Developer Course

Customization

T210 Customized Forms and Master-Detail Relationship 2022 R1

Revision: 3/31/2022

Acumatica
The Cloud ERP

Contents

Copyright

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

Restricted Rights

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

Disclaimer

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

Trademarks

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

Software Version: 2022 R1

Last Updated: 03/31/2022

Introduction

The *T210 Customized Forms and Master-Detail Relationship* course shows how to customize existing Acumatica ERP forms and define the master-detail relationship. In this course, you will learn how to add custom controls and tabs to an existing Acumatica ERP form. You will also learn how to implement the UI logic, calculations, and insertion of a default detail record by using Acumatica Framework.

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. It is designed to give you ideas about how to develop your own embedded applications by using the customization tools. As you go through the course, you will continue development of the customization for the cell phone repair shop, which was started in the *T200 Maintenance Forms* training course. We recommend completing *T200 Maintenance Forms* before you complete this one; if you have not completed that course, a prerequisite step will explain how to publish the customization project that contains the results of *T200 Maintenance Forms*.

After you complete all the lessons of the course, you will be familiar with the basic programming techniques for the customization of existing Acumatica ERP forms and for the implementation of the business logic of an Acumatica ERP form.

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 *Course Prerequisites*, perform *Initial Configuration*, and carefully read *Company Story and Customization Description*.
2. Complete the lessons in all parts of the training guide.
3. In Partner University, take *T210 Certification Test: Master-Detail Relationship and Customized Forms*.

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

What Is in a Part?

Each part of the course is dedicated to implementation of a particular scenario of customization of an existing Acumatica ERP form or to implementation of a particular business logic on example of a custom form.

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

What Is in a Lesson?

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

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

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

What Are the Documentation Resources?

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

Licensing Information

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

Course Prerequisites

To complete this course, you should be familiar with the basic concepts of Acumatica Framework. We strongly recommend that you complete the *T200 Maintenance Forms* training course before you begin this course.

Required Knowledge and Background

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

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

Initial Configuration

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

If you have deployed an instance for the *T200 Maintenance Forms* course and have the customization project and the source code for this course, you can perform only *Step 3: Creating the Database Tables*.

Step 1: Preparing the Environment

If you have completed the *T200 Maintenance Forms* training course and are using the same environment for the current course, you can skip this step.

You should 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 for Acumatica ERP 2022 R1*.
2. Make sure that the Web Server (IIS) features that are listed in *Configuring Web Server (IIS) Features* are turned on.
3. Install the Acuminator extension for Visual Studio.
4. Clone or download the customization project and the source code of the extension library from the *Help-and-Training-Examples* repository in Acumatica GitHub to a folder on your computer.
5. Install Acumatica ERP. On the Main Software Configuration page of the 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. Reinstallation of Acumatica ERP does not affect existing Acumatica ERP instances. For details, see *To Install the Acumatica ERP Tools*.

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

If you have completed the *T200 Maintenance Forms* training course, instead of deploying a new instance, you can use the Acumatica ERP instance that you have deployed for the *T200 Maintenance Forms* training course.

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

1. Open the Acumatica ERP Configuration Wizard, and do the following:
   a. Click **Deploy New Application Instance for T-series Developer Courses**.
   b. On the **Database Configuration** page, make sure the name of the database is `PhoneRepairShop`.
   c. On the **Instance Configuration** page, do the following:
      a. In the **Local Path of the Instance** box, select a folder that is outside of the `C:\Program Files (x86)` and `C:\Program Files` folders. We recommend that you store the website folder outside of these folders to avoid an issue with permission to work in these folders when you perform customization of the website.

b. In the **Training Course** box, select the training course you are taking.

The system creates a new Acumatica ERP instance, adds a new tenant, loads the data to it, and publishes the customization project that is needed for this training course.

2. Make sure a Visual Studio solution is available in the `App_Data\Projects\PhoneRepairShop` folder of the Acumatica ERP instance folder. This is the solution of the extension library that you will modify in this course.
3. Sign in to the new tenant by using the following credentials:
   - Username: `admin`
   - Password: `setup`

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

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

If for some reason you cannot complete instructions in this step, you can create an Acumatica ERP instance as described in *Appendix: Deploying the Needed Acumatica ERP Instance for the Training Course* and manually publish the needed customization project as described in *Appendix: Publishing the Required Customization Project*.

Step 3: Creating the Database Tables

Create the database tables that are necessary for the *T210 Customized Forms and Master-Detail Relationship* training course and include the scripts in the customization project as follows:

1. In SQL Server Management Studio, execute the `T210_DatabaseTables.sql` script to create the database tables that are necessary for the *T210 Customized Forms and Master-Detail Relationship* training course. The script creates the following tables: `RSSVRepairPrice`, `RSSVRepairItem`, `RSSVLabor`, `RSSVWarranty`, and `RSSVStockItemDevice`.

   The script is provided in the `Customization\T210\SourceFiles\DBScripts` folder, which you have downloaded from Acumatica GitHub.
2. On the Database Scripts page of the Customization Project Editor, for each added table, do the following:
   a. On the page toolbar, click **Add > Custom Table Schema**.
   b. In the dialog box that opens, select the table and click **OK**.
3. Publish the project.

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*.

Company Story and Customization Description

In this course, you will continue the development for the cell phone repair shop of the Smart Fix company that you have started in the *T200 Maintenance Forms* training course.

> If you have not completed this training course, you have loaded and published the customization project with the results of this course as described in *Initial Configuration*.

In the *T200 Maintenance Forms* training course, you have created two simple maintenance forms, Repair Services (RS201000) and Serviced Devices (RS202000), which the Smart Fix company uses to manage the lists of, respectively, repair services that the company provides and devices that can be serviced.

In this course, you will create another maintenance form, Services and Prices (RS203000), which gives users the ability to define and maintain the price for each provided repair service. You will also customize the *Stock Items* (IN202500) form of Acumatica ERP to mark particular stock items as repair items—that is, the items that are supplied to the customer (such as replacement screens and batteries) as part of the repair services.

Services and Prices Form

The Services and Prices (RS203000) form, which you will create in this course, will look as shown in the following screenshot.

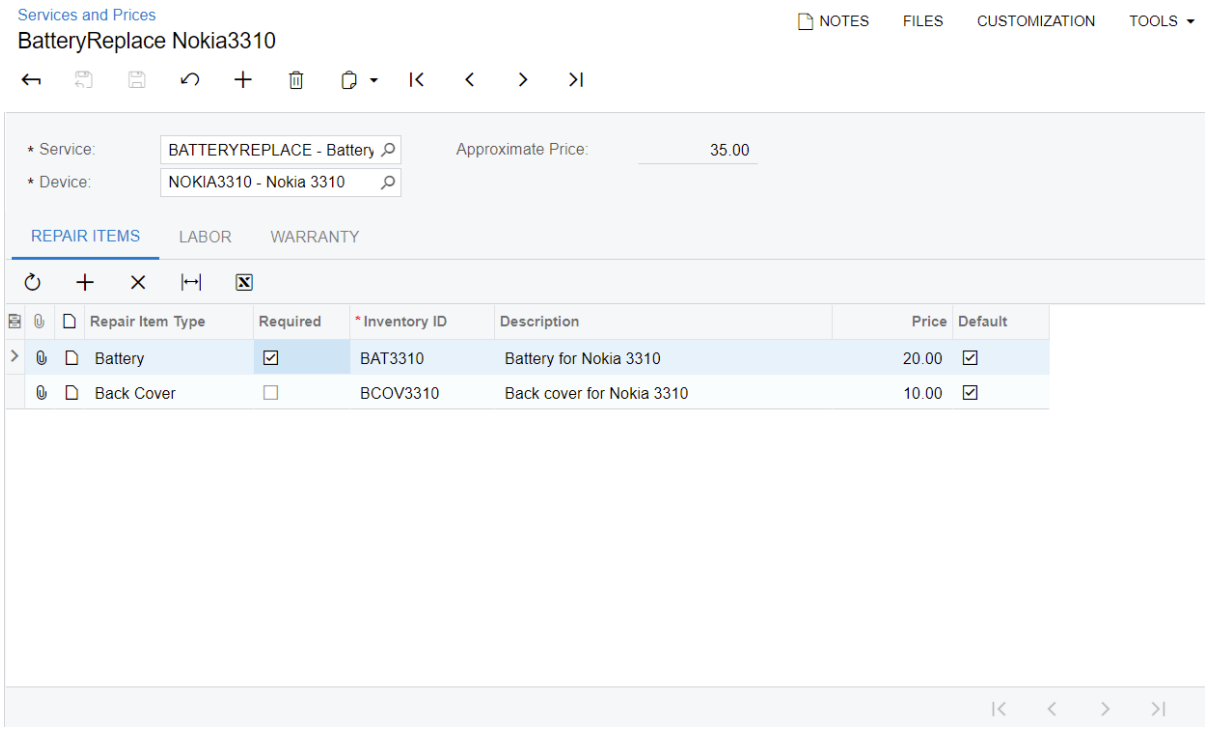


Figure: Services and Prices form

The form will contain the following tabs:

- **Repair Items**: Will show the list of repair items (stock items) necessary for the repair service and device selected in the Summary area of the form.
- **Labor**: Will represent the work (the list of non-stock items) that is performed for the repair service and device selected in the Summary area of the form.
- **Warranty**: Will include the list of contract templates for the repair service and device selected in the Summary area of the form.

You will also create a substitute form of the inquiry type, which will serve as the entry point to the form.

> This form displays the data entered on the Services and Prices form in a tabular format. The generic inquiry will function as a *substitute form* because it will be brought up instead of the form when a user clicks the form name in a workspace.

These forms will use the following custom tables, which you have added to the application database in *Initial Configuration*:

- `RSSVRepairPrice`: The data of this table is displayed in the Summary area of the form.
- `RSSVRepairItem`: The data of this table is displayed on the **Repair Items** tab.
- `RSSVLabor`: The data of this table is displayed on the **Labor** tab.
- `RSSVWarranty`: The data of this table is displayed on the **Warranty** tab.

Customization of the Stock Items Form

The custom elements that you will add to the *Stock Items* (IN202500) form are shown in the following screenshot.

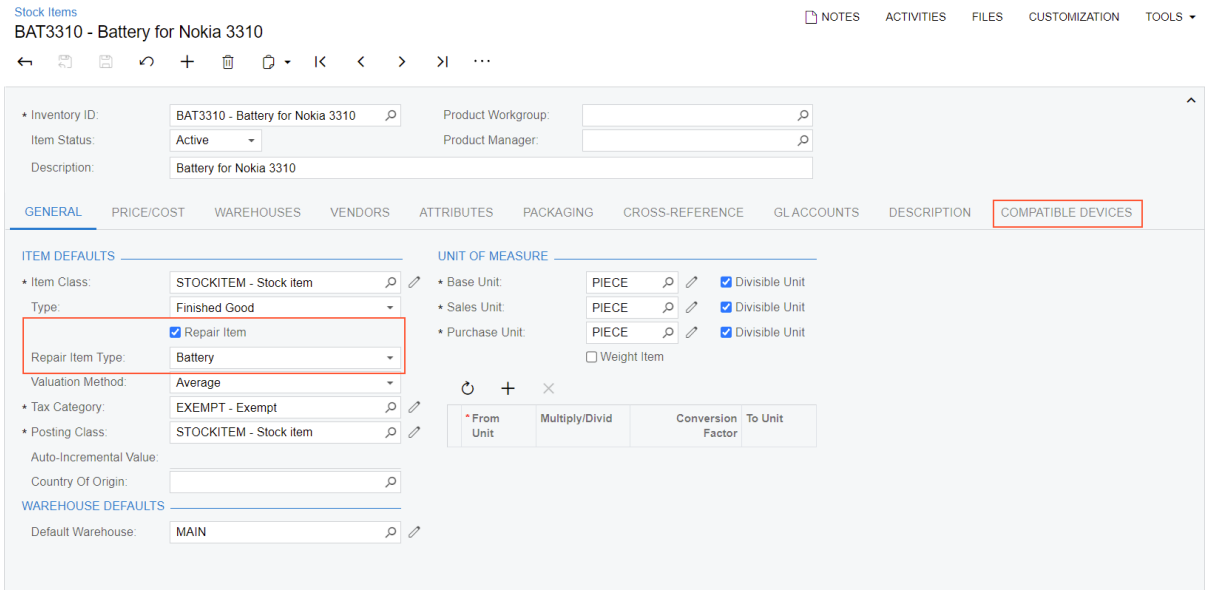


Figure: Stock Items form

The customization of the *Stock Items* form will include the following changes:

- The **Repair Item** check box, which you will add to the **Item Defaults** section of the **General** tab, will be used to define whether the selected stock item is a repair item.
- The **Repair Item Type** box, which you will also add to the **Item Defaults** section of the **General** tab, will hold the repair item type to which the repair item belongs, which is one of the following: *Battery*, *Screen*, *Screen Cover*, *Back Cover*, or *Motherboard*.
- You will add the **Compatible Devices** tab, which will be used to define and maintain a list of compatible serviced devices for the selected repair item. This tab will appear on the form only if the **Repair Item** check box is selected.

The customized *Stock Items* form will use the `RSSVStockItemDevice` custom table, which you have added to the application database in *Initial Configuration*. The form also uses the custom `UsrRepairItem` and `UsrRepairItemType` fields of the `InventoryItem` database table. You will add these custom fields in *Part 1: Custom Fields (Stock Items Form)*.

Part 1: Custom Fields (Stock Items Form)

A repair item is an item, such as a battery, that is supplied to the customer as part of the repair service. In Acumatica ERP, such items will be defined on the *Stock Items* (IN202500) form.

This form, however, was not designed to track whether the item is a repair item or what type of repair item it is. Therefore, to make it possible for a user to define a stock item as a repair item, the Smart Fix company needs to customize the *Stock Items* form. In this part of the course, you will add custom fields to this form.

After you complete the lessons of this part, you will be able to test the ability to create a repair item on the *Stock Items* form.

Lesson 1.1: Adding Custom Fields

A manager of the Smart Fix company needs to specify that particular stock items on the *Stock Items* (IN202500) form of Acumatica ERP are repair items and select the type of each repair item.

To implement this scenario, you need to change the UI of the *Stock Items* form and the database table that stores data for this form. In this lesson, you will use two approaches to perform these changes:

- Using the Customization Project Editor to create the custom database column, the data access class (DAC) field, and the UI control
- Using Visual Studio to add the DAC field and the Customization Project Editor to create the custom database column and the UI control

UI Changes

In this lesson, you will add the following custom controls to the *Stock Items* (IN202500) form of Acumatica ERP:

- **Repair Item**: A check box that indicates (if selected) that the stock item can be used during the provision of the repair services of the Smart Fix company
- **Repair Item Type**: A drop-down list box for the repair item type, which is one of the following:
  - *Battery*
  - *Screen*
  - *Screen Cover*
  - *Back Cover*
  - *Motherboard*

You will add these controls to the **Item Defaults** section of the **General** tab of the form (see the following screenshot).

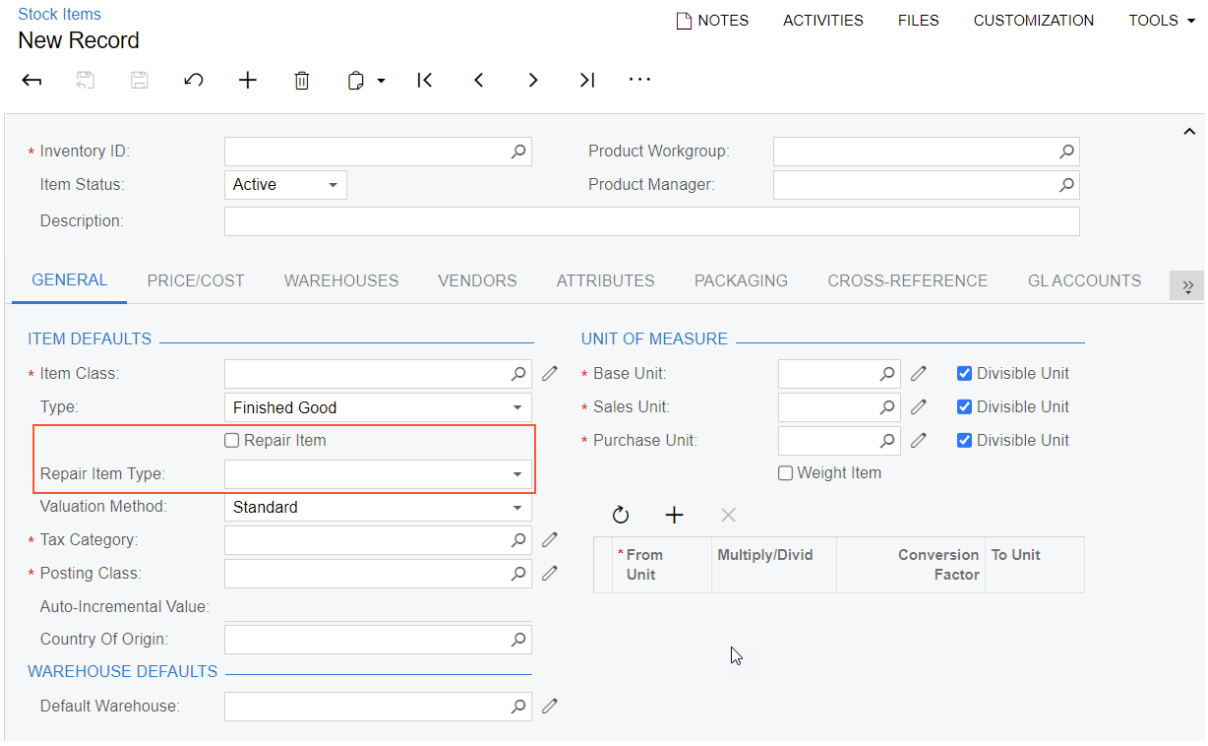


Figure: Custom elements to be added to the Stock Items form

Database Changes

The **General** tab displays the stock item's general information, which is stored in the data record of the `IN.InventoryItem` data access class. Hence, you will add the custom fields to this class. To be able to save the repair item data to the database, you will add the database columns for the new values. The `IN.InventoryItem` class records are stored in the `InventoryItem` database table; therefore, you will add columns for the new fields to this table.

Lesson Objectives

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

- Add a custom column to an Acumatica ERP database table
- Add a custom field to an Acumatica ERP data access class
- Add the control for the custom field to the form

Step 1.1.1: Creating a Custom Column and Field with the Project Editor

In this step, you will create a custom column in the `InventoryItem` database table and a custom field in the `IN.InventoryItem` data access class for this column. This column and field will be used to store and edit the value of the **Repair Item** check box. You will use the Customization Project Editor to add the column and field.

> The approach described in this step is the easiest way to create both the column in the database and the bound field in the corresponding data access class.
>
> We recommend that you not write custom SQL scripts to add changes to the database schema. If you add a custom SQL script, you must adhere to platform requirements that apply to custom SQL scripts, such as the support of multitenancy and the support of SQL dialects of the target database management systems. If you use the approach described in this topic, during the publication of the customization project, the platform generates SQL statements to alter the existing table so that this statement conforms to all platform requirements.

You will create an extension of the `IN.InventoryItem` DAC to hold custom fields (which is referred to as a *DAC extension* or *cache extension*). Acumatica Customization Platform creates an extension for every customized DAC to hold custom fields and customized attributes. At runtime, during the first initialization of a base class, the platform automatically finds the extension for the class and applies the customization by replacing the base class with the merged result of the base class and the extension it found. For more information about DAC extensions, see *BLC and DAC Extensions*.

You will also move the generated code to the extension library and adjust it with Acuminator.

Creating the Custom Column and Field

To create the custom column and the custom field, perform the following steps:

1. Open the *Stock Items* (IN202500) form, and then open the Screen Editor for it as follows:
   a. On the form title bar, click **Customization > Inspect Element**, as shown in the following screenshot.

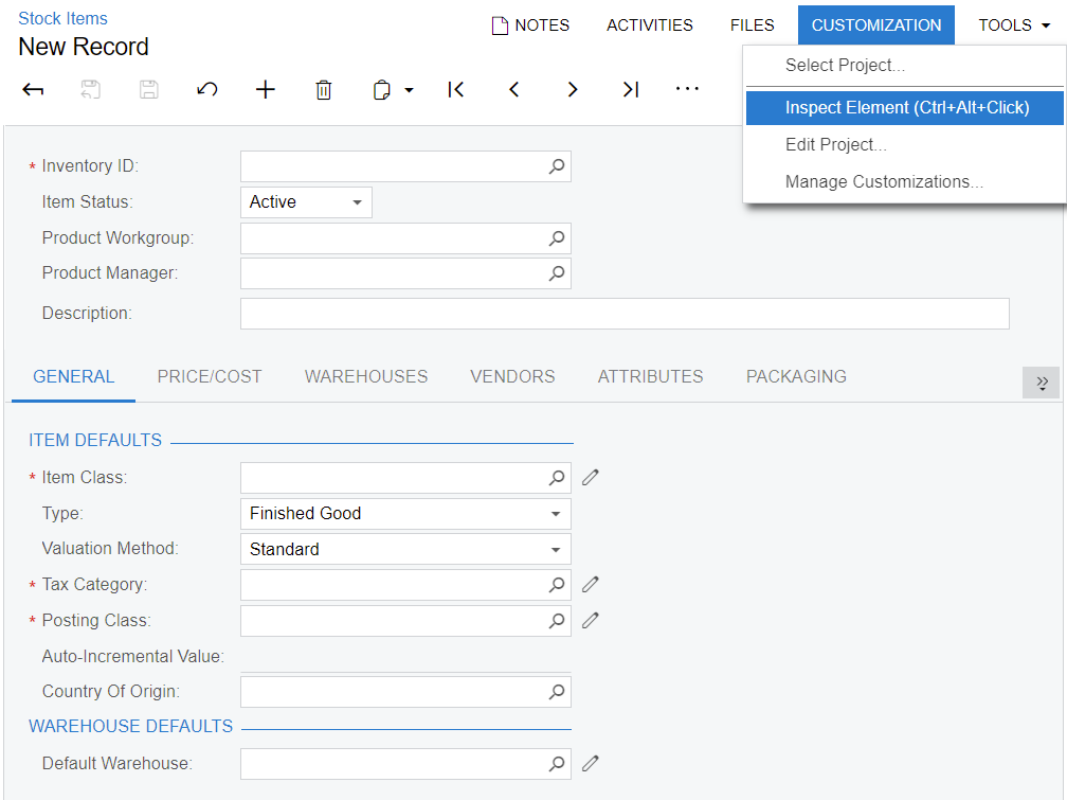


Figure: Customization menu

   b. Click the name of the **General** tab to open the **Element Properties** dialog box for the tab control, as shown in the following screenshot. In the dialog box, notice the following:

   - *Tab* (the `PXTab` control) is the type of UI container whose area you have clicked for inspection.

- The `InventoryItem` data access class provides the data fields for the controls on the inspected tab.

  > By clicking the link with the name of the DAC you can view details about this DAC in the DAC Schema Browser.

- The `ItemSettings` data view provides data for the container.
- The `InventoryItemMaint` graph provides business logic for this form.

