Development

T200 Maintenance Forms

Training Guide

Acumatica
The Cloud ERP

Last Revision: 5/23/2020

Acumatica ERP 2019 R2

Contents

Copyright

11235 SE 6th Street, Suite 140
Bellevue, WA 98004

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: 2019 R2

Introduction

Acumatica Framework provides the application programming interface (API) and tools for developing cloud business applications. Acumatica Framework is a part of the Acumatica Cloud xRP Platform, which provides various ways to develop the following:

- Add-on applications that interact with Acumatica ERP through the web services API
- Applications embedded into Acumatica ERP through the built-in customization tools
- Completely new applications based purely on Acumatica Framework

The *T200 Maintenance Forms* course introduces to you the main concepts of Acumatica Framework and the customization of Acumatica ERP 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.

How to Use This Course

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

1. Complete (or be sure you meet) *Course Prerequisites*, perform *Initial Configuration*, and carefully read *Getting Started*.
2. Complete the lessons in both parts of the training guide.
3. In Partner University, take *T200 Certification Test: Maintenance Forms*.

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

What Is in a Part?

The first part of the course explains how to create a custom Acumatica ERP form by using 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 consists of steps that outline the procedures you are completing and describe the related concepts you are learning.

What Are the Documentation Resources?

All the links listed in the *Related Links* sections refer to the documentation available on the *https://help.acumatica.com/* website. These and other topics are also included in the Acumatica ERP instance, and you can find them under the **Help** menu.

Course Prerequisites

Before you begin completing lessons, you should make sure you have the needed knowledge and background, and that you have completed the configuration steps, as described in the following section.

Required Knowledge and Background

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

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

Initial Configuration

You need to perform the prerequisite actions described in this part before you start to complete the course.

Step 1: Preparing the Environment

Prepare the environment for the training course as follows:

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

   If you have already installed Acumatica ERP without debugger tools, you should remove Acumatica ERP and install it again with the **Install Debugger Tools** check box selected. For details, see *To Install the Acumatica ERP Tools*.

Step 2: Deploying the Needed Acumatica ERP Instance for the Training Course

Deploy an Acumatica ERP instance and configure it as follows:

1. Open the Acumatica ERP Configuration Wizard, and deploy a new application instance as follows:
   a. On the Database Configuration page, type the name of the database: `PhoneRepairShop`.
   b. On the Tenant Setup page, set up a tenant with the *I100* data inserted by specifying the following settings:
      - **Login Tenant Name**: `MyTenant`
      - **New**: Selected
      - **Insert Data**: *I100*
      - **Parent Tenant ID**: *1*
      - **Visible**: Selected
   c. On the **Instance Configuration** page, in the **Local Path of the Instance** box, select the folder that is outside of the `C:\Program Files (x86)` or `C:\Program Files` folder. We recommend that you store the website folder outside of these folders to avoid an issue with permission to work in these folders when you will perform customization of the website.

   The system creates a new Acumatica ERP instance, adds a new tenant, and loads the selected data to it.
2. Sign in to the new tenant by using the following credentials:
   - User name: `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 user name and then click **My Profile**. On the **General Info** tab of the User Profile (SM203010) form, which the system has opened, select *YOGIFON* in the **Default Branch** box; then click **Save** on the form toolbar.

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

Step 3: Creating the Database Tables

Add the `RSSVRepairService` and `RSSVDevice` tables to the instance database by executing the `T200_DatabaseTables.sql` script, which you can find in the course files.

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*.

Getting Started

In this part of the course, you will review the company story and requirements to the customization that will be performed in this training course. You will also get an overview of application programming with Acumatica Framework.

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 added these tables to the database when you completed the initial configuration, which is described in *Step 3: Creating the Database Tables*.

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`: Is 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 *Table and Column Naming Conventions*.
- `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.

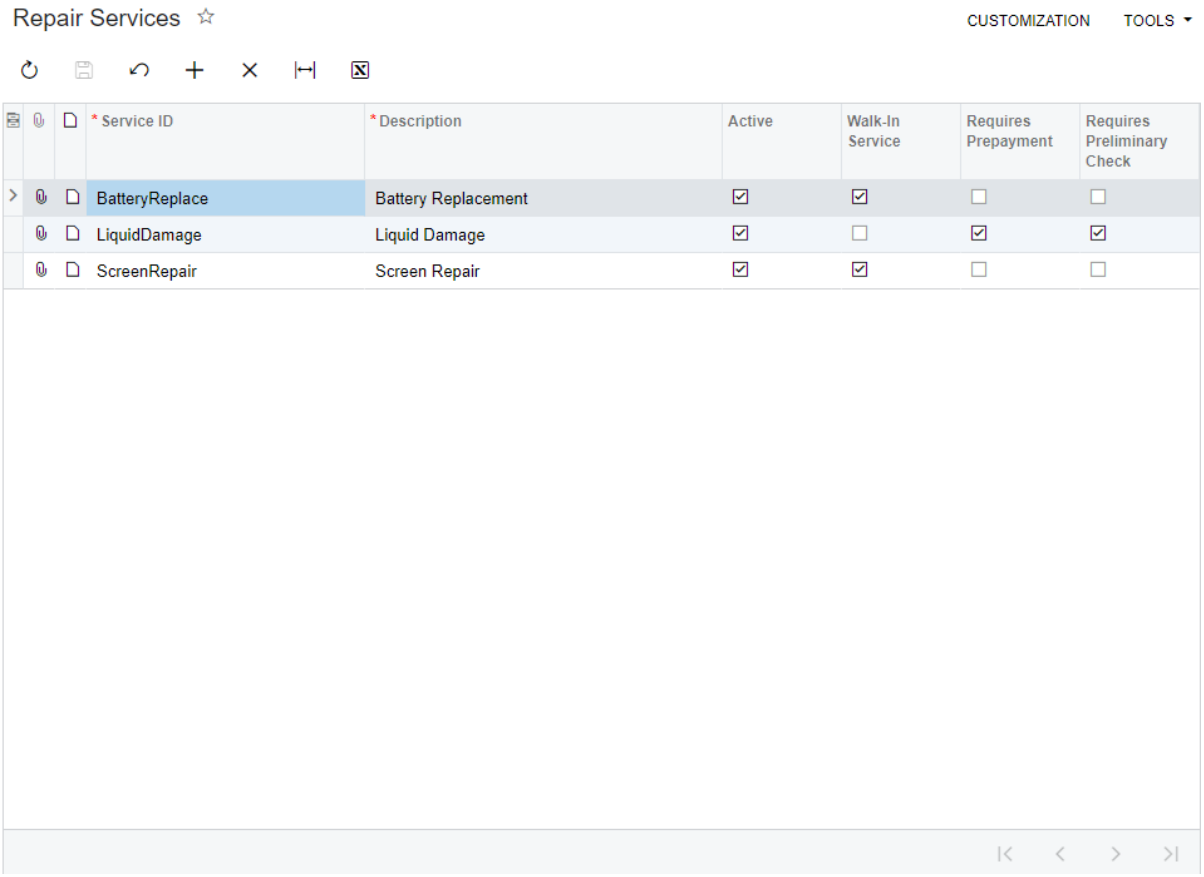


Figure: Service list on the Repair Services form

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.)

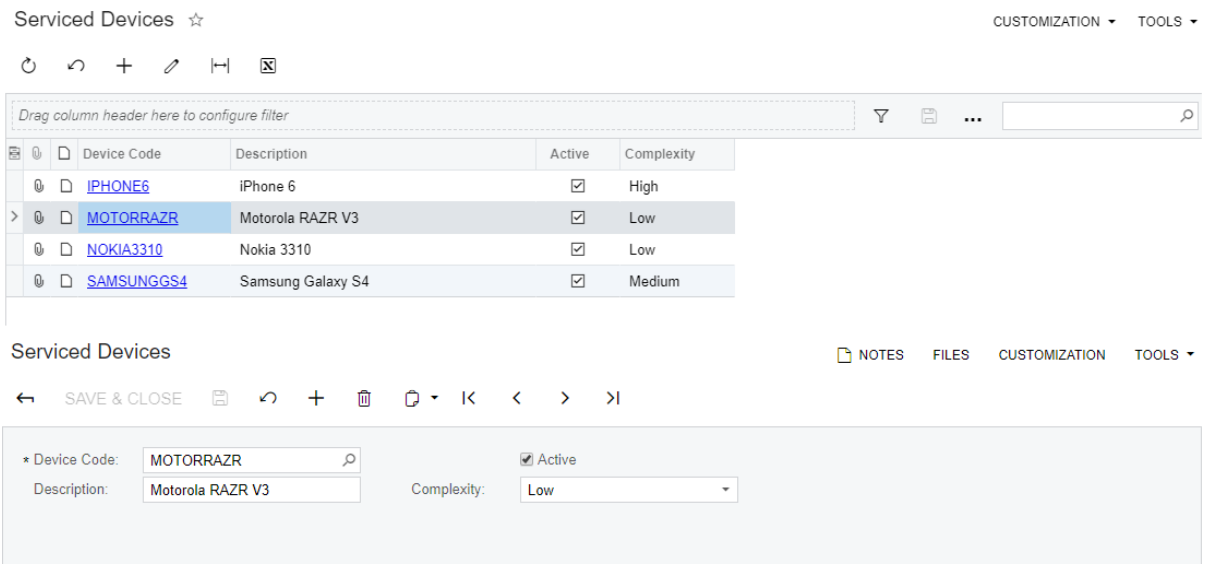


Figure: List and detail views of the Serviced Devices form

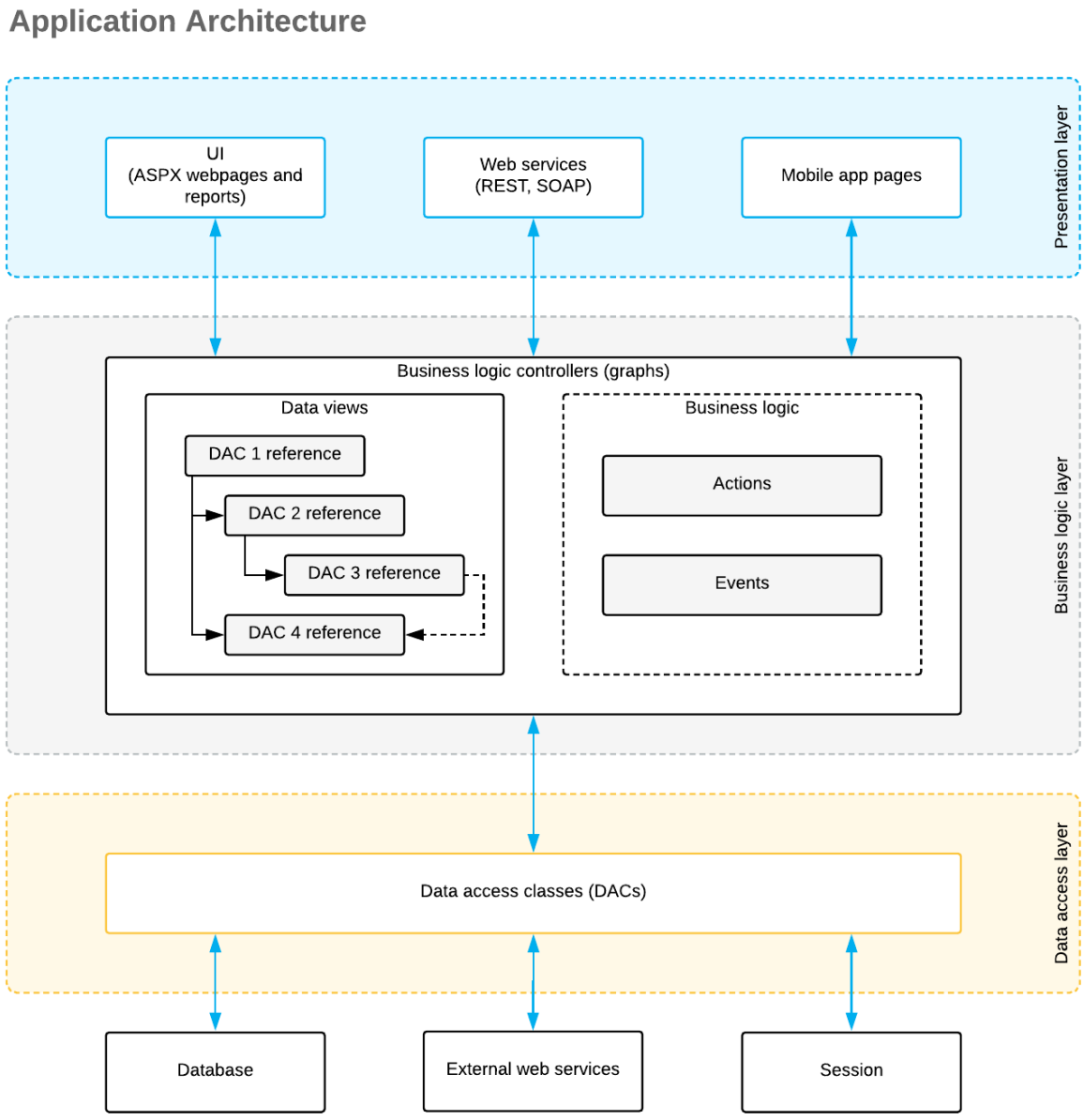
The Serviced Devices form will use the `RSSVDevice` table.

Application Programming Overview

Acumatica Framework provides the platform and tools for developing cloud business applications. This topic explains the runtime structure of Acumatica Framework, introduces the main components of this platform, and illustrates the relationships between these components by using simple examples.

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.



