visual Studio, you should add the database table schema for the `RSSVDevice` table to the customization project by performing similar instructions to those described in [Step 1.1.2: Add a Database Table Schema](#).



The design of database tables is outside of the scope of this course. For details on designing database tables for Acumatica ERP, see [Designing the Database Structure and DACs](#).

### Lesson 2.1: Create a Graph and a DAC in Visual Studio

---

In this lesson, you will create items that you need to define the Serviced Devices maintenance form in Visual Studio: a DAC and a graph.

#### Lesson Objectives

As you complete this lesson, you will learn how to do the following:

- Define a graph in Visual Studio
- Use Acuminator code snippets to define a DAC and its fields in Visual Studio
- Define a selector control for a field and a drop-down menu for a field

#### Step 2.1.1: Define the `RSSVDeviceMaint` Graph

---

To define a graph in Visual Studio, do the following:

1. Open the `PhoneRepairShop_Code` solution in Visual Studio.
2. In Solution Explorer, right-click the `PhoneRepairShop_Code` project, and click **Add > Class...**
3. In the **Add New Item** dialog box, type the name of the class being created, `RSSVDeviceMaint.cs`, and click **Add**.
4. Add the following directives to the `RSSVDeviceMaint.cs` file.

```
using PX.Data;
using PX.Data.BQL.Fluent;
```

5. Replace the code on the `PhoneRepairShop` namespace that is generated in the `RSSVDeviceMaint.cs` file with the following code.

```
namespace PhoneRepairShop
```

```
{
    public class RSSVDeviceMaint : PXGraph<RSSVDeviceMaint>
    {
    }
}
```

In the code above, you are defining the `RSSVDeviceMaint` graph.

6. Save your changes, and rebuild the project.

## Step 2.1.2: Create a DAC in Visual Studio

You will use Acuminator code snippets to simplify creation of the data access class (DAC) in Visual Studio. To create the `RSSVDevice` DAC in Visual Studio, do the following:

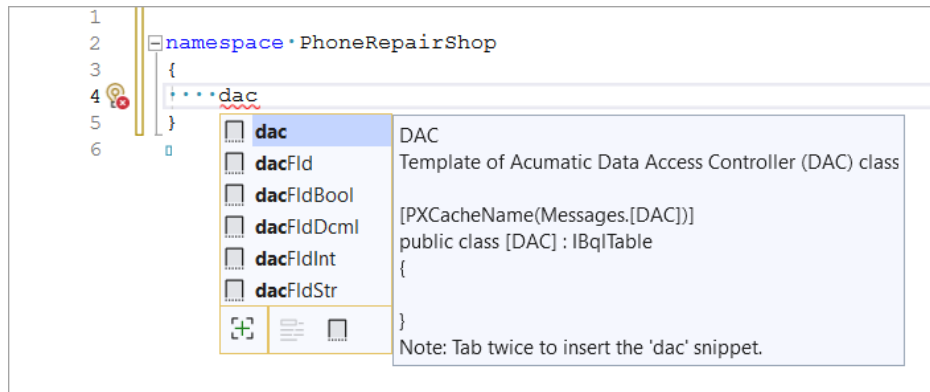
1. Open the `PhoneRepairShop_Code` project in Visual Studio.
2. Add messages for the user-friendly names of the DACs as follows:
  - a. In the project, add the `Helper` folder.
  - b. In the `Helper` folder, add the `Messages.cs` file.
  - c. In the `Messages.cs` file, add the messages as shown in the following code. You will use these messages in the `PXCacheName` attributes of the DACs.

```
namespace PhoneRepairShop
{
    public static class Messages
    {
        //DAC names
        public const string RSSVDevice = "Serviced Device";
        public const string RSSVRepairService = "Repair Service";
    }
}
```

3. In Solution Explorer, add a new item named `RSSVDevice.cs` based on the *Class* template in the DAC folder of the project.
4. Replace the generated code of the `RSSVDevice.cs` file with the following code.

```
namespace PhoneRepairShop
{
}
```

5. Use Acuminator code snippets to create the `RSSVDevice` DAC as follows:
  - a. Inside the `PhoneRepairShop` namespace, type `dac`. The list of available code snippets is displayed, as shown in the following screenshot.



**Figure: Insertion of the DAC code snippet**

- b. Tab twice to insert the template for the DAC code. The following template code is inserted.

```
using PX.Data;

namespace PhoneRepairShop
{
    [PXCachedName(Messages.DAC)]
    public class DAC : IBqlTable
    {

    }
}
```

- c. Replace the name of the class and the constant of the messages class with `RSSVDevice`. Extend the `PXBqlTable` class in the class declaration. Every DAC must be declared as extending the `PX.Data.PXBqlTable` class and implementing the `PX.Data.IBqlTable` interface. The final code should look as follows.

```
using PX.Data;

namespace PhoneRepairShop
{
    [PXCachedName(Messages.RSSVDevice)]
    public class RSSVDevice : PXBqlTable, IBqlTable
    {

    }
}
```

6. Replace the string in the `PXCachedName` attribute of the `RSSVRepairService` DAC with the `RSSVRepairService` constant that you defined in the `Messages.cs` file, as shown in the following code.

```
[PXCachedName(Messages.RSSVRepairService)]
public class RSSVRepairService : PXBqlTable, IBqlTable
```

7. Save your changes.

### Step 2.1.3: Define DAC Fields in Visual Studio

Now you will define fields for the `RSSVDevice` data access class (DAC) in Visual Studio. Do the following:

1. Open the `PhoneRepairShop_Code` project in Visual Studio.
2. In the `Helper` folder, add the `Constants.cs` file.
3. In the `Constants.cs` file, add the constants shown in the following code. You will need these constants to define a drop-down list.

```
namespace PhoneRepairShop
{
    public static class RepairComplexity
    {
        public const string Low = "L";
        public const string Medium = "M";
        public const string High = "H";
    }
}
```

4. In the `Messages.cs` file, add the messages as shown in the following code. You will need these messages to define the values in a drop-down list.

```
//Complexity of repair
public const string High = "High";
public const string Medium = "Medium";
public const string Low = "Low";
```

5. In the `RSSVDevice.cs` file, add a definition of the `DeviceID` field to the `RSSVDevice` class by using an Acuminator code snippet, as follows:
  - a. In the `RSSVDevice` class, type `dac`.
  - b. In the list of available code snippets, select `dacFldInt` and tab twice to insert the template code for an integer DAC field. The `PX.Data.BQL` using directive and the following template code are inserted.

```
#region Field
public abstract class bqlField : BqlInt.Field<bqlField> { }

[PXDBInt]
[PXUIField(DisplayName = "Field")]
public virtual int? Field
{
    get;
    set;
}
#endregion
```

- c. Replace the name of the field in the `region` block with `DeviceID`. The name of the property field of the DAC is adjusted automatically to `DeviceID`.
- d. Replace the name of the class field of the DAC with `deviceID`.
- e. Replace all attributes of the field with `PXDBIdentity` because this is an identity field. The final code of the DAC field looks as follows.

```
#region DeviceID
public abstract class deviceID : BqlInt.Field<deviceID> { }
```

```
[PXDBIdentity]
public virtual int? DeviceID
{
    get;
    set;
}
#endregion
```

6. In the `RSSVDevice.cs` file, add to the `RSSVDevice` class the definitions of the rest of the fields, which are the following:



You can use Acuminator code snippets to add these fields.