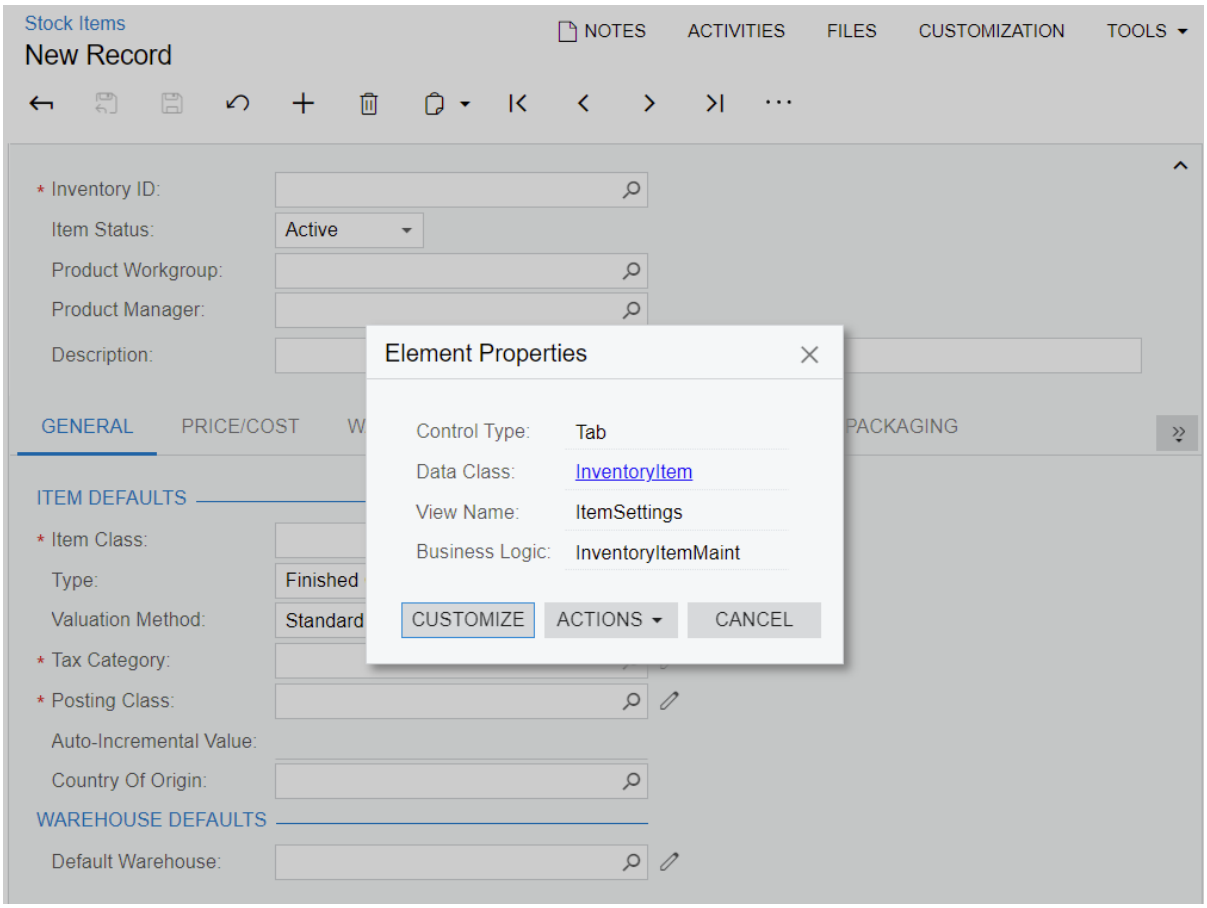


Figure: Element Properties dialog box

c. Click **Customize**.

d. In the **Select Customization Project** dialog box, which opens, select the *PhoneRepairShop* customization project, and click **OK**.

   The Customization Project Editor opens for the *PhoneRepairShop* project; the Screen Editor is displayed for the **Tab: ItemSettings** node, which is selected in the control tree.

2. To add a custom field for the **Repair Item** check box in the customization project, do the following:

   a. On the Screen Editor page, click the **Add Data Fields** tab.

   b. On the table toolbar, click **New Field**.

   c. In the **Create New Field** dialog box, which opens, specify the following settings for the new field:

   - **Field Name**: `RepairItem`

> As soon as you move the focus out of the **Field Name** box, the system adds the `Usr` prefix to the field name, which provides a distinction between the original fields and the new custom fields that you add to the class. Keep the prefix in the field name.

- **Display Name**: `Repair Item`
- **Storage Type**: `DBTableColumn`
- **Data Type**: `bool`

d. Click **OK** to create the specified extension to both the data access class and the database table. The DAC extension name contains the name of the original DAC and the `Ext` suffix. The **IN.InventoryItem** customization item is added to the Customized Data Classes page of the Customization Project Editor.

Once you click **OK**, the platform automatically saves the changes to the customization project that is opened in the Project Editor. However, the changes have not yet been applied to the application because the project has not been republished.

3. Move the data access class extension to the `PhoneRepairShop_Code` extension library:

a. In the navigation pane, click **Data Access**.

The Customized Data Classes page opens.

b. On the Customized Data Classes page, click the line with InventoryItem.

c. On the page toolbar, click **Convert to Extension**.

The InventoryItemExtensions *Code* item appears in the Code Editor.

d. On the toolbar of the Code Editor, click **Move to Extension Lib**.

> For details how to move a DAC to an extension library, see *To Move a DAC Item to an Extension Library* in the documentation.

4. In Visual Studio, adjust the DAC extension as follows:

a. Move the `InventoryItemExtensions.cs` file to the `DAC` folder and open the file.

Notice that Acuminator displays the *PX1016* error and the *PX1011* warning for the `InventoryItemExt` class.

The PX1016 error indicates that the class does not implement the `IsActive` method, which conditionally makes the extension active or inactive. For details, see *To Enable a DAC Extension Conditionally (IsActive)*. In this course, for simplicity, the extension will be always active and you will suppress the error.

The PX1011 warning shows that the `sealed` modifier can be removed because C#-style inheritance from `PXCacheExtension` is not supported. You will use the fix provided by Acuminator.

b. To suppress the PX1016 error, place the cursor to the `InventoryItemExt` class name and in the Quick Actions menu select **Suppress the PX1016 diagnostic with Acuminator > in a comment**, as shown in the screenshot below. Acuminator adds the suppression comment.

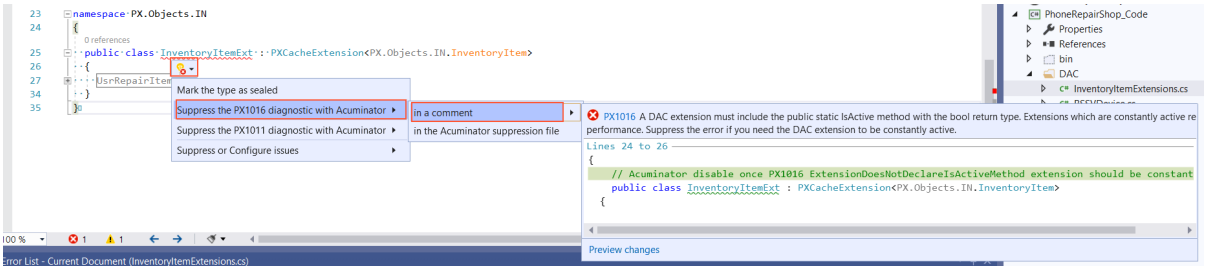


Figure: Suppression of the error in a comment

c. Place the cursor to the `InventoryItemExt` class name and in the Quick Actions menu select **Mark the type as sealed**, as the following screenshot shows. Acuminator adds the sealed modifier.

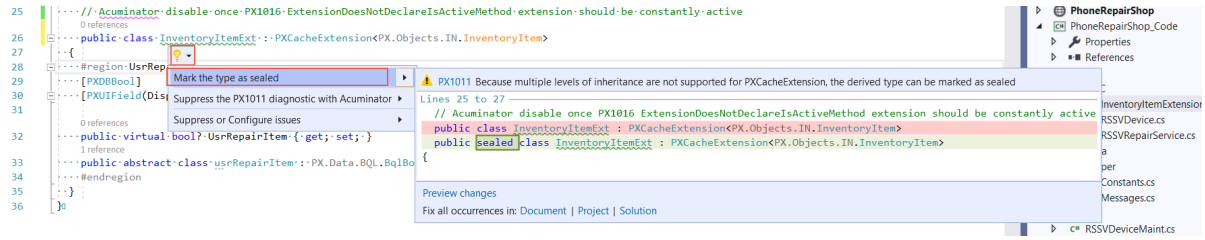


Figure: Fix of the warning

d. In the `PhoneRepairShop_Code` project, add an assembly reference for the `PX.Objects.dll` file, which is located in the `Bin` folder of the PhoneRepairShop instance folder.

e. Remove `virtual` from the `UsrRepairItem` property field.

f. Make sure the `UsrRepairItem` field has the attributes shown in the following code.

```
[PXDBBool]
[PXUIField(DisplayName="Repair Item")]
```

g. Add the `PXDefault` attribute as shown in the following code. The check box that will correspond to the field will be cleared by default and the value of the field will not be required.

```
[PXDefault(false, PersistingCheck = PXPersistingCheck.Nothing)]
```

h. Build the project.

Related Links

- *To Add a Custom Data Field*
- *To Move a DAC Item to an Extension Library*
- *To Publish the Current Project*
- *Changes in the Application Code (C#)*
- *To Enable a DAC Extension Conditionally (IsActive)*

Step 1.1.2: Creating a Control for the Custom Field

Now you will create a control for the custom field that you added to the *PhoneRepairShop* customization project in the previous step.

For you to create a control for a field on a form in an application instance, both of the following conditions must be met:

- The field exists in the instance.
- The field is available through a data view that refers to the data access class containing the field declaration.

Creating the Control

To create the control for the custom field, perform the following actions:

1. Open the Screen Editor for the *Stock Items* (IN202500) form.
2. In the control tree of the Screen Editor, click the **Tab: ItemSettings** node.

3. On the **Add Data Fields** tab, select the **Custom** filter tab to view the custom fields that are available through the data view of the container. Notice that the **Control** column displays the available control type for the custom field.
4. Create the control for the custom field as follows:
   a. In the control tree of the Screen Editor, select the **Type** node (shown in the following screenshot) to position the new control beneath it.

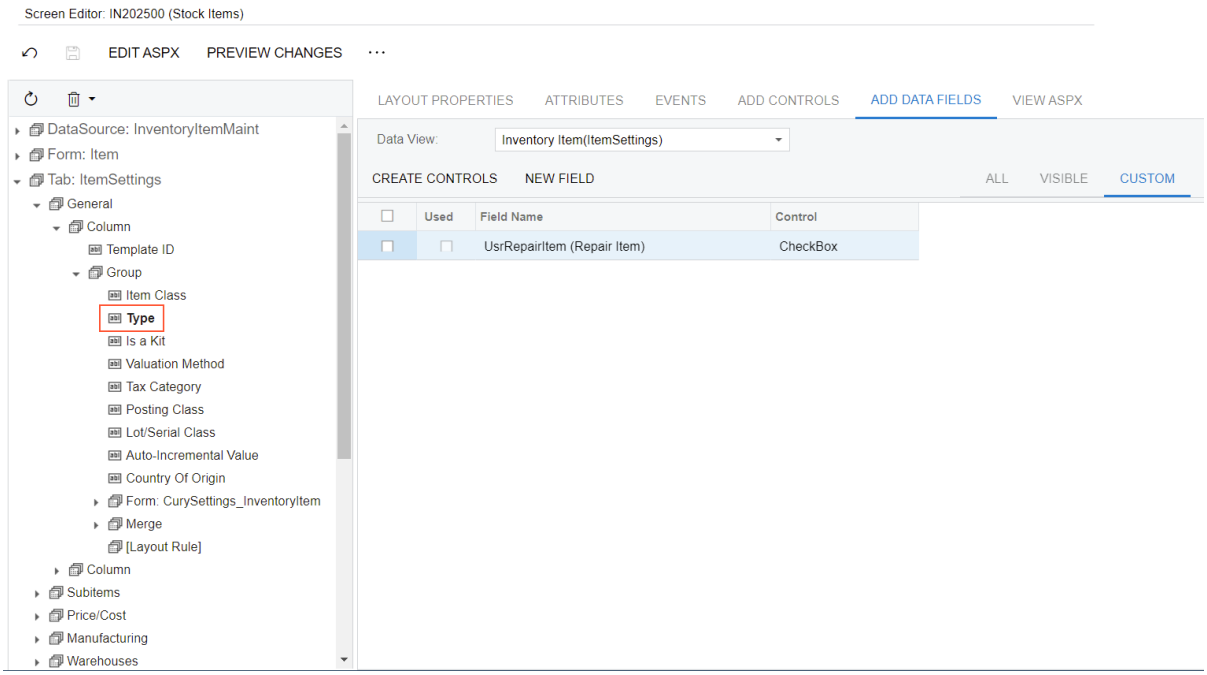


Figure: The Type node in the control tree

   b. On the **Add Data Fields** tab, select the unlabeled check box for the row with the custom field.
   c. On the table toolbar, click **Create Controls** to create the control for the selected field.

      The control appears in the control tree of the Screen Editor beneath the **Type** node (see the following screenshot). Notice that the **Used** check box has been selected for the field, meaning that a control for this field has been added to the layout.

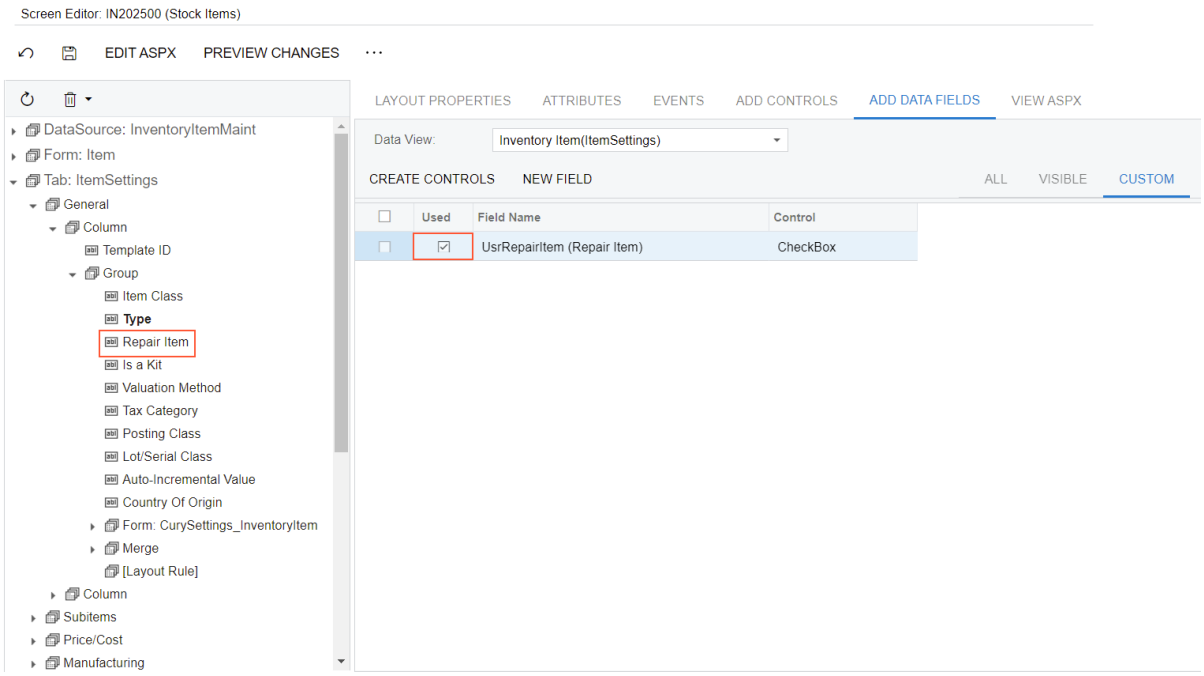


Figure: The added control

5. On the menu of the Customization Project Editor, click **Publish > Publish Current Project** to apply the customization to the site.

   > The **Modified Files Detected** dialog box opens before publication because you have rebuilt the extension library in the `PhoneRepairShop_Code` Visual Studio project in *Step 1.1.1: Creating a Custom Column and Field with the Project Editor*. The `Bin\PhoneRepairShop_Code.dll` file has been modified and you need to update it in the project before the publication.

   As the system applies the customization to the website, the system generates the proper SQL script by using the definition of the new database column added to the project in *Step 1.1.1: Creating a Custom Column and Field with the Project Editor*; it then executes the script on the database. The system also generates ASPX code for the custom control.

   > If you unpublish the project, the changes to the database schema and any custom data already entered remain in the database; the UI changes are removed.

6. Close the **Compilation** window.
7. Refresh the Stock Items form in the browser to view the added control on the **General** tab of the form, which is shown in the following screenshot.

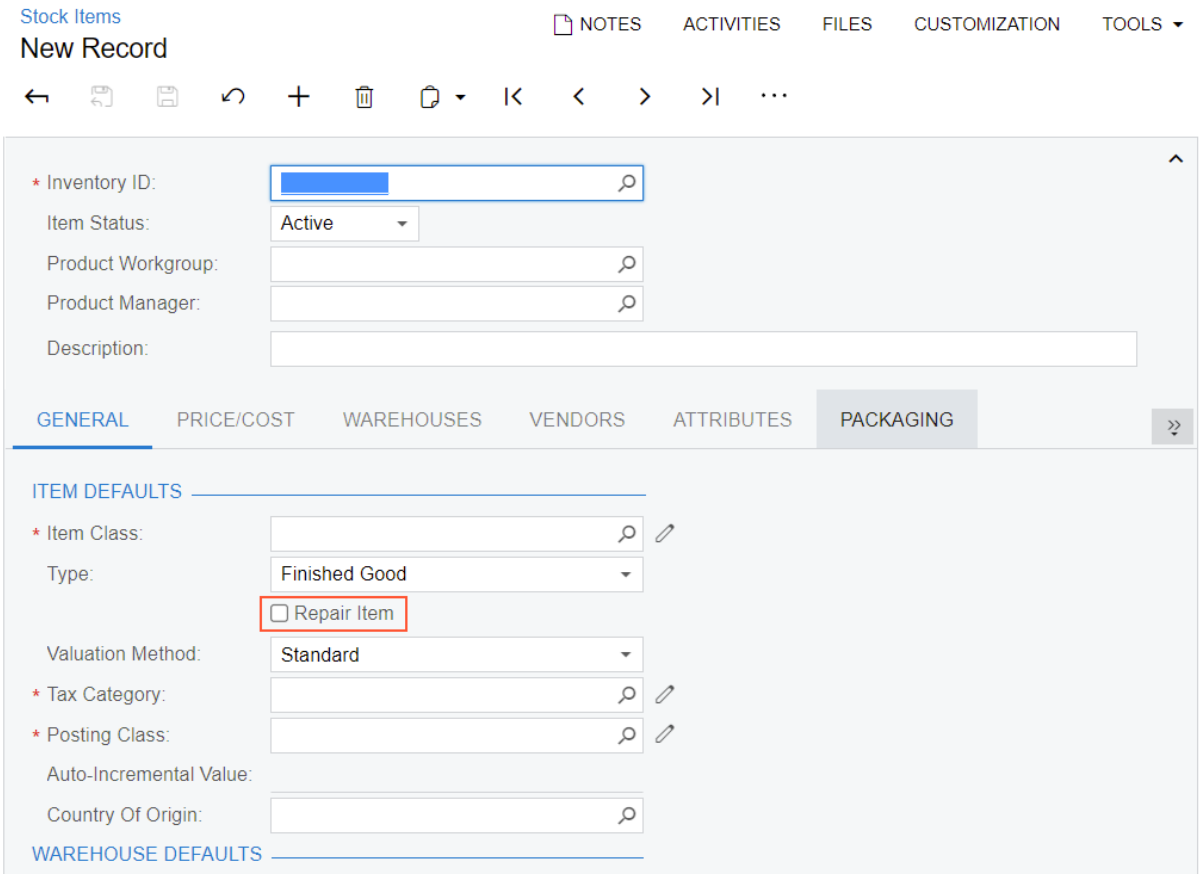


Figure: The Repair Item check box

Related Links

- *To Add a Box for a Data Field*
- *Changes in Webpages (ASPX)*

Step 1.1.3: Viewing the Content of the Customization Project

In this lesson, you have defined the following items in the *PhoneRepairShop* customization project:

- A custom column in the `InventoryItem` table of the database.
- A custom field in the `IN.InventoryItem` data access class. You then have moved this definition to the extension library.
- A custom control on the *Stock Items* (IN202500) form.

In this step, you will view the XML content of these items.

Viewing the Content of the Customization Project

To view the content of the customization project for items you have created, click **File > Edit Project Items** on the menu of the Customization Project Editor.

The Project Editor opens the Edit Project Items page, which displays the list of items of the customization project and the work area to review and edit the XML code of the item selected in the list (see the following screenshot).

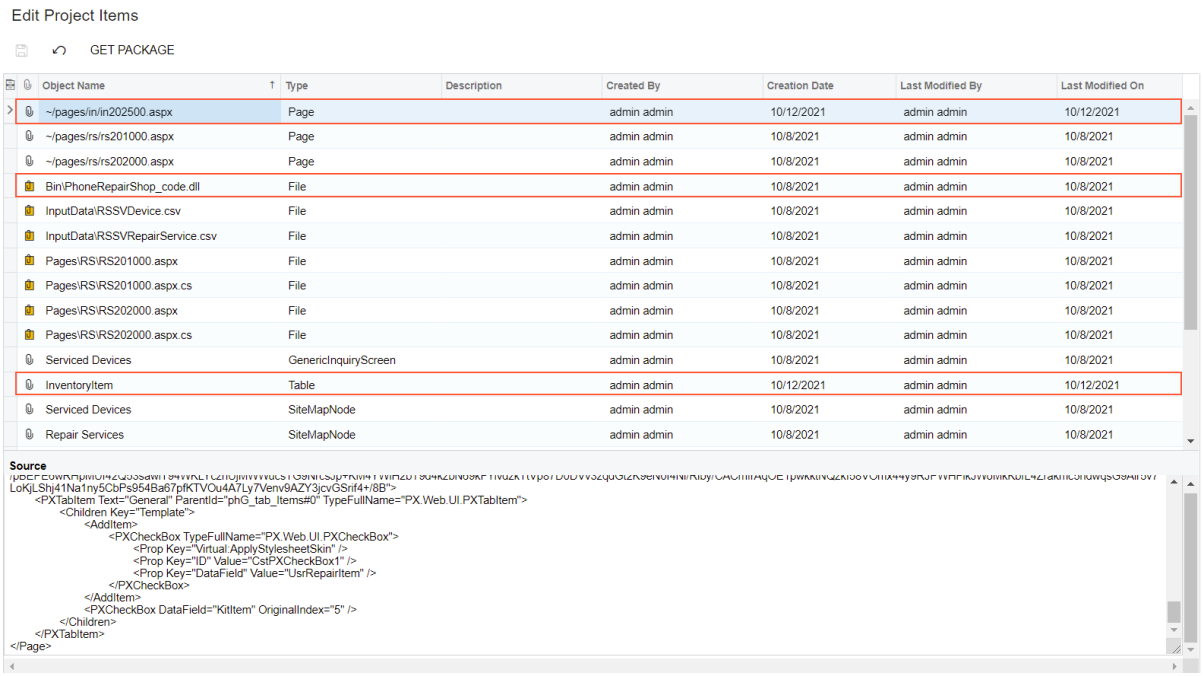


Figure: The list of the project items

Notice the following items in the *PhoneRepairShop* customization project, which are highlighted in the screenshot above.

Item	Description
`Bin\PhoneRepairShop_Code.dll`	This *File* item contains a relative path to the DLL file of the extension library.
`~/pages/in/in202500.aspx`	This *Page* item contains layout change instructions that have to be applied by the platform to the `.aspx` code of the *Stock Items* (IN202500) form during the publication of the project. When you publish the project, the platform creates a customized version of the `in202500.aspx` file with the same name in the `pages_in` subfolder of the `CstPublished` folder of the website.
`InventoryItem`	This *Table* item contains information about the custom column added to the `InventoryItem` table for the bound custom field created in the extension of the `InventoryItem` DAC.