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 though 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 Webpages 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 on 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
```

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;

// Types used in BQL statements should derive from special interfaces:
// table  is derived from IBqlTable, and column is derived from IBqlField.
[PXCacheName("Product")]
public class Product : 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*) to reference a field in the BQL command

You will learn more about data access classes later in the training.

Related Links

Querying Data in Acumatica Framework

Part 1: Repair Services Maintenance 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 customers, inventory items included, and 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 a result of completing this lesson, you will do the following:

- Learn what customization project is
- Create a customization project
- Learn 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.
2. In the navigation pane, click **Database Scripts**. The Acumatica Customization Platform opens the Database Scripts page.
3. On the page toolbar, 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 platform opens, start typing *RSSVRepairService* in the **Table** box, and select the *RSSVRepairService - 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 learned about customization projects, created a customization project to be used for the training course, and added a database script to the 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 and a graph.

About the Repair Services Form

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

Lesson Objective

As you complete this lesson, you will create the first form of the application, which displays the list of repair services in a grid, 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 access 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 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.

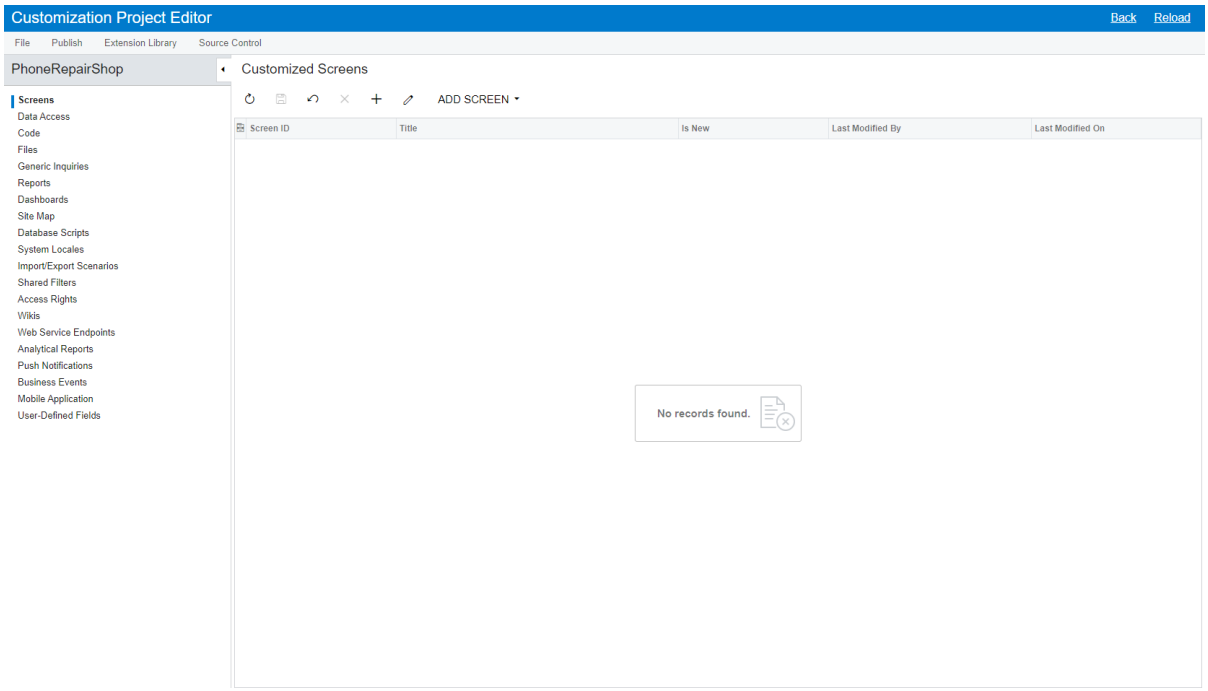


Figure: The Customized Screen page of the Customization Project Editor

3. On the page toolbar, click **Add Screen** > **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 module (for Acumatica ERP) or subject area (which in this case is phone repair), *20* indicates a maintenance type of page, and *10* is the first sequential number of the maintenance page in *RS*. 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 module name (in this case, *RS*) and a two-letter application module prefix (in this case, *SV* to indicate service). The suffix indicates the type of the webpage 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*.

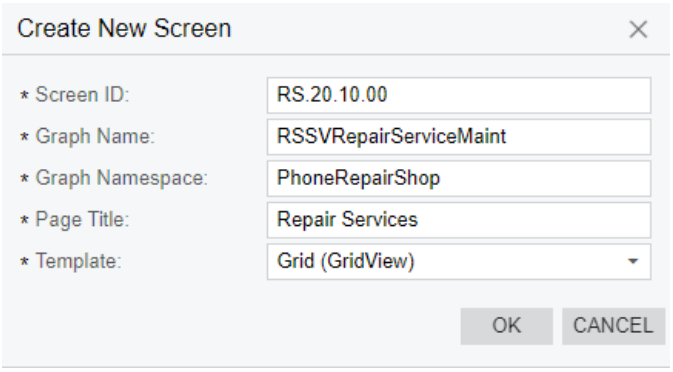


Figure: The Create New Screen dialog box

5. 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
RS201000	This *Screen* item contains the new page content.
RSSVRepairServiceMaint	This *Code* 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 `RSSVRepairServiceMaint.cs` file in the `App_RuntimeCode` folder of the Acumatica ERP application instance.
Pages\RS\RS201000.aspx **Pages\RS\RS201000.aspx.cs**	These *File* items contain ASPX page code for the new form. When you publish the project for the first time, the platform creates the files in the `Pages\RS` folder of the Acumatica ERP application instance, and the platform creates copies of these files in the `pages_RS` subfolder of the `CstPublished` folder of the instance.
Repair Services	This *SiteMapNode* item contains the site map object of the new form.

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

Related Links

To Create a Custom Form Template
To Add a New Custom Form to a Project

Graph

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

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

    [Serializable]
    public class MasterTable : IBqlTable
    {
    }

    [Serializable]
    public class DetailsTable : 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 by using the `PXSave` attribute. For details, see *PXSaveButtonAttribute Class*.
- The `Cancel` action by using the `PXCancel` attribute. For details, see *PXCancelButtonAttribute Class*.
- 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.

Related Links

PXGraph Class

Lesson Summary

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

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

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 pages 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 the New Workspace

Before adding a form to the Acumatica ERP UI, you need to know 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 new workspace, which will contain all forms related to the phone repair shop.

To create this new 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.

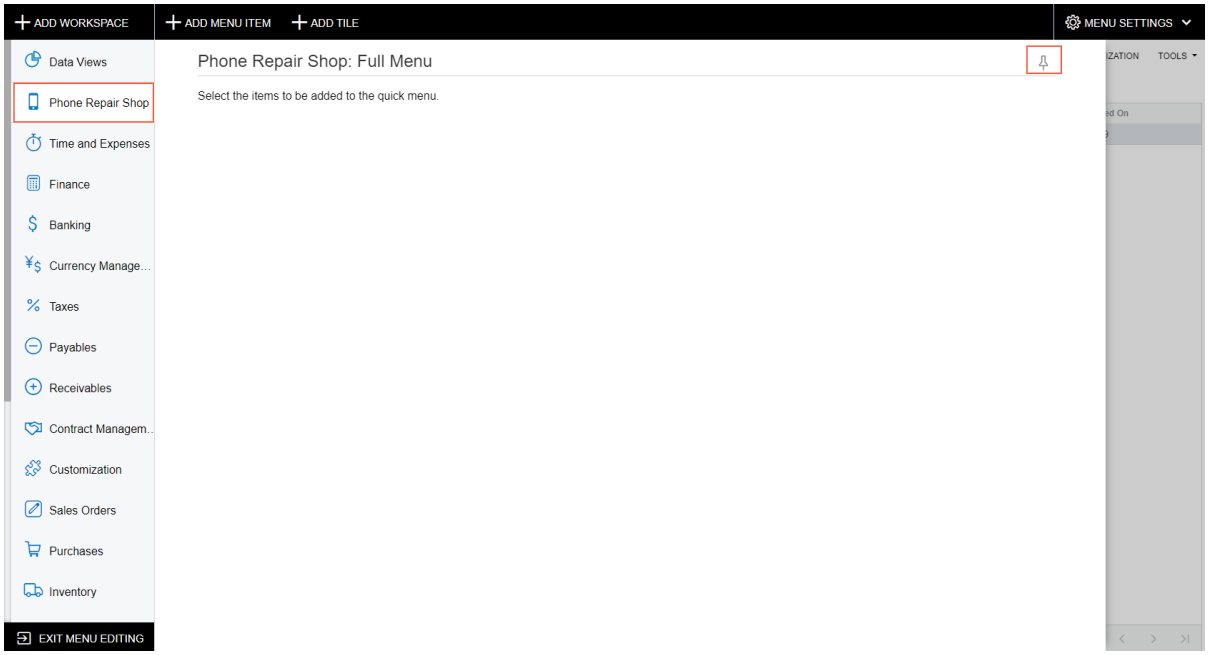


Figure: The Phone Repair Shop workspace in Menu Editing mode

Related Links

To Add and 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.

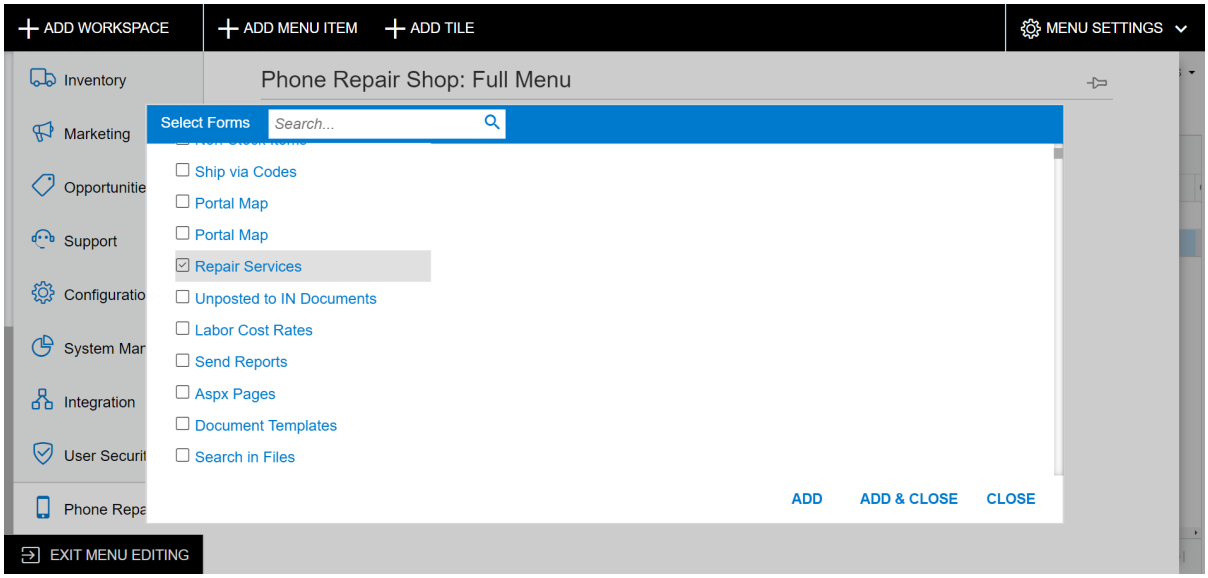


Figure: The Select Forms dialog box

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, as shown in the following screenshot.

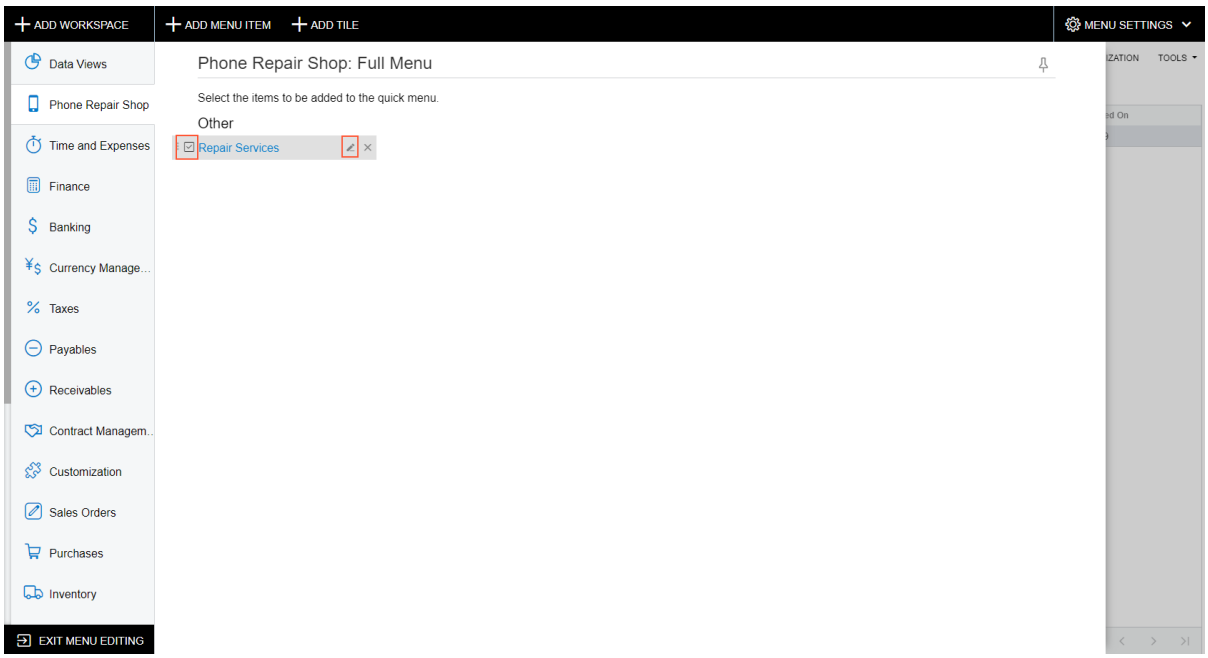


Figure: The Edit button

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 Mode** to save your changes and exit this 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.

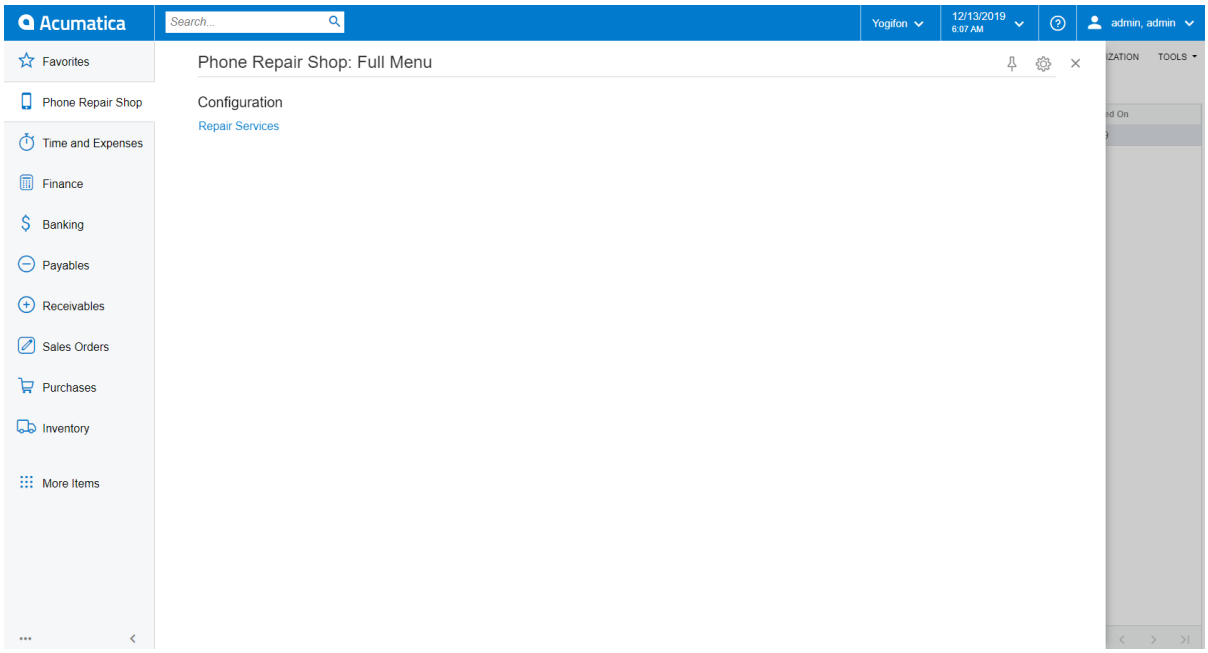


Figure: The Phone Repair Shop workspace

Related Links

To Add a Link to a Workspace

Step 1.3.3: Update the SiteMapNode Item

When you configured the new form's location in Acumatica ERP in Steps 1.3.1 and 1.3.2, 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.

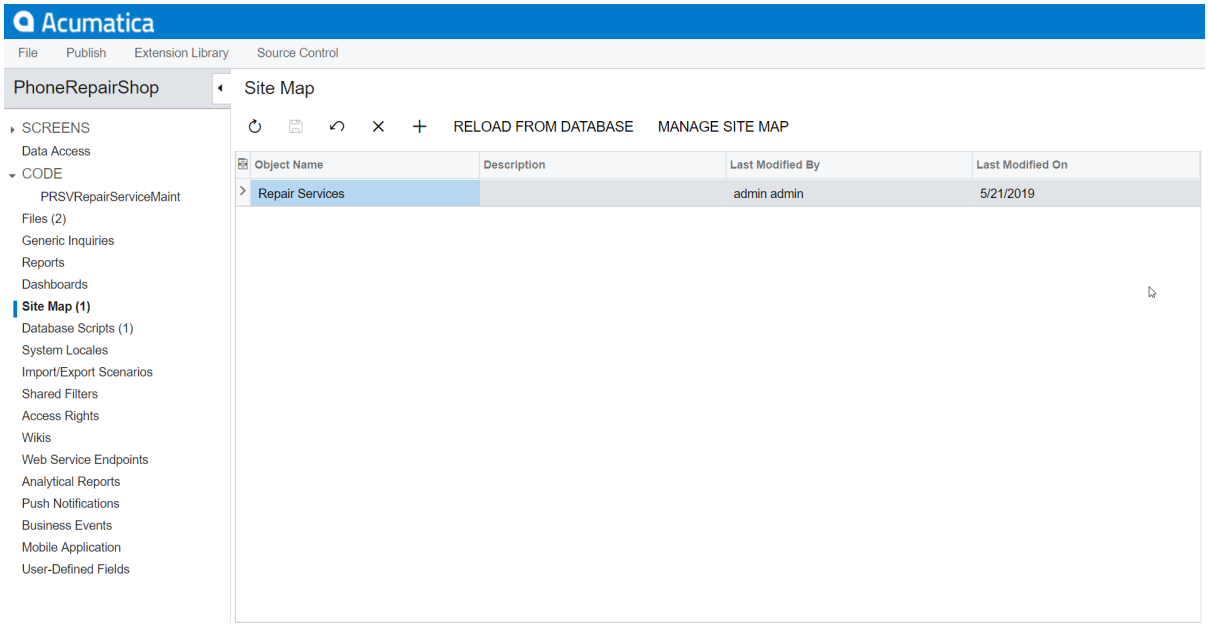


Figure: The Site Map page

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

Related Links

To Update a Site Map Node in a Project

Lesson Summary

In this lesson, you 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 page. You need this class to access data from the database.

Lesson Objectives

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

- Learn about the purpose and structure of DACs
- Generate and configure the `RSSVRepairService` DAC
- Configure a view in the `RSSVRepairServiceMaint` graph

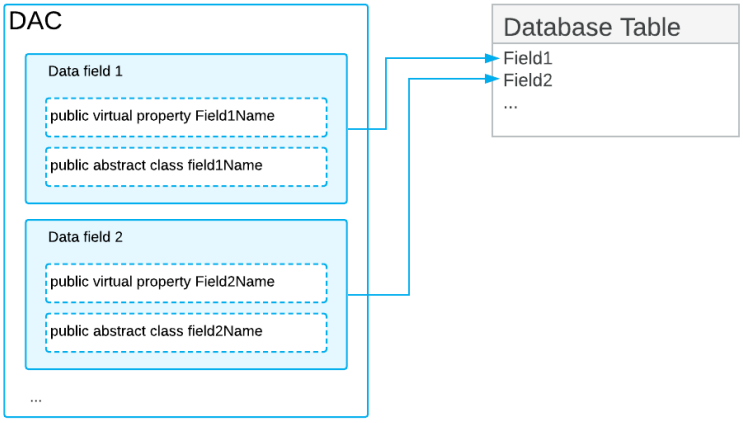
Definition of Data Access Classes

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 that represents the data field in BQL statements, such as `companyType`.
- A public virtual property that holds the field value, such as `CompanyType`. The property should have a nullable type.

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

The connection between DAC fields and database fields



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` attributes hides the DAC from generic inquiries, reports, and web services API clients.

- The class must be declared as implementing the `PX.Data.IBqlTable` interface.
- Abstract classes of data fields must be defined as implementing interfaces of the `PX.Data.BQL` namespace.

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 Customization Project Editor.
2. 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*.
3. On the Code page toolbar, click **Add New Record**.
4. 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

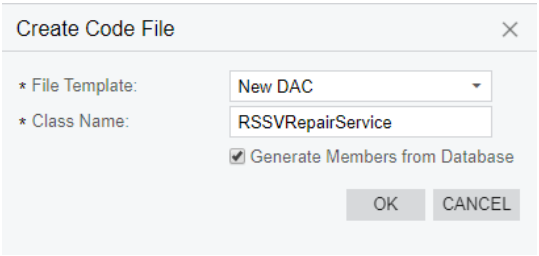


Figure: The Create Code File dialog box

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

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

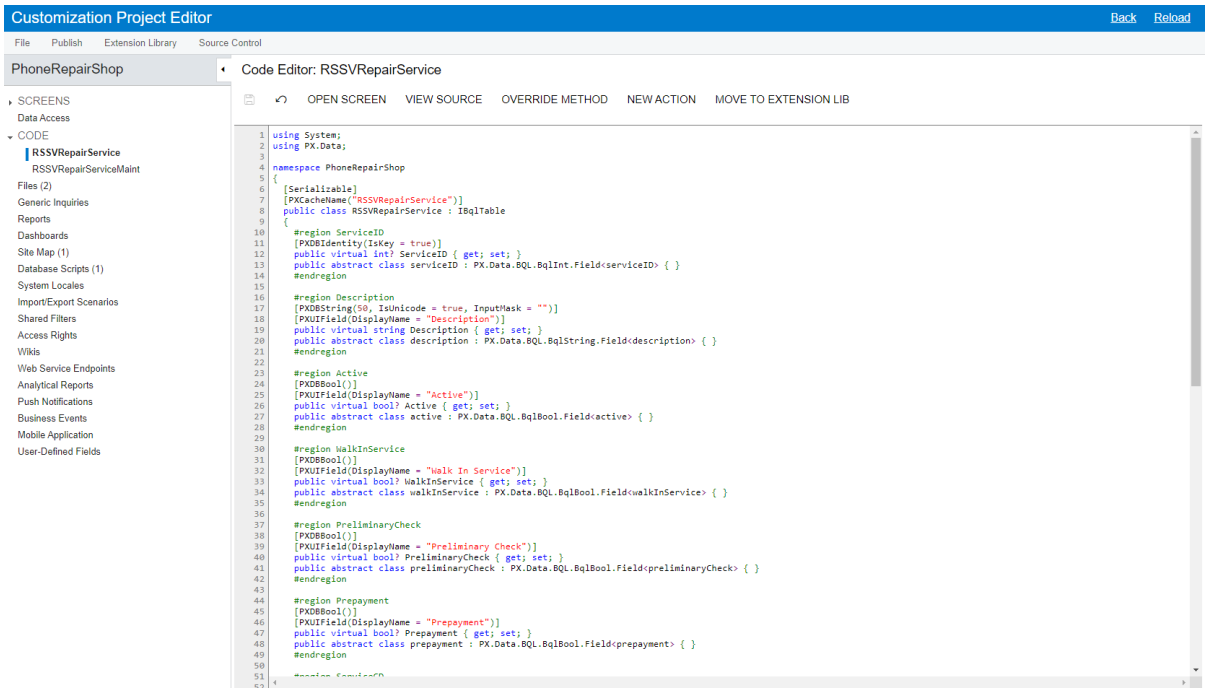


Figure: The RSSVRepairService DAC code

It is important to pay attention to the order in which fields are declared in a DAC: Every roundtrip 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

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. 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 (the text in bold shows which attributes should be placed):
   - For the `ServiceID` field, remove `IsKey=true` for the `PXDBIdentity` attribute as follows.

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

     The `PXDBIdentity` attribute is intended to identify the identity column in the database. For details, see *PXDBIdentity Attribute*.

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

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

  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 = ">aaaaaaaaaaaaaaa"` parameter to the `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. For details, see *IsKey Property*.

  The `PXDefault` attribute makes the field required. However, to add an asterisk symbol to the field, you need to add to specify `Required = true` on the `PXEUIField` attribute as follows: `PXUIField(..., Required= true)`.

  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*.

- For the `Description` field, add the `PXDefault` attribute as follows.

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

  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.

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

  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.