- **Device Code:** This field is a selector field, so you should add the `PXSelector` attribute for it, as shown in the following code.

```
#region DeviceCD
[PXDBString(15, IsUnicode = true, IsKey = true,
    InputMask = ">aaaaaaaaaaaaa")]
[PXDefault]
[PXUIField(DisplayName = "Device Code")]
[PXSelector(typeof(Search<RSSVDevice.deviceCD>),
    typeof(RSSVDevice.deviceCD),
    typeof(RSSVDevice.active),
    typeof(RSSVDevice.avgComplexityOfRepair))]
public virtual string DeviceCD { get; set; }
public abstract class deviceCD : PX.Data.BQL.BqlString.Field<deviceCD> { }
#endregion
```

In this code, you have done the following:

- Bound the `DeviceCD` field to the string column in the database by using the `PXDBString` attribute
- Made the field mandatory for the input by using the `PXDefault` attribute without parameters
- Defined the name of the corresponding UI control by using the `PXUIField` attribute
- Configured the selector control by using the `PXSelector` attribute

In the first parameter of the `PXSelector` attribute constructor, you specify a `Search<>` BQL query to select data records for the control. The rest of the parameters define the list of columns to be displayed in the lookup table. When a user selects a data record in the control, the system copies the value of the key field of the selected row and assigns it to the data field. For details, see [PXSelectorAttribute Class](#).

- **Description:** In the following code, you bind the `Description` field to a string column in the database by using the `PXDBString` attribute and define the name of the corresponding UI control by using the `PXUIField` attribute.

```
#region Description
[PXDBString(256, IsUnicode = true, InputMask = "")]
[PXUIField(DisplayName = "Description")]
public virtual string Description { get; set; }
public abstract class description :
    PX.Data.BQL.BqlString.Field<description>
{ }
#endregion
```

- **Active:** In the following code, you bind the `Active` field to a Boolean column in the database by using the `PXDBBool`, set the default value of the check box by using the `PXDefault` attribute, and specify the name of the check box by using the `PXUIField` attribute.

```
#region Active
[PXDBBool()]
[PXDefault(true)]
[PXUIField(DisplayName = "Active")]
public virtual bool? Active { get; set; }
public abstract class active : PX.Data.BQL.BqlBool.Field<active> { }
#endregion
```

- **Complexity:** This field is a drop-down list, so you should add the `PXStringList` attribute for it, as the following code demonstrates.

```
#region AvgComplexityOfRepair
[PXDBString(1, IsFixed = true)]
[PXDefault(RepairComplexity.Medium)]
[PXUIField(DisplayName = "Complexity")]
[PXStringList(
    new string[]
    {
        RepairComplexity.Low,
        RepairComplexity.Medium,
        RepairComplexity.High
    },
    new string[]
    {
        Messages.Low, Messages.Medium, Messages.High
    })]
public virtual string AvgComplexityOfRepair { get; set; }
public abstract class avgComplexityOfRepair :
    PX.Data.BQL.BqlString.Field<avgComplexityOfRepair>
{ }
#endregion
```

In the code above, you have done the following:

- Bound the `AvgComplexityOfRepair` field to the string column in the database by using the `PXDBString` attribute.
- Set the default value of the UI control by using the `PXDefault` attribute.
- Defined the name of the UI control by using the `PXUIField` attribute.
- Configured the drop-down list by using the `PXStringList` attribute. In the first parameter of the `PXStringList` attribute constructor, you specify the list of possible values for the control. In the second parameter, you specify the list of labels displayed in the UI. For details, see [PXStringListAttribute Class](#).



For fields that store combo box values (such as the `AvgComplexityOfRepair` field), we recommend using a non-Unicode string field with fixed length of 1 symbol. You specify this string as follows: `[PXDBString(1, IsFixed = true)]`.

- Fields for the mandatory system columns, which are the following:
  - `CreatedDateTime`
  - `CreatedByID`
  - `CreatedByScreenID`
  - `LastModifiedDateTime`

- LastModifiedByID
- LastModifiedByScreenID
- Tstamp
- NoteID

You can copy these fields from the `RSSVRepairService` class. You will also need to add the `using System;` directive to the `RSSVDevice.cs` file. For details on system columns, see [Audit Fields](#), [Concurrent Update Control \(TStamp\)](#), and [Attachment of Additional Objects to Data Records \(NoteID\)](#) in the documentation.

7. Save your changes.

#### Related Links

- [Working with Attributes](#)

## Step 2.1.4: Configure the RSSVDeviceMaint Graph

---

Now that you have defined the `RSSVDevice` data access class, you can adjust the `RSSVDeviceMaint` graph by doing the following:

1. Open the `RSSVDeviceMaint.cs` file.
2. Add the `RSSVDevice` type parameter to the graph definition as follows.

```
public class RSSVDeviceMaint : PXGraph<RSSVDeviceMaint, RSSVDevice>
```

By specifying this type parameter, you define the set of standard buttons for the form toolbar.

3. Declare the following data view inside the `RSSVDeviceMaint` graph.

```
public SelectFrom<RSSVDevice>.View ServDevices;
```

4. Rebuild the project.

Now you can use this view as a data member of the Serviced Devices (RS202000) form.

## Lesson Summary

---

In this lesson, you have learned how to create a graph and a DAC in Visual Studio. Also, you learned how to configure a lookup control by using the `PXSelector` attribute and a drop-down list by using the `PXStringList` attribute.

## Lesson 2.2: Create an ASPX Page in Visual Studio

---

In this lesson, you will learn how to create an ASPX and ASPX.CS files for the Serviced Devices (RS202000) form.

### Lesson Objectives

As you complete this lesson, you will learn how to do the following:

- Create an ASPX file
- Configure a data member for an Acumatica ERP form



- Add the ASPX and ASPX.CS files to the customization project

## Step 2.2.1: Create the RS202000.aspx Page

To create an ASPX page in Visual Studio, do the following:

1. Open the PhoneRepairShop\_Code solution in Visual Studio.
2. In Solution Explorer, select the SmartFix\_T200 project, which is the website project.
3. Add a new item named RS202000.aspx based on the *Web Form* template in the **SmartFix\_T200 > Pages > RS** folder.

The RS202000.aspx and RS202000.aspx.cs files are created.



The RS folder already contains the RS201000.aspx and RS201000.aspx.cs files, which were created in Part 1 of the course.

4. In the RS202000.aspx file, replace the code with the following code. Note the values of the TypeName and PrimaryView attributes of the PXDataSource control, and the values of the DataSourceID and DataMember attributes of the PXFormView control.