Related Links

- *Customization Project*
- *Customization Project Editor*

Step 1.1.4: Creating a Custom Column with the Project Editor and a Custom Field with Visual Studio

In this step, you will create a custom column in the `InventoryItem` database table and a custom field in the `IN.InventoryItem` data access class for this column. This column will hold the value of the **Repair Item Type**

box of the *Stock Items* (IN202500) form, which corresponds to the field. You will use Visual Studio to add the DAC field and the Customization Project Editor to add the database column.

> ⚠ We recommend that you not write custom SQL scripts to add changes to the database schema. If you add a custom SQL script, you must adhere to platform requirements that apply to custom SQL scripts, such as the support of multitenancy and the support of SQL dialects of the target database management systems. If you use the approach described in this topic, during the publication of the customization project, the platform generates SQL statements to alter the existing table so that this statement conforms to all platform requirements.

You will define the `UsrRepairItemType` data field in the `InventoryItemExt` DAC extension. The fields in the DAC extensions are defined in the same way as they are in DACs. For details about the definition of DACs, see *Data Access Classes*.

You will define the **Repair Item Type** combo box as the input control for the `UsrRepairItemType` data field by adding the `PXStringList` attribute to the field. The control will give the user the ability to select one of the following repair item types: *Battery*, *Screen*, *Screen Cover*, *Back Cover*, or *Motherboard*.

Creating the Custom Column and Field

Do the following to create the custom column and field:

1. Add the database column as follows:
   a. In the Customization Project Editor, open the *PhoneRepairShop* project.
   b. On the left pane, click **Database Scripts**.
   c. On the More menu of the Database Scripts page of the Customization Project Editor, click **Add Custom Column to Table**.

      > ⓘ The InventoryItem database script is already present on the page. So, alternatively, you can click on the InventoryItem row, and in the **Edit Table Columns** dialog box which appears, click **Add > Add New Column**.

   d. In the dialog box that opens, specify the following values:
      - **Table**: *InventoryItem*
      - **Field Name**: `UsrRepairItemType`
      - **Data Type**: *string*
      - **Length**: `2`
   e. Click **OK** to close the dialog box.

      The Acumatica Customization Platform adds the column to the `InventoryItem` *Table* item in the customization project.
2. In Visual Studio, in the `Helper\Constants.cs` file, define the `RepairItemTypeConstants` class (if it has not been defined yet) as shown in the following code.

```
//Constants for the repair item types
public static class RepairItemTypeConstants
{
    public const string Battery = "BT";
    public const string Screen = "SR";
    public const string ScreenCover = "SC";
    public const string BackCover = "BC";
    public const string Motherboard = "MB";
}
```

3. In the `Helper\Messages.cs` file, add the constants for the repair item types (if they have not been added yet), as shown in the following code.

```
//Repair item types
public const string Battery = "Battery";
public const string Screen = "Screen";
public const string ScreenCover = "Screen Cover";
public const string BackCover = "Back Cover";
public const string Motherboard = "Motherboard";
```

4. In the `InventoryItemExt` class of the `InventoryItemExtensions.cs` file, add a custom field for the **Repair Item Type** box, as the following code shows.

```
#region UsrRepairItemType
[PXDBString(2, IsFixed = true)]
[PXStringList(
    new string[]
    {
        PhoneRepairShop.RepairItemTypeConstants.Battery,
        PhoneRepairShop.RepairItemTypeConstants.Screen,
        PhoneRepairShop.RepairItemTypeConstants.ScreenCover,
        PhoneRepairShop.RepairItemTypeConstants.BackCover,
        PhoneRepairShop.RepairItemTypeConstants.Motherboard
    },
    new string[]
    {
        PhoneRepairShop.Messages.Battery,
        PhoneRepairShop.Messages.Screen,
        PhoneRepairShop.Messages.ScreenCover,
        PhoneRepairShop.Messages.BackCover,
        PhoneRepairShop.Messages.Motherboard
    })]
[PXUIField(DisplayName = "Repair Item Type")]
public string UsrRepairItemType { get; set; }
public abstract class usrRepairItemType :
  PX.Data.BQL.BqlString.Field<usrRepairItemType>
{ }
#endregion
```

5. Build the project.

Related Links

- *To Add a Custom Column to an Existing Table*
- *PXStringListAttribute Class*
- *Data Access Classes*

Step 1.1.5: Creating a Control for the Custom Field—Self-Guided Exercise

Now you will create a control on your own for the **Repair Item Type** custom field, which you added to the *PhoneRepairShop* customization project in the previous step. The addition of a control for the field was described earlier in this lesson.

> For custom forms (that is, the forms that have been created from scratch and added to the customization project), you can edit the ASPX code in the `Pages` folder of the site. For customized forms, the `Pages` folder of the site contains the original version of the ASPX code for this form; the customized version is available only in the `CstPublished` folder of the site. However, you cannot edit the custom ASPX code in the `CstPublished` folder because your changes will be overridden once you publish the customization project.

Once you complete this step, the *Stock Items* (IN202500) form will look as shown in the following screenshot. The `InventoryItem` database table contains the `UsrRepairItemType` column of the `nvarchar(2)` type.

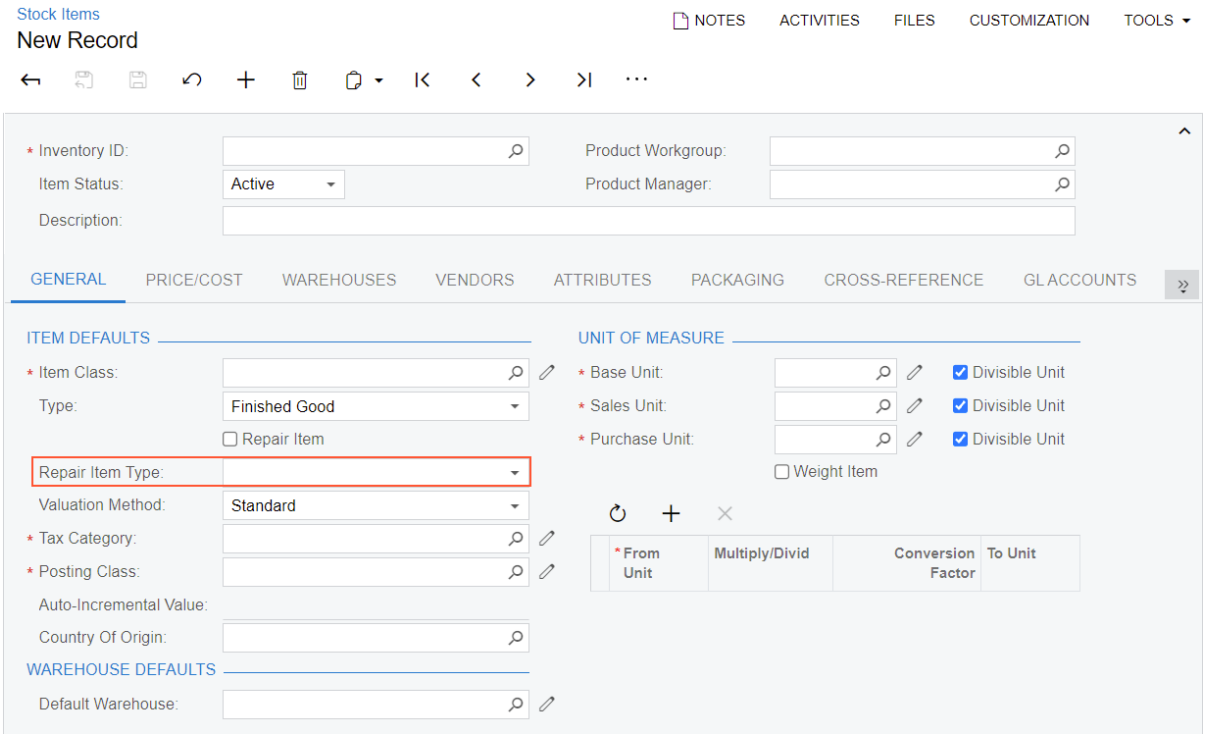


Figure: The Repair Item Type box

Related Links

- *To Add a Box for a Data Field*
- *Changes in Webpages (ASPX)*

Step 1.1.6: Making the Custom Field Conditionally Available (with RowSelected)

In this step, you will learn how to work with a custom control that is available only conditionally. The **Repair Item Type** box should be unavailable on the *Stock Items* (IN202500) form unless the **Repair Item** check box is selected.

Changes in the DAC

You will make the **Repair Item Type** box unavailable by default by setting the `Enabled` property of the `PXUIField` attribute to `false`.

The user can edit a field value in the UI if the control for the field is available on the form. You can make a control available or unavailable by specifying the `Enabled` parameter of the `PXUIField` attribute in the data access class, or by specifying the `Enabled` property of the control in the `.aspx` page. Generally, for a data field that should be unconditionally unavailable in the UI, you set the `Enabled` property of the control to `False` in the `.aspx` page. For example, you use this approach for calculated fields with totals that users will never edit. As the result, the UI controls are unconditionally unavailable, regardless of the logic implemented in event handlers.

Changes in the Graph

You will add the `RowSelected` event handler to make the box become available when a user selects the **Repair Item** check box. You will use the `RowSelected` event handler because it is intended for the implementation of UI presentation logic. In the `RowSelected` event handler, you will do the following:

- Access the `UsrRepairItem` extension field of the `InventoryItem` DAC by invoking the `GetExtension` method on the cache. (For details on this method, see *Access to a Custom Field* in the documentation.)
- Use the `PXUIFieldAttribute.SetEnabled<>()` method to change the value of the `Enabled` property of the `PXUIField` attribute of the `UsrRepairItemType` extension field.

You will use the Customization Project Editor to create the graph extension, and you will edit the business logic in Visual Studio.

Changes in the ASPX Page

To make the **Repair Item Type** box available if a user has selected the **Repair Item** check box and then the **Repair Item** check box has lost input focus, you will set the `CommitChanges` property of this control to `True`. If you do not set the `CommitChanges` property to `True`, then when a user selects the **Repair Item** check box, the **Repair Item Type** box will become available only when the stock item record is saved or when the value of another field with the `CommitChanges` property set to `True` is changed. For details about the `CommitChanges` property, see *Use of the CommitChanges Property of Boxes* in the documentation.

Instructions for Adding the UI Presentation Logic

To add this presentation logic, perform the following steps:

1. In Visual Studio, in the `InventoryItemExtensions.cs` file, make the **Repair Item Type** box unavailable by default by setting the `Enabled` property of the `PXUIField` attribute of the `UsrRepairItemType` field to `false`, as the following code shows.

```
[PXUIField(DisplayName = "Repair Item Type", Enabled = false)]
public string UsrRepairItemType { get; set; }
public abstract class usrRepairItemType :
  PX.Data.BQL.BqlString.Field<usrRepairItemType>
{ }
```

2. Add the event handler as follows:
   a. Open the Screen Editor for the *Stock Items* (IN202500) form.
   b. In the control tree, open the **Repair Item** node, and click the **Events** tab.
   c. On the tab, in the event list, click the row with the `RowSelected` event. Notice that the **Handled in Source** check box is cleared for the `RowSelected` event of the `InventoryItem` DAC, which means that the Acumatica ERP source code does not include an implementation of this event handler. However, to not override possible future implementations of this event handler in the source code of Acumatica ERP, you will extend the base method with your own code.

d. On the table toolbar, click **Add Handler > Keep Base Method** to create a `RowSelected` event handler for the selected DAC, as shown in the following screenshot.

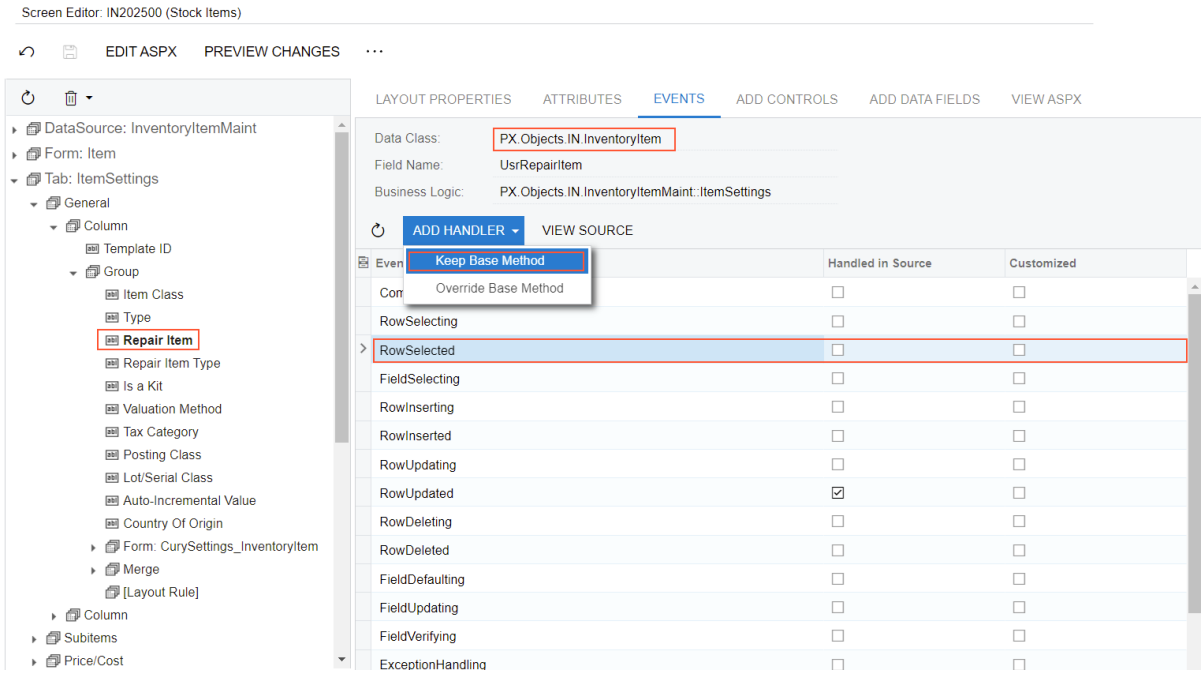


Figure: The generation of the event handler

The platform creates a template for the `InventoryItem_RowSelected` event handler in the extension for the `InventoryItemMaint` graph. The platform opens the `InventoryItemMaint` *Code* item in the Code Editor of the Customization Project Editor.

e. To move the generated template to the extension library, click **Move to Extension Lib** on the toolbar of the Code Editor.

> Alternatively, you can add the `InventoryItemMaint.cs` file in Visual Studio and add the event handler in the file.

3. In Visual Studio, adjust the graph extension as follows:

a. In the `InventoryItemMaint.cs` file, use Acuminator to suppress the *PX1016* error in a comment as you have done for the DAC extension in the first step of this lesson.

b. Use Acuminator to change the signature of the event to a generic one.

> For details about generic event handlers, see *Types of Graph Event Handlers*.

c. Remove unnecessary `using` directives.

> While Acumatica Customization Platform creates an extension for an original class of Acumatica ERP, the platform inserts all the `using` directives from the original class to the extension. Some `using` directives are unused in the customization code and can be removed.

d. Redefine the `RowSelected` event handler as follows.

```
protected void _(Events.RowSelected<InventoryItem> e)
{
    InventoryItem item = e.Row;
```

```
            InventoryItemExt itemExt = PXCache<InventoryItem>.
                GetExtension<InventoryItemExt>(item);
            bool enableFields = itemExt != null &&
                itemExt.UsrRepairItem == true;
            //Make the Repair Item Type box available
            //when the Repair Item check box is selected.
            PXUIFieldAttribute.SetEnabled<InventoryItemExt.usrRepairItemType>(
                e.Cache, e.Row, enableFields);
        }
```

The code above makes the `usrRepairItemType` custom field available for editing if the value of the `UsrRepairItem` field of the row in `PXCache` is `true`. Otherwise, it makes the custom field unavailable.

e. Build the project.

4. Open the Screen Editor for the *Stock Items* (IN202500) form.
5. Set the `CommitChanges` property to `True` for the `UsrRepairItem` data field, as the following screenshot shows.

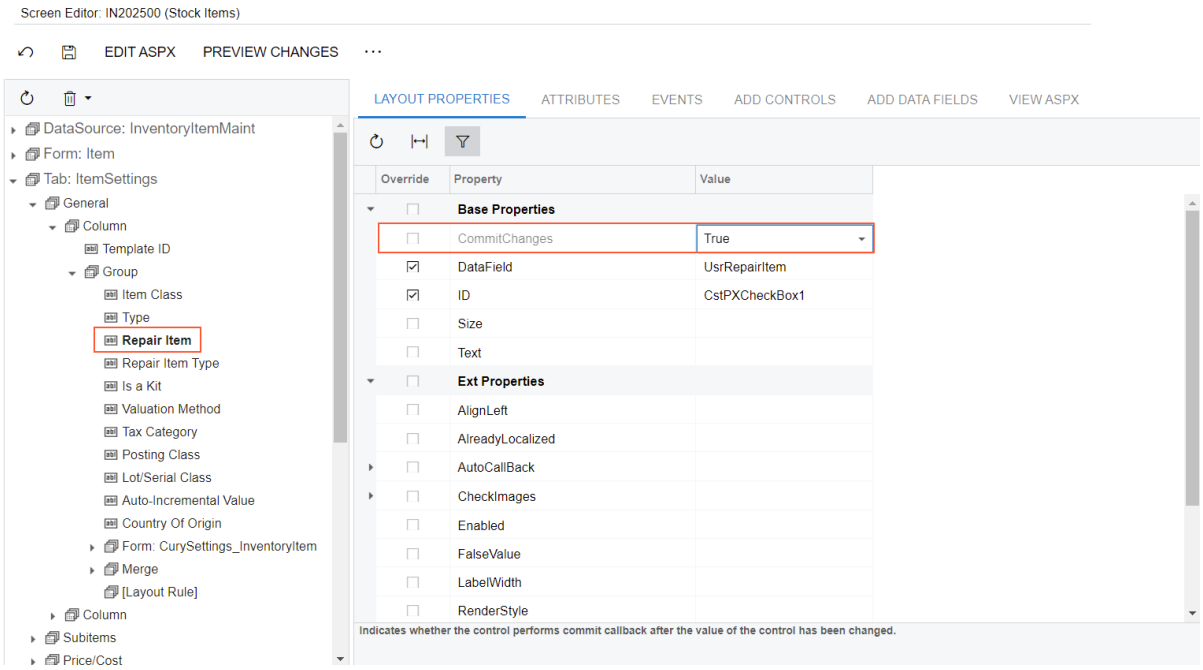


Figure: The CommitChanges property

6. Click **Save** to save the changes to the customization project.
7. Update the customization project with the changes you have made in this lesson, and publish the project.

Related Links

- *PXUIFieldAttribute Class*
- *Use of the CommitChanges Property of Boxes*
- *RowSelected Event*
- *Access to a Custom Field*
- *Configuration of the User Interface in Code*

Step 1.1.7: Creating Repair Items

Now you will create the *BAT3310*, *BAT3310EX*, and *BCOV3310* repair item records, which you will need later.

To do this, perform the following actions:

1. On the *Stock Items* (IN202500) form, click **Add New Record** on the form toolbar.
2. In the Summary area, specify the following settings:
   - **Inventory ID**: `BAT3310`
   - **Description**: `Battery for Nokia 3310`
3. Notice that the **Repair Item Type** box in the **Item Defaults** section of the **General** tab is unavailable because the **Repair Item** check box is cleared.
4. In the **Item Defaults** section of the **General** tab, specify the following settings:
   - **Item Class**: *Stock Item*
   - **Repair Item**: Selected

     Notice that once you selected the check box, the **Repair Item Type** box becomes available.
   - **Repair Item Type**: *Battery*
5. In the **Warehouse Defaults** section of the **General** tab, select *MAIN* in the **Default Warehouse** box.
6. In the **Price Management** section of the **Price/Cost** tab, enter `20` as the **Default Price**.
7. On the form toolbar, click **Save** to save the record in the database.
8. Repeat the previous instructions to create repair items with the following values.

UI Element (Location)	First Record	Second Record
Inventory ID (Summary area)	`BAT3310EX`	`BCOV3310`
Description (Summary area)	`Extended Battery for Nokia 3310`	`Back cover for Nokia 3310`
Item Class (**Item Defaults** section of the **General** tab)	*Stock Item*	*Stock Item*
Repair Item (**Item Defaults** section of the **General** tab)	Selected	Selected
Repair Item Type (**Item Defaults** section of the **General** tab)	*Battery*	*Back Cover*
Default Warehouse (**Warehouse Defaults** section of the **General** tab)	*MAIN*	*MAIN*
Default Price (**Price Management** section of the **Price/Cost** tab)	`30`	`10`

To ensure that the `InventoryItem` table of the database contains the custom columns and the data entered, you can review the table by using SQL Server Management Studio.

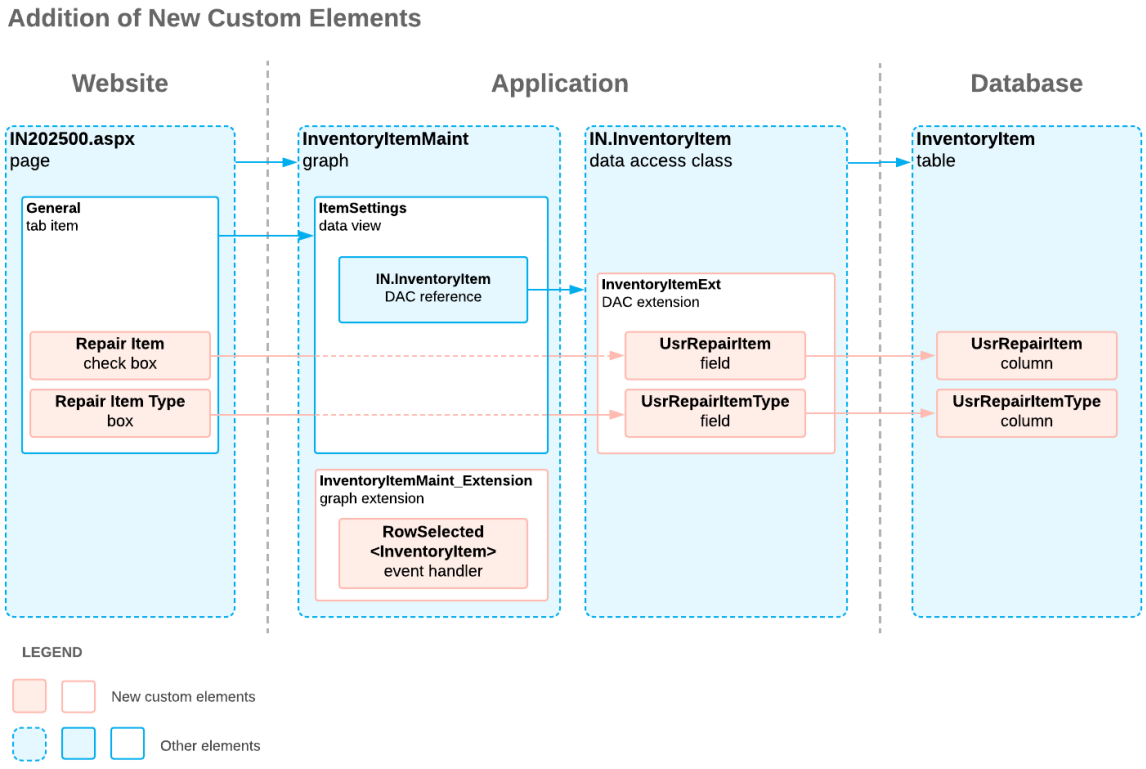
Lesson Summary

In this lesson, you have learned how to create a control so that you can display on a form a custom field bound to the database. To implement this customization, you have learned how to add the necessary modifications to a customization project and how to publish the project to apply the changes to the system.