```
[PXDBBool()]
[PXDefault(false)]
[PXUIField(DisplayName = "Walk-In Service")]
public virtual bool? WalkInService { get; set; }
public abstract class walkInService :
 PX.Data.BQL.BqlBool.Field<walkInService> { }
```

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 `Display Name` parameter value of the `PXUIField` attribute, you change the label for the corresponding checkbox 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.

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

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

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

- For the system fields, make sure that only the following attributes are specified. For details, see *Audit Fields*.

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

- Remove the `PXUIField` attribute from the `Tstamp` field.

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 in the Code Editor.
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 using 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;
```

5. Save your changes.
6. Publish the customization project.
7. In the Screen Editor, specify `RepairService` as the data member for the Repair Services (RS201000) form by doing the following:
   a. In the **Screen** node on the navigation pane, select **RS201000**.

      The Repair Services screen opens in the Screen Editor.
   b. In the control tree, select 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.

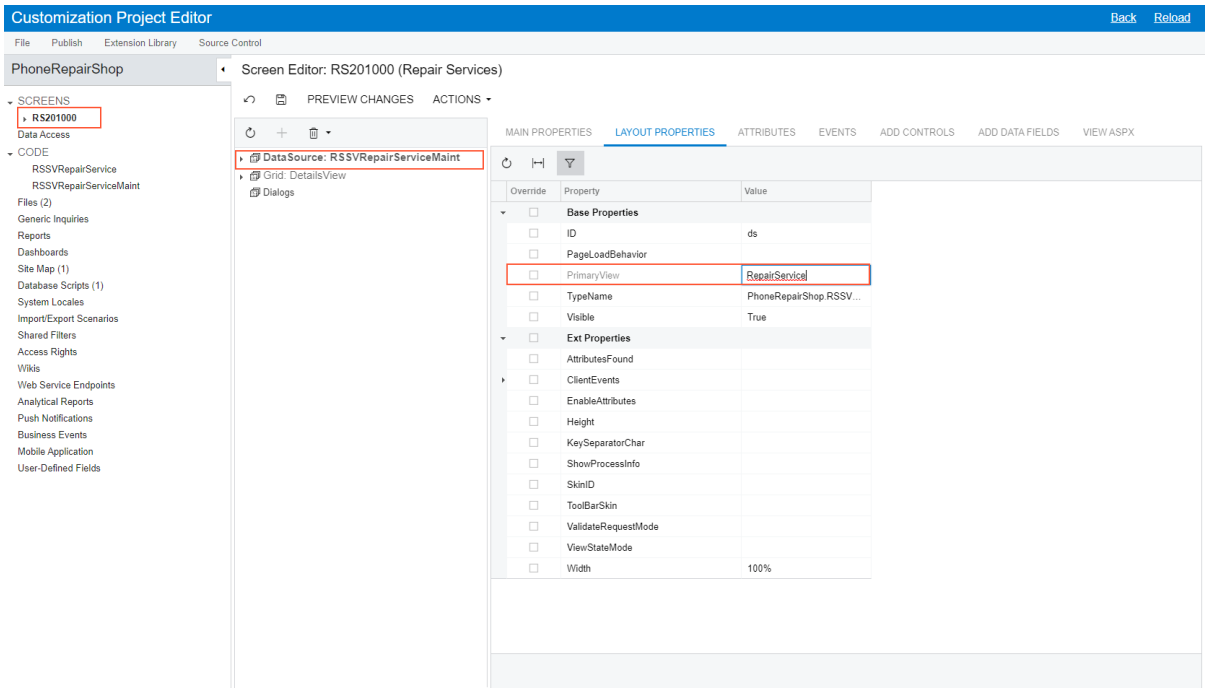


Figure: Specifying the PrimaryView property in the Screen Editor

d. Save your changes.

e. In the control tree, select 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.

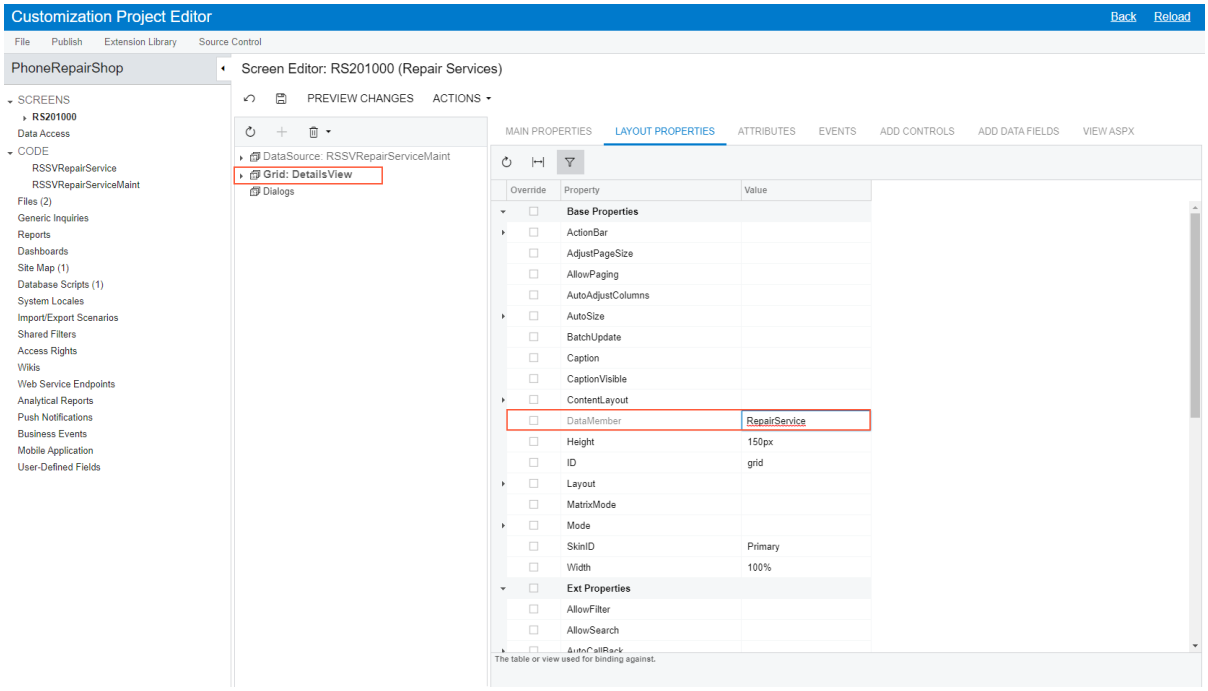


Figure: Specifying the DataMember property in Screen Editor

8. Save your changes.

9. Publish your customization project.

After you configured the form in Screen Editor, you can remove the definitions of the `MasterView`, `DetailsView`, `MasterTable`, and `DetailsTable` members from the `RSSVRepairServiceMaint` graph as 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 do the following:

- Add columns to a form grid
- Configure the appearance of the columns in the grid
- Test the configured Repair Services form

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.

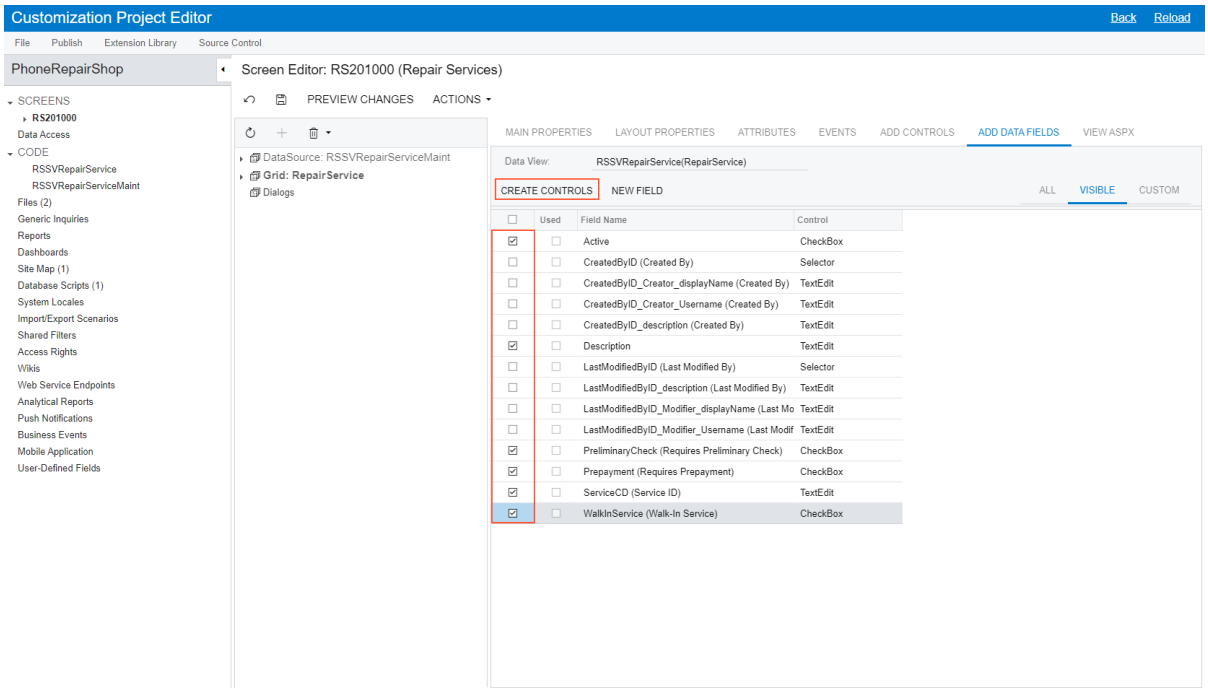


Figure: Columns to be added

You can generate all columns automatically by setting the **AutoGeneratedColumns** 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 in 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.