Templates for different types of forms are located in the AppData/Templates folder of the instance folder.

```
<%@ Page Language="C#" MasterPageFile="~/MasterPages/FormView.master"
AutoEventWireup="true" ValidateRequest="false" CodeFile="RS202000.aspx.cs"
Inherits="Page_RS202000" Title="Untitled Page" %>
<%@ MasterType VirtualPath="~/MasterPages/FormView.master" %>
<asp:Content ID="cont1" ContentPlaceHolderID="phDS" Runat="Server">
<px:PXDataSource ID="ds" runat="server" Visible="True" Width="100%"
    TypeName="PhoneRepairShop.RSSVDeviceMaint"
    PrimaryView="ServDevices"
>
<CallbackCommands>
</CallbackCommands>
</px:PXDataSource>
</asp:Content>
<asp:Content ID="cont2" ContentPlaceHolderID="phF" Runat="Server">
<px:PXFormView ID="form"
    runat="server" DataSourceID="ds" DataMember="ServDevices"
    Width="100%" AllowAutoHide="false">
<Template>
    <px:PXLayoutRule ID="PXLayoutRule1" runat="server" StartRow="True" />
</Template>
<AutoSize Container="Window" Enabled="True" MinHeight="200" />
</px:PXFormView>
</asp:Content>
```

For the `TypeName` attribute, you specify the name of the graph that defines the business logic of the form. For the `PrimaryView` and the `DataMember` attributes, you specify the name of the view that you defined in [Step 2.1.4: Configure the RSSVDeviceMaint Graph](#). For the `DataSourceID` attribute, you specify the ID of the datasource control on the page.

5. Save your changes.
6. In the RS202000.aspx.cs file, replace the code with the following code.

```
using System;

public partial class Page_RS202000 : PX.Web.UI.PXPage
{
    protected void Page_Load(object sender, EventArgs e)
    {
    }
}
```

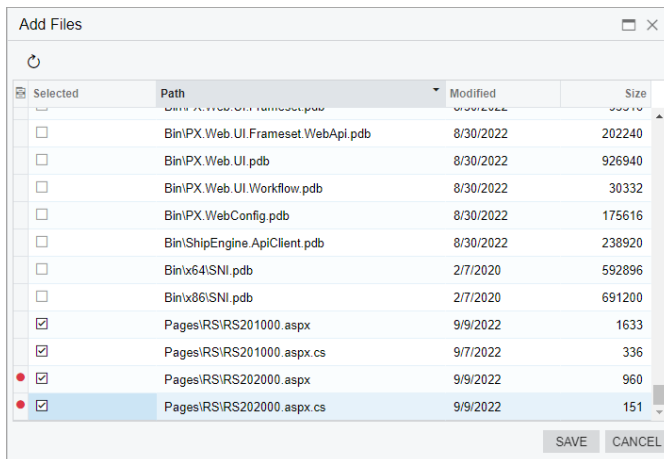


You can create the same form in the Screen Editor of the Customization Project Editor by using the New Screen wizard. The form should be based on the *Form (FormView)* template. You add boxes on a form by using the **Add Data Fields** tab (as described in [Step 1.5.1: Add Columns to the Grid](#)) and layout rules by using the **Add Controls** tab of the Screen Editor. For details, see [To Add a Layout Rule](#).

## Step 2.2.2: Add ASPX and ASPX.CS Files to the Customization Project

After you create the `RS202000.aspx` and `RS202000.aspx.cs` files in Visual Studio, you add them to the customization project as follows:

1. Open the *PhoneRepairShop* customization project in Customization Project Editor.
2. In the navigation pane, click **Files**. The Custom Files page opens.
3. On the page toolbar, click **Add New Record**.
4. In the **Add Files** dialog box, select the unlabeled check boxes for the `Pages\RS\RS202000.aspx` and `Pages\RS\RS202000.aspx.cs` items, as shown in the following screenshot, and click **Save**.



**Figure:** The Add Files dialog box

5. Publish the customization project.



The form is not yet available in the Acumatica ERP because you have not yet added the form to the site map. This step is described in [Lesson 2.4: Add the Form to the Site Map and Workspace](#). After performing the step, you will be able to access the form in Acumatica ERP.

## Lesson Summary

---

In this lesson, you have learned how to create an ASPX page in Visual Studio and configure a data member for it. Also, you have added created files to the customization project.

## Lesson 2.3: Configure a Form in Visual Studio

---

In this lesson, you will learn how to add input controls on a form and configure the layout of the form in Visual Studio. After that, you will test the form by specifying values and saving them to the database.

### Lesson Objectives

As you complete this lesson, you will learn how to use Visual Studio to add controls and configure layout of a form.

### Step 2.3.1: Add Input Controls

---

In this step, you will add the following input controls to a form:

- A selector for the `Device Code` field
- A text box for the `Description` field
- A check box for the `Active` field
- A combo box for the `Complexity` field

You add input controls to a form by defining them in the ASPX code of the form. The type of an input control correlates with the attributes of the DAC field. For example, to add a check box, you add an input control of the `PXCheckBox` type.

In ASP.NET markup, the following properties are required for every input control:

- `ID`: Identifies the control within the page. This property is required by ASP.NET.
- `runat="Server"`: Indicates that the server should create an object of the specified class. This property is required by ASP.NET.
- `DataField`: Specifies the DAC field represented by the control.

To add these input controls, do the following:

1. In Visual Studio, open the `RS202000.aspx` file.
2. Define the fields by adding the following code after the `px:PXLayoutRule` tag:
  - For the `Device Code` field:

```
<px:PXSelector ID="DeviceCD" runat="server" DataField="DeviceCD">
</px:PXSelector>
```

- For the `Description` field:

```
<px:PXTextEdit ID="Description" runat="server" DataField="Description"
  DefaultLocale=""></px:PXTextEdit>
```

- For the `Active` field:

```
<px:PXCheckBox ID="Active" runat="server" DataField="Active">
```

```
</px:PXCheckBox>
```

- For the Complexity field:

```
<px:PXDropDown ID="AvgComplexityOfRepair" runat="server"
    DataField="AvgComplexityOfRepair"></px:PXDropDown>
```

3. Save your changes.

#### Related Links

- [Customizing Elements of the User Interface](#)
- [Input Controls](#)

## Step 2.3.2: Configure the Layout

---

You should organize the elements added to the form into two columns. To do this, you will use the `PXLayoutRule` component to configure the layout rules on the form as follows:

1. In Visual Studio, open the `RS202000.aspx` file.
2. Replace the `PXLayoutRule` tag before the `PXSelector` tag with the tag shown in the following code. This `PXLayoutRule` tag indicates that all elements after it are placed in a new row.

```
<px:PXLayoutRule ID="PXLayoutRule1" runat="server" StartRow="True"
    ControlSize="M" LabelsWidth="S"></px:PXLayoutRule>
```

For details, see [Use of the StartRow and StartColumn Properties of PXLayoutRule](#).