As you have completed the lesson, you have added the following elements to the *PhoneRepairShop* customization project:

- Two column definitions in the `InventoryItem` table of the database.
- Two custom field declarations in the extension of the `IN.InventoryItem` data access class (in the `PhoneRepairShop_Code` extension library).
- Two controls to display the custom fields on the *Stock Items* (IN202500) form.
- One custom event handler, which you have added to the `InventoryItemMaint` graph. You have used the `RowSelected` event handler to configure the UI presentation logic.

The following diagram shows the results of the lesson.



Review Questions

1. Select all objects that together make up the minimum set of objects that are customized when you add a control for a custom field with the `DBTableColumn` storage type to an Acumatica ERP form.
   a. The database table
   b. The data access class

   c. The graph
   d. The *.aspx* page
2. Suppose that you have to add a control for a custom field to a form of Acumatica ERP. Select the tool of the Customization Project Editor that is designed to do this.
   a. Customization Menu
   b. Data Class Editor
   c. Screen Editor
   d. Project Items Editor
   e. Project XML Editor
3. Which event handler would you use to configure the UI presentation logic?
   a. `FieldUpdated`
   b. `RowUpdated`
   c. `RowSelected`

Answer Key

1. a, b, d
2. c
3. c

Additional Information: DAC Extensions

You can not only create custom fields but also customize existing Acumatica ERP fields and include your customizations in DAC extensions of different levels. These scenarios are outside of the scope of this course but may be useful to some readers.

Customization of Field Attributes

In addition to adding custom fields in DAC extensions, you can customize existing fields by changing the attributes assigned to the fields in Acumatica ERP DACs. For more information about the customization of field attributes, see *Customization of Field Attributes in DAC Extensions*.

Different Levels of DAC Extensions

The Acumatica Customization Platform supports multilevel extensions, which are required when you develop off-the-shelf software that is distributed in multiple editions. Precompiled extensions provide a measure of protection for your source code and intellectual property.

You can use multilevel extensions to develop applications that extend the functionality of Acumatica ERP or other software based on Acumatica Framework in multiple markets (that is, specified categories of potential client organizations). You may have a base extension that contains the solution common to all markets as well as multiple market-specific extensions. Every market-specific solution is deployed along with the base extension. Moreover, you can later customize deployed extensions for the end user by using DAC and graph extensions.

For additional details about multilevel extensions, see *DAC Extensions* and *Graph Extensions*.

Lesson 1.2: Configuring the UI—Self-Guided Exercise

Consultants and managers of the Smart Fix company would like that all Acumatica ERP forms that are necessary for their work be available in one place, in the **Phone Repair Shop** workspace of Acumatica ERP. This workspace was added as part of the *T200 Maintenance Forms* training course. (If you did not complete this course, the creation of the workspace is part of the customization project that have been published in preparation to take the current course.)

In this lesson, which you will complete on your own, you will add the *Stock Items* (IN202500) form to the **Phone Repair Shop** workspace of Acumatica ERP. You will also include this change to the workspace in the customization project.

Tips

As you add the *Stock Items* form to the **Phone Repair Shop** workspace of Acumatica ERP, use the following tips:

- Use Menu Editing mode. For details about Menu Editing mode, see *Menu Editing Mode*.
- Add a link to the *Stock Items* form to the **Profiles** category of the **Phone Repair Shop** workspace and make the form be available in the quick menu of the workspace. For details about this process, see *User Interface: Workspaces* and *User Interface: To Configure a Workspace*.

Result

As the result, the **Phone Repair Shop** workspace will look as shown in the following screenshot.

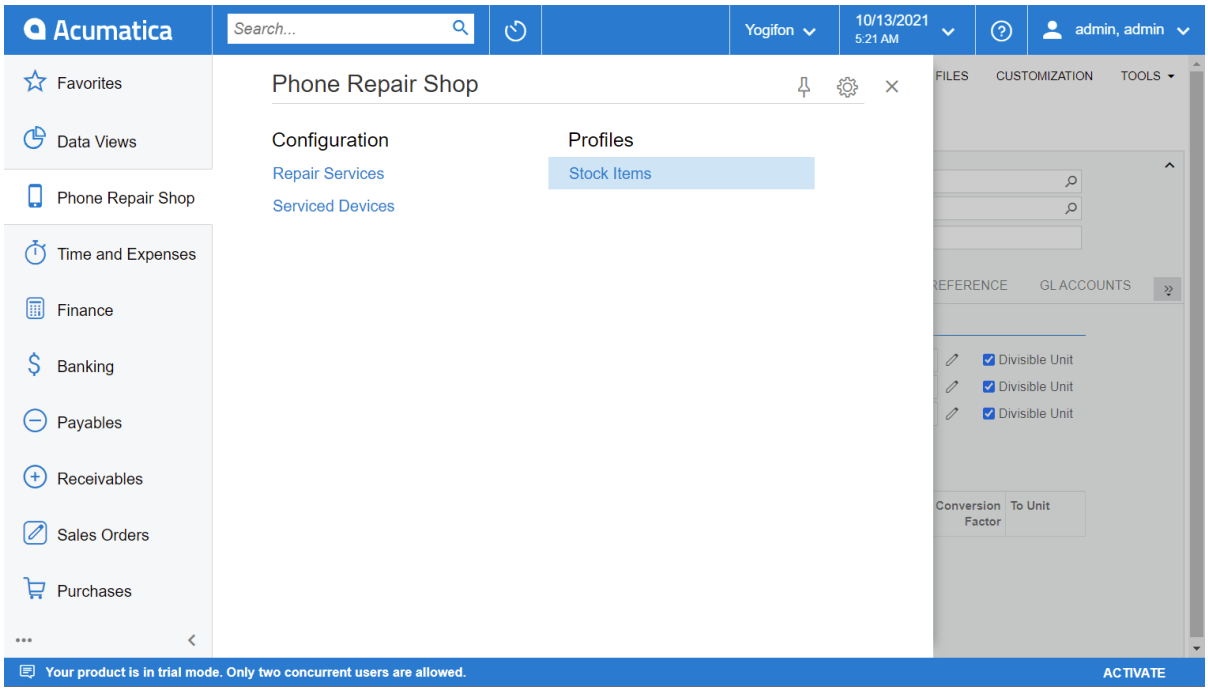


Figure: Stock Items form in the Phone Repair Shop workspace

The customization project will include the *SiteMapNode* item for the *Stock Items* form, as shown in the following screenshot.

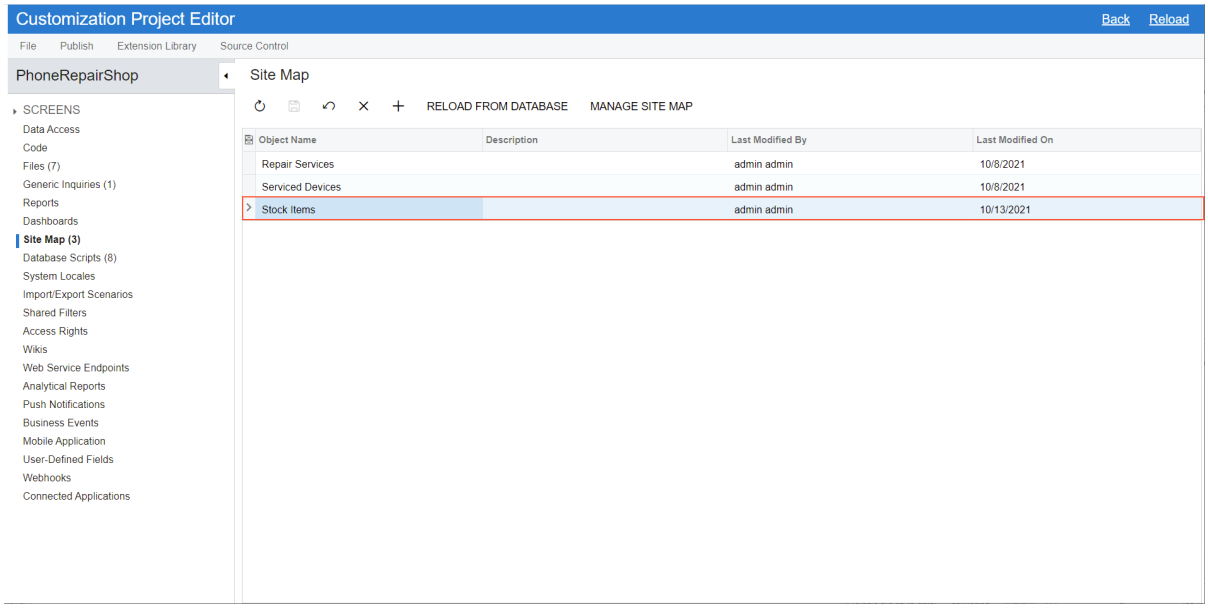


Figure: SiteMapNode item for the Stock Items form

Part 2: Master-Detail Relationship and Business Logic (Services and Prices Form)

To simplify the creation of the work orders for repairs, the Smart Fix company needs to have a custom Acumatica ERP form on which users can maintain the price of the selected repair service for the selected device. For this purpose, you will create the Services and Prices (RS203000) custom maintenance form, which is described in *Company Story and Customization Description*. In this part, you will create and configure the Summary area and the **Repair Items** tab of the form, for which you will implement the master-detail relationship and basic business logic. You will add other tabs in *Part 4: Calculations and Insertion of a Default Record (Services and Prices Form)*.

After you complete the lessons of this part, you will be able to test the functionality of the **Repair Items** tab of the Services and Prices form.

Lesson 2.1: Defining a Master-Detail Relationship

In this lesson, you will create the custom Services and Prices (RS203000) form, design its **Repair Items** tab, and define the master-detail relationship between the records displayed in the Summary area of the form and the records displayed on the **Repair Items** tab. For the particular service and device selected in the Summary area of the form, the **Repair Items** tab will display the records of the stock items that are repair items.

A manager of the Smart Fix company will select a service and device in the Summary area of the form and will add or remove repair items for this service and device on the **Repair Items** tab of the form. For each repair item on the tab, the manager can select the type of the repair item, the stock item, whether the repair item is required, and whether it should be used by default if multiple repair items of the same type can be used for the selected service and device.

You will implement the business logic for the Summary area and the **Repair Items** tab of the form in *Lesson 2.2: Defining the Business Logic* and *Lesson 4.1: Calculating Field Values*.

Description of Form Elements That Are Created in This Lesson

The Summary area of the form will contain the following elements:

- **Service**: A box in which the user can select one of the services that has been entered on the custom Repair Services (RS201000) form. You created this form in the *T200 Maintenance Forms* training course or published a customization project that includes it while completing the prerequisites of the current course.
- **Device**: A box in which the user can select one of the devices that has been entered on the custom Serviced Devices (RS202000) form, which was also created in the *T200 Maintenance Forms* training course or included in the customization project that you published before completing the current course.
- **Approximate Price**: A read-only box that displays the price of the selected service for the selected device, which is an approximate price for the corresponding work order. In this lesson, you will only specify the default value for this field; the calculation of the field value will be defined in *Lesson 4.1: Calculating Field Values*.

The **Repair Items** tab will contain the following columns:

- **Repair Item Type**: A drop-down list box that contains the type of the repair item, which is one of the following:
  - *Battery*
  - *Screen*
  - *Screen Cover*

  - *Back Cover*
  - *Motherboard*
- **Required**: A check box that indicates (if selected) that this repair item type is required
- **Inventory ID**: A box in which the user can select one of the stock items defined on the *Stock Items* (IN202500) form
- **Description**: A read-only box with the description of the stock item selected in the **Inventory ID** column
- **Price**: The price of the repair item
- **Default**: A check box that indicates (if selected) that this repair item should be used by default if multiple repair items of the same repair item type can be used for the selected service and device

The following screenshot shows the form as it will look after you complete this lesson.

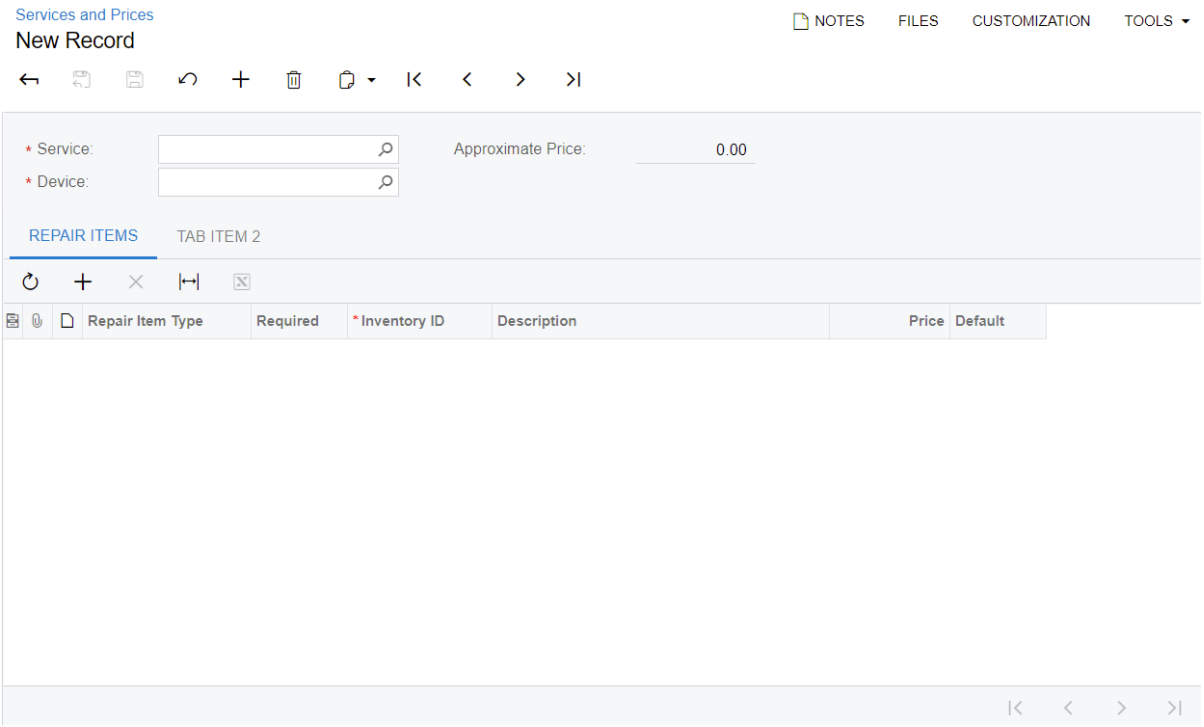
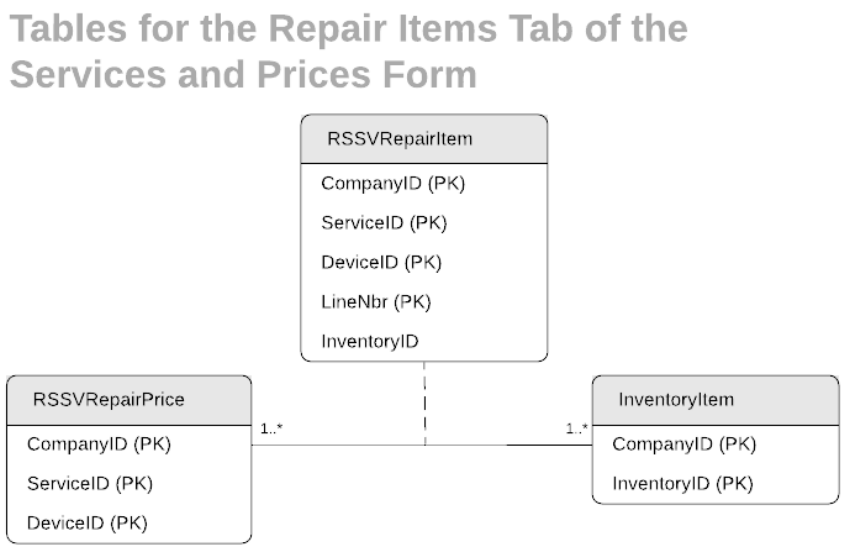


Figure: Services and Prices form

Relationships Between Database Tables

The repair prices for particular services and devices (represented by the records of the `RSSVRepairPrice` database table) and the stock items (represented by the records of the `InventoryItem` database table) will have a many-to-many relationship: The same stock item can be included in the repair prices for multiple services and devices, and a repair price for a particular service and device can include multiple stock items. The many-to-many links between records will be stored in the separate `RSSVRepairItem` custom table (see the following diagram). For the Services and Prices form, `RSSVRepairItem` records will be details for the `RSSVRepairPrice` class. Once a repair price (master record) is deleted, the links to stock items (detail records) should also be deleted from the database.

The following diagram illustrates the relationships between the database tables that will be used in this lesson.



Lesson Objectives

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

- Define the master-detail relationship between data
- Implement automatic numbering of detail records

Step 2.1.1: Creating the Form—Self-Guided Exercise

In this step, you will create the Services and Prices (RS203000) form on your own. Although this is a self-guided exercise, you can use the details and suggestions in this topic as you create the form. The creation of a form is described in detail in the *T200 Maintenance Forms* training course.

If you are using the Customization Project Editor to complete the self-guided exercise, you can perform the following instructions:

1. Create the form and graph as follows:
   a. On the toolbar of the Customized Screens page of the Customization Project Editor, click **Create New Screen**.
   b. In the **Create New Screen** dialog box, which opens, specify the following values:
      - **Screen ID**: `RS.20.30.00`
      - **Graph Name**: `RSSVRepairPriceMaint`
      - **Graph Namespace**: `PhoneRepairShop`
      - **Page Title**: `Services and Prices`
      - **Template**: *FormTab*
   c. Move the generated `RSSVRepairPriceMaint` graph to the extension library.
2. Create and configure the DACs by doing the following:
   a. In Code Editor, generate the `RSSVRepairPrice` and `RSSVRepairItem` DACs, and move them to the extension library.
   b. Configure the `RSSVRepairPrice` and `RSSVRepairItem` DACs in Visual Studio as follows:

- `RSSVRepairPrice`: Specify the system attributes and other attributes as shown in the code fragments below:
  - For the DAC:

```
    [PXCacheName("Repair Price")]
```

  - For the system fields:

```
        #region CreatedDateTime
        [PXDBCreatedDateTime()]
        public virtual DateTime? CreatedDateTime { get; set; }
        public abstract class createdDateTime :
            PX.Data.BQL.BqlDateTime.Field<createdDateTime>
        { }
        #endregion

        #region CreatedByID
        [PXDBCreatedByID()]
        public virtual Guid? CreatedByID { get; set; }
        public abstract class createdByID :
            PX.Data.BQL.BqlGuid.Field<createdByID>
        { }
        #endregion

        #region CreatedByScreenID
        [PXDBCreatedByScreenID()]
        public virtual string CreatedByScreenID { get; set; }
        public abstract class createdByScreenID :
            PX.Data.BQL.BqlString.Field<createdByScreenID>
        { }
        #endregion

        #region LastModifiedDateTime
        [PXDBLastModifiedDateTime()]
        public virtual DateTime? LastModifiedDateTime { get; set; }
        public abstract class lastModifiedDateTime :
            PX.Data.BQL.BqlDateTime.Field<lastModifiedDateTime>
        { }
        #endregion

        #region LastModifiedByID
        [PXDBLastModifiedByID()]
        public virtual Guid? LastModifiedByID { get; set; }
        public abstract class lastModifiedByID :
            PX.Data.BQL.BqlGuid.Field<lastModifiedByID>
        { }
        #endregion

        #region LastModifiedByScreenID
        [PXDBLastModifiedByScreenID()]
        public virtual string LastModifiedByScreenID { get; set; }
        public abstract class lastModifiedByScreenID :
            PX.Data.BQL.BqlString.Field<lastModifiedByScreenID>
        { }
        #endregion

        #region Tstamp
        [PXDBTimestamp()]
```

```
        public virtual byte[] Tstamp { get; set; }
        public abstract class tstamp :
            PX.Data.BQL.BqlByteArray.Field<tstamp> { }
        #endregion

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

- For the `ServiceID` field:

```
        #region ServiceID
        [PXDBInt(IsKey = true)]
        [PXDefault]
        [PXUIField(DisplayName = "Service", Required = true)]
        [PXSelector(typeof(Search<RSSVRepairService.serviceID>),
            typeof(RSSVRepairService.serviceCD),
            typeof(RSSVRepairService.description),
            SubstituteKey = typeof(RSSVRepairService.serviceCD),
            DescriptionField = typeof(RSSVRepairService.description))]
        public virtual int? ServiceID { get; set; }
        public abstract class serviceID :
            PX.Data.BQL.BqlInt.Field<serviceID> { }
        #endregion
```

- For the `DeviceID` field:

```
        #region DeviceID
        [PXDBInt(IsKey = true)]
        [PXDefault]
        [PXUIField(DisplayName = "Device", Required = true)]
        [PXSelector(typeof(Search<RSSVDevice.deviceID>),
            typeof(RSSVDevice.deviceCD),
            typeof(RSSVDevice.description),
            SubstituteKey = typeof(RSSVDevice.deviceCD),
            DescriptionField = typeof(RSSVDevice.description))]
        public virtual int? DeviceID { get; set; }
        public abstract class deviceID :
            PX.Data.BQL.BqlInt.Field<deviceID> { }
        #endregion
```

The `PXSelector` attributes added to the `ServiceID` and `DeviceID` fields must have the `SubstituteKey` property set to `RSSVRepairService.serviceCD` and `RSSVDevice.deviceCD`, respectively. Whenever you need to create a lookup control based on a DAC that contains a CD key field and a numeric ID field mapped to identity column, you must use the ID field in the BQL query for the lookup and substitute the ID field with the CD field.

- For the `Price` field:

```
        #region Price
        [PXDBDecimal()]
        [PXDefault(TypeCode.Decimal, "0.0")]
        [PXUIField(DisplayName = "Approximate Price", Enabled = false)]
        public virtual Decimal? Price { get; set; }
```

```
        public abstract class price : PX.Data.BQL.BqlDecimal.Field<price> { }
        #endregion
```

You will define other fields of this DAC in the remaining steps of this lesson.

- `RSSVRepairItem`: Specify the system attributes and other attributes as shown in the code fragments below:
  - For the DAC:

```
    [PXCacheName("Repair Item")]
```

  - For the system fields add the same attribute as for the `RSSVRepairPrice` DAC
  - For the `InventoryID` field:

```
        #region InventoryID
        [Inventory]
        [PXDefault]
        public virtual int? InventoryID { get; set; }
        public abstract class inventoryID :
            PX.Data.BQL.BqlInt.Field<inventoryID> { }
        #endregion
```

    You use the `Inventory` attribute defined in the `PX.Objects.IN` namespace in the Acumatica ERP code. This attribute, which is defined from the `PXDimensionSelector` attribute, defines a selector for the inventory ID. This selector displays only the inventory items for which the current user has access rights and that do not have the *Inactive* or *Marked for Deletion* status.

You can find the attribute that can be used for a selector control that retrieves the data from an Acumatica ERP database table by investigating the source code of a similar selector in the application. For example, if you want to find an attribute that can be used with the inventory ID selector in a document detail, you can do the following:

a. On the *Shipments* (SO302000) form, click **Customization > Inspect Element** and click the header of the **Inventory ID** column on the **Details** tab.

b. In the **Element Properties** dialog box, notice the data field name (`InventoryID`) and click **Actions > View Data Class Source**.

c. In the Source Code Browser, find the needed attribute of the `InventoryID` field, which is shown in the following screenshot.

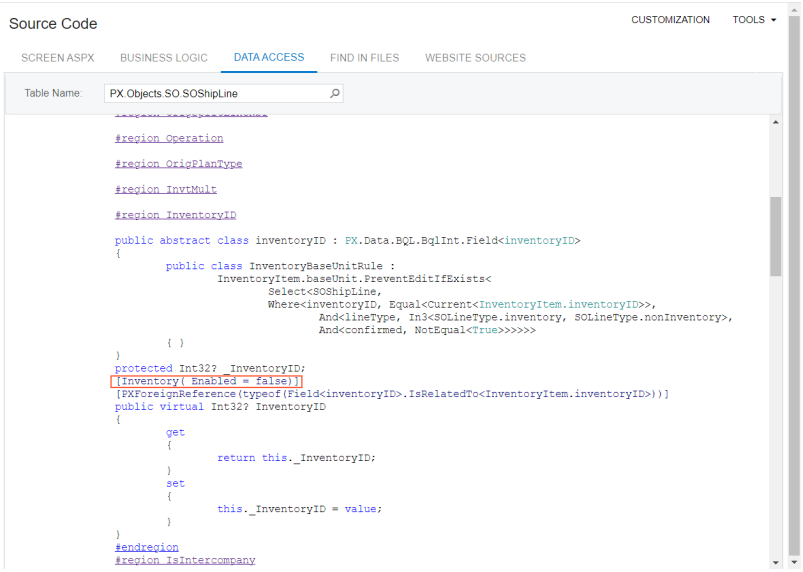


Figure: `InventoryID` attribute

- For the `BasePrice` field:

```
#region BasePrice
[PXDBDecimal()]
[PXDefault(TypeCode.Decimal, "0.0")]
[PXUIField(DisplayName = "Price")]
public virtual Decimal? BasePrice { get; set; }
public abstract class basePrice :
    PX.Data.BQL.BqlDecimal.Field<basePrice> { }
#endregion
```

- For the `Required` field:

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

- For the `IsDefault` field:

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

- For the `RepairItemType` field

```
        #region RepairItemType
        [PXDBString(2, IsFixed = true)]
        [PXStringList(
            new string[]
            {
                RepairItemTypeConstants.Battery,
                RepairItemTypeConstants.Screen,
                RepairItemTypeConstants.ScreenCover,
                RepairItemTypeConstants.BackCover,
                RepairItemTypeConstants.Motherboard
            },
            new string[]
            {
                Messages.Battery,
                Messages.Screen,
                Messages.ScreenCover,
                Messages.BackCover,
                Messages.Motherboard
            })]
        [PXUIField(DisplayName = "Repair Item Type")]
        public virtual string RepairItemType { get; set; }
        public abstract class repairItemType :
            PX.Data.BQL.BqlString.Field<repairItemType> { }
        #endregion
```

You will define other fields of this DAC in the remaining steps of this lesson.

3. Configure the `RSSVRepairPriceMaint` graph: Define a data view in the generated graph and make the full list of standard system actions available by specifying the second generic type parameter in the base `PXGraph` class, as the following code shows.

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

namespace PhoneRepairShop
{
    public class RSSVRepairPriceMaint :
        PXGraph<RSSVRepairPriceMaint, RSSVRepairPrice>
    {
        #region Data Views
        public SelectFrom<RSSVRepairPrice>.View RepairPrices;
        #endregion
    }
```

```
}
```

> The fluent BQL classes, which are used for the definition of the data view, are available in the `PX.Data.BQL.Fluent` namespace.