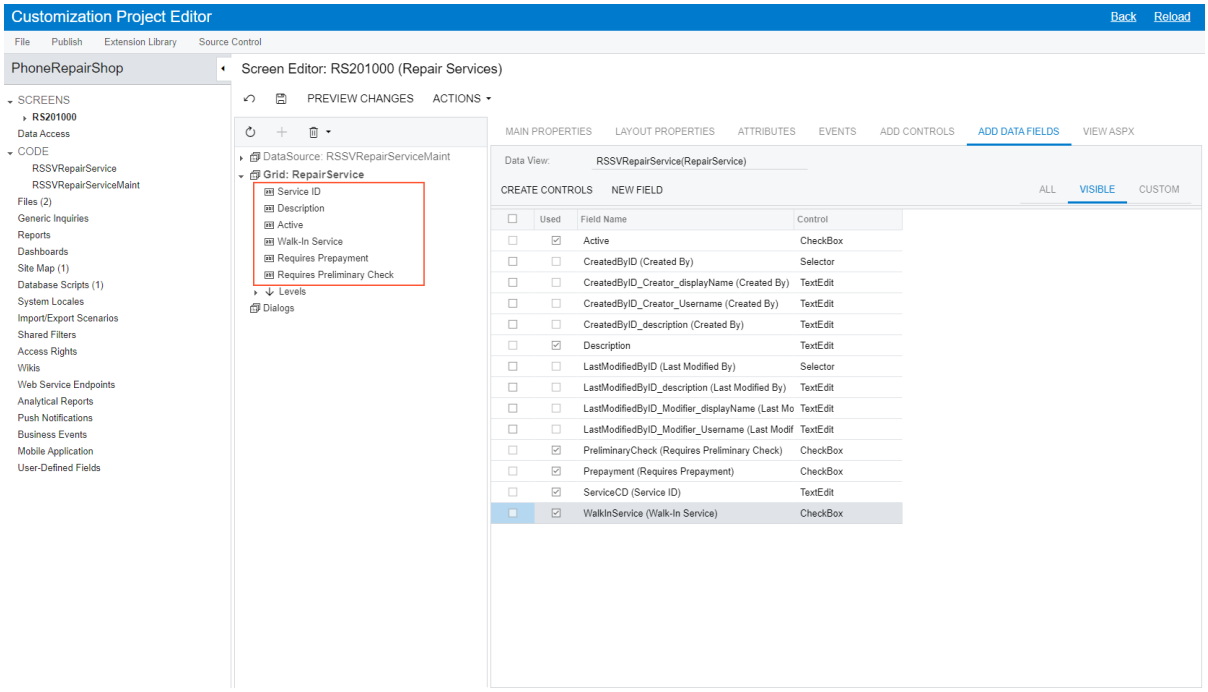


Figure: Control tree with new columns

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.

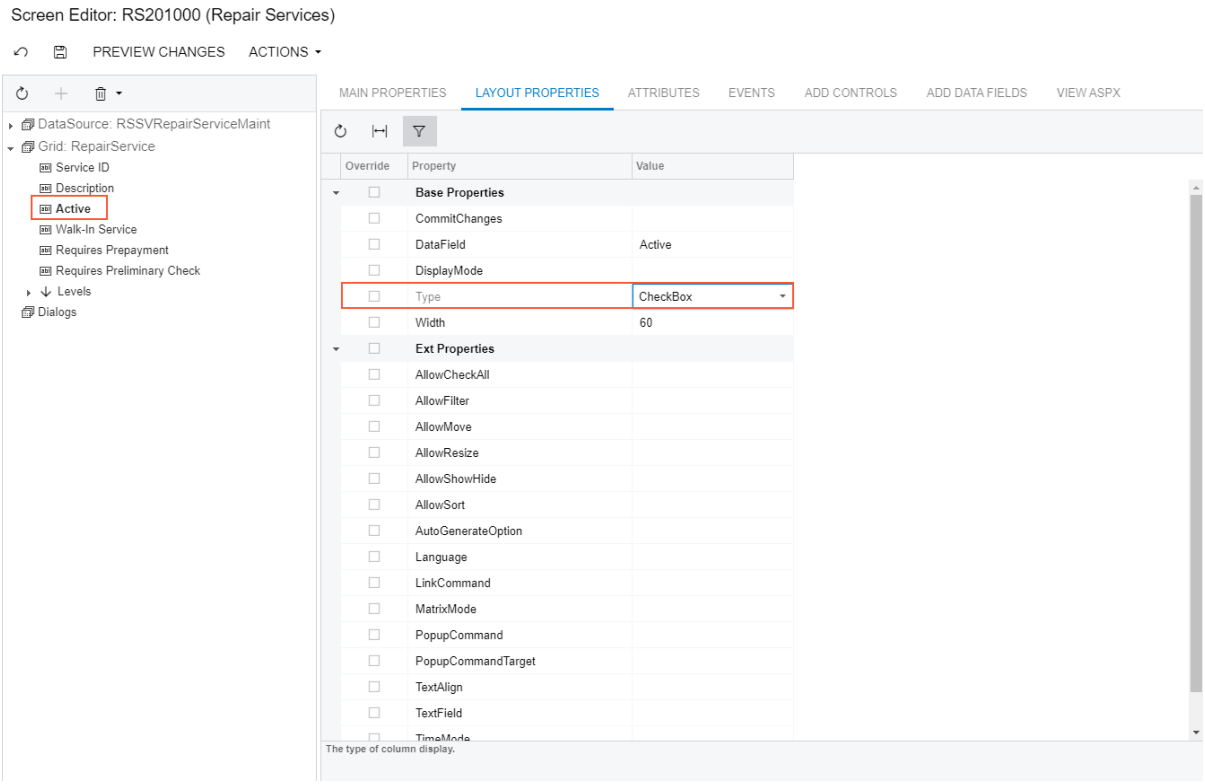


Figure: Property to define the Active column as a check box

c. Save your changes.

d. Repeat the three previous substeps for the **Walk-In Service**, **Requires Preliminary Check**, and **Requires Prepayment** columns.

8. In the control tree, select the **Grid: Repair Service** node.

9. In the **Base Properties** section of the **Layout Properties** tab, set the **AutoAdjustColumns** property to *True*.

10. Save your changes.

11. Make sure the elements on the form are displayed correctly. To do this, on the page toolbar, click **Preview Changes**.

The Repair Services form is opened in a new window as shown in the following screenshot.

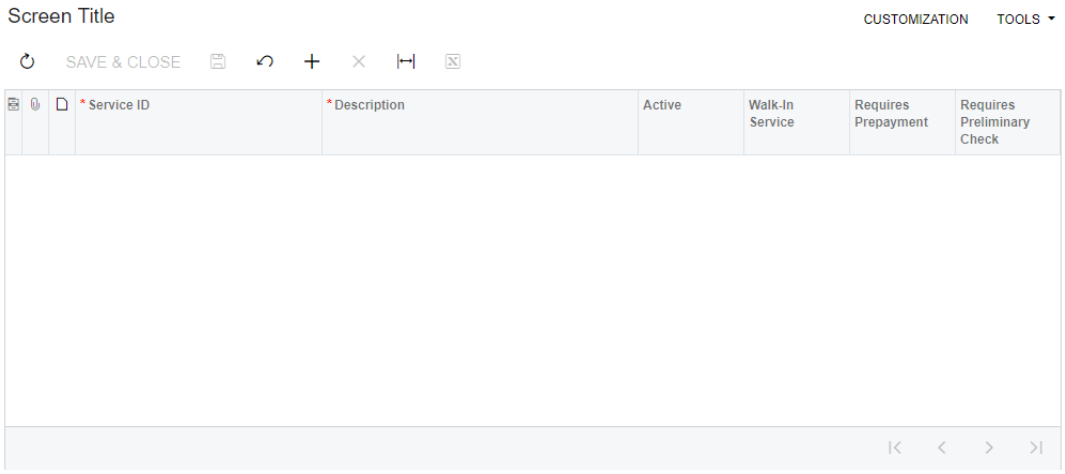


Figure: The Repair Services form in the preview mode

12. Publish the customization project.

Related Links

To Add a Column for a Data Field

Step 1.5.2: Test the Form

After you have configured the column on the Repair Services (RS201000) form, you should test it as follows:

1. In Acumatica ERP, open the Repair Services form. The form should look like the one shown in the following screenshot.

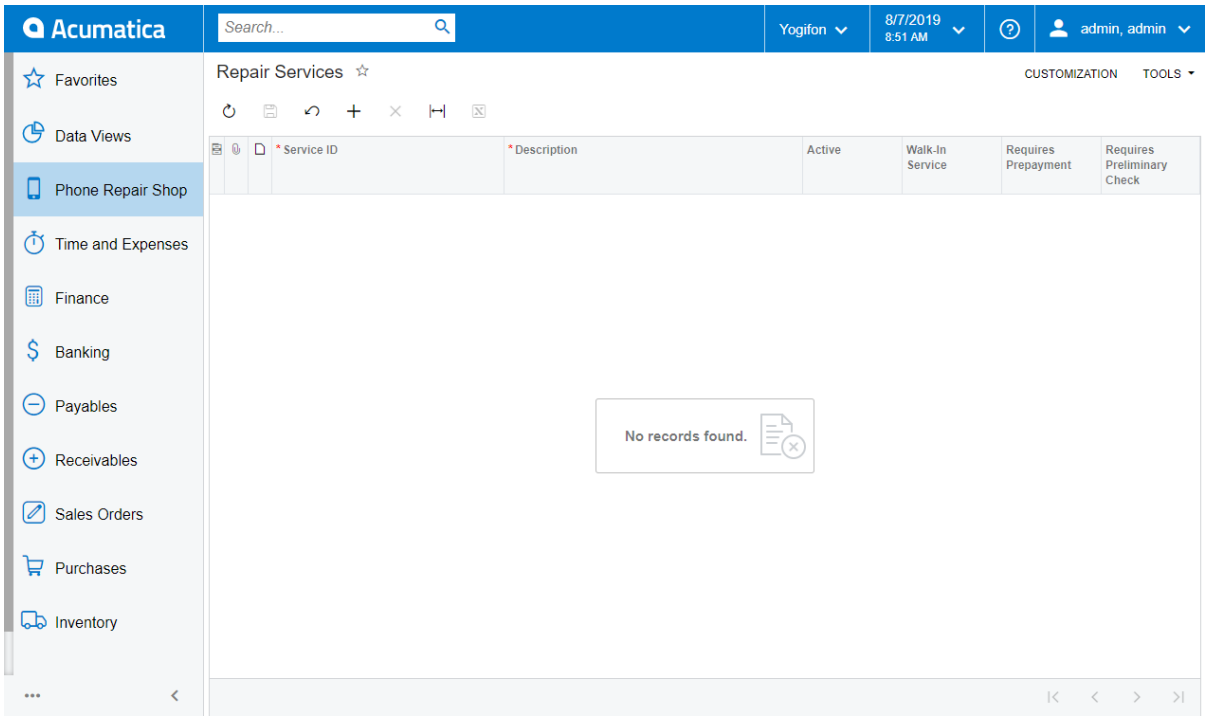


Figure: The Repair Services form

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.

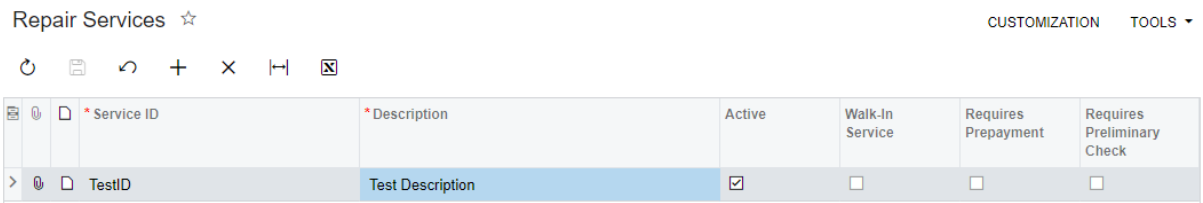


Figure: The new row on the Repair Services form

5. Add the following rows to the table.

Service ID	Description	Active	Walk-In Service	Requires Prepayment	Requires Preliminary Check
BatteryReplace	Battery Replacement	Selected	Selected	Cleared	Cleared
LiquidDamage	Liquid Damage	Selected	Cleared	Selected	Selected
ScreenRepair	Screen Repair	Selected	Selected	Cleared	Cleared

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

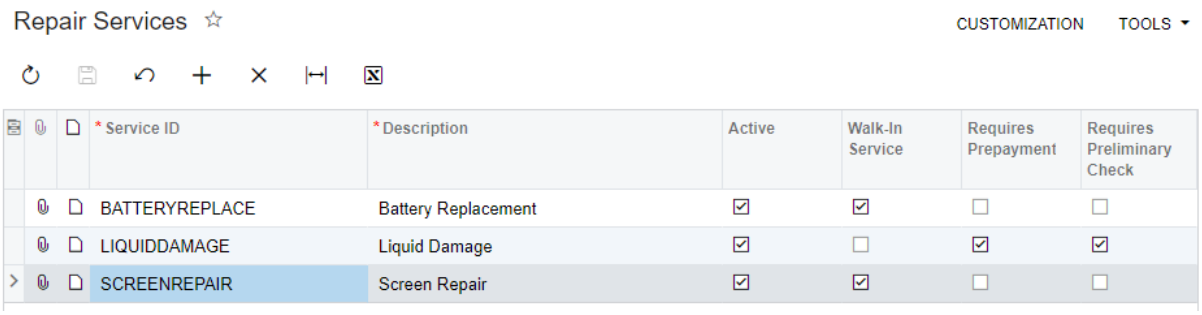


Figure: New rows on the Repair Services form

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 column of a grid. 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.

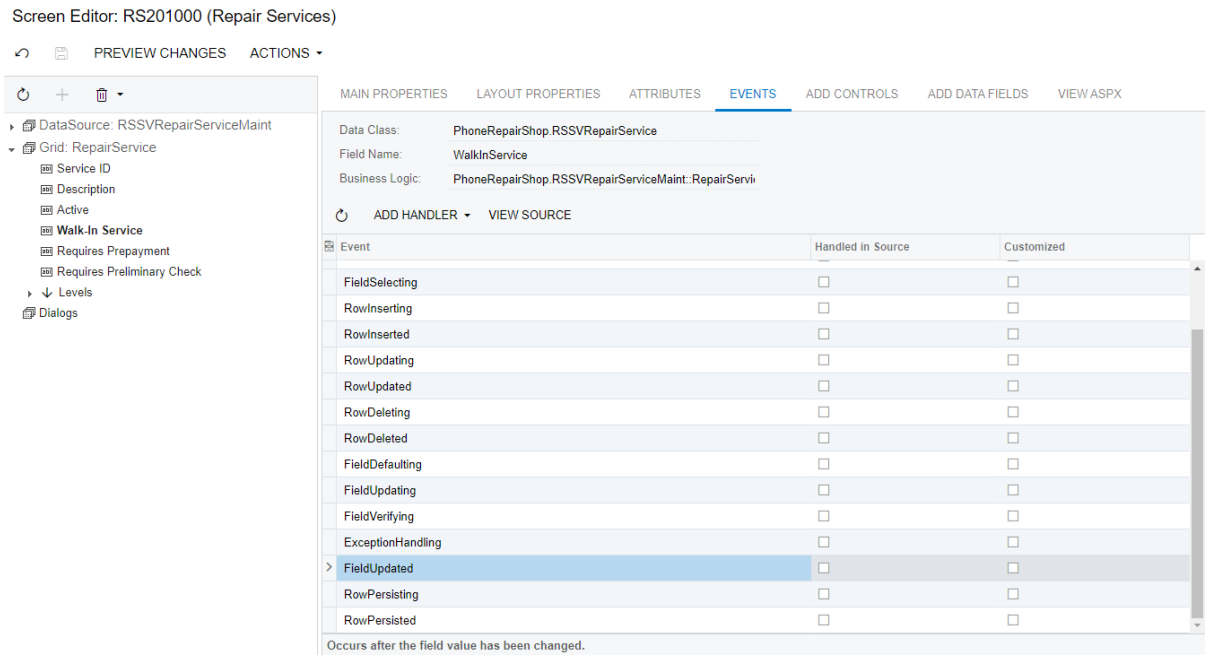


Figure: The FieldUpdated event

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*.

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.

```
if (row.WalkInService == true)
{
  row.PreliminaryCheck = false;
}
else
{
  row.PreliminaryCheck = true;
}
```

6. Save your changes.

7. Publish the customization project.

Related Links

Working with Events
To Add an Event Handler
PXFieldUpdated Delegate

Step 1.6.2: Configure the CommitChanges Property

To enable a callback for a column, you should configure 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 configure 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 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.