3. Put the following `PXLayoutRule` tag before the `<px:PXCheckBox ID="Active">` tag.

```
<px:PXLayoutRule ID="PXLayoutRule2" runat="server" StartColumn="True"
    ControlSize="M" LabelsWidth="S"></px:PXLayoutRule>
```

This `PXLayoutRule` tag indicates that all elements after it are placed in a new column.

4. Save your changes.

#### Related Links

- [Layout Rule \(PXLayoutRule\)](#)

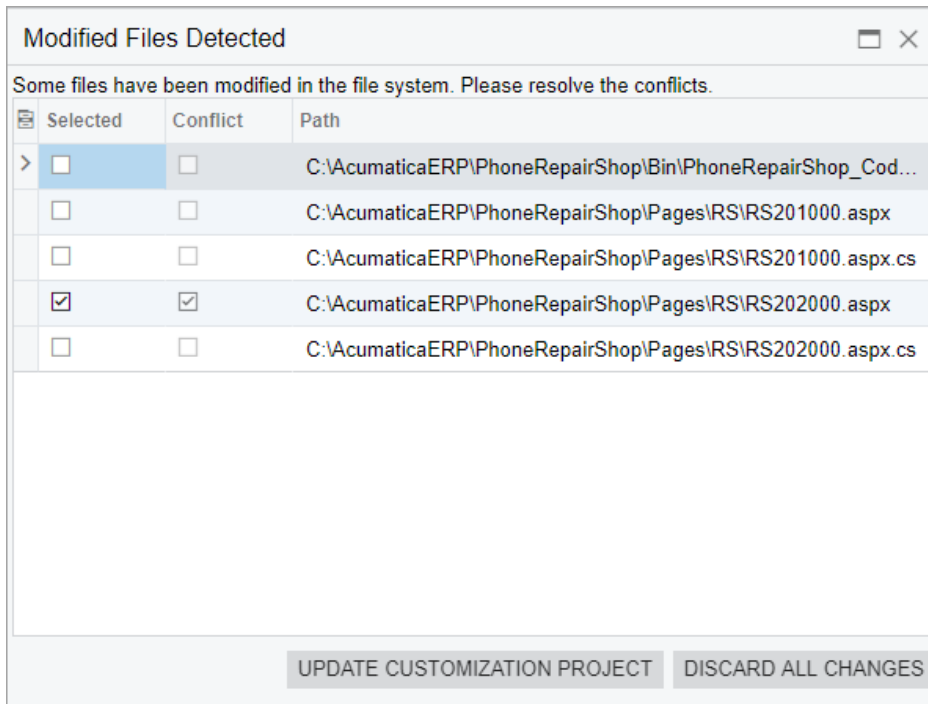
## Step 2.3.3: Update the Files in the Customization Project

---

When you use Visual Studio to modify files that are added to a customization project, you should update them in the customization project as follows:

1. Open the *PhoneRepairShop* project in the Customization Project Editor.
2. Click **Files** in the navigation pane.
3. On the page toolbar, click **Detect Modified Files**.

The **Modified Files Detected** dialog box opens, as shown in the following screenshot.



*Figure: The Modified Files Detected dialog box*

4. In the dialog box, make sure check boxes in the row of the `RS202000.aspx` file are selected, and click **Update Customization Project**. Close the dialog box.
5. Publish the customization project.



You can skip going to the Custom Files page to update files if you intend to publish the project from the start: If any custom files were modified outside of the Customization Project Editor, when you click **Publish Current Project** on the main menu, the same dialog box is opened.

#### Related Links

- [Detecting the Project Items Modified in the File System](#)

## Lesson Summary

In this lesson, you have learned how to configure input controls on a form in Visual Studio and organize the layout of the form.

To configure controls on the Serviced Devices (RS202000) form, you have used `PXSelector`, `PXTextEdit`, `PXCheckBox`, and `PXDropDown` tags in ASPX. To organize the layout of the form, you have added the `PXLayoutRule` tags. You have also updated the modified ASPX file in the customization project.

## Lesson 2.4: Add the Form to the Site Map and Workspace

To make the Serviced Devices (RS202000) form visible in Acumatica ERP, you should create a site map item for the form. In the created site map item, you specify in which workspace the form should be displayed. You also need to configure the access rights for the form before adding it to the specified workspace.

## Lesson Objectives

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

- Create a site map item for a custom form and configure the access rights for the form
- Save the created site map item to the customization project
- Add a form created in Visual Studio to the Screen Editor

### Step 2.4.1: Create a Site Map Item for the Form

---

To create a site map item for a form, do the following:

1. Open the *PhoneRepairShop* customization project in the Customization Project Editor.
2. In the navigation pane, click **Site Map**. The Site Map page opens.
3. On the page toolbar, click **Manage Site Map**.

The [Site Map](#) (SM200520) form opens.

4. On the form toolbar, click **Add Row**.
5. In the new row specify the following settings:
  - **Screen ID:** RS202000
  - **Title:** Serviced Devices
  - **URL:** ~/Pages/RS/RS202000.aspx (specified automatically)
  - **Workspaces:** *Phone Repair Shop*
  - **Category:** *Configuration*

6. Save your changes and close the form.

The new site map item is created. Now you can configure the access rights for the form and include them in your customization project.

7. To configure the access rights for the form, perform similar instructions to those described in [Step 1.2.2: Configure Access Rights for the Created Form](#).



On the [Access Rights by Screen](#) (SM201020) form, you can find the Serviced Devices form by expanding the **Phone Repair Shop** node instead of the **Hidden** node because you have already specified *Phone Repair Shop* as the workspace for this form in the preceding Instruction 5.

8. To include the configured access rights in your customization project, perform similar instructions to those described in [Step 1.2.3: Include Access Rights of the Created Form in the Customization Project](#).
9. Open the **Phone Repair Shop** workspace. Notice that the Serviced Devices form is unavailable in the quick menu.
10. Click **... > Edit Menu**.
11. In the **Phone Repair Shop** workspace, select the check box for the Serviced Devices form to display the form in the quick menu.
12. Click **Exit Menu Editing**.