4. Build the project in Visual Studio and publish the customization project.
5. Configure the `RS203000.aspx` page as follows:
   a. Set the `PrimaryView` property value of the `DataSource` control to `RepairPrices`.

      Before you did this, the `PrimaryView` property value was *MasterView*. The `MasterView` view has been removed from the `RSSVRepairPriceMaint` graph, so you cannot open the form in the Screen Editor to edit its properties. Instead, you can click **Files** in the navigation pane of the Customization Project Editor and open the `RS203000.aspx` file, which is present in the list of files on the Custom Files page.

   b. Set the `DataMember` property value of the `Form` control to `RepairPrices`.

      Now the `RS203000.aspx` file contains only valid values, so you can proceed with editing the form by using the Screen Editor.

   c. Add the `ServiceID`, `DeviceID`, and `Price` fields to the Summary area of the form.

   d. Organize the layout of the Summary area of the form by putting the **Service ID** and **Device ID** boxes in the first column and the **Price** box in the second column, as shown in the following screenshot.

      > To split the controls into two columns, you can use the approach described in *Step 2.3.2: Configure the Layout* in the *T200 Maintenance Forms* training course.

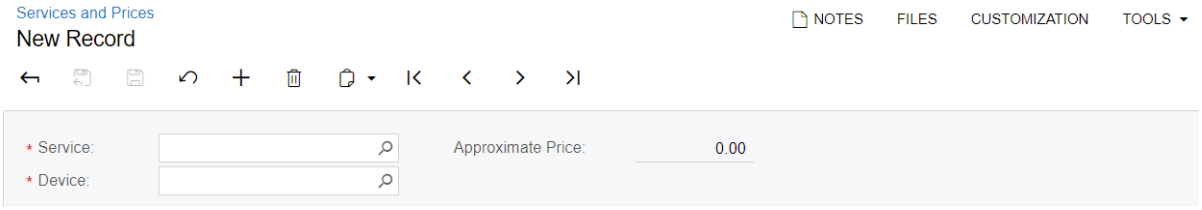
      

      Figure: Two columns in the Summary area

   e. Clear the `Height` property value of the `Form: RepairPrices` element.

   f. For the row, specify the following values:

      - `ControlSize`: `M`
      - `LabelsWidth`: `S`

   g. For the right column, specify the following values:

      - `ControlSize`: `M`
      - `LabelsWidth`: `SM`

6. Publish the customization.
7. Include a link to the form in the **Configuration** group of the **Phone Repair Shop** namespace, and make it available in the quick menu.
8. As a substitute form, use the generic inquiry from the `ServicesAndPrices.xml` file, which is provided with this course. For details about how to add a substitute form, see *Lesson 2.5: Create a Substitute Form* in the *T200 Maintenance Forms* training course.

> The generic inquiry is provided in the `Customization\T210\SourceFiles\ListAsEntryPoint` folder, which you have downloaded from Acumatica GitHub.

9. Add the Services and Prices generic inquiry to the customization project and update the *SiteMapNode* item for the Services and Prices form in the customization project.

Step 2.1.2: Defining the Master-Detail Relationship Between Data (with PXParent and PXDBDefault)

In this step, you will set up the master-detail relationship between the `RSSVRepairPrice` and `RSSVRepairItem` DACs and define the detail data view, which selects the items that are used for a particular repair service provided for a particular device.

Changes to the Detail DAC

To set up the master-detail relationship between the `RSSVRepairPrice` and `RSSVRepairItem` data access classes, you will add the `PXDBDefault` and `PXParent` attributes to the key fields of the detail class, which is `RSSVRepairItem`.

The `PXDBDefault` attribute specifies the value that is inserted into a field of the detail records from the field of the current (at runtime) master data record.

The `PXParent` attribute defines the master-detail relationship between the data access classes. In particular, the attribute enables cascading deletion of the detail records when the master data record is deleted. When a price record is deleted, the corresponding detail records that match the specified query will also be deleted. The `PXParent` attribute can be added to any data field of the detail class in the master-detail relationship. However, we recommend that you add it to the declaration of the first foreign key.

Changes to the Graph

You will use fluent BQL to define the detail data view.

To retrieve data from the database, Acumatica Framework converts the fluent BQL query to an SQL command and executes this command. You can trace executed SQL commands on the *Request Profiler* (SM205070) form. For more information, see *To Validate a BQL Statement* and *To Measure the Execution Time of a BQL Statement*.

Instructions for Defining the Master-Detail Relationship

To define the master-detail relationship, do the following:

1. In the `RSSVRepairItem` DAC, add the attributes to the `ServiceID` and `DeviceID` fields as follows:
   a. Add the `PXDBInt` and `PXDBDefault` attributes to the `ServiceID` field, as shown below.

```
#region ServiceID
[PXDBInt(IsKey = true)]
[PXDBDefault(typeof(RSSVRepairPrice.serviceID))]
public virtual int? ServiceID { get; set; }
public abstract class serviceID : PX.Data.BQL.BqlInt.Field<serviceID> { }
#endregion
```

   The `PXDBDefault` attribute inserts the default value into the key field of the detail class, which is the key to the master record. For each new `RSSVRepairPrice` object, the framework inserts the ID of the current service into the `RSSVRepairItem.ServiceID` data field. The field isn't intended for editing in the UI; thus, it does not have the `PXUIField` attribute.

   b. Add the `PXParent` attribute to the `ServiceID` field as follows.

```
#region ServiceID
[PXDBInt(IsKey = true)]
[PXDBDefault(typeof(RSSVRepairPrice.serviceID))]
[PXParent(typeof(SelectFrom<RSSVRepairPrice>.
    Where<RSSVRepairPrice.serviceID.
        IsEqual<RSSVRepairItem.serviceID.FromCurrent>.
        And<RSSVRepairPrice.deviceID.
        IsEqual<RSSVRepairItem.deviceID.FromCurrent>>>))]
public virtual int? ServiceID { get; set; }
public abstract class serviceID : PX.Data.BQL.BqlInt.Field<serviceID> { }
#endregion
```

The `PXParent` attribute specifies the master-detail relationship between classes.

> Make sure you have added the `using PX.Data.BQL.Fluent;` directive.

c. Add the `PXDBInt` and `PXDBDefault` attributes to the `DeviceID` field, as shown below.

```
#region DeviceID
[PXDBInt(IsKey = true)]
[PXDBDefault(typeof(RSSVRepairPrice.deviceID))]
public virtual int? DeviceID { get; set; }
public abstract class deviceID : PX.Data.BQL.BqlInt.Field<deviceID> { }
#endregion
```

You do not need to specify the `PXParent` attribute for the second key field.

2. In the `RSSVRepairPriceMaint` graph, define the detail data view, as the following code shows.

```
public SelectFrom<RSSVRepairItem>.
    Where<RSSVRepairItem.serviceID.
        IsEqual<RSSVRepairPrice.serviceID.FromCurrent>.
    And<RSSVRepairItem.deviceID.
        IsEqual<RSSVRepairPrice.deviceID.FromCurrent>>>.View
    RepairItems;
```

In the `RepairItems` data view, you pass the device ID and service ID to the query by using the `FromCurrent` parameter of the fluent BQL statement. When the framework executes the statement, it takes the value from the `Current` property of the `PXCache` object that holds `RSSVRepairPrice` objects in this graph and retrieves records that match the query.

> The detail data view must be declared after the master data view.

3. Build the project.

Related Links

- *Master-Detail Relationship Between Data with PXDBDefault and PXParent*
- *PXDBDefaultAttribute Class*
- *PXParentAttribute Class*
- *Creating Fluent BQL Queries*
- *Selection of the Linked Data Through the Current Property*

Step 2.1.3: Numbering Detail Records (with PXLineNbr)

Now you will implement the numbering of repair items by using the predefined `PXLineNbr` attribute. You will use the line number as a key field of a repair item. (These numbers will be maintained internally and not made visible on the form.) You need one more key (besides the service ID and device ID) for the repair items and cannot use as a key the inventory ID because users can add multiple repair items with the same service ID, device ID, and inventory ID to the **Repair Items** tab of the Services and Prices (RS203000) form.

The `PXLineNbr` attribute uses the value of the specified field from the parent data record to keep the last assigned number, increments this value, and assigns the incremented value to the `LineNbr` field of the new detail data records. The `PXLineNbr` attribute does not work without the `PXParent` attribute being assigned to a field of the same DAC.

Numbering Detail Records

Do the following to implement the numbering of detail records:

1. Add the attributes shown in the following code to the `LineNbr` field of the `RSSVRepairItem` DAC.

```
#region LineNbr
[PXDBInt(IsKey = true)]
[PXLineNbr(typeof(RSSVRepairPrice.repairItemLineCntr))]
[PXUIField(DisplayName = "Line Nbr.", Visible = false)]
public virtual int? LineNbr { get; set; }
public abstract class lineNbr : PX.Data.BQL.BqlInt.Field<lineNbr> { }
#endregion
```

   You set the `Visible` property of the `PXUIField` attribute to `false` so that the field is not displayed in the UI by default.

2. To set the starting number, for the `RepairItemLineCntr` field of the `RSSVRepairPrice` DAC, specify the following `PXDefault` attribute, and add the `PXDBInt` attribute. (See the following code.)

```
#region RepairItemLineCntr
[PXDBInt()]
[PXDefault(0)]
public virtual Int32? RepairItemLineCntr { get; set; }
public abstract class repairItemLineCntr :
    PX.Data.BQL.BqlInt.Field<repairItemLineCntr> { }
#endregion
```

   In this code, you specify `0` as the default value, so the first `LineNbr` value assigned will be 1.

3. Rebuild the project.

Related Links

- *PXLineNbrAttribute Class*

Step 2.1.4: Creating Controls on the Form

In this step, you will bind the data view to the grid that displays the detail data on the Services and Prices (RS203000) form and create controls for the grid.

Creating Controls

Create the controls on the `RS203000.aspx` page as follows:

> You can use either the Screen Editor or Visual Studio to create controls. The addition of controls with the Screen Editor and with Visual Studio is described in *Step 1.5.1: Add Columns to the Grid* and *Step 2.3.1: Add Input Controls* (respectively) of the *T200 Maintenance Forms* training course. The instructions below are presented in general terms to accommodate both methods.

1. For the first tab item control, type `Repair Items` as the value of the `Text` property.
2. Add a nested `PXGrid` control to the first tab item. If you are using Customization Project Editor, do the following:
   a. In the Screen Editor, with the **Repair Items** node selected, click the **Add Controls** tab.
   b. Drag the **Grid** (`PXGrid`) container onto the **Repair Items** node in the tree.
   c. Click the new **Grid** node in the tree.
   d. On the **Layout Properties** tab, for the **DataMember** property, type the `RepairItems` data view name, which you have created in *Step 2.1.2: Defining the Master-Detail Relationship Between Data (with PXParent and PXDBDefault)*, to bind the container to the data view.
   e. On the page toolbar, click **Save**.
   f. Refresh the control tree.

      The platform assigns the *Grid: RepairItems* name to the grid node in the control tree.
3. Specify the following properties for the grid:
   - `SkinID` (in the `PXGrid` control in ASPX): `Details`
   - `Enabled` in `AutoSize` (in the `AutoSize` control inside `PXGrid` in ASPX): *True*
   - `Width` (in the `PXGrid` control in ASPX): `100%`
   - `InitNewRow` in `Mode`: *True*

     Normally, when a user clicks the **Add Row** button on a grid toolbar, the new empty row is created only on the web page and the data is not sent to the application server. It means that the C# code with all event handlers is not performed. For the data to be sent to the server, you need to specify `InitNewRow = True`. Also, by specifying this property, you can avoid a new extra row created automatically when a user clicks the **Save** button.
4. Add grid columns for the following fields and arrange them in the following order:
   - `RepairItemType`
   - `Required`
   - `InventoryID`
   - `InventoryID_description`
   - `BasePrice`
   - `IsDefault`
5. For the `Required` and `IsDefault` fields, specify the following values of the properties:
   - `Type`: `CheckBox`
   - `Width`: `80`

   The resulting ASPX for the grid columns should look as follows.

```
<Columns>
 <px:PXGridColumn DataField="RepairItemType" Width="70" ></px:PXGridColumn>
 <px:PXGridColumn DataField="Required" Width="80" Type="CheckBox" ></px:PXGridColumn>
 <px:PXGridColumn DataField="InventoryID" Width="70" ></px:PXGridColumn>
```

```
  <px:PXGridColumn DataField="InventoryID_description" Width="280" ></px:PXGridColumn>
  <px:PXGridColumn DataField="BasePrice" Width="100" ></px:PXGridColumn>
  <px:PXGridColumn Type="CheckBox" DataField="IsDefault" Width="80" ></px:PXGridColumn>
</Columns>
```

In ASPX, you specify a data field from the selector of another field (in this example, the description of the inventory ID) as two field names separated by one underscore character, such as `InventoryID_description`.

6. Publish the customization project.

Step 2.1.5: Testing the Tab

In this step, on the Services and Prices (RS203000) form, you will test the master-detail relationship that you have developed.

Do the following:

1. Open the Services and Prices form, and add a new record with the following values in the Summary area:
   - **Service**: *Battery Replacement*
   - **Device**: *Nokia 3310*
2. On the **Repair Items** tab, add rows with the following settings.

Repair Item Type	Required	Inventory ID	Price	Default
Battery	Cleared	*BAT3310*	20	Cleared
Back Cover	Cleared	*BCOV3310*	10	Cleared

You need to specify the price because the `Price` columns in the database must be `Not Null`. You will define automatic change of the **Price** box value in *Lesson 2.2: Defining the Business Logic*.

3. On the form toolbar, click **Save**.
4. In SQL Server Management Studio, make sure the data you specified has been added to the `RSSVRepairPrice` and `RSSVRepairItem` tables.
5. On the Services and Prices form, delete the record you have created.
6. In SQL Server Management Studio, make sure the data has been removed from the `RSSVRepairPrice` and `RSSVRepairItem` tables.

Lesson Summary

In this lesson, you have learned how to set up a master-detail relationship between data.

To create a master-detail form, you have completed the following actions:

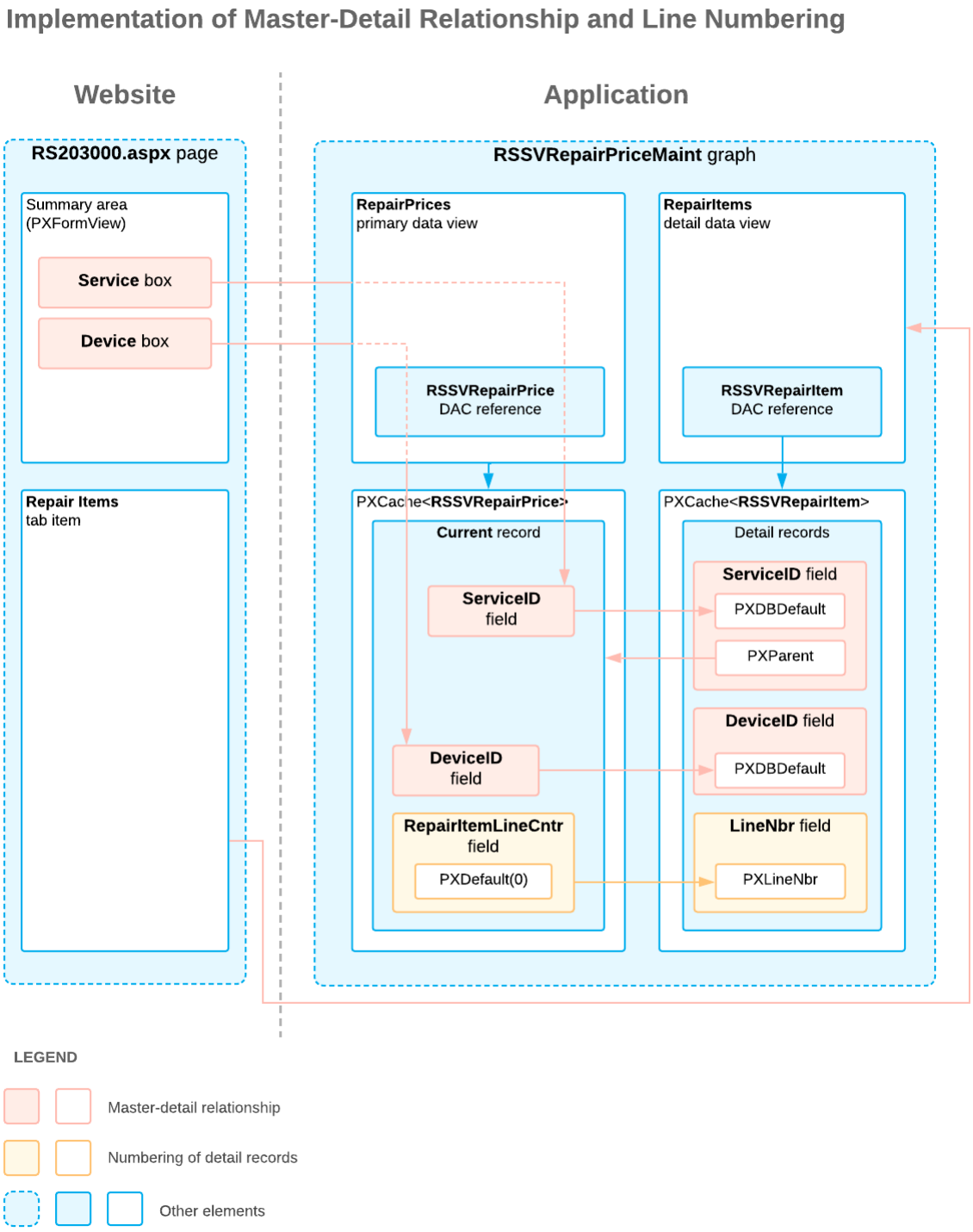
1. Set up the master-detail relationship between data access classes as follows:

- Added the `PXDBDefault` attribute to the key data fields of the detail DAC that are the keys to the master record. The `PXDBDefault` attribute provides the default value for the key field of the detail DAC.
- Added the `PXParent` attribute to the first foreign key data field of the detail DAC. The `PXParent` attribute enables cascading deletion of detail records on deletion of the master record.

2. Defined two data views that select the master-detail data for the form. You have used the `FromCurrent` parameter in the detail data view type to select records for a particular master record.
3. Bound the UI controls that display the data on the ASPX page as follows:
   - Specified the master data view for the datasource control on the page (in the `PrimaryView` property of the control).
   - Specified the master data view for the form (in the `DataMember` property of the control).
   - Specified the detail data view for the grid (in the `DataMember` property of the control).

In the lesson, you have also implemented the numbering of detail data records by using the `PXLineNbr` attribute.

The implementation of the master-detail relationship and the numbering of detail records are shown in the following diagram.



Review Questions

1. Which attributes would you use to define the master-detail relationship?
   a. `PXParent`
   b. `PXDBDefault`

c. `PXLineNbr`

d. `PXUIField`

2. How would you define the starting number for the numbering of detail lines?

   a. Add the `PXDefault` attribute with the starting number for a field in the master DAC and the `PXLineNbr` attribute for a field in the detail DAC.

   b. Add the `PXLineNbr` attribute with the starting number for a field in the master DAC and the `PXDefault` attribute for a field in the detail DAC.

   c. Add the `PXDefault` attribute with the starting number for a field in the detail DAC and the `PXLineNbr` attribute for a field in the master DAC.

Answer Key

1. a, b
2. a

Additional Information: Relationships Between DACs

In this lesson, you have defined the relationship between DACs with the `PXParent` attribute. To define relationships between DACs, you can also use other approaches, which are outside of the scope of this course but may be useful to some readers.

Relationship Between Data with PrimaryKeyOf and ForeignKeyOf

In the code of an Acumatica Framework-based application, you can define the relationship between two tables as follows:

- To define a primary key of a table, for the set of key fields of the data access class (DAC) that corresponds to the table, you set the `IsKey` property of the data type attribute to `true`.
- To define a foreign key of a table, in the DAC that corresponds to the table, you mark the field that contains the foreign key with one of the following attributes: `PXForeignReference`, `PXSelector`, or `PXParent`.

Another way to define a relationship between two tables is to use the `PrimaryKeyOf` and `ForeignKeyOf` classes that are specially designed for the definition of primary and foreign keys.

For details about these classes, see *Relationship Between Data with PrimaryKeyOf and ForeignKeyOf* in the documentation.

Lesson 2.2: Defining the Business Logic

In this lesson, you will define the business logic of the Services and Prices (RS203000) form. You will implement the logic to meet the following requirements for the **Repair Items** tab:

- For a particular row, if a value is selected in the **Repair Item Type** column, the **Inventory ID** column will display only the stock items that are repair items and have the selected repair item type. If no value is selected in the **Repair Item Type** column, the **Inventory ID** column will display all stock items that are repair items.
- For a particular row, if a value is selected in the **Inventory ID** column, the values in the **Repair Item Type** and **Price** columns will be changed to the repair item type and base price (respectively) of the selected stock item as specified on the *Stock Items* (IN202500) form.