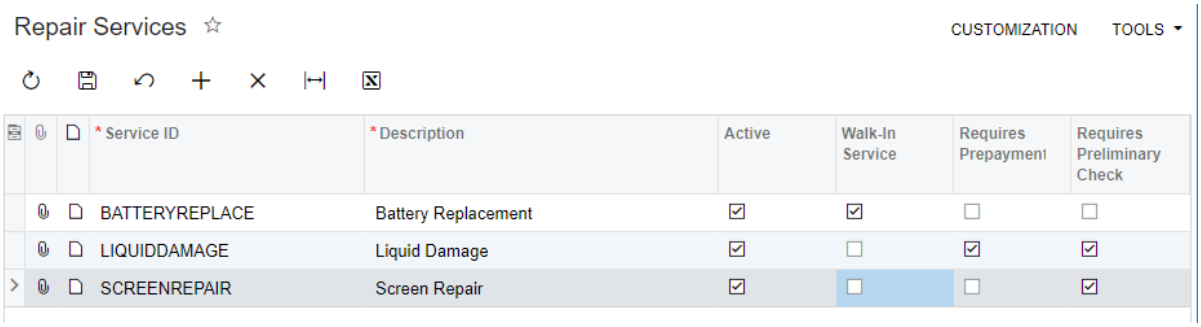


Figure: Clearing the Walk-In Service check box

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 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, configured 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.
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 the *PhoneRepairShop* 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 the text editor the `web.config` file that is located in the root folder of the *PhoneRepairShop* 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" ...>
```

4. Make sure the *PhoneRepairShop* 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, select the `PhoneRepairShop` 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.

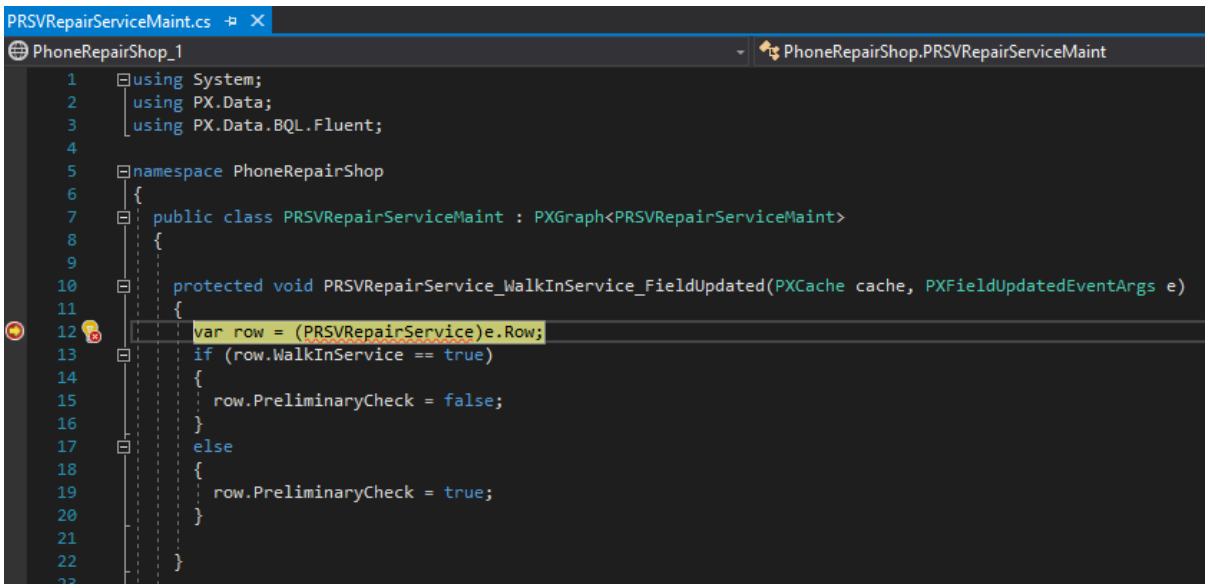


Figure: Breakpoint hit

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 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 do the following:

- Learn about extension libraries
- Learn how to use extension libraries as follows:
    - Create an extension library
    - Open the created extension library in Visual Studio
    - Build the created extension library in Visual Studio

About 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*.

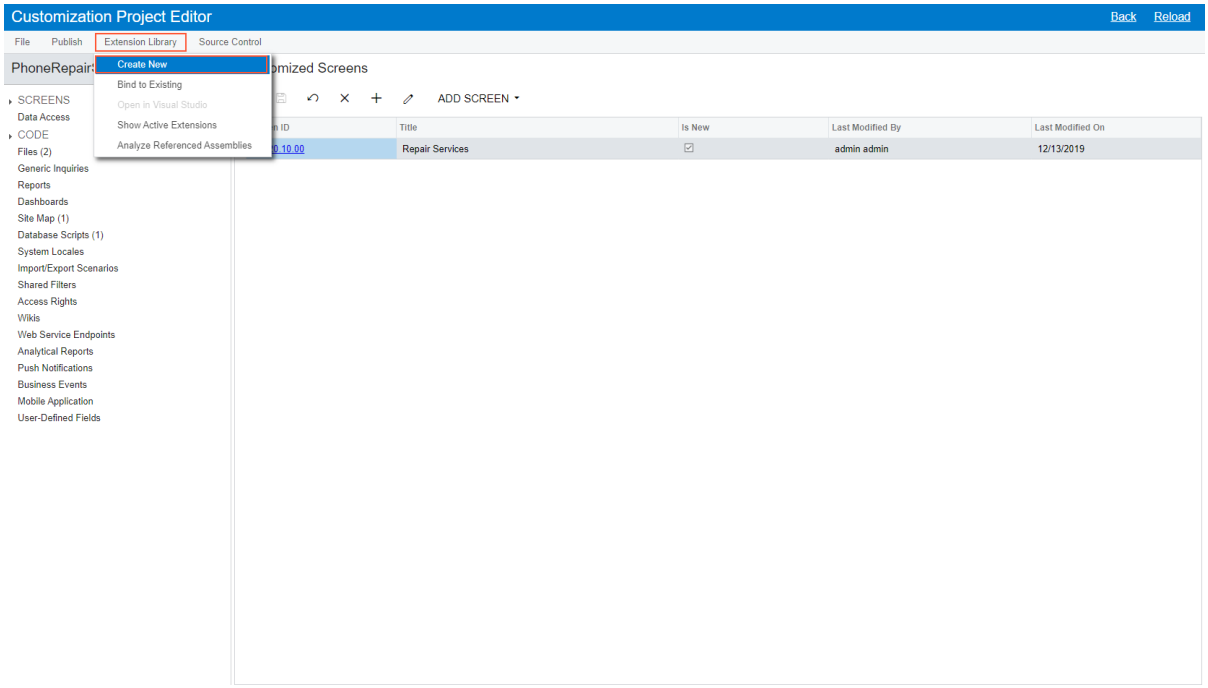
See *Extension Library (DLL) Versus Code in a Customization Project* in the Acumatica ERP Customization Guide for our recommendations about where you should keep your customization code.

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 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)` and `C:\Program Files` 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)` and `C:\Program Files` 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

File	Description
`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

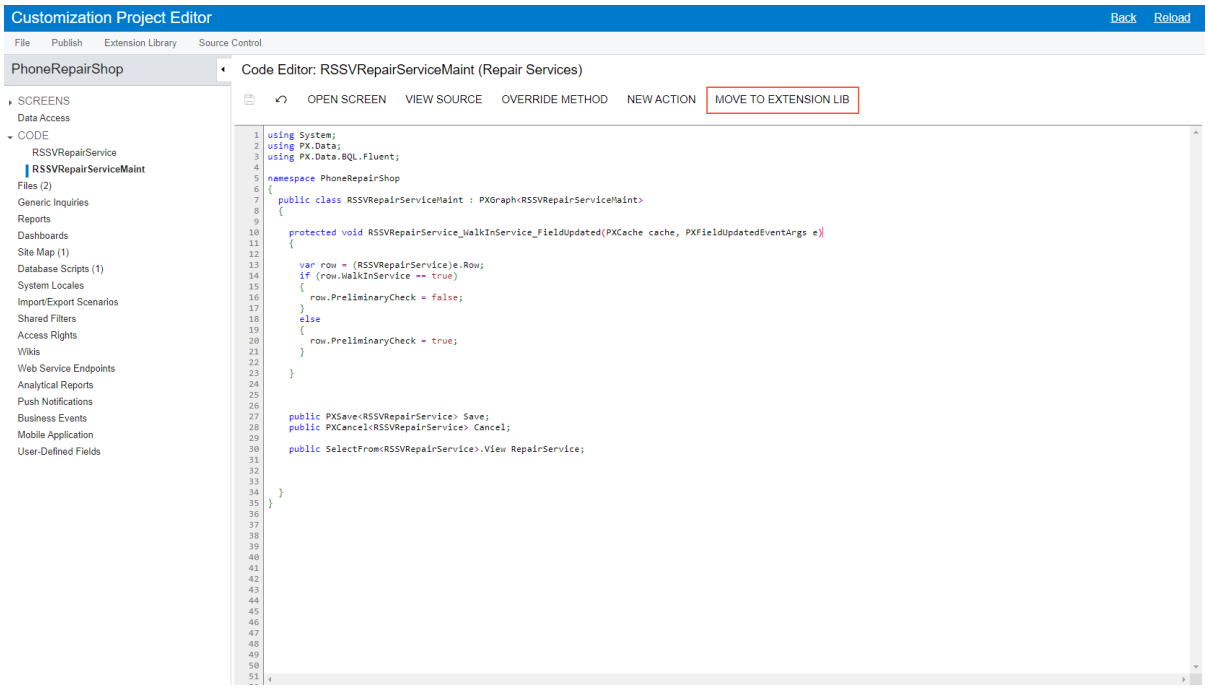
Step 1.8.2: Move Code from the Customization Project to the Extension Library

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 Extension Lib**, 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).

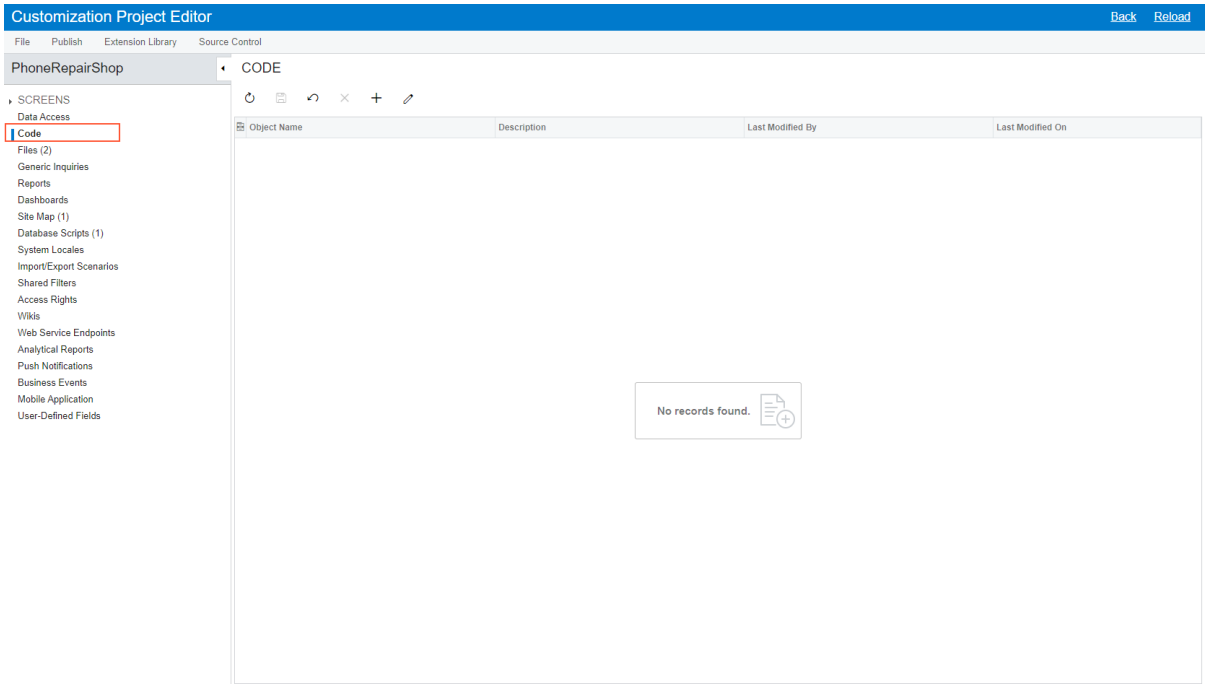


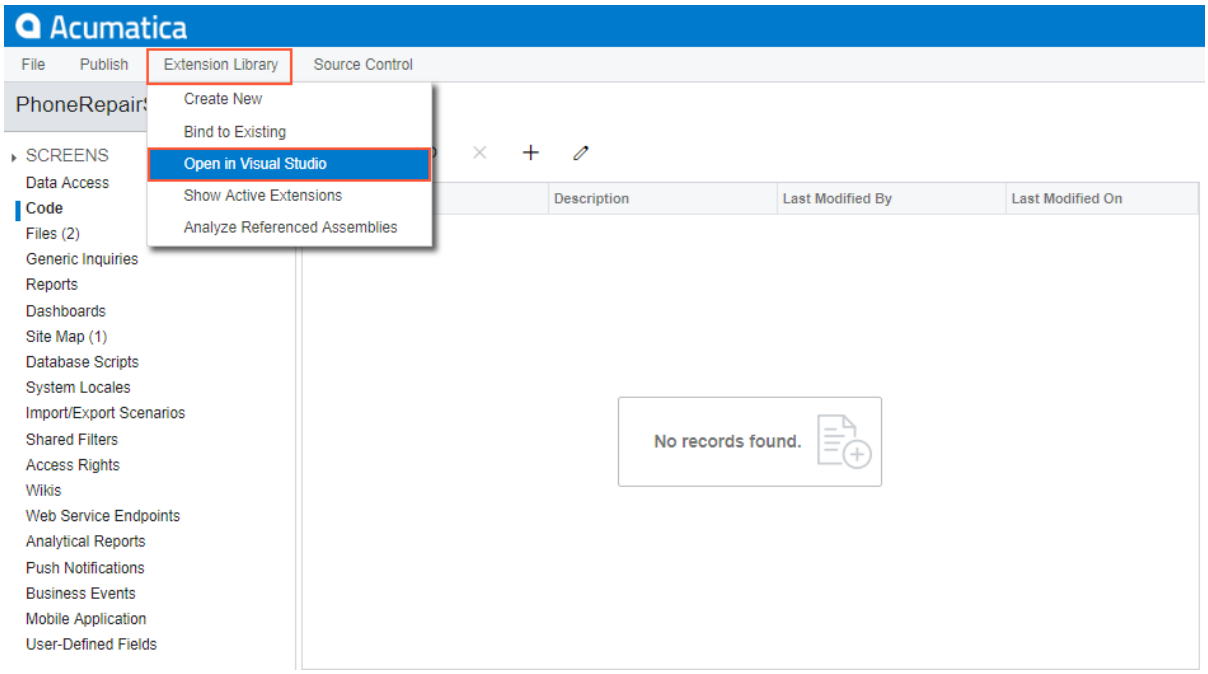
Figure: The empty list of the Code page

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 Customization Project Editor.
2. In the main menu of 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` code item 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 an assembly reference for the `PX.Data.BQL.Fluent.dll` file, which is located in the `Bin` folder of the PhoneRepairShop instance folder.
2. Build the `PhoneRepairShop_Code` project in Visual Studio.

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

Lesson Summary

In this lesson, you 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 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 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;
  if (row.PreliminaryCheck == true)
  {
    row.WalkInService = false;
  }
  else
  {
    row.WalkInService = true;
  }
}
```

4. Rebuild the project.
5. In the Customization Project Editor, open the RS201000 screen in the Screen Editor.
6. In the control tree, click **Grid: RepairService** > **Requires Preliminary Check**.
7. On the **Layout Properties** tab, set the **CommitChanges** property of **Requires Preliminary Check** field to *True*.
8. On the page toolbar, click **Save**.
9. 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.
10. Test the added handler as follows:
    a. In Acumatica ERP, open the Repair Services (RS201000) form.
    b. On the table toolbar, click add **New Row** to add a row for a new repair service.
    c. In the new row, select the **Walk-In Service** check box, and then select the **Requires Preliminary Check** check box. When you select the **Requires Preliminary Check** check box, the **Walk-In Service** check box should be cleared automatically.