#### Related Links

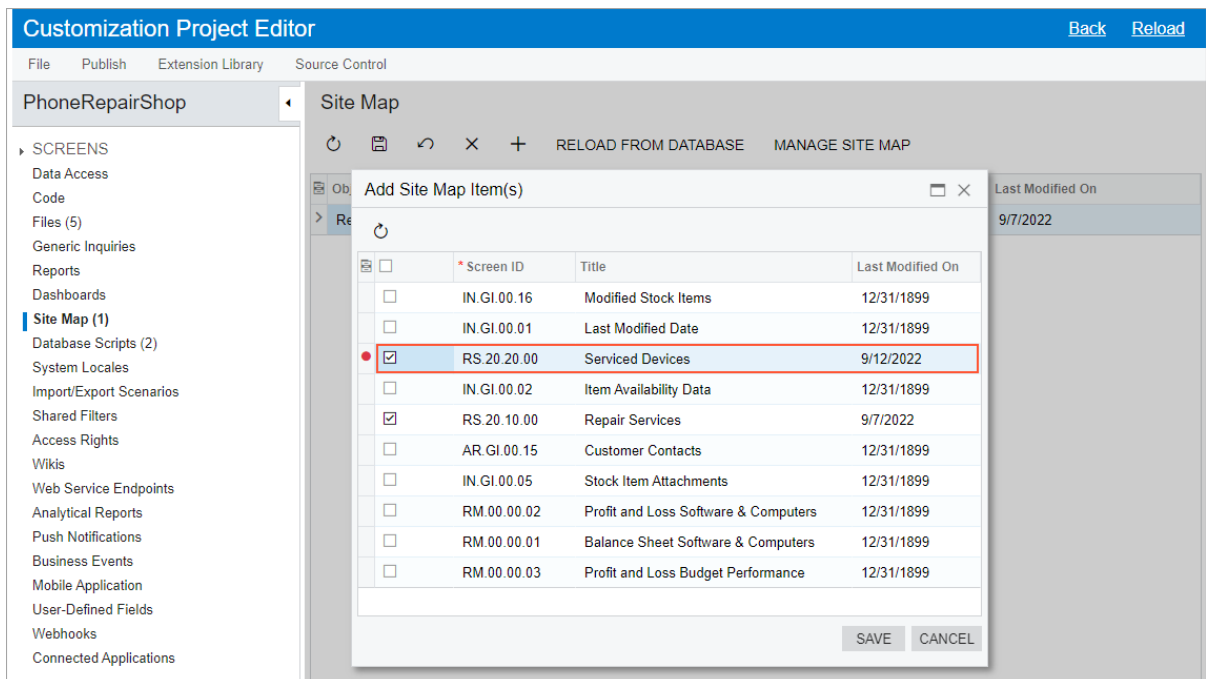
- [UI Navigation Options: Site Map](#)

## Step 2.4.2: Add the Site Map Item to the Customization Project

Now that you have added the Serviced Devices (RS202000) form to the **Phone Repair Shop** workspace, you will add the created site map item to your customization project, so that if you publish the customization project on another instance, the information about the site map location of the form will also be applied.

To add a site map item to the customization project, do the following:

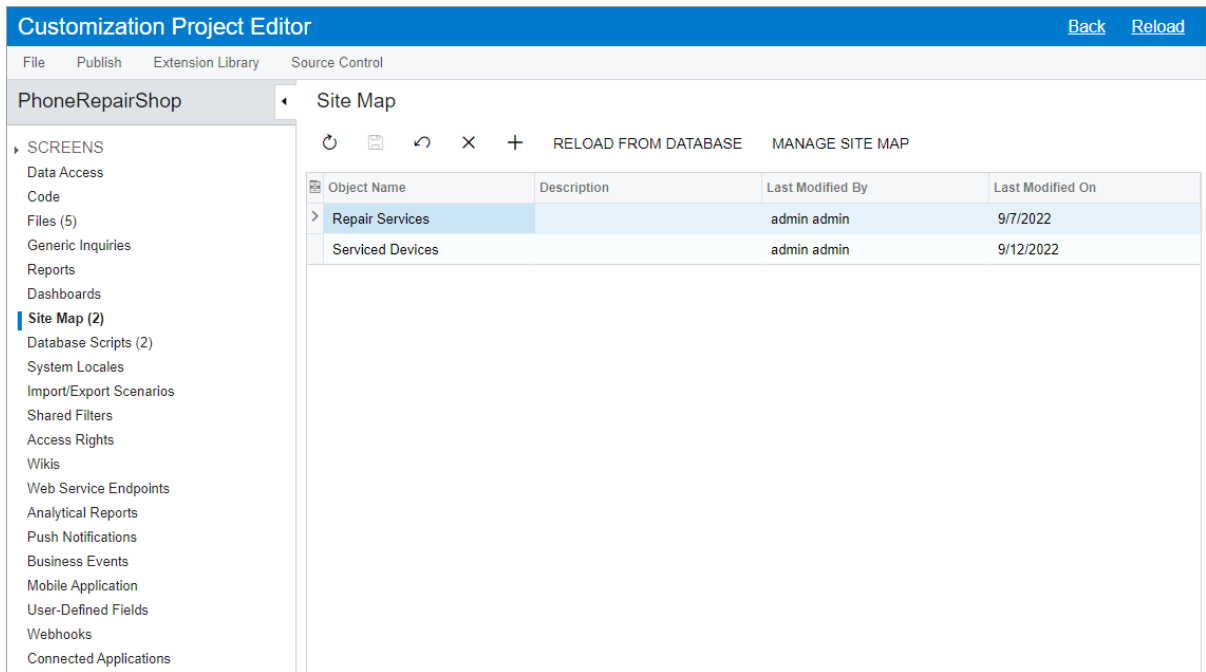
1. Open the *PhoneRepairShop* customization project in the Customization Project Editor.
2. On the navigation pane, click **Site Map** to open the Site Map page.
3. On the page toolbar, click **Add New Record**.
4. In the **Add Site Map** dialog box, which is opened, find the row with the *RS202000* screen ID and select the unlabeled check box in that row, as shown in the following screenshot.



**Figure: The selected site map item**

5. Click **Save**.

The site map item is added to the customization project. The Site Map page should look as shown in the following screenshot.