- If the **Default** check box is selected for a repair item listed in the grid, this check box must be cleared for all other repair items of the same repair item type.
- For all repair items of the same repair item type, the state of the **Required** check box must be identical.
- If a value is selected in the **Repair Item Type** column for any row, the system should set the state of the **Required** check box for this row to the state of the **Required** check box specified in other rows that have the same repair item type as the selected one.

Lesson Objectives

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

- Restrict the possible values of a field by using the `PXRestrictor` attribute
- Mark localizable messages in code
- Update the fields of the same data record on update of a field of this record
- Update the fields of other records on update of a field
- Learn one of the possible ways to retrieve a data record from the database in code by using the static `PXSelectorAttribute.Select<>()` method

Step 2.2.1: Restricting the Values of a Field (with PXRestrictor)

In this step, you will define the logic of the selector in the **Inventory ID** column of the **Repair Items** tab of the Services and Prices (RS203000) form. You will restrict the values available in the selector when a repair item type is selected in the **Repair Item Type** column.

Changes in the DAC

To configure the restriction, you will use the `PXRestrictor` attribute for the `InventoryID` field of the `RSSVRepairItem` DAC. In the first parameter of the `PXRestrictor` attribute constructor, you will specify the restriction that limits the records displayed in the **Inventory ID** selector. In the constructor, you will also use an error message defined in a class with the `PXLocalizable` attribute specified to make the text available for localization.

For the `PXRestrictor` attribute to work correctly, the `PXRestrictor` attribute must be used along with a `PXSelector` or `PXDimensionSelector` attribute. In this example, the `InventoryID` field has the `Inventory` attribute, which is derived from the `PXDimensionSelector` attribute.

Changes in the ASPX Page

To update the data in the **Inventory ID** selector based on the restriction configured in the `PXRestrictor` attribute, you will set to `True` the following properties in ASPX:

- The `CommitChanges` property for the **Repair Item Type** column
- The `AutoRefresh` property for the **Inventory ID** selector
- The `SyncPosition` property of the `RepairItems` grid

To specify the `AutoRefresh` property, you will add the `PXSegmentMask` control that corresponds to the `InventoryID` field. To add this control, you will add the `RowTemplate` element to the grid. You will use the `PXSegmentMask` control instead of `PXSelector` because the inventory ID is a segmented key and `PXSegmentMask` is intended for editing of segmented keys. For details about segmented keys, see *Managing Segmented Keys*.

Instructions for Restricting the Values of a Field

Proceed as follows:

1. In Visual Studio, add the `using PX.Common;` directive to the `Messages.cs` file and the `using PX.Data.BQL;` directive to the `RSSVRepairItem.cs` file.
2. Add the `PXLocalizable` attribute to the `Messages` class, and include in the class the constant that defines the error message to be used in the `PXRestrictor` attribute, as the following code shows.

```
[PXLocalizable()]
public static class Messages
{
    ...

    //Messages
    public const string StockItemIncorrectRepairItemType =
        "This stock item has a repair item type that differs from {0}.";
}
```

3. In the `RSSVRepairItem` DAC, add the `PXRestrictor` attribute to the `InventoryID` field, as shown in the following code.

```
    #region InventoryID
    [PXRestrictor(typeof(
        Where<InventoryItemExt.usrRepairItem.IsEqual<True>.
            And<Brackets<
                RSSVRepairItem.repairItemType.FromCurrent.IsNull.
                    Or<InventoryItemExt.usrRepairItemType.
                        IsEqual<RSSVRepairItem.repairItemType.FromCurrent>>>>>),
        Messages.StockItemIncorrectRepairItemType,
        typeof(RSSVRepairItem.repairItemType))]
    [Inventory]
    [PXDefault]
    public virtual int? InventoryID { get; set; }
    public abstract class inventoryID : PX.Data.BQL.BqlInt.Field<inventoryID> { }
    #endregion
```

4. Build the project.
5. For the `RepairItemType` field of the `RS203000.aspx` page, set the `CommitChanges` property to `True`.
6. Set the `AutoRefresh` property to `True` for the `InventoryID` field in one of the following ways:
   - In the Screen Editor, do the following:
     a. In the control tree, click the **Tab > Repair Items > Grid: RepairItems > Levels > RepairItems** node.
     b. On the **Add Data Fields** tab, select the unlabeled check box for the *InventoryID (Inventory ID)* field.
     c. On the toolbar of the tab, click **Create Controls**. The **Field Editor** node appears in the **Tab > Repair Items > Grid: RepairItems > Inventory ID** node.
     d. Click the **Field Editor** node.
     e. On the **Layout Properties** tab, set the **AutoRefresh** property to *True*.
   - In Visual Studio, in the `PhoneRepairShop/Pages/RS/RS203000.aspx` file, add the following `RowTemplate` element inside `<px:PXGridLevel DataMember="RepairItems" >`.

```
<RowTemplate>
 <px:PXSegmentMask runat="server" ID="CstPXSegmentMask6"
```

```
  DataField="InventoryID" AutoRefresh="True" >
 </px:PXSegmentMask>
</RowTemplate>
```

7. For the `PXGrid` element of the **Repair Items** tab, set the `SyncPosition` property to `True`.
8. Publish the customization project.

Instructions for Testing the Restriction

On the Services and Prices (RS203000) form, do the following:

1. Create a new record and specify the *Battery Replacement* service and the *Nokia 3310* device for it.
2. Add a new row on the **Repair Items** tab, and do not select anything in the **Repair Item Type** column. Click the magnifier icon in the **Inventory ID** column. Notice the list of stock items displayed in the lookup table. This list includes all active stock items that are repair items.
3. In the **Repair Item Type** column, select *Battery*. Click the magnifier icon in the **Inventory ID** column. The list of stock items contains only two stock items—the ones with the *Battery* repair item type—as shown in the following screenshot.

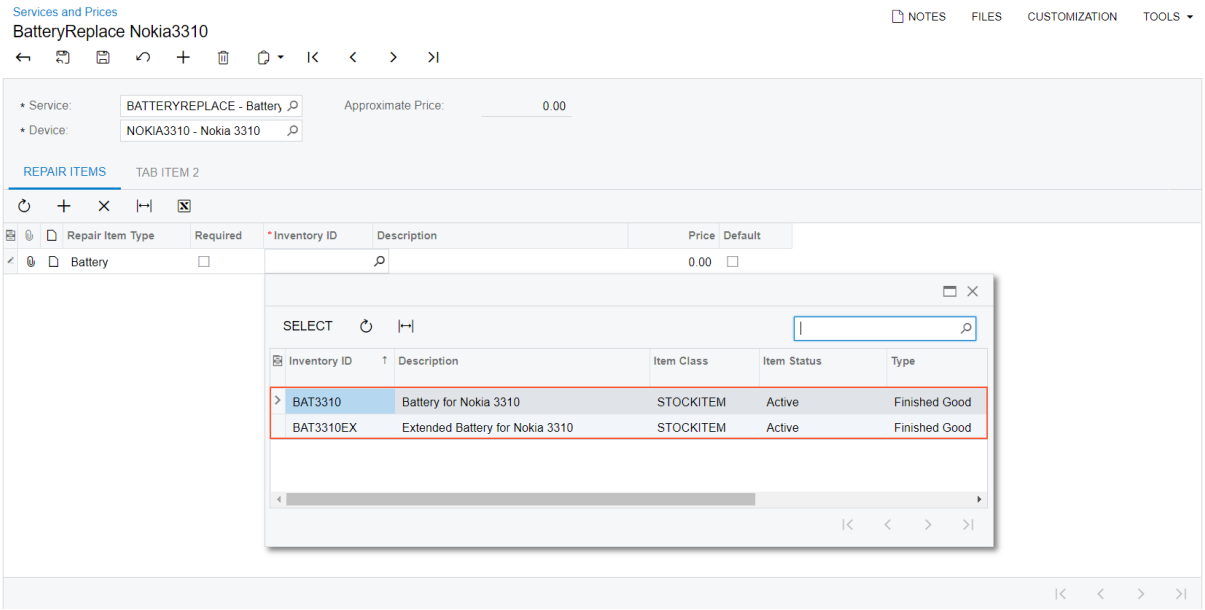


Figure: Inventory ID selector

4. Do not save the record.

Related Links

- *PXRestrictorAttribute Class*
- *PXSelectorAttribute Class*
- *PXDimensionSelectorAttribute Class*
- *To Localize Application Messages*
- *Managing Segmented Keys*

Step 2.2.2: Updating Fields of the Same Record on Update of a Field (with FieldUpdated and FieldDefaulting)

In this step, you will add code that does the following when the `RSSVRepairItem.InventoryID` value is changed: It copies the `RSSVRepairItem.BasePrice` and `RSSVRepairItem.RepairItemType` values from the stock item record that has the ID equal to the new `RSSVRepairItem.InventoryID` value.

You will use the `FieldUpdated` event handler for the `RSSVRepairItem.InventoryID` field to update the values of the following fields of the same record:

- `RSSVRepairItem.RepairItemType`: Instead of directly assigning the value to this field, you will call the `SetValueExt<field>` method to assign the value and invoke the `FieldUpdated` event handler for this field.
- `RSSVRepairItem.BasePrice`: You will trigger the `FieldDefaulting` event for this field by using the `SetDefaultExt<field>` method of `PXCache`. You will assign the value of the `RSSVRepairItem.BasePrice` field in the `FieldDefaulting` event handler.

> You will not assign the value of the `RSSVRepairItem.BasePrice` field in the `FieldUpdated` event handler, because this field may depend on multiple fields of the same record. For example, the price can depend on not only the item selected in the line but also the discount specified for this line. In this example, the `RSSVRepairItem.BasePrice` field depends only on the `RSSVRepairItem.InventoryID` value, but we recommend that you use this approach for the fields that may depend on multiple fields of the same record.

In the `FieldUpdated` and `FieldDefaulting` handlers, you will use the `PXSelectorAttribute.Select<>()` method to select the stock item record with the inventory ID that has been selected in the updated field. The `PXSelectorAttribute.Select<>()` method uses the BQL query from `PXSelector` on the specified field.

In the `FieldDefaulting` handler, you will use the `PK.Find()` method, which selects a record by using the values of the key fields of the record, to retrieve the value of the base price of the stock item. For details about definition of primary keys, see *Relationship Between Data with PrimaryKeyOf and ForeignKeyOf* in the documentation.

Updating Fields of the Same Record

To update multiple fields of the same record, do the following:

1. In the `RSSVRepairPriceMaint.cs` file, add the `PX.Objects.IN using` directive.
2. Define the `FieldUpdated` event handler for the `RSSVRepairItem.InventoryID` field in the `RSSVRepairPriceMaint` class as follows.

```
        //Update the price and repair item type when the inventory ID of
        //the repair item is updated.
        protected void _(Events.FieldUpdated<RSSVRepairItem,
            RSSVRepairItem.inventoryID> e)
        {
            RSSVRepairItem row = e.Row;

            if (row.InventoryID != null && row.RepairItemType == null)
            {
                //Use the PXSelector attribute to select the stock item.
                InventoryItem item = PXSelectorAttribute.
                    Select<RSSVRepairItem.inventoryID>(e.Cache, row)
```

```
                    as InventoryItem;
                //Copy the repair item type from the stock item to the row.
                InventoryItemExt itemExt = item.GetExtension<InventoryItemExt>();
                e.Cache.SetValueExt<RSSVRepairItem.repairItemType>(
                    row, itemExt.UsrRepairItemType);
            }
            //Trigger the FieldDefaulting event handler for basePrice.
            e.Cache.SetDefaultExt<RSSVRepairItem.basePrice>(e.Row);
        }
```

3. Define the `FieldDefaulting` event handler for the `RSSVRepairItem.basePrice` field in the `RSSVRepairPriceMaint` class as follows to calculate the default value of the field.

```
        //Set the value of the Price column.
        protected void _(Events.FieldDefaulting<RSSVRepairItem,
            RSSVRepairItem.basePrice> e)
        {
            RSSVRepairItem row = e.Row;
            if (row.InventoryID != null)
            {
                //Use the PXSelector attribute to select the stock item.
                InventoryItem item = PXSelectorAttribute.
                    Select<RSSVRepairItem.inventoryID>(e.Cache, row)
                    as InventoryItem;
                //Retrieve the base price for the stock item.
                InventoryItemCurySettings curySettings =
                    InventoryItemCurySettings.PK.Find(
                    this, item.InventoryID, Accessinfo.BaseCuryID ?? "USD");
                //Copy the base price from the stock item to the row.
                    e.NewValue = curySettings.BasePrice;
            }
        }
```

4. Rebuild the project.
5. On the `RS203000.aspx` page (in the `Pages\RS` folder of the site), for the `InventoryID` control of the `Repair Items` tab item, set the `CommitChanges` property to `True` to enable a callback for the control.
6. Save your changes to the page.
7. Publish the customization project.

Testing the Logic

On the Services and Prices (RS203000) form, do the following:

1. In the Summary area, select the *Battery Replacement* service and the *Nokia 3310* device.
2. On the **Repair Items** tab, add a row, and select *Battery* in the **Repair Item Type** column and *BAT3310* in the **Inventory ID** column. Shift the focus away from the column. Make sure the system has filled in values in the **Description** and **Price** columns.
3. Save the record.

Related Links

- *Access to a Custom Field*
- *PXSelectorAttribute Class*
- *FieldUpdated Event*

Step 2.2.3: Updating a Field of Another Record on Update of a Field (with RowUpdated)

In this step, you will change the business logic so that when a field of a detail record is updated, the values that depend on this field will be changed in other detail records. On the **Repair Items** tab of the Services and Prices (RS203000) form, when the **Default** check box is selected for a repair item, this check box must be cleared for other repair items with the same repair item type.

To update the field in other detail lines, you will use the `RowUpdated` event handler.

If you are wondering if you could use a `FieldUpdated` event handler in this case we do not recommend this approach, because it can cause data inconsistency. With `FieldUpdated`, if the update of the current detail record did not finish—for example, due to a validation error—the changes in other detail records would not be discarded.

In the `RowUpdated` event handler, you will use the following techniques:

- You will use LINQ to filter the data of the data view that provides data for the **Repair Items** tab.

  You can use language-integrated query (LINQ) provided by the `System.Linq` library when you need to select records from the database in the code of Acumatica Framework-based applications or if you want to apply additional filtering to the data of a BQL query. However, you still have to use business query language (BQL) to define the data views in graphs and to specify the data queries in attributes of data fields.

- You will use the `Update()` method of the `RepairItems` data view to update repair items in `PXCache`. The data view's `Update()` method just invokes the `Update()` method of the `PXCache` object. The `Update()` method of the `PXCache` object raises field-level events for each field of the modified record and row-level events for the modified record. You need to use this method to update `PXCache` if the event handler modifies records other than the record for which the event has been raised. If you do not call this method in the `RowUpdated` event handler, the changes that you've made in the event handler may not be saved to the database and can cause data inconsistency issues.

  Because you cannot modify another record in `FieldUpdated` event handlers, you should never call the `PXCache.Update()` method in one of these handlers.

- You will use the `PXView.RequestRefresh()` method. To optimize the performance, by default, in the `RowUpdated` event, the system synchronizes the values in the UI with the data in `PXCache` for only the current record. Because you have updated the values in other detail records, you need to refresh the UI by using the `PXView.RequestRefresh()` method.

Updating Fields in Detail Records

Do the following to update fields in detail records on update of a field:

1. In the `RSSVRepairPriceMaint.cs` file, add the `using System.Linq;` directive.
2. In the `RSSVRepairPriceMaint` class, add the following event handler.

```
        //Update the IsDefault field of other records with the same repair item type
        //when the IsDefault field is updated.
        protected void _(Events.RowUpdated<RSSVRepairItem> e)
        {
            // Make sure the handler runs only when the IsDefault field is edited.
```

```
            if (e.Cache.ObjectsEqual<RSSVRepairItem.isDefault>(e.Row, e.OldRow))
                return;

            RSSVRepairItem row = e.Row;

            //Use LINQ to select the repair items
            // with the same repair item type as in the updated row.
            var repairItems = RepairItems.Select().Where(item =>
                item.GetItem<RSSVRepairItem>().RepairItemType == row.RepairItemType);

            foreach (RSSVRepairItem repairItem in repairItems)
            {
                if (repairItem.LineNbr == row.LineNbr) continue;

                //Set IsDefault to false for all other items.
                if (row.IsDefault == true && repairItem.IsDefault == true)
                {
                    repairItem.IsDefault = false;
                    RepairItems.Update(repairItem);
                }
            }

            //Refresh the UI.
            RepairItems.View.RequestRefresh();
        }
```

3. Build the project.
4. On the `RS203000.aspx` page, for the `IsDefault` control, set the `CommitChanges` property to `True` to enable a callback for the control.
5. Publish the customization project.

Testing the Default Field

On the Services and Prices (RS203000) form, do the following:

1. Select the price record for the *Battery Replacement* service and the *Nokia 3310* device.
2. On the **Repair Items** tab, add a new line with the following settings:
   - **Repair Item Type**: *Battery*
   - **Inventory ID**: *BAT3310EX*
   - **Default**: Selected
3. Add another line with the following settings:
   - **Repair Item Type**: *Back Cover*
   - **Inventory ID**: *BCOV3310*
   - **Default**: Selected
4. Save the record.
5. On the **Repair Items** tab, select the **Default** check box for the *BAT3310* repair item (which has the *Battery* repair item type). Make sure that this check box has become cleared for the *BAT3310EX* repair item (which also has the *Battery* repair item type) and remains selected for the *BCOV3310* repair item (which has the *Back Cover* repair item type).
6. Save the record, and make sure the values of the **Default** check box in all rows have not changed after you saved it.

Related Links

- *Creating LINQ Queries*
- *PXView Class*
- *RowUpdated Event*

Step 2.2.4: Updating Fields on Update of Another Field—Self-Guided Exercise

In this step, you will define the UI logic of the **Repair Item Type** and **Required** columns on the Services and Prices (RS203000) form on your own.

The **Required** and **Repair Item Type** columns must meet the following criteria:

- For all repair items of the same repair item type, the state of the **Required** check box must be identical.
- If a value is selected in the **Repair Item Type** column for any row, the system should set the state of the **Required** check box for this row to the state of the **Required** check box specified in other rows that have the same repair item type as the selected one.
- For the **Repair Item Type** column, when its value is edited for a row, the system should clear the values of the **Inventory ID** and **Default** columns in this row.

As you implement the logic, use the tips described in the following sections.

Updating Fields of the Same Record on Update of the RepairItemType Field

You should use the same approach as was described in *Step 2.2.2: Updating Fields of the Same Record on Update of a Field (with FieldUpdated and FieldDefaulting)*. Specifically, you should do the following:

- Define the `FieldUpdated` event handler for the `RSSVRepairItem.repairItemType` field in the `RSSVRepairPriceMaint` class.
- In the event handler, clear the `RSSVRepairItem.inventoryID` and `RSSVRepairItem.isDefault` field values when a repair item type is updated by using the `SetValueExt<field>` and `SetValue<field>` methods of `PXCache`. Use the `SetValueExt<field>` method to set the value of the `InventoryID` field and to trigger the `FieldUpdated` event for this field right after the `RSSVRepairItem.repairItemType` field is updated. Use the `SetValue<field>` method for the `IsDefault` field.
- In the event handler, trigger the `FieldDefaulting` event for the `RSSVRepairItem.required` field by using the `SetDefaultExt<field>` method of `PXCache`.
- In the `FieldDefaulting` event handler for the `RSSVRepairItem.required` field, use LINQ to check whether there are any repair items with the same repair item type and copy the `Required` value from the previous records.
- Do not forget to set the `CommitChanges` property to `True` for the `RSSVRepairItem.repairItemType` field.

The following code fragment shows the result of this section.

```
        //When Repair Item Type is updated,
        //clear the values of the Inventory ID and Default columns and
        //trigger FieldDefaulting for the Required column.
        protected void _(Events.FieldUpdated<RSSVRepairItem,
            RSSVRepairItem.repairItemType> e)
        {
            RSSVRepairItem row = e.Row;
            e.Cache.SetDefaultExt<RSSVRepairItem.required>(row);
            if (e.OldValue != null)
            {
                e.Cache.SetValueExt<RSSVRepairItem.inventoryID>(row, null);
```

```
            e.Cache.SetValue<RSSVRepairItem.isDefault>(row, false);
        }
    }

    //Set the value of the Required column.
    protected void _(Events.FieldDefaulting<RSSVRepairItem,
        RSSVRepairItem.required> e)
    {
        RSSVRepairItem row = e.Row;
        if (row.RepairItemType != null)
        {
            // Use LINQ to check whether there are any repair items
            // with the same repair item type.
            var repairItem = (RSSVRepairItem)RepairItems.Select()
                .FirstOrDefault(item =>
                    item.GetItem<RSSVRepairItem>().RepairItemType ==
                        row.RepairItemType &&
                    item.GetItem<RSSVRepairItem>().LineNbr != row.LineNbr);
            //Copy the Required value from the previous records.
            e.NewValue = repairItem?.Required;
        }
    }
```

Updating Fields of Another Record on Update of the Required Field

You should use the same approach as was described in *Step 2.2.3: Updating a Field of Another Record on Update of a Field (with RowUpdated)*.

As you implement the logic, use the following tips:

- Modify the `RowUpdated<RSSVRepairItem>` event handler to implement modifications of the repair item records with the same repair item type as is selected in the updated record.
- In the `RowUpdated` event handler, check whether the value of the `Required` field of the updated record has changed and update other records by using the following code.

```
            //Make the Required field identical for all items of the type.
            if (row.Required != e.OldRow.Required &&
                repairItem.Required != row.Required)
            {
                repairItem.Required = row.Required;
                RepairItems.Update(repairItem);
            }
```

- Do not forget to set the `CommitChanges` property to `True` for the `Required` field.