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.**

   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*.

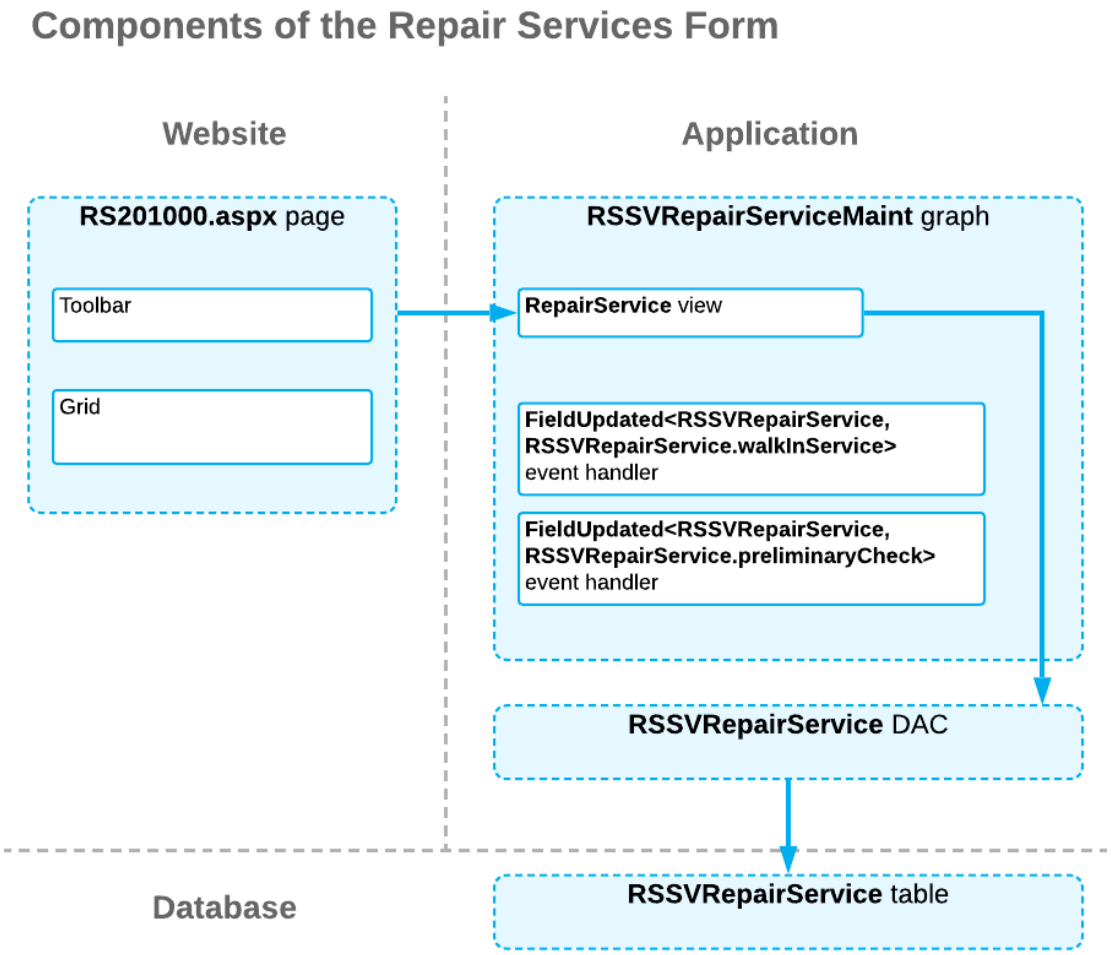
When you are testing the event handlers, make sure the check boxes are selected and cleared automatically in the needed ways for both the **Walk-In Service** and **Preliminary Check** check boxes.

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.

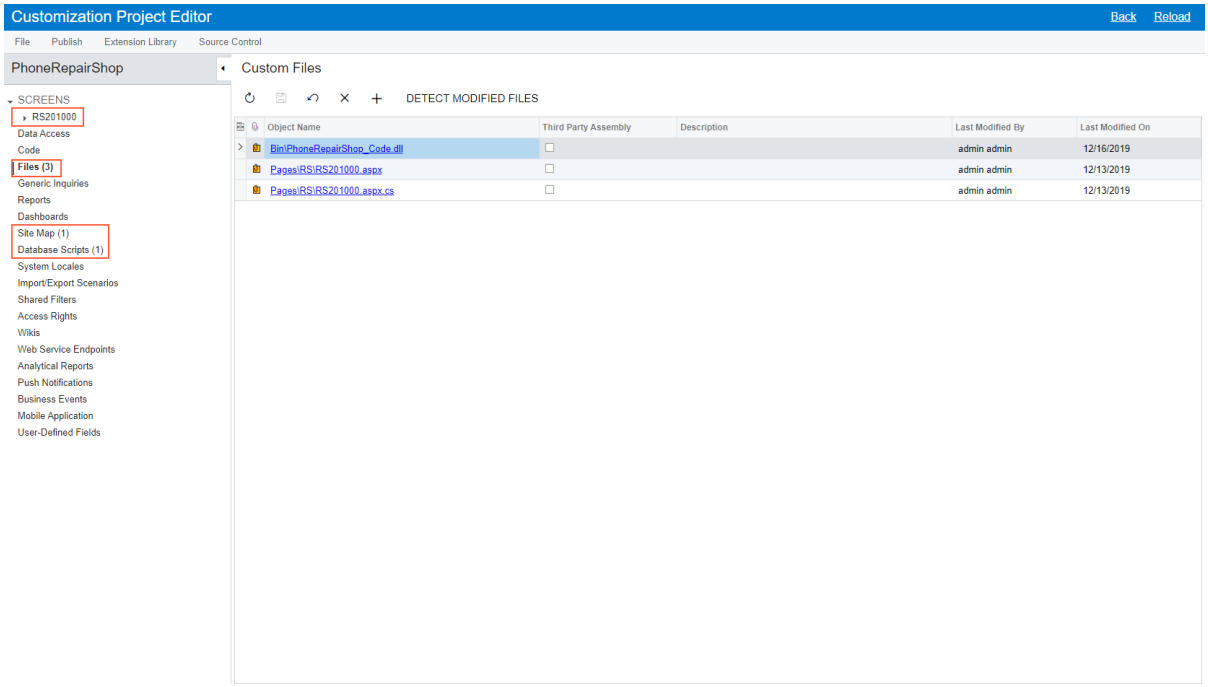


Figure: Customization project items

Review Questions

1. How do you save information about a custom database table to a customization project?
   a. Export a table from the database, and add it as a File item to the customization project.
   b. Add an SQL script for the table to the customization project.
   c. Add the table schema to the customization project.
2. Select all of the items that are created when you use the New Screen wizard to create a new form.
   a. A site map node
   b. An ASPX file and a corresponding ASPX.CS file
   c. A graph
   d. The `web.config` file
   e. A Screen item
   f. The `Solution.bat` file
3. Why do you need a data access class?
   a. To implement the business logic of the application
   b. To hold information about the items of the customization project
   c. To map the fields of a database table
4. What do you specify as a data member for a form?
   a. A view declared in a graph
   b. A DAC
   c. A site map node
5. What tool do you use to add columns to a form grid?
   a. Screen Editor
   b. Code Editor
   c. Layout Editor
6. How do you enable a callback?
   a. On the **Events** tab of the Screen Editor, select the `FieldUpdated` event.
   b. Set the `Active` property of the callback to *true*.
   c. Set the `CommitChanges` property of the field to *true*.
7. Which customization code can you debug by using Visual Studio? Select all correct responses.
   a. Code located in the `App_RuntimeCode` folder
   b. Code created in Visual Studio
   c. Code created by using the Code Editor of the Customization Project Editor
8. What tool can you use to refactor customization code?
   a. Element Inspector
   b. Acuminator

c. Screen Editor

Answer Key

1. C

2. A, B, C, E

3. C

4. A

5. A

6. C

7. A, B, C

8. B

Part 2: Serviced Devices Maintenance 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 a database script 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 and DAC in Visual Studio
- Configure a selector 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** > **New Item**.
3. In the **Add New Item** dialog box, select the *Class* template provided by Visual Studio.
4. Type the name of the class being created, `RSSVDeviceMaint.cs`, and click **Add**.
5. Add the following directives to the `RSSVDeviceMaint.cs` file.

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

6. 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` class.

7. Save your changes, and rebuild the project.

Step 2.1.2: Create a DAC in Visual Studio

To create a data access class (DAC) in Visual Studio, do the following:

1. Open the `PhoneRepairShop_Code` project in Visual Studio.
2. In Solution Explorer, add a new item named `RSSVDevice.cs` based on the *Class* template in the `DAC` folder of the project.
3. Add the following `using` directive.

```
using PX.Data;
```

4. Replace the generated code of the `RSSVDevice` class with the following code.

```
namespace PhoneRepairShop
{
  [PXCacheName("Serviced Device")]
  public class RSSVDevice : IBqlTable
  {
  }
}
```

5. In the project, add the `Helper` folder.
6. In the `Helper` folder, add the following files: `Messages.cs` and `Constants.cs`.
7. 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";
    }
}
```

8. 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.

```
namespace PhoneRepairShop
{
    public static class Messages
    {
        //Complexity of repair
        public const string High = "High";
        public const string Medium = "Medium";
        public const string Low = "Low";
    }
}
```

9. Add definitions of the following fields to the `RSSVDevice` class:
   - `Device ID`

     This is an identity field so you add the `PXDBIdentity` attribute as shown in the following code.

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

```
#endregion
```

- `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 =
 ">aaaaaaaaaaaaaaa")]
[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 do the following:

  - Bind the `DeviceCD` field to the string column in the database by using the `PXDBString` attribute
  - Make the field mandatory for the input by using the `PXDefault` attribute without parameters
  - Configure the name of the corresponding UI control by using the `PXUIField` attibute
  - Configure the lookup box (selector) 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 box. When a user selects a data record in the control, the control 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 configure 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 configure 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
```

- `Avg. Complexity of Repair`

  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 do the following:

  - Bind 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
  - Configure the name of the UI control by using the `PXUIField` attribute
  - Configure 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 (as the `AvgComplexityOfRepair` field), we recommend using a non-Unicode string field with fixed length of 1 symbol. You configure such 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. For details on system columns, see *Audit Fields*, *Concurrent Update Control*, and *Attachment of Additional Objects to Data Records* in the documentation.