*Figure: The Site Map page*

#### Related Links

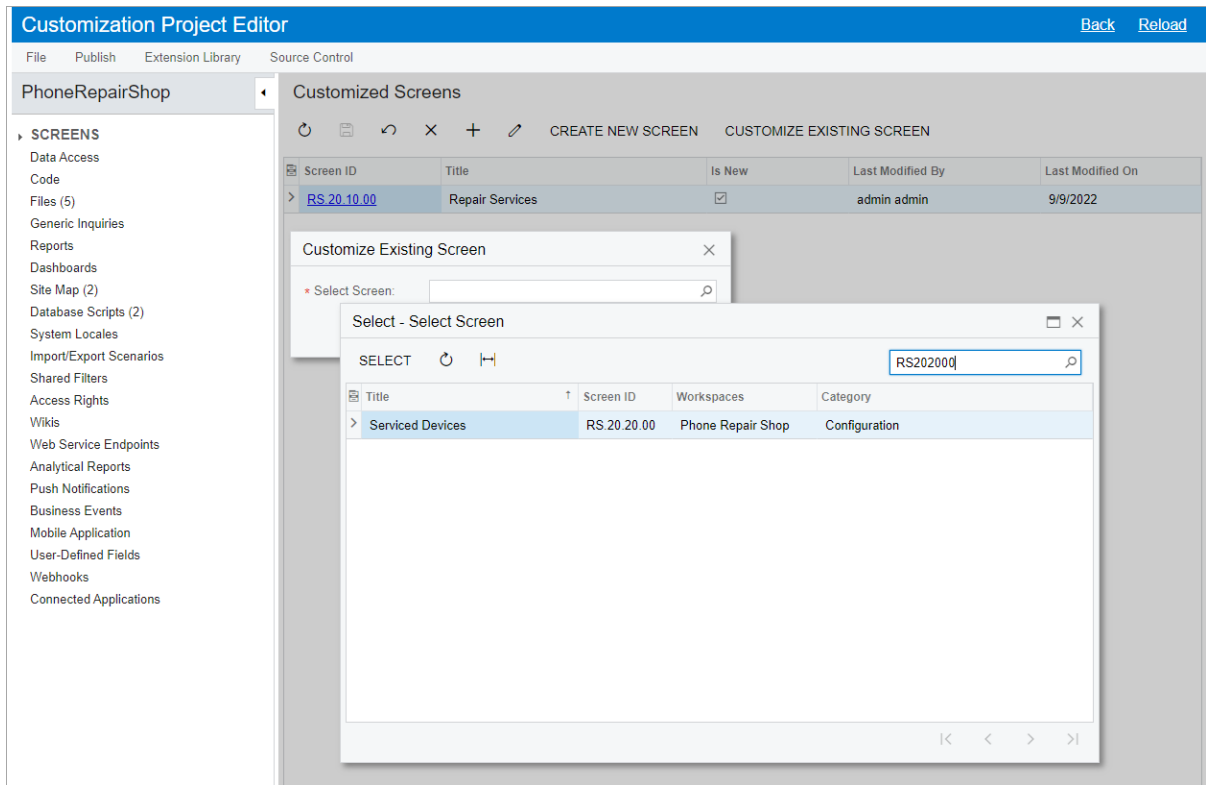
- [To Add a Site Map Node to a Project](#)

### Step 2.4.3: Add the Form to the Screen Editor

You can edit a form created in Visual Studio both in Visual Studio and in the Screen Editor. To be able to edit the form in the Screen Editor, you should add it to the Screen Editor by doing the following:

1. Open the *PhoneRepairShop* customization project in the Customization Project Editor.
2. In the navigation pane, click **Screens**. The Customized Screens page opens.
3. On the page toolbar, click **Customize Existing Screen**.
4. In the **Customize Existing Screen** dialog box which opens, select the Serviced Devices (RS202000) form as shown in the following screenshot.

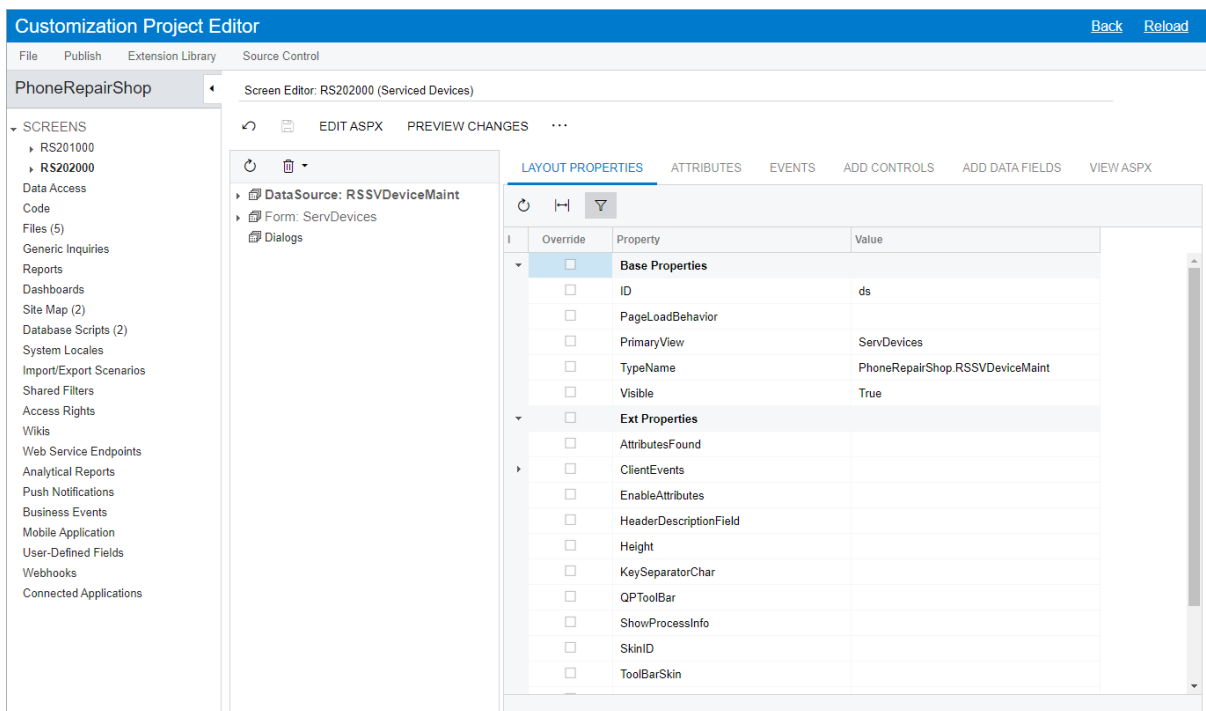




**Figure: Selecting the Serviced Devices form**

- Click **OK**.

The Serviced Devices form opens in the Screen Editor as shown in the following screenshot.



**Figure: Viewing the Serviced Devices form in the Screen Editor**

- Publish the customization project.

## Step 2.4.4: Test the Form

In this step, you will test the Serviced Devices (RS202000) form.

### Adding Records

To test the ability to add a record to the form, do the following:

1. In Acumatica ERP, open the Serviced Devices (RS202000) form (shown in the following screenshot).

**Figure: The Serviced Devices form**

2. Enter the following settings:

- a. **Device Code:** NOKIA3310



For the DAC field of the **Device Code** box, you have specified the input mask in the `PXDBString` attribute. Thus, a mask is applied to the control so that all letters entered in the box are uppercase.

- b. **Description:** Nokia 3310
  - c. **Active:** Selected
  - d. **Complexity:** Low
3. On the form toolbar, click **Save**.
  4. By performing the same actions you did in the previous two instructions, add the devices listed in the table below with the settings indicated for each.

Device Code	Description	Active	Repair Complexity
MotorRAZR	Motorola RAZR V3	Selected	Low
SamsungGS4	Samsung Galaxy S4	Selected	Medium
iPhone6	iPhone 6	Selected	High

### Editing Records

To test loading and editing a record, do the following:

1. On the Serviced Devices (RS202000) form, in the **Device Code** box, click the selector icon.

The lookup table opens, as shown in the following screenshot.

The screenshot shows the 'Serviced Devices' form in Visual Studio. The 'Device Code' field is set to 'IPHONE6' and the 'Active' checkbox is checked. A 'Select - Device Code' dialog box is open, displaying a table of device codes and their complexities.