The following code fragment shows the result of this section.

```
    //Update the IsDefault and Required fields of other records
    //with the same repair item type when these fields are updated.
    protected void _(Events.RowUpdated<RSSVRepairItem> e)
    {
        if (e.Cache.ObjectsEqual<RSSVRepairItem.isDefault,
                                 RSSVRepairItem.required>(e.Row, e.OldRow)) return;

        RSSVRepairItem row = e.Row;
        //Use LINQ to select the repair items
        // with the same repair item type as in the updated row.
        var repairItems = RepairItems.Select()
```

```
                .Where(item => item.GetItem<RSSVRepairItem>()
                .RepairItemType == row.RepairItemType);
            foreach (RSSVRepairItem repairItem in repairItems)
            {
                if (repairItem.LineNbr == row.LineNbr) continue;

                //Set IsDefault to false for all other items.
                if (row.IsDefault == true && repairItem.IsDefault == true)
                {
                    repairItem.IsDefault = false;
                    RepairItems.Update(repairItem);
                }
                //Make the Required field identical for all items of the type.
                if (row.Required != e.OldRow.Required &&
                    repairItem.Required != row.Required)
                {
                    repairItem.Required = row.Required;
                    RepairItems.Update(repairItem);
                }
            }
            //Refresh the UI.
            RepairItems.View.RequestRefresh();
        }
```

Testing the Logic

Test the added logic as follows:

1. On the Services and Prices (RS203000) form, add the price record for the *Liquid Damage* service and the *Nokia 3310* device.
2. On the **Repair Items** tab, add a new line with the following settings:
   - **Repair Item Type**: *Battery*
   - **Required**: Selected
   - **Inventory ID**: *BAT3310*
   - **Default**: Selected
3. Add another line with the following settings:
   - **Repair Item Type**: *Battery*
   - **Inventory ID**: *BAT3310EX*

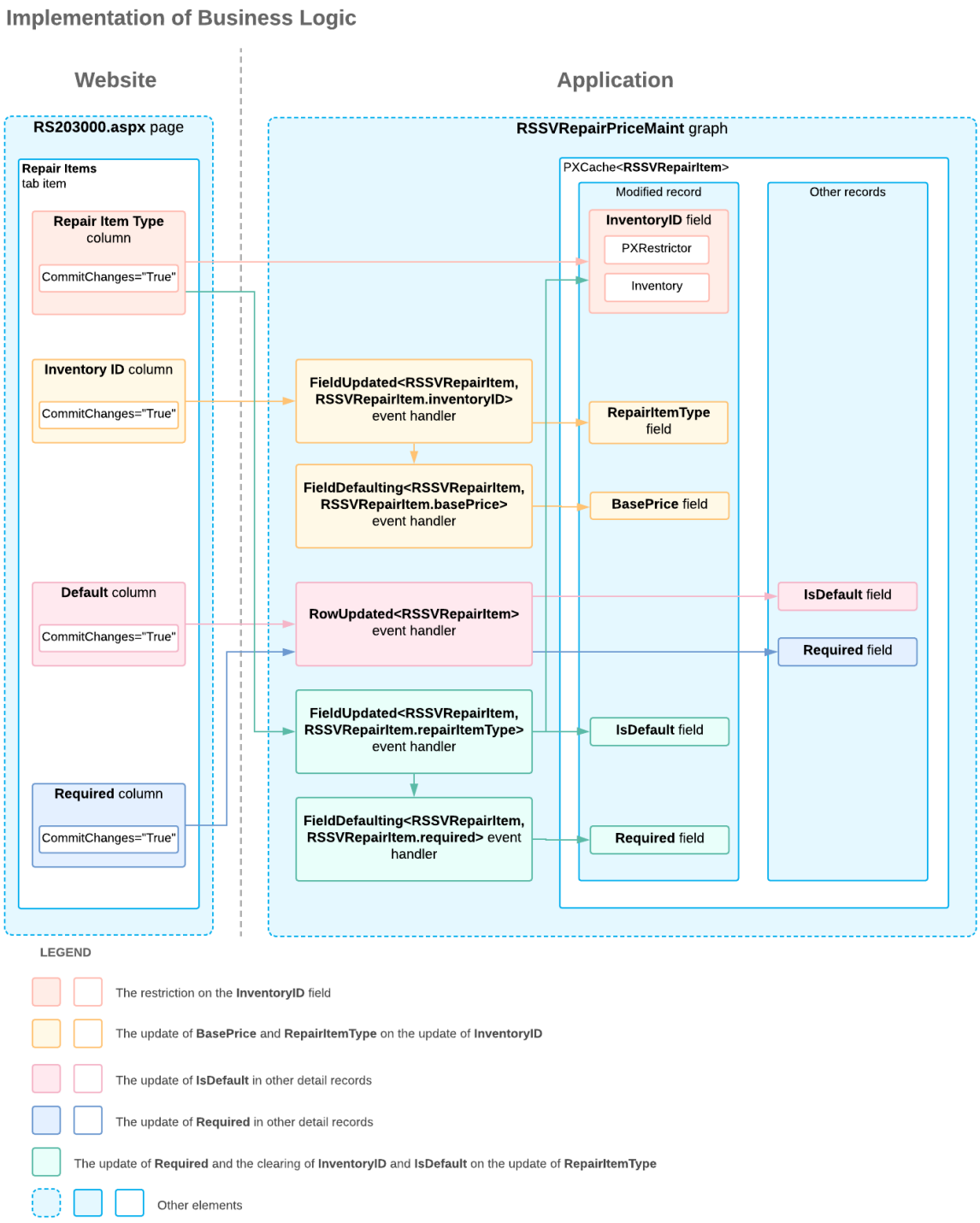
   Make sure that the **Required** check box becomes selected once you select the value in the **Repair Item Type** box because you have already defined the *Battery* repair item type as being required in the first row.
4. Save the record.
5. Clear the **Required** check box in the row for the *BAT3310EX* stock item. Make sure the **Required** check box becomes cleared in the row for the *BAT3310* stock item.
6. Save the record.
7. For the first row in the table, change the value in the **Repair Item Type** column to *Screen* and click another column. Make sure the values in the **Inventory ID** and **Default** columns become cleared.
8. Do not save the latest changes.

Lesson Summary

In this lesson, you have configured the business logic of the **Repair Items** tab of the Services and Prices (RS203000) form as follows:

- You have used the `PXRestrictor` attribute to configure a restriction on the values in the **Inventory ID** column.
- You have used a class with the `PXLocalizable` attribute specified to make the text available for localization.
- You have used the `FieldUpdated` and `FieldDefaulting` event handlers to modify the values of a detail record on update of the column of this detail record. In the event handlers, you have used the `PXSelectorAttribute.Select<>()` method to obtain the stock item record with the inventory ID selected in the updated field. You have also used the `SetValueExt<field>` and `SetDefaultExt<field>` methods to trigger additional events for particular fields.
- You have used the `RowUpdated` event handler so that when particular fields of one detail record are updated, the values in the other detail records are modified. In this event handler, you have used the `PXCache.Update()` method to update in `PXCache` the records other than the record for which the event has been raised. You have also used the `PXView.RequestRefresh()` method to display in the UI the changes in `PXCache` that you have made in the event handler. You have also used LINQ for the filtering of the records of the data view.

The implementation of the business logic is shown in the following diagram.



Review Questions

1. Which event handler would you use if when the field of a detail record is updated, you need to update the fields of the related detail records?
   a. `FieldUpdated`

   b. `RowUpdated`
   c. `RowSelected`
2. In which cases do you need to call the `PXCache.Update()` method in a `FieldUpdated` event handler?
   a. If you modify records other than the record for which the event has been raised.
   b. In no case; you should never call the `PXCache.Update()` method in a `FieldUpdated` event handler.
   c. For each updated field.

Answer Key

1. b
2. b

Additional Information: Application Localization

In this lesson, you have used messages that can be localized in the `PXRestrictor` attribute. Application localization is outside of the scope of this course, but this information may be useful to some readers.

Localization Process

Acumatica Framework provides a built-in utility that makes it possible for a user to localize the product. Once localization is entered and applied, the application does not require any recompilation or re-installation. Also, localization can be exported, imported, and merged.

For more information about how to use the built-in localization mechanism, see *Translation Process* in the documentation.

Preparation of the Application Code and DACs for Localization

To prepare an application for localization, you must prepare data access classes (DACs) and the application code.

For details about the preparation of the application code and DACs, see *To Prepare DACs for Localization* and *To Localize Application Messages*.

Multi-Language Fields

With Acumatica Framework, you can create fields into which a user can type values in multiple languages if multiple locales are configured in the applicable Acumatica Framework application. For example, in Acumatica ERP, if an instance works with English and French locales, a user can specify the value of the **Description** box on the *Stock Items* (IN202500) form in English and French.

For the information about configuring multi-language fields in code, see *To Work with Multi-Language Fields*.

Optimization of Memory Consumption of Localized Data

To optimize the memory consumption of static data, you can move the localization data from all customer application instances to centralized storage. By default, the localization data is kept in the database of every Acumatica ERP instance, and the total size of this data therefore equals the number of instances times the size of the data. If you move the localization data to centralized storage, there is only one copy of this data.

See details about this optimization in *To Optimize Memory Consumption of Localized Data*.

Additional Information: Data Querying

In this lesson, you have used fluent BQL for data querying. Details about the data querying in Acumatica Framework are outside of the scope of this course but may be useful to some readers.

Data Querying in Acumatica Framework

In Acumatica Framework, you generally use business query language (BQL) to query data from the database. BQL statements represent specific SQL queries and are translated into SQL by Acumatica Framework, which helps you to avoid the specifics of the database provider and validate the queries at the time of compilation. Acumatica Framework provides two dialects of BQL: traditional BQL and fluent BQL.

To query data from the database, you can also use language-integrated query (LINQ), which is a part of the .NET Framework. In the code of Acumatica Framework-based applications, you can use both the standard query operators (provided by LINQ libraries) and the Acumatica Framework-specific operators that are designed to query database data.

For details about building queries, see the following chapters in the documentation:

- *Creating Fluent BQL Queries*
- *Creating Traditional BQL Queries*
- *Creating LINQ Queries*

For a comparison of these approaches in data querying, see *Comparison of Fluent BQL, Traditional BQL, and LINQ*.

Execution of Data Queries in Acumatica Framework

If you want to know how data queries are executed in the system, such as how a BQL statement is converted to an SQL query, see the following topics in the documentation:

- *Data Query Execution*
- *Translation of a BQL Command to SQL*
- *Merge of the Records with PXCache*

Part 3: Custom Tab (Stock Items Form)

In this part of the course, you will add a custom tab to the *Stock Items* (IN202500) form. If the selected item is a repair item (that is, if the **Repair Item** check box is selected on the **General** tab of the form), this tab displays a list of devices that can be serviced by using the item.
After you complete the lessons of this part, you will be able to test the ability of a user to add compatible devices for a repair item.

Lesson 3.1: Adding a New Tab

In this lesson, you will learn how to add a new data view to the business logic of an Acumatica ERP form. Also, you will learn how to use the Screen Editord to add a new container control to an Acumatica ERP form.

On the *Stock Items* (IN202500) form, you will create the **Compatible Devices** tab, which will contain a grid with the compatible serviceable devices for each repair item. On this tab, managers of the Smart Fix company will select the devices that can be serviced by using the item. You will place the new tab right of the other tabs on the form, as the following screenshot shows.

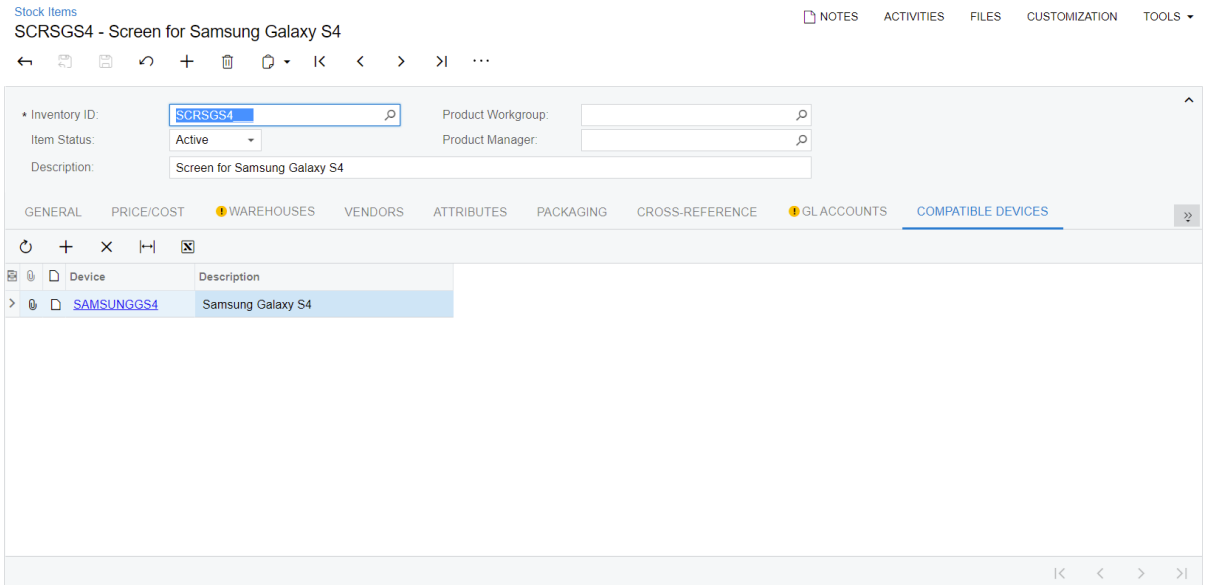


Figure: The Compatible Devices tab

A form may contain controls for only data fields that are accessible through a data view. To provide access to data fields for a container control on a form, you have to define a separate data view for the container in the graph. Each data view should refer to a unique main data access class (DAC) unless you want to display the same data record in multiple containers.

To provide data for the new tab item, you have to create a data view in the `InventoryItemMaint` graph that provides the business logic for the *Stock Items* form.

Lesson Objectives

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

- Add a custom data view for an Acumatica ERP form
- Create a custom tab on an Acumatica ERP form

- Conditionally hide a custom tab on an Acumatica ERP form

Step 3.1.1: Creating a DAC—Self-Guided Exercise

As you completed the *Initial Configuration*, you created the `RSSVStockItemDevice` database table. In this step, you will create a data access class for this table. The ways to create a DAC are described in detail in *Lesson 1.4: Configure the Data Access Class* and *Step 2.1.2: Create a DAC in Visual Studio* in the *T200 Maintenance Forms* training course.

As you add the DAC, you perform the following general actions:

1. Create the `RSSVStockItemDevice` data access class and define its system fields. For more information about definition of the system fields, see *Audit Fields*, *Concurrent Update Control (TStamp)*, and *Attachment of Additional Objects to Data Records (NoteID)* in the documentation.
2. For the DAC, specify the following attribute.

```
[PXCacheName("Device Compatible with Stock Item")]
```

3. In the `RSSVStockItemDevice` DAC, define the attributes as follows:
   - Mark the `InventoryID` and `DeviceID` fields as the key fields.
   - Specify the display names for the fields as follows:
     - For the `DeviceID` field, `Device`
     - For the `InventoryID` field, no display name
   - Set up a master-detail relationship between the `InventoryItem` and `RSSVStockItemDevice` DACs. You have learned how to define a master-detail relationship in *Step 2.1.2: Defining the Master-Detail Relationship Between Data (with PXParent and PXDBDefault)*.
   - Add the following `PXSelector` attribute to the `DeviceID` field.

```
[PXSelector(
    typeof(RSSVDevice.deviceID),
    typeof(RSSVDevice.deviceCD),
    typeof(RSSVDevice.description),
    SubstituteKey = typeof(RSSVDevice.deviceCD),
    DescriptionField = typeof(RSSVDevice.description))]
```

The full code of the `RSSVStockItemDevice` DAC, which is the result of this step, is provided in the `Customization\T210\CodeSnippets\Step3.1.1` folder, which you have downloaded from Acumatica GitHub.

Step 3.1.2: Creating a Data View

Now you will define the `CompatibleDevices` data view in the `InventoryItemMaint` graph.

Creating the Data View

To create the `CompatibleDevices` data view, perform the following actions:

1. In the `InventoryItemMaint.cs` file in Visual Studio, add the following `using` directives.

```
using PhoneRepairShop;
```

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

2. In the `InventoryItemMaint_Extension` extension class, define the `CompatibleDevices` data view as shown in the following code.

```
        #region Data Views
        public SelectFrom<RSSVStockItemDevice>.
            Where<RSSVStockItemDevice.inventoryID.
                IsEqual<InventoryItem.inventoryID.FromCurrent>>.View
            CompatibleDevices;
        #endregion
```

   In the data view declaration, the fluent BQL statement selects the devices that are compatible with the current stock item.

3. Rebuild the project.

Step 3.1.3: Creating the Tab Item, Grid, and Columns

Now you will customize the *Stock Items* (IN202500) form to have a new tab item that displays the compatible devices of the selected stock item.

You will create the following interface elements on the *Stock Items* form:

- The **Compatible Devices** tab
- A grid on the tab
- The following columns in the grid:
  - **Device**
  - **Description**

In the **Device** column, you will provide hyperlinks to the Serviced Devices (RS202000) form so that a user can review the details of the device and edit them.

You will perform the following tasks, each of which is described in the sections below:

1. *1. Creating the Tab*
2. *2. Creating the Grid in the Tab Item*
3. *3. Creating the Columns in the Grid*
4. *4. Adjusting the Grid Layout*
5. *5. Providing Hyperlinks in the Device Column*
6. *6. Reviewing Results*

In this step, you will use the Customization Project Editor to add the interface elements; however, you could perform the same tasks in Visual Studio.

1. Creating the Tab

You create the tab as follows:

1. Open the Screen Editor for the `IN202500.aspx` page.
2. In the control tree, expand the **Tab: ItemSettings** node, and on the right pane, open the **Add Controls** tab.

3. Drag the **Tab Item** (`PXTabItem`) container below the current last node in the tree, as shown in the following screenshot.

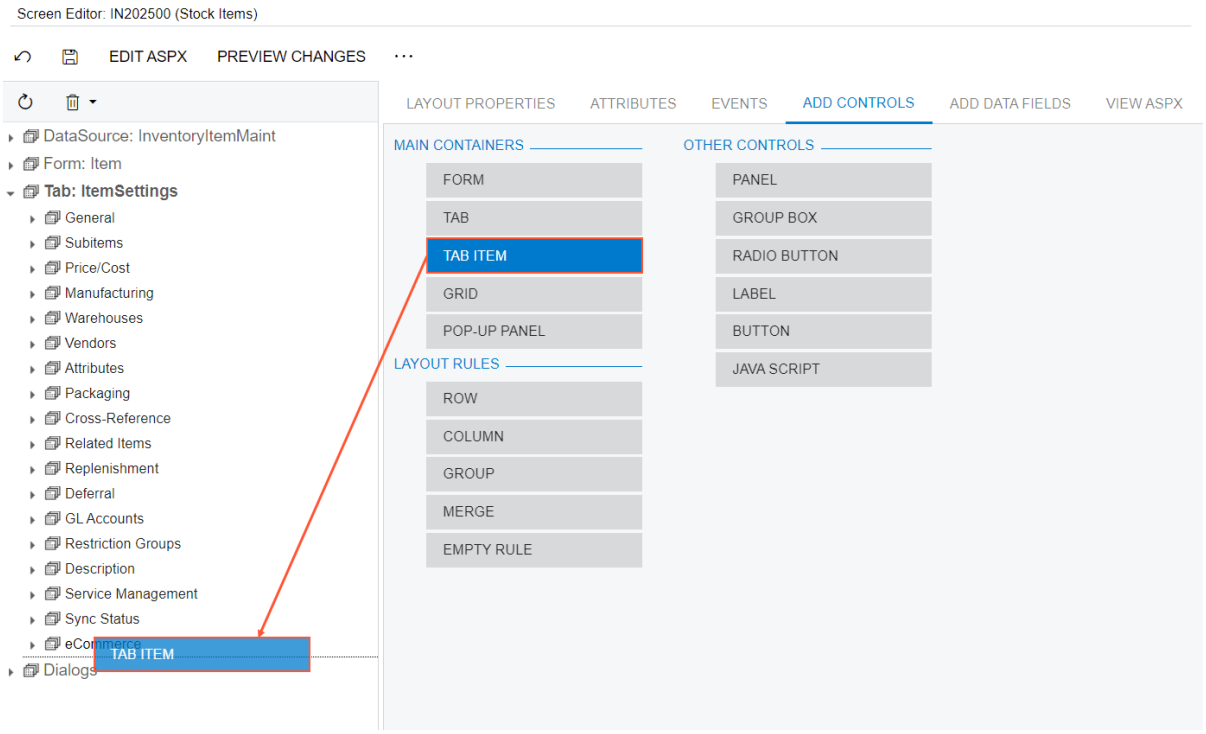


Figure: Addition of the tab item

4. Click the **TabItem** node that you have added.
5. On the **Layout Properties** tab, for the **Text** property, type `Compatible Devices` to specify the name of the new tab.
6. On the page toolbar, click **Save** to save the changes to the customization project and refresh the control tree.

   The platform assigns the *Compatible Devices* name to the new `TabItem` node in the control tree.

2. Creating the Grid in the Tab Item

Perform the following instructions to create the grid in the tab item:

1. In the Screen Editor, with the **Compatible Devices** node selected, click the **Add Controls** tab.
2. Drag the **Grid** (`PXGrid`) container onto the **Compatible Devices** node in the tree.
3. Click the new **Grid** node in the tree.
4. On the **Layout Properties** tab, for the **DataMember** property, type the `CompatibleDevices` data view name, which you have created in the previous step, to bind the container to the data view.
5. On the page toolbar, click **Save**.
6. Refresh the control tree.

   The platform assigns the *Grid: CompatibleDevices* name to the grid node in the control tree.

3. Creating the Columns in the Grid

Proceed as follows to create the columns in the grid:

1. In the Screen Editor with the **Grid: CompatibleDevices** node selected, click the **Add Data Fields** tab.
2. In the list of the **Visible** fields, select the unlabeled check box for the following fields:
   - `DeviceID`
   - `DeviceID_RSSVDevice_description`

   > This is the `Description` field of the `RSSVDevice` DAC, which is available through `PXSelector` assigned to the `DeviceID` field of the `RSSVStockItemDevice` DAC.
3. On the tab toolbar, click **Create Controls** to add the controls to the grid column.
4. In the tree, expand the grid node and reorder the nodes of the created controls, as shown in the following screenshot.

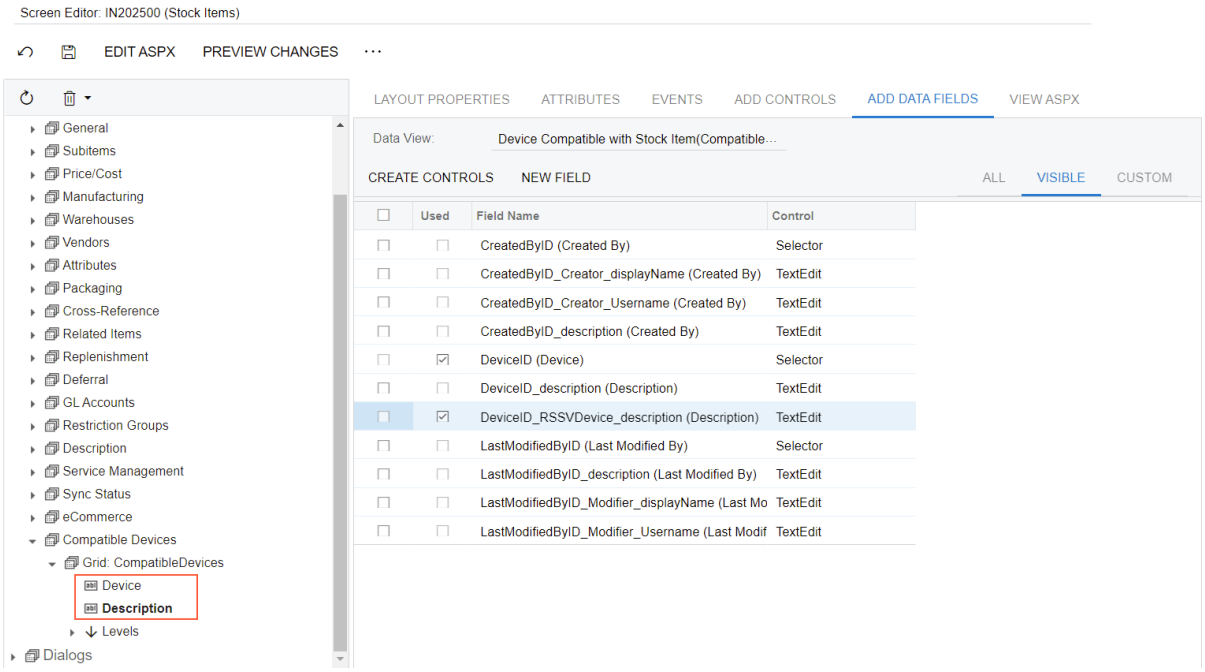


Figure: The Device and Description nodes

5. On the page toolbar, click **Save** to save the changes to the *PhoneRepairShop* customization project.
6. Publish the project to apply the customization to the site.
7. Refresh the *Stock Items* (IN202500) form to view the content of the created tab, which is shown in the following screenshot.