10. Save your changes.

Related Links

Working with Attributes

Step 2.1.3: Configure the RSSVDeviceMaint Graph

Now that you have defined the `RSSVDevice` data access class, you can configure 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 `PhoneRepairShop` project, which is the website project.
3. Add a new item named `RS202000.aspx` based on the *Web Form* template in the **PhoneRepairShop** > **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 attribute values highlighted with bold.

```
<%@ 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 2.1.3: 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**.

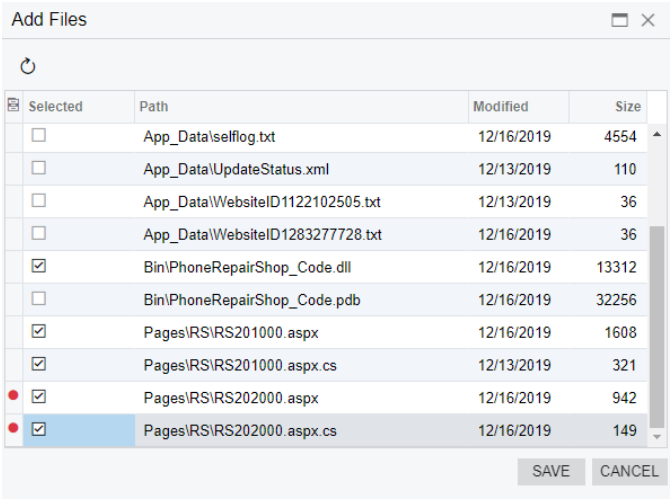


Figure: The Add Files dialog box

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 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 `Avg. Complexity of Repair` field

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 `Avg. Complexity of Repair` field:

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

   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.

   For details, see *Configuring ASPX Webpages and Reports*.
3. Save your changes.

Related Links

Customizing Elements of the User Interface

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.

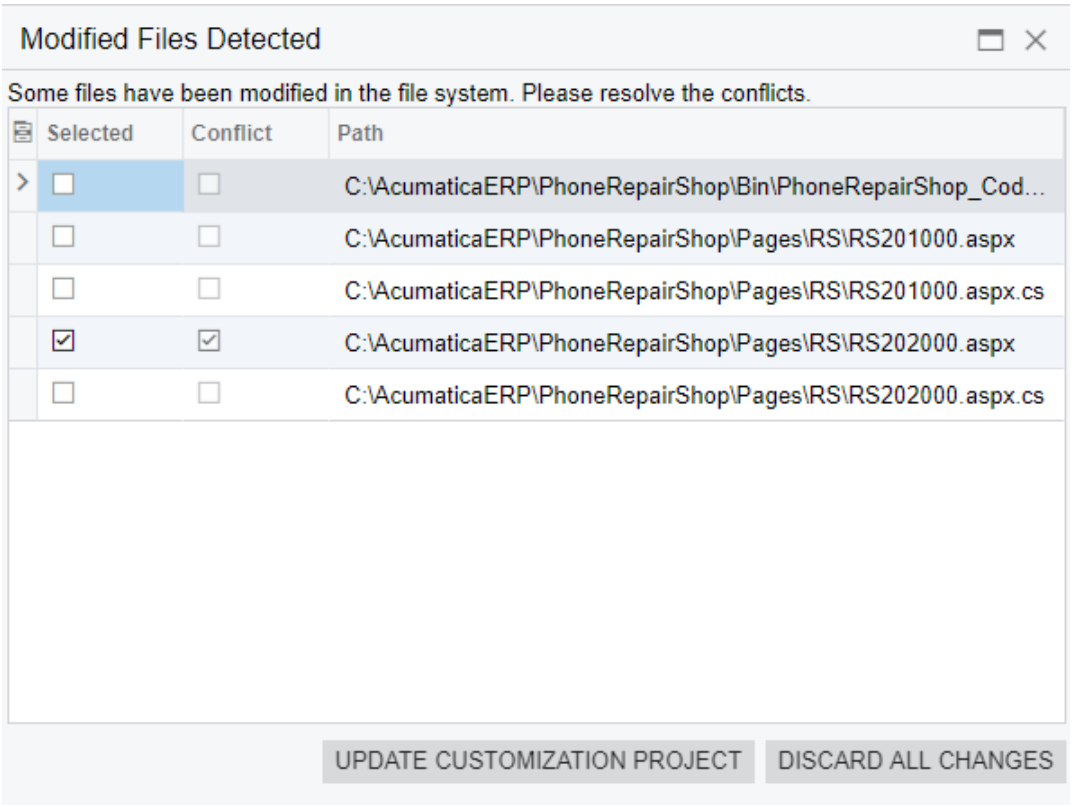


Figure: The Modified Files Detected dialog box

4. In the dialog box, make sure check boxes in the row of the `RS202000.aspx` file is selected, and click **Update Customization Project**.
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 configured input controls on the Serviced Devices (RS202000) form and organized the layout of the form.

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.

Lesson Objectives

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

- Create a site map item for a custom 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 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`
   - **Workspaces**: *Phone Repair Shop*
   - **Category**: *Configuration*
6. Save your changes.
7. Open the **Phone Repair Shop** workspace. Notice that the Serviced Devices (RS202000) form is available in the quick menu.

Related Links

To Add an Item to the 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.

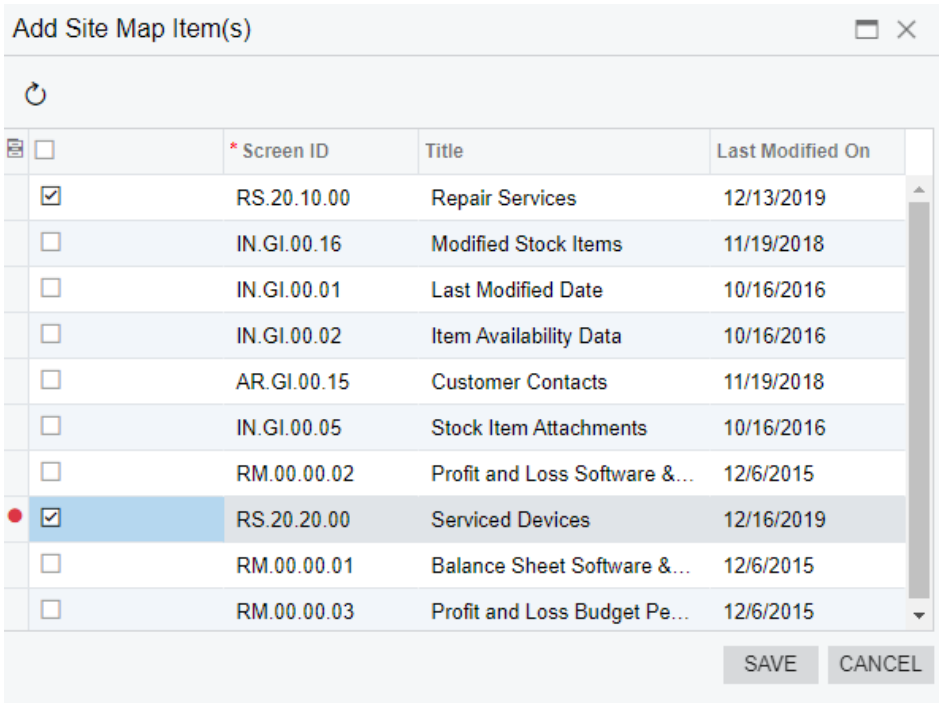


Figure: The selected site map item

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.

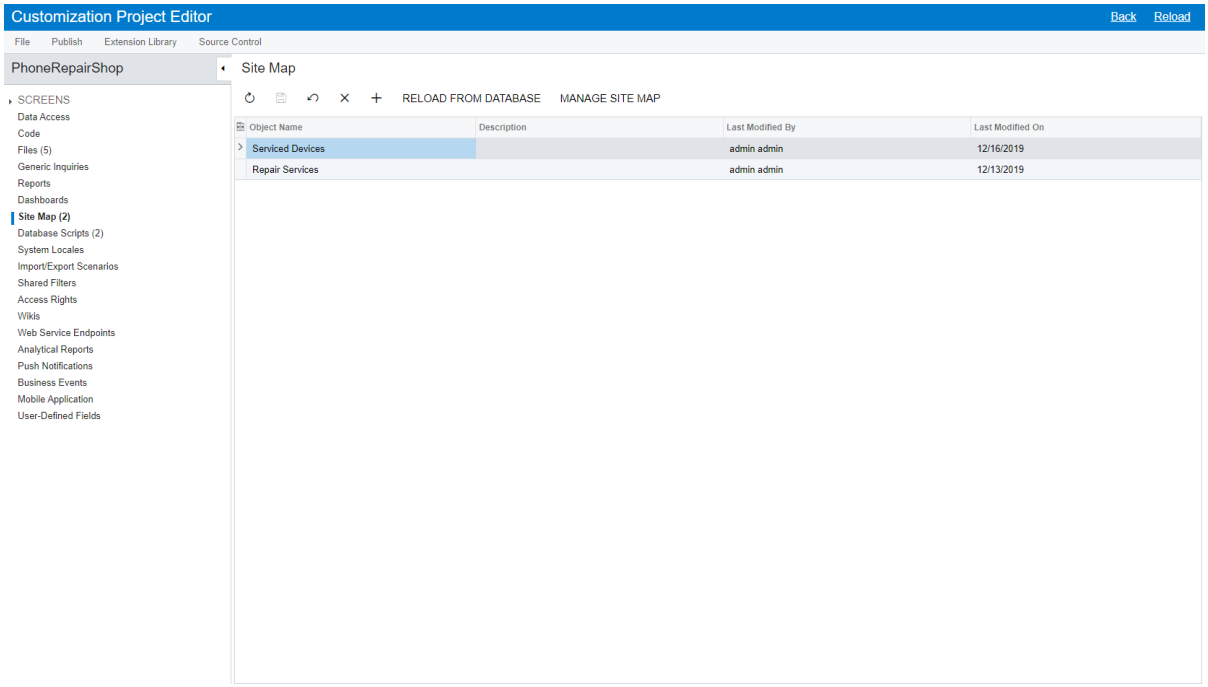


Figure: The Site Map page

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 Customization Project Editor.
2. In the navigation pane, click **Screens**. The Customized Screens page opens.
3. On the page toolbar, click **Add Screen** > **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.

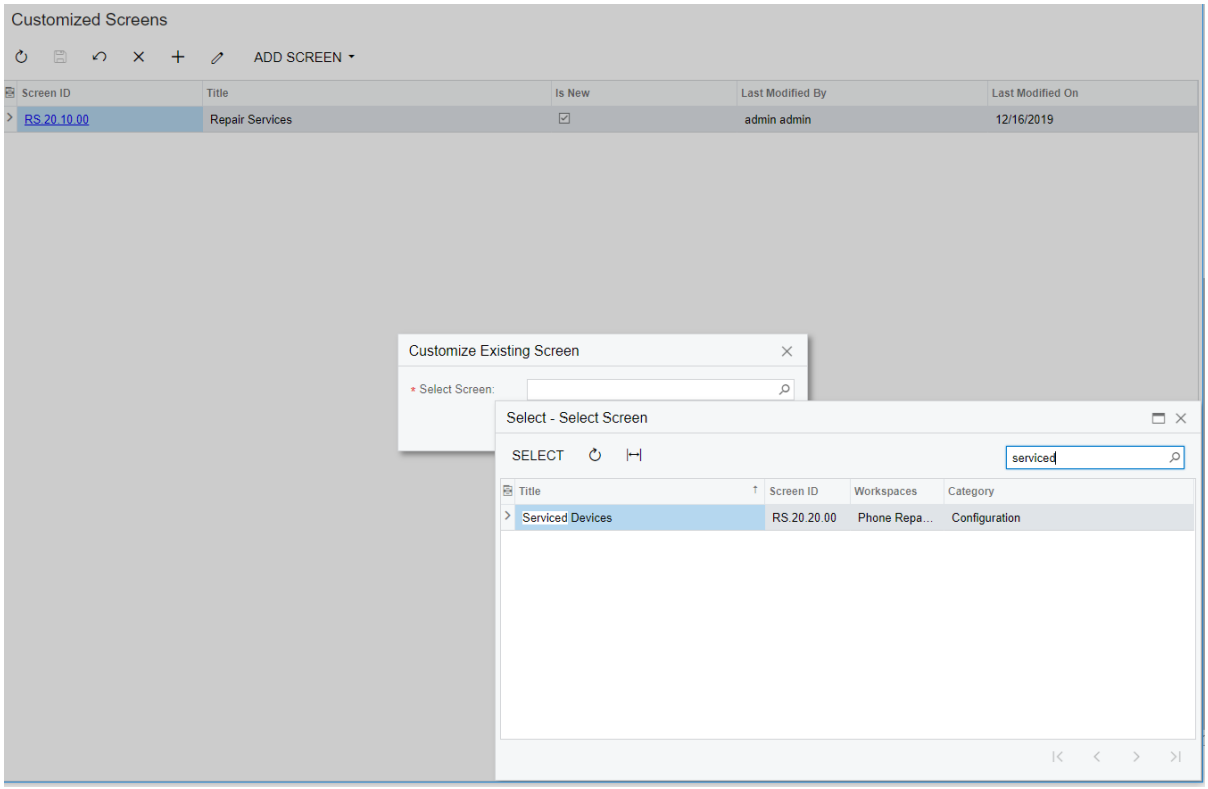


Figure: Selecting the Serviced Devices form

5. Click **OK**.

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

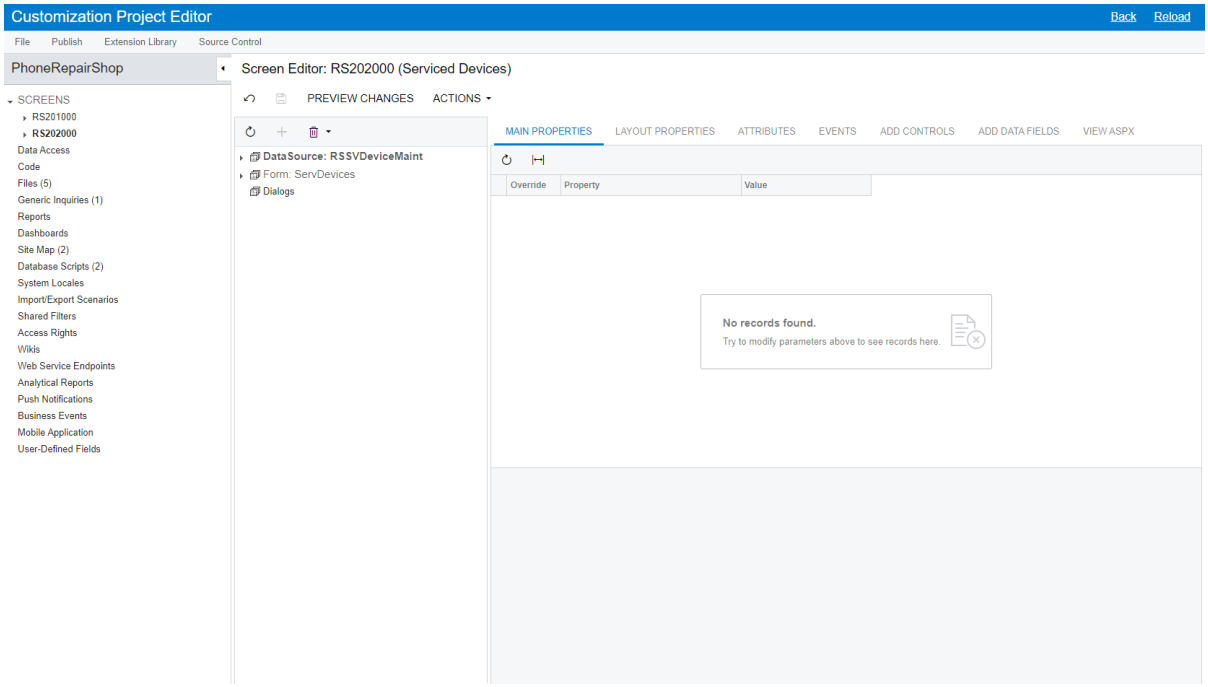


Figure: Viewing the Serviced Devices form in the Screen Editor

6. 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).

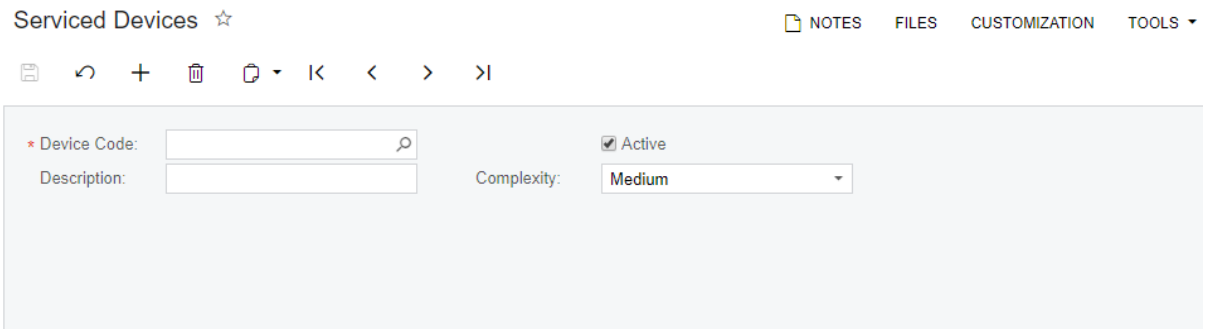


Figure: The Serviced Devices form

2. Enter the following settings:
   a. **Device Code**: `NOKIA3310`

      The DAC field for the **Device Code** box is marked as a key field 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. **Repair 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.