Device Code	Active	Complexity
IPHONE6	<input checked="" type="checkbox"/>	High
MOTORRAZR	<input checked="" type="checkbox"/>	Low
NOKIA3310	<input checked="" type="checkbox"/>	Low
SAMSUNGGS4	<input checked="" type="checkbox"/>	Medium

**Figure: The lookup table**

- In the lookup table, notice all the devices you have added and select the *MotorRAZR* device.

The rest of the elements on the form are filled in with the *MotorRAZR* device properties, as shown in the following screenshot.

The screenshot shows the 'Serviced Devices' form with the 'Device Code' field set to 'MOTORRAZR' and the 'Active' checkbox checked. The 'Description' field is filled with 'Motorola RAZR V3' and the 'Complexity' dropdown is set to 'Low'.

**Figure: The device properties**

- Clear the **Active** check box.
- On the form toolbar, click **Save**.

## Lesson Summary

In this lesson, you have added the Serviced Devices (RS202000) form to a workspace, saved a site map item to the customization project, added the form to the Screen Editor, and tested the form.

## Lesson 2.5: Create a Substitute Form

As of this point in the design of the Serviced Devices (RS202000) maintenance form, a user can access a specific device record by opening the form and clicking the magnifier icon in the **Device Code** box; the list of records then

opens in the lookup table. When there are only a small number of serviced devices, this sequence of events is sufficient, but as the number of records grows, it may not give the user the needed information quickly enough.

In Acumatica ERP, you can create a generic inquiry that presents the data entered on a particular data entry or maintenance form (called the *entry form* in this context) in a tabular format. You can then define the generic inquiry as the *substitute form*, which will be brought up instead of the entry form. Thus, when you click the name of the entry form in a workspace or the search results, the system will open the substitute form, which contains the list of records. When you click a record identifier in the list, the system opens the entry form.



The process of creating a generic inquiry is outside of the scope of this course. For details on working with generic inquiries, see the *S130 Data Retrieval and Analysis* training course.

## Lesson Objectives

In this lesson, you will learn how to create a substitute form for a custom form and save it to the customization project.

### Related Links

- [Generic Inquiry as a Substitute Form: General Information](#)

## Step 2.5.1: Upload a Predefined Generic Inquiry

---

The first step in creating a substitute form is creating a generic inquiry based on the particular entry form (which is beyond the scope of this course). For this lesson, you will load a generic inquiry that has been prepared for this training course. Do the following:

1. In Acumatica ERP, open the [Generic Inquiry](#) (SM208000) form.
2. On the form toolbar, click **Clipboard > Import from XML**.
3. In the **Upload XML File** dialog box, select the `Customization\T200\SourceFiles\ListAsEntryPoint\GI_ServicedDevices.xml` file, which you have downloaded from Acumatica GitHub.
4. Click **Upload**.

The predefined generic inquiry is uploaded to the form, as shown in the following screenshot. You can check the inquiry settings on the **Results Grid** tab.

The screenshot shows the 'Generic Inquiry' configuration window. The 'Inquiry Title' is 'Serviced Devices'. The 'Workspace' is 'Data Views'. The 'Category' is 'Inquiries'. The 'Screen ID' is 'RS2020PL'. The 'RESULTS GRID' tab is selected, showing a table with columns: Object, Data Field, Schema Field, Width (px), Style, Visible, Default Navigation, and Navigate To. The table contains four rows of data for 'ServicedDevices'.

Object	Data Field	Schema Field	Width (px)	Style	Visible	Default Navigation	Navigate To
ServicedDevices	DeviceCD	ServicedDevices.Devic...			<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
ServicedDevices	Description	ServicedDevices.Descr...			<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
ServicedDevices	Active	ServicedDevices.Active			<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
ServicedDevices	AvgComplexityOfRepair	ServicedDevices.AvgC...			<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	

Figure: The Serviced Devices generic inquiry

## Step 2.5.2: Configure the Generic Inquiry as a Substitute Form

To make a generic inquiry a substitute form, do the following:

1. Open the *Serviced Devices* generic inquiry on the *Generic Inquiry* (SM208000) form.
2. Open the **Entry Point** tab.
3. In the **Entry Screen** box, select the *Serviced Devices* (RS202000) form, as shown in the following screenshot.

The screenshot shows the 'Generic Inquiry' configuration window with the 'ENTRY SCREEN SET' tab selected. A 'Select - Entry Screen' dialog box is open, showing a table with columns: Title, Screen ID, Workspaces, and Category. The table contains one row for 'Serviced Devices'.

Title	Screen ID	Workspaces	Category
Serviced Devices	RS.20.20.00	Phone Repai...	Configuration

The 'Entry Screen' box is set to 'Serviced Devices'. Below the dialog, there are checkboxes for 'Replace Entry Screen with this Inquiry in Menu', 'Enable Mass Record Deletion', 'Auto-Confirm Custom Delete Confirmations', 'Enable Mass Record Update', and 'Enable New Record Creation'. The 'New Record Defaults' section is also visible.

Figure: Selecting the Serviced Devices form

4. Select the **Replace Entry Screen with this Inquiry in Menu** check box and the **Enable New Record Creation** check box.
5. Save your changes.

You can now access the substitute form by using any of the following ways:

- By searching using *RS2020PL* or *Serviced Devices* as a search string
- By selecting the Serviced Devices form in the **Phone Repair Shop** workspace

#### Related Links

- [Generic Inquiry as a Substitute Form: To Configure an Inquiry as an Entry Point](#)

## Step 2.5.3: Save the Generic Inquiry to the Customization Project

To save the Serviced Devices (RS2020PL) generic inquiry to the customization project, do the following:

1. Open the *PhoneRepairShop* customization project in the Customization Project Editor.
2. On the navigation pane, click **Generic Inquiries**. The Generic Inquiries page opens.
3. On the page toolbar, click **Add New Record**.
4. In the **Add Generic Inquiries** dialog box, select the unlabeled check box in the row with the Serviced Devices generic inquiry, as shown in the following screenshot, and click **Save**.

	*Inquiry Title	Last Modified By	Last Modified
<input type="checkbox"/>	Customer Contacts	admin ad...	1/2/2022
<input type="checkbox"/>	Item Availability Data	admin ad...	1/2/2022
<input type="checkbox"/>	Modified Stock Items	admin ad...	1/2/2022
<input checked="" type="checkbox"/>	Serviced Devices	admin ad...	9/12/2022
<input type="checkbox"/>	Stock Item Attachments	admin ad...	1/2/2022
<input type="checkbox"/>	Stock Items: Last Modified Date	admin ad...	1/2/2022

SAVE CANCEL

**Figure: The Add Generic Inquiries dialog box**

5. To include the access rights for the *Serviced Devices* generic inquiry in your customization project, perform similar instructions to those described in [Step 1.2.3: Include Access Rights of the Created Form in the Customization Project](#). When you perform these instructions, select *RS.20.20.PL* in the **Screen Name** box, and select the **Apply and Keep** option button in the **Merge Rule** section.
6. Publish the customization project.