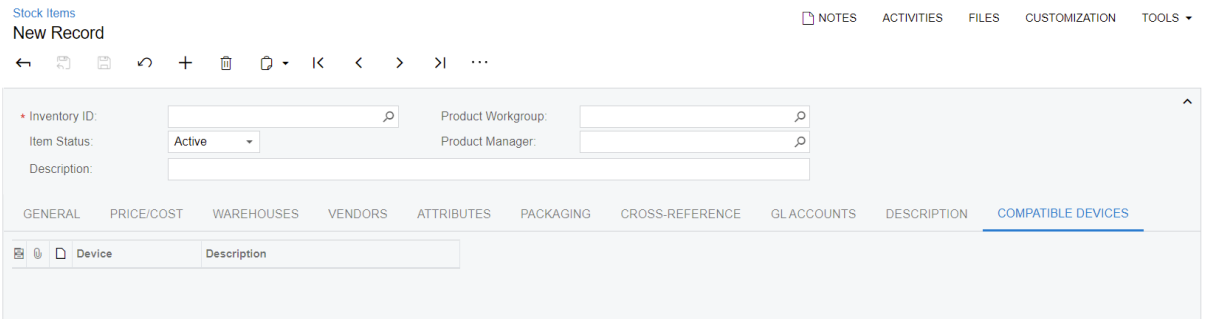


Figure: The new tab

On the form, you can see the awkward layout of the table on the created tab. You have to change the properties of the columns and the grid to display the table properly, and you need to save the new layout in the customization project.

> The user that is currently logged in can rearrange and configure the columns of a grid. But this new grid configuration cannot be saved to a customization project. Conversely, any change of a property of an original or custom control is a part of a customization that can be finally included in a deployment package that can be exported to the production environment.

4. Adjusting the Grid Layout

To adjust the grid layout, do the following:

1. In the Screen Editor, select the **Grid: CompatibleDevices** node in the control tree, and specify the following properties on the **Layout Properties** tab:
   - **AutoSize > Enabled**: *True*
   - **AutoSize > MinHeight**: `200`
   - **SkinID**: `Details`
   - **Width**: `100%`
   - **DataSourceID**: `ds`
2. On the page toolbar, click **Save** to save the changes to the customization project.
3. Publish the project to apply the customization to the site.
4. Refresh the *Stock Items* (IN202500) form to view the content of the created tab, which is shown in the following screenshot.

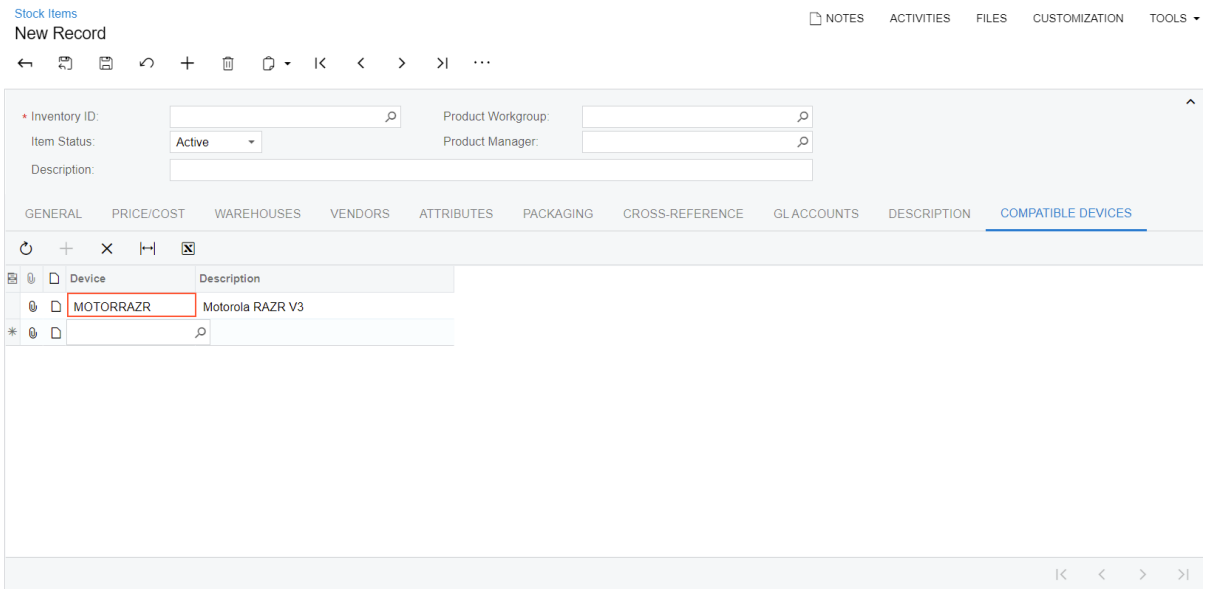


Figure: The final layout of the tab

Try to add a record to the tab and notice that the **Device** column values are displayed as text. You have to customize the control to display hyperlinks instead of the text.

5. Providing Hyperlinks in the Device Column

You provide hyperlinks in the **Device** column as follows:

1. In the control tree of the Screen Editor, expand the **Levels** node in the **Grid: CompatibleDevices** node, and click the **CompatibleDevices** subnode.
2. On the **Add Data Fields** tab, in the list of the **Visible** fields, select the unlabeled check box in the first column for the **DeviceID (Device)** field.
3. On the tab toolbar, click **Create Controls** to create a control for the field in the `RowTemplate` element of the grid.

   Acumatica Customization Platform creates the control in the `RowTemplate` element of the grid and displays it in the tree as a **Field Editor** subnode for the **Device** node.
4. Click the **Field Editor** node in the tree, and on the **Layout Properties** tab, select *True* for the **AllowEdit** property.
5. On the page toolbar, click **Save** to save the changes to the *PhoneRepairShop* customization project.
6. Select the **Device** node and set the **CommitChanges** property to *True*.
7. On the page toolbar, click **Save**.
8. In Visual Studio, assign the `PXPrimaryGraph` attribute to the `RSSVDevice` DAC, as shown in the following code.

   ```
   [PXPrimaryGraph(typeof(RSSVDeviceMaint))]
   ```

   You use the `PXPrimaryGraph` attribute to specify the graph that corresponds to the default editing form for records of the DAC.
9. Build the `PhoneRepairShop_Code` project.

6. Reviewing Results

Review the results of the tasks you have performed as follows:

1. On the Screen Editor page, click the **Compatible Devices** node in the control tree.
2. Click the **View ASPX** tab to view the webpage code of the **Compatible Devices** tab, as shown in the following screenshot.

Figure: ASPX code of the tab

3. Publish the project to apply the customization to the site.
4. Refresh the *Stock Items* (IN202500) form to view the content of the created tab.

Related Links

- *Tab Item Container (PXTabItem)*

Step 3.1.4: Hiding the Tab from the Form (with RowSelected)

In this step, you will define the **Compatible Devices** tab of the *Stock Items* (IN202500) form to be hidden if the **Repair Item** check box is cleared on the **General** tab of the form.

You will modify the `RowSelected` event handler, which you have added in *Step 1.1.6: Making the Custom Field Conditionally Available (with RowSelected)*. You will also set the `RepaintOnDemand` property of the `PXTabItem` element that corresponds to the **Compatible Devices** tab to `False`. If the `RepaintOnDemand` property is `True`, the system repaints the tab only when a user opens the tab or works with this tab.

Hiding the Tab from the Form

To conditionally hide the tab from the form, do the following:

1. In Visual Studio, in the `InventoryItemMaint_Extension` class, modify the `RowSelected` event handler for the `InventoryItem` DAC as shown in the following code.

```
protected void _(Events.RowSelected<InventoryItem> e)
{
    InventoryItem item = e.Row;
    InventoryItemExt itemExt = PXCache<InventoryItem>.
        GetExtension<InventoryItemExt>(item);
    bool enableFields = itemExt != null &&
        itemExt.UsrRepairItem == true;
    //Make the Repair Item Type box available
    //when the Repair Item check box is selected.
    PXUIFieldAttribute.SetEnabled<InventoryItemExt.usrRepairItemType>(
        e.Cache, e.Row, enableFields);

    //Display the Compatible Devices tab when the Repair Item check box
    //is selected.
    CompatibleDevices.Cache.AllowSelect = enableFields;
}
```

2. Build the project.
3. In the Screen Editor opened for the `IN202500.aspx` page, select the **Compatible Devices** node and set the `RepaintOnDemand` property to `False`.
4. Publish the customization project.

Related Links

- *Conditional Hiding of a Tab Item*

Step 3.1.5: Using the New Tab

In this step, you will test the new **Compatible Devices** tab of the *Stock Items* (IN202500) form as follows:

1. On the *Stock Items* form, select the *BAT3310* item, and make sure the **Compatible Devices** tab is displayed for this item.

2. On the **Compatible Devices** tab, add the *Nokia3310* device to the list.
3. Save your changes.
4. Create the *SCRSGS4* item with the following settings:
   - **Inventory ID**: `SCRSGS4`
   - **Description**: `Screen for Samsung Galaxy S4`
   - **Item Class** (**Item Defaults** section of the **General** tab): *STOCKITEM*
   - **Default Warehouse** (**Warehouse Defaults** section of the **General** tab): *MAIN*
   - **Default Price** (**Price Management** section of the **Price/Cost** tab): `50`
5. Make sure that the **Compatible Devices** tab is not displayed.
6. Also, make sure that in the **Item Defaults** section of the **General** tab, the **Repair Item Type** box is unavailable for editing.
7. On the **General** tab, select the **Repair Item** check box, and select *Screen* in the **Repair Item Type** box.
8. On the **Compatible Devices** tab, add the *Samsung Galaxy S4* device.
9. Save your changes.

Lesson Summary

In this lesson, you have practiced implementing extensions for both the business logic and the user interface of an existing form, and you have learned the following tasks:

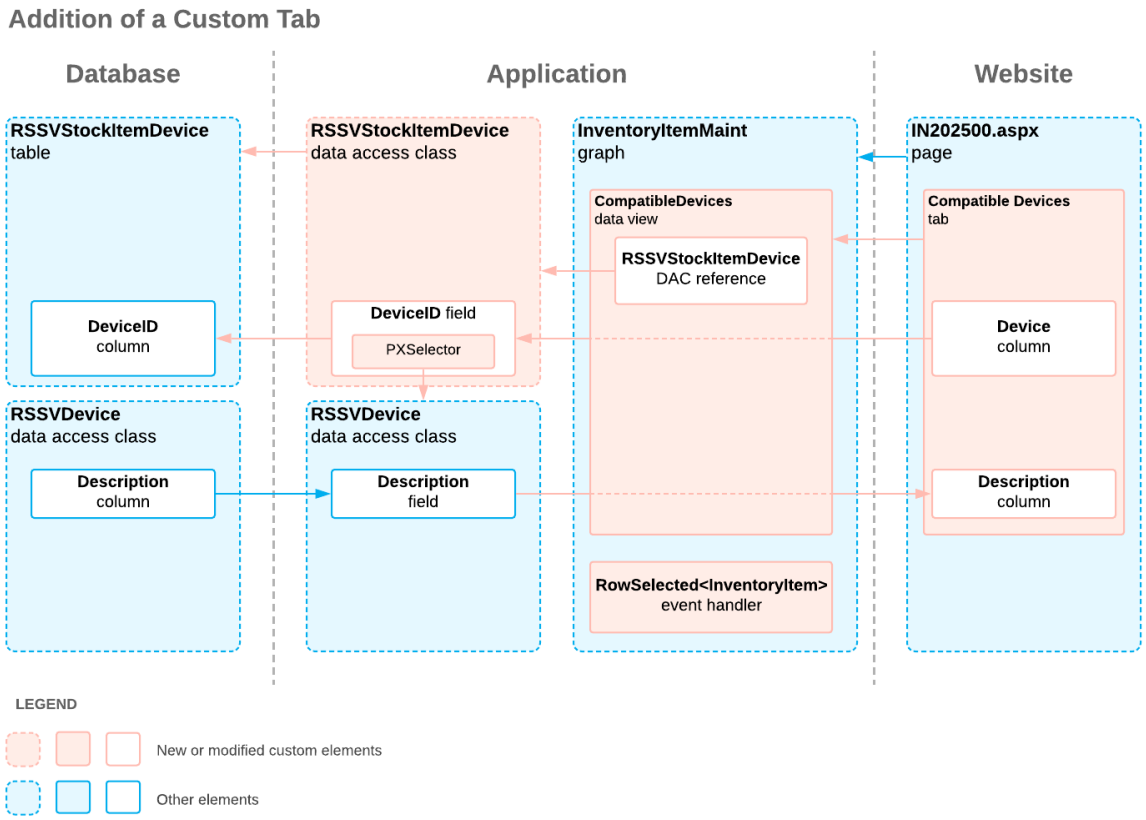
- Creating a control container on a form
- Creating the data view in the extension of the graph for the form to provide data for the container
- Conditionally hiding a tab

During the lesson, you have added to the customization project the following items:

- On the *Stock Items* (IN202500) form, the **Compatible Devices** tab, which contains the **Device** and **Description** columns
- The `CompatibleDevices` data view in the `InventoryItemMaint_Extension` graph
- The `RSSVStockItemDevice` DAC

You have also modified the `RowSelected<InventoryItem>` event handler.

The following diagram shows the results of the lesson.



Review Questions

1. In which order should you perform the following actions to create custom controls in a new container on an Acumatica ERP form by using the Customization Project Editor?

   (1) Bind the container control to the data view.

   (2) Create an extension for the graph that provides the business logic for the form.

   (3) Create the custom fields in a data access class by using either the Data Class Editor or the Screen Editor.

   (4) Create the container control.

   (5) Define a data view that provides a reference to the data access class that includes the custom fields.

   (6) For the custom fields, create the controls.

   (7) Publish the customization project.

   a. 4, 3, 2, 5, 1, 7, 6, 7

   b. 2, 5, 3, 7, 4, 1, 6, 7

   c. 4, 3, 2, 5, 7, 6, 1, 7

   d. 5, 2, 3, 4, 7, 6, 1, 7

   e. 2, 5, 3, 4, 6, 1, 7

2. Select all the conditions that must be met for you to be able to create a control for a field on an Acumatica ERP form.

   a. The graph is bound to the `.aspx` page of the form.

b. The data access class is mapped to a table in the database.
c. The `.aspx` page contains a container to create a control.
d. The data access class includes the field declaration.
e. The data view is bound to the `datasource` control of the `.aspx` page.
f. The container on the page refers to the data view of the graph.
g. The data access class is bound to the `.aspx` page of the form.
h. The graph includes the data view declaration.
j. The data view contains a reference to the data access class.
k. The field is mapped to a table column in the database.

Answer Key

1. b
2. a, c, d, f, h, i

Part 4: Calculations and Insertion of a Default Record (Services and Prices Form)

In this part, you will add the **Labor** and **Warranty** tabs to the Services and Prices (RS203000) form. You will also add the business logic of these tabs: You will implement calculations of the fields on the **Labor** tab and in the Summary area of the form; you will also add a default record to the **Warranty** tab.

After you complete the lessons of this part, you will be able to test the functionality of the Services and Prices form.

Lesson 4.1: Calculating Field Values

In this lesson, you will implement the calculation of the value in the **Approximate Price** box on the Services and Prices (RS203000) form. This value will be calculated as a sum of the values in the **Price** column on the **Repair Items** tab and the values of the **Ext. Price** column on the **Labor** tab, which you will add as a self-guided exercise. In the **Approximate Price** box, a manager of the Smart Fix company can view the price of the selected service for the selected device, which is an approximate price for the corresponding work order.

Description of Form Elements That Are Created in This Lesson

The **Labor** tab will display the non-stock item records, which represent work for the particular service and device that are selected in the Summary area of the form. The tab will have the following columns:

- **Inventory ID**: One of the non-stock items (which the user can select) defined on the *Non-Stock Items* (IN202000) form
- **Description**: The description of the non-stock item that has been selected in the **Inventory ID** column
- **Default Price**: The price of the non-stock item
- **Quantity**: The quantity of the included non-stock item
- **Ext. Price**: The extended price, which the system calculates as the default price multiplied by the quantity

The following screenshot shows the form as it will look after you complete this lesson.

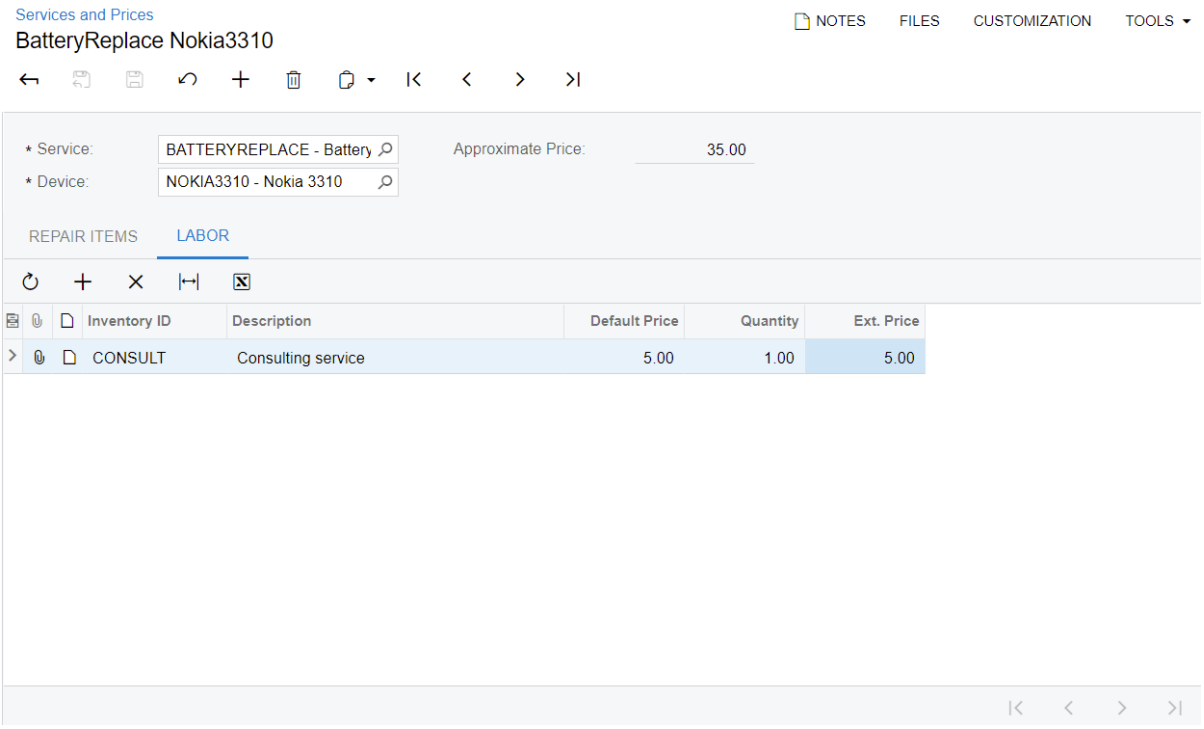
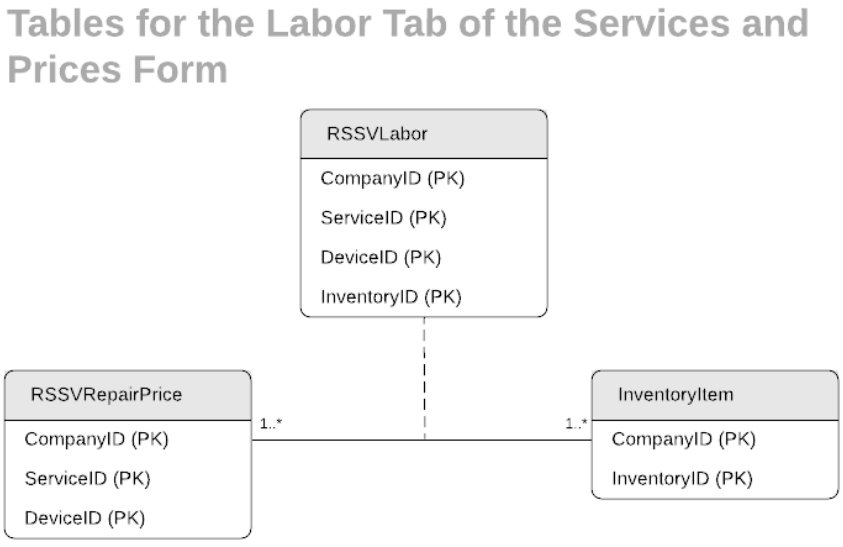


Figure: The Labor tab

Relationships Between Database Tables

The repair prices for particular services and devices (`RSSVRepairPrice`) and the non-stock items (`InventoryItem`) have a many-to-many relationship. The many-to-many links between records are stored in the separate `RSSVLabor` custom table. The following diagram illustrates the relationships between the database tables that are used in this lesson.



Lesson Objectives

You will learn how to use the `PXFormula` attribute for calculations.

Step 4.1.1: Adding the Labor Tab—Self-Guided Exercise

In this step, you will add the **Labor** tab of the Services and Prices (RS203000) form on your own. Although this is a self-guided exercise, you can use the details and suggestions in this topic as you modify the form. You have learned how to add a tab to a form in *Lesson 3.1: Adding a New Tab*.

DAC

You should define the `RSSVLabor` DAC, and set up its fields as follows:

- For the DAC, define the following attribute.

```
[PXCacheName("Repair Labor")]
```

- For the system fields, define the attributes as described in *Step 1.4.2: Configure the Attributes of the New DAC* in the *T200 Maintenance Forms* training course or in *Audit Fields*, *Concurrent Update Control (TStamp)*, and *Attachment of Additional Objects to Data Records (NoteID)* in the documentation.
- For the `InventoryID` field, replace the existing attributes with the `[Inventory(IsKey = true)]` attribute from the `PX.Objects.IN` namespace. (This selector displays only the inventory items for which the current user has access rights and that do not have the *Inactive* or *Marked for Deletion* status.)
- For the `InventoryID` field, add the `PXRestrictor` attribute to display only the inventory items that are non-stock items (that is, only those for which `InventoryItem.stkItem` is equal to `False`). Make sure that you have added the message specified in the restrictor to the localizable `Messages` class.
- Mark the `ServiceID` and `DeviceID` fields as the key fields.

> Unlike the `RSSVRepairItem` DAC, the `RSSVLabor` DAC does not contain additional `LineNbr` key field because users cannot add multiple labor items with the same inventory ID, service ID, and device ID on the **Labor** tab of the Services and Prices (RS203000) form.

- Define a master-detail relationship between the `RSSVRepairPrice` DAC and the `RSSVLabor` DAC. You have learned how to define a master-detail relationship in *Step 2.1.2: Defining the Master-Detail Relationship Between Data (with PXParent and PXDBDefault)*.
- For the `DefaultPrice` field, add attributes, as shown in the following code.

```
#region DefaultPrice
[PXDBDecimal()]
[PXDefault(TypeCode.Decimal, "0.0")]
[PXUIField(DisplayName = "Default Price")]
public virtual Decimal? DefaultPrice { get; set; }
public abstract class defaultPrice :
    PX.Data.BQL.BqlDecimal.Field<defaultPrice> { }
#endregion
```

- For the `Quantity` field, add attributes, as shown in the following code.

```
#region Quantity
[PXDBDecimal()]
[PXDefault(TypeCode.Decimal, "0.0")]
[PXUIField(DisplayName = "Quantity")]
```

```
        public virtual Decimal? Quantity { get; set; }
        public abstract class quantity : PX.Data.BQL.BqlDecimal.Field<quantity> { }
        #endregion
```

- For the `ExtPrice` field, add attributes, as shown in the following code.

```
        #region ExtPrice
        [PXDBDecimal()]
        [PXDefault(TypeCode.Decimal, "0.0")]
        [PXUIField