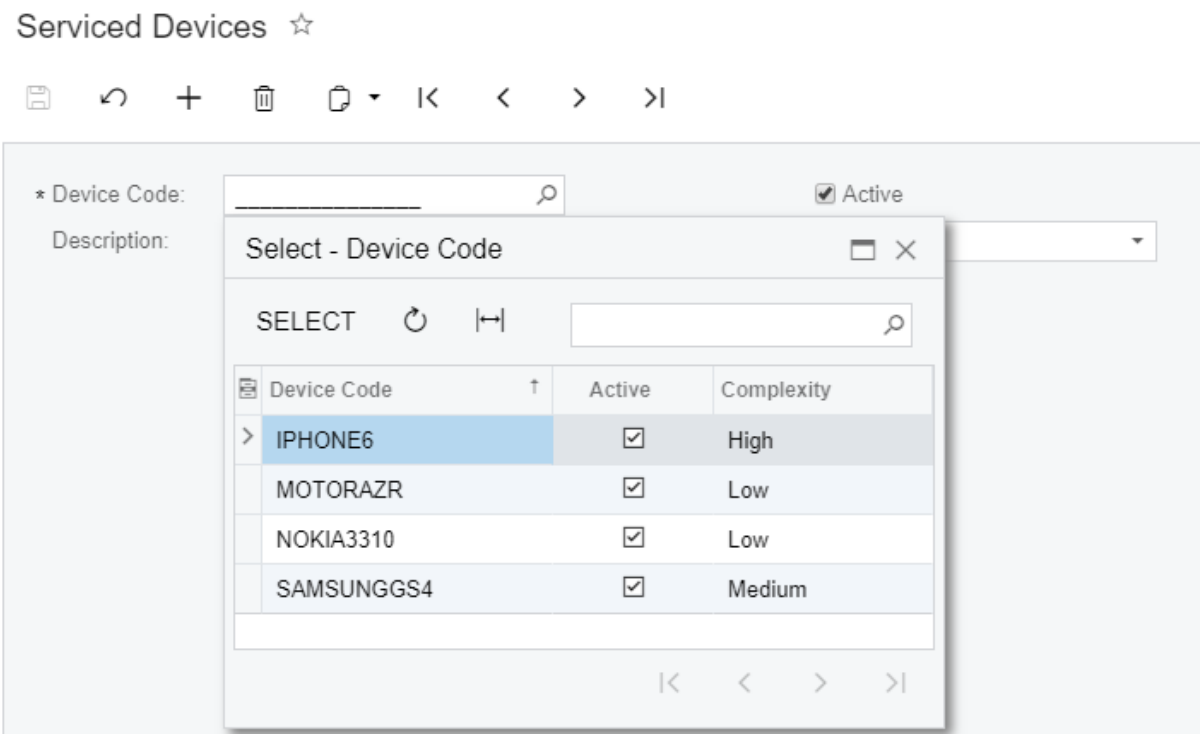


Figure: The lookup table

2. 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.

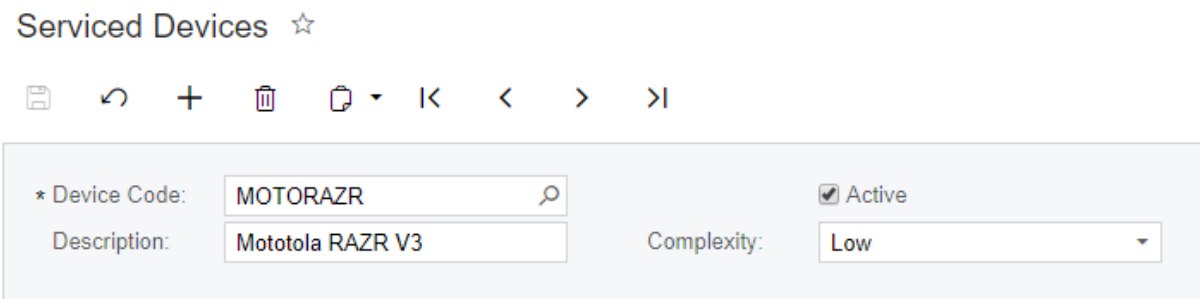


Figure: The device properties

3. Clear the **Active** check box.
4. 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 create a substitute form for a custom form and save it to the customization project.

Related Links

Managing Substitutes of Entry Forms

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 `GI_ServicedDevices.xml` file, which is provided with this course.
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 **Tables** and **Results Grid** tabs.

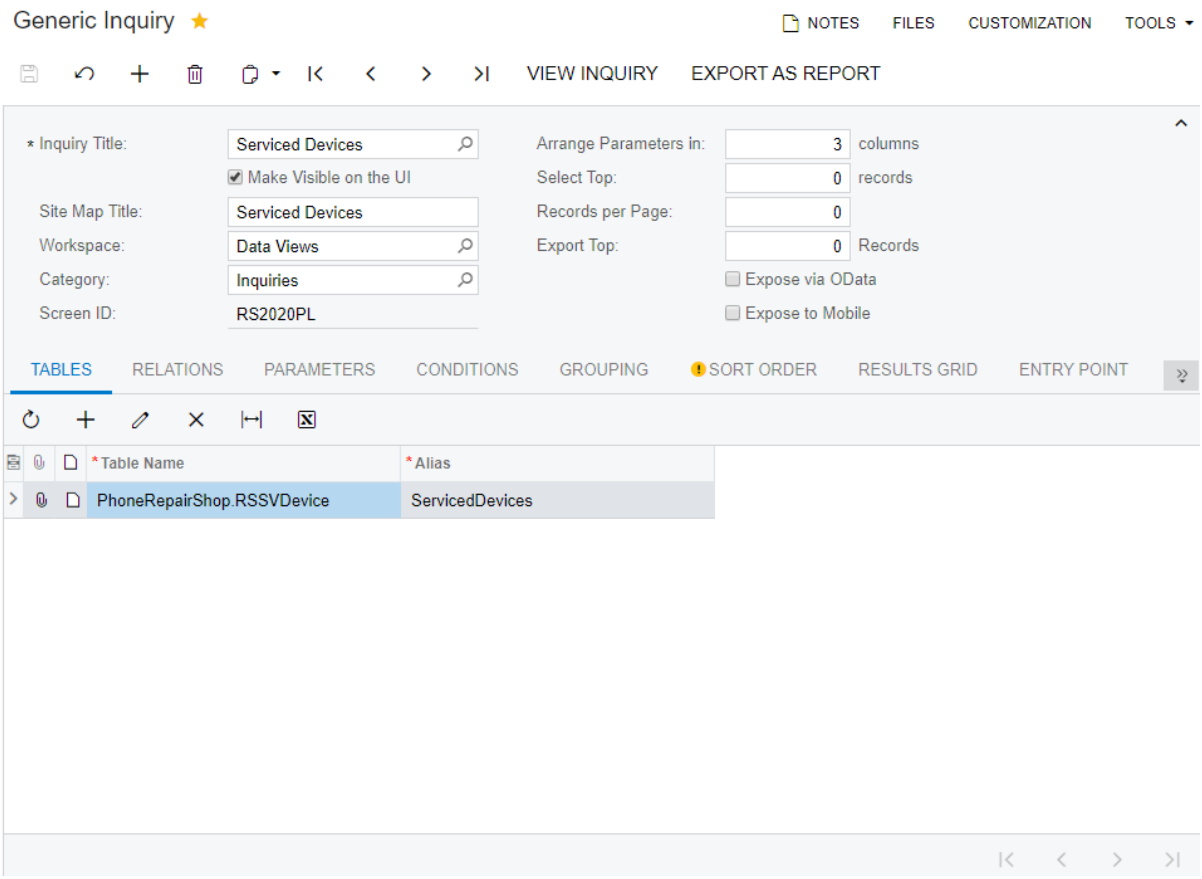


Figure: The Serviced Devices generic inquiry

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.

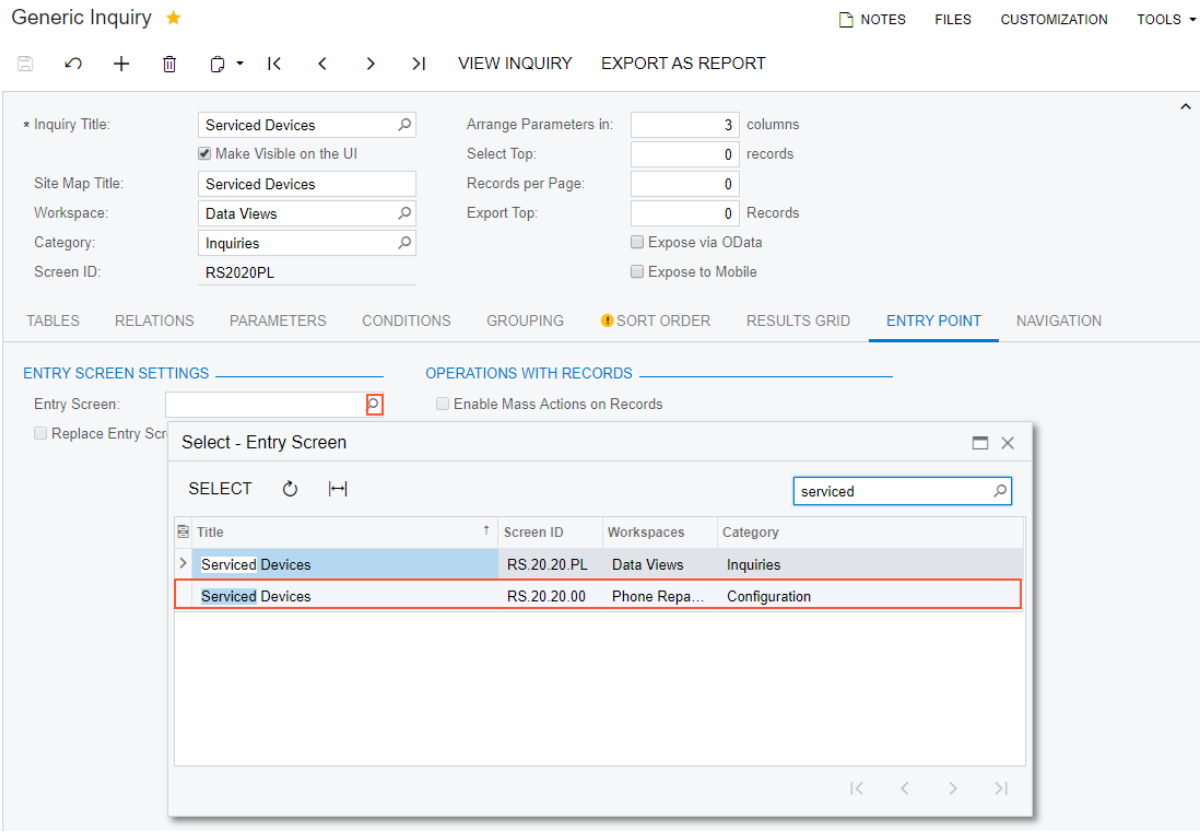


Figure: Selecting the Serviced Deviced form

4. Select the **Replace Entry Screen with this Generic 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 of the following ways:

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

Related Links

To Configure a Generic Inquiry to Replace an Entry Form

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 Inquires**. 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**.

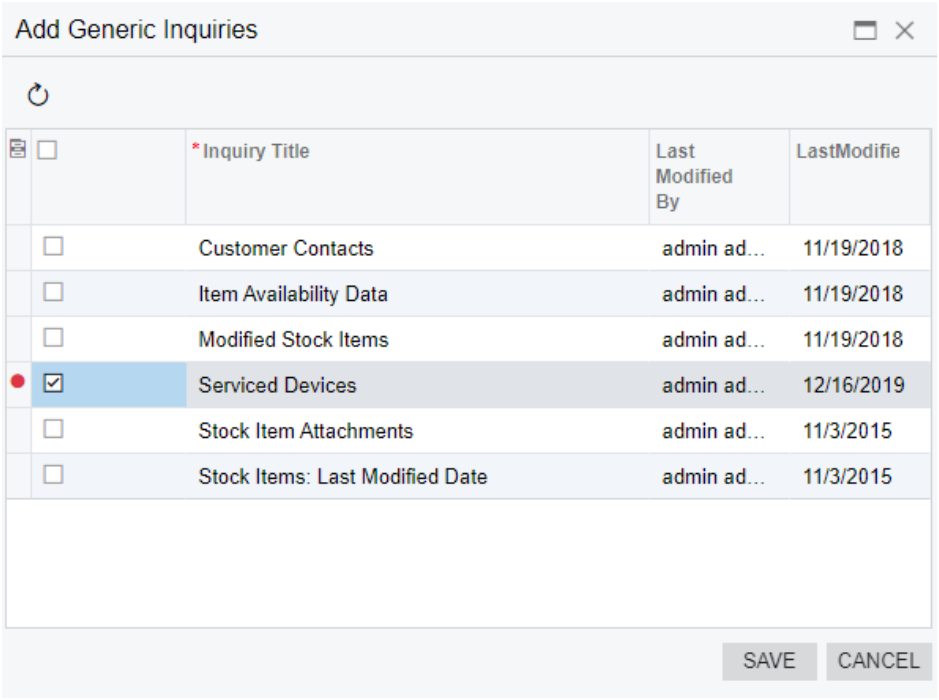


Figure: The Add Generic Inquiries dialog box

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.

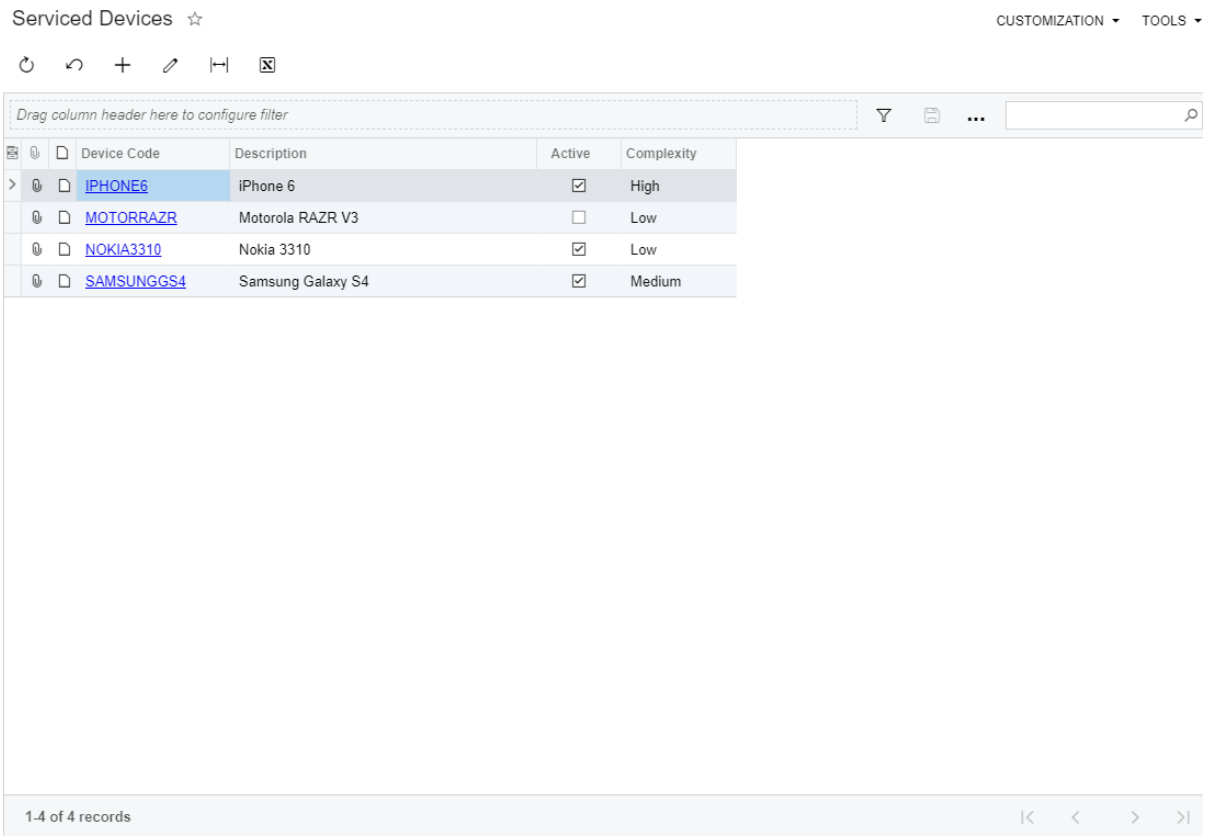


Figure: The Serviced Devices substitute form

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 & 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

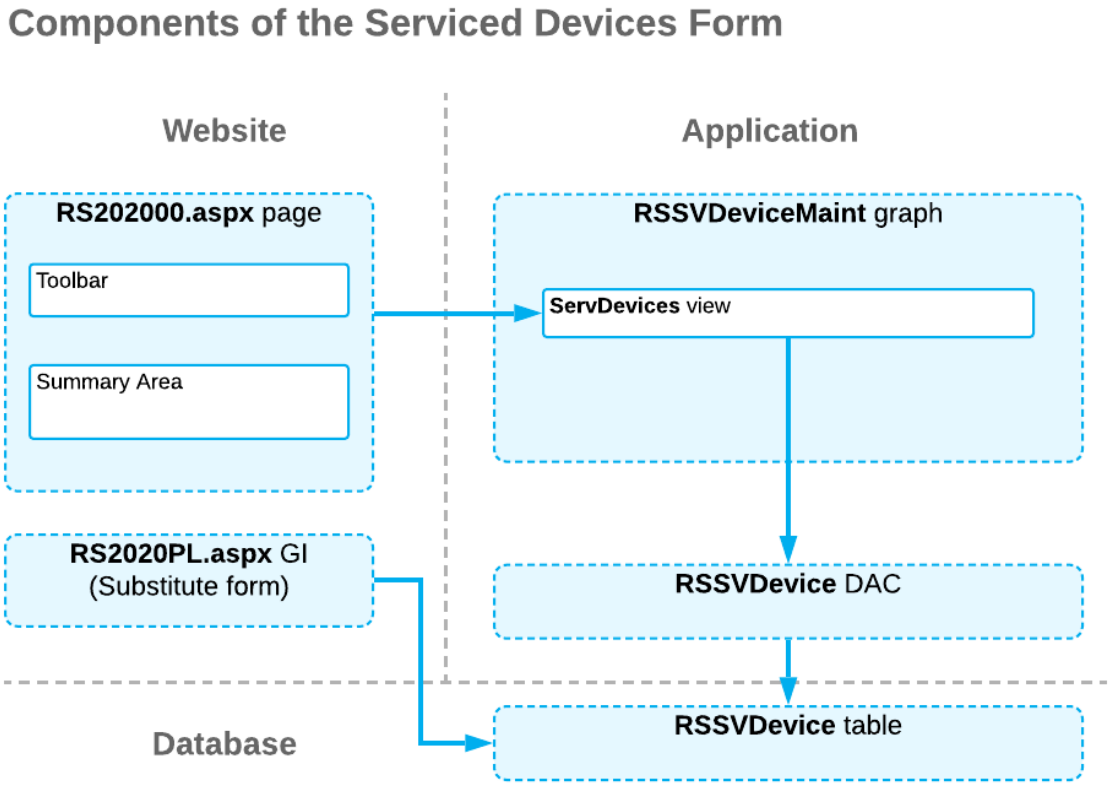
In this lesson, you have learned about substitute forms and have created one for the Serviced Devices (RS202000) form.

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.



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.

Review Questions

1. When you need to specify a DAC name as a second parameter of the graph definition?
   a. When you need to bind a graph and DAC
   b. When you need to specify that a DAC will be used in the graph's code
   c. When you need to define a set of standard buttons for the form toolbar
2. What attribute do you use to configure a drop-down control?
   a. `PXStringList`
   b. `PXSelector`
   c. `PXDropDown`
3. What do you use to configure the layout of a form in Visual Studio?
   a. Screen Editor
   b. The `PXLayoutRule` tag in ASPX code
   c. Attributes in the definitions of DAC fields
4. Which of the following types of forms can be used as a substitute form?
   a. A generic inquiry
   b. A maintenance form
   c. A setup form

Answer Key

1. C
2. A
3. B
4. A

Course Summary

In this course, you have learned the basic concepts of developing a custom form in Acumatica ERP using Customization Project Editor and Visual Studio. Now you know the following:

- How to create a maintenance form
- How to access and manipulate data by using DACs and graph members
- How to implement business logic on a form by using event handlers
- How to configure controls that display data on a form and adjust the layout of controls on the form

Appendix: Reference Implementation

You can find the reference implementation of the customization described in this course in the `Customization\T200` folder of the *Help-and-Training-Examples* repository in Acumatica GitHub.