#### Related Links

- [To Add a Generic Inquiry to a Project](#)

## Step 2.5.4: Test the Substitute Form

Now you can test how the substitute form is used to access a particular serviced device record. Do the following:

1. On the main menu of Acumatica ERP, select **Phone Repair Shop**.
2. In the **Phone Repair Shop** workspace, which opens, click the *Serviced Devices* link.

The Serviced Devices (RS2020PL) substitute form opens, as shown in the following screenshot.

The screenshot displays the 'Serviced Devices' form interface. At the top, there are tabs for 'CUSTOMIZATION' and 'TOOLS'. Below the title bar is a toolbar with icons for refresh, undo, add, edit, split, and print. A filter configuration bar allows users to drag column headers to configure filters. The main area contains a table with the following data:

Device Code	Description	Active	Complexity
<a href="#">IPHONE6</a>	iPhone 6	<input checked="" type="checkbox"/>	High
<a href="#">MOTORRAZR</a>	Motorola RAZR V3	<input type="checkbox"/>	Low
<a href="#">NOKIA3310</a>	Nokia 3310	<input checked="" type="checkbox"/>	Low
<a href="#">SAMSUNGGS4</a>	Samsung Galaxy S4	<input checked="" type="checkbox"/>	Medium

At the bottom, a status bar indicates '1-4 of 4 records' and provides navigation controls.

**Figure: The Serviced Devices substitute form**

3. In the **Device Code** column, click the *MotorRAZR* link.

The Serviced Devices (RS202000) form opens displaying the device properties of the *MotorRAZR* record.

4. Select the **Active** check box.
5. On the form toolbar, click **Save and Close**.

The system returns you to the Serviced Devices (RS2020PL) substitute form.

6. Check the *MotorRAZR* record in the list, to verify that your change is reflected: The **Active** check box is now selected.

## Lesson Summary

In this lesson, you have learned about substitute forms and have created one for the Serviced Devices (RS202000) form.

To configure the substitute form, you have completed the following steps:

1. Loaded a predefined generic inquiry on the *Generic Inquiry* (SM208000) form.

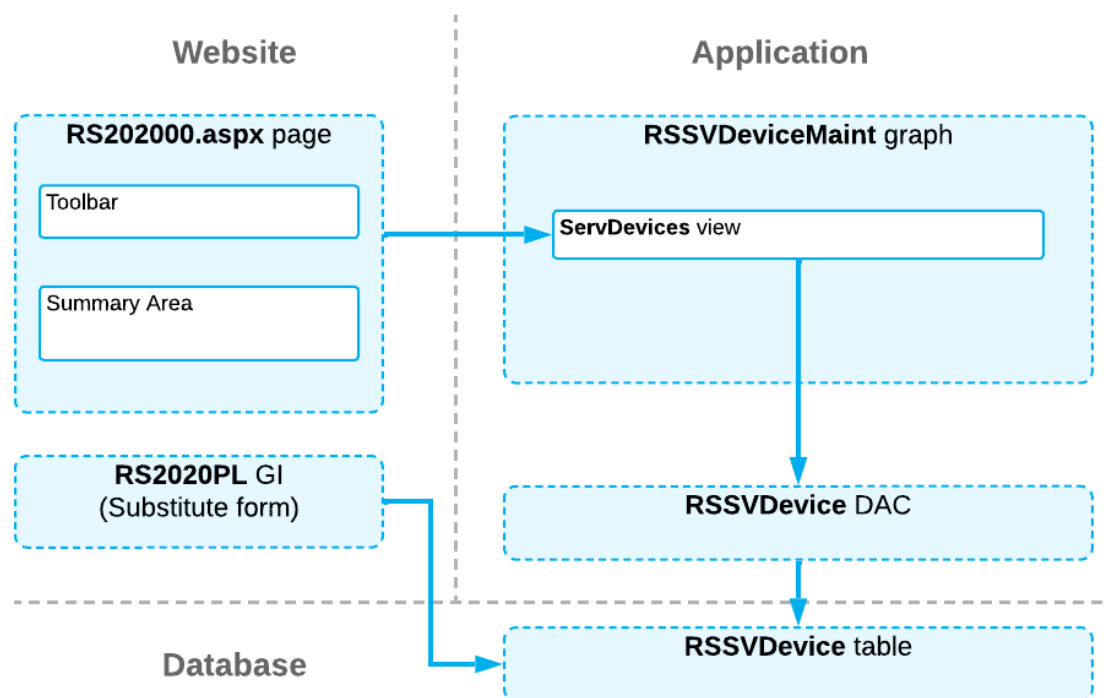
2. Configured the properties of the generic inquiry on the **Entry Point** tab of this form.
3. Saved the generic inquiry to the customization project.

## Part 2 Summary

---

While completing all of the lessons of Part 2, you learned how to create a new form by using Visual Studio and set up a substitute form for it. The components that were needed for the new Serviced Devices (RS202000) maintenance form are shown in the following diagram.

### Components of the Serviced Devices Form



The RS2020PL substitute form displays information from the RSSVDevices database table. The RS202000 ASPX page uses the *ServDevices* view as the data member and the RSSVDeviceMaint graph as a data source. The *ServDevices* data view of the graph uses the RSSVDevice DAC to select records from the RSSDevice database table.



# Appendix: Customization Projects

---

In this topic, you can find links to additional information related to customization projects.

## Types of Items in a Customization Project

When you customize an instance of Acumatica ERP by using the [Customization Project Editor](#), the platform stores all items of a customization project as records in the `CustObject` table of the database. Each record in this table contains the data of an item, including the XML code of the item in a specific field. When you add an item to the customization project, the platform adds the new record to the table, creates the XML code of the item, and stores the code within the `Content` field of the record.

For details about the types of items in customization projects and the ways the items are stored in the database, see [Types of Items in a Customization Project](#).

## Deployment of a Customization Result

Once you have finished a customization project, you can export the project as a deployment package that can then be imported and published as a customization project in the end-user systems.

For more information about deployment packages, see [Deployment of a Customization Result](#).

## Simultaneous Management of Multiple Customization Projects

With the Acumatica Customization Platform, you can simultaneously manage multiple customization projects by using the [Customization Projects](#) (SM204505) form. You can publish multiple customization projects to an Acumatica ERP instance at once.

For details about how the system works with multiple customization projects, see [Project Publication: General Information](#).

## Publication of Customization Projects in a Multitenant Site

A customization project is stored in the instance database. The data of each tenant that uses the same instance of Acumatica ERP is isolated from the data of other tenants in the database. However, the website files of the instance are shared by all tenants.

For more information about publication of a customization project on a multitenant site and the customization items that are applied to one or all tenants, see [Publication of Customization Projects in a Multitenant Site](#).

## Unpublishing of Customization Projects

When you unpublish all customization projects, the system reverses the changes introduced by the customization project as follows:

- The forms of Acumatica ERP return to their original layout.
- The `.cs` files of the project with customization code are removed from the website folder in the file system.
- The custom files of these projects are removed from the website folder on the file system.

For details about the changes that cannot be unpublished, see [Unpublishing Customization Projects](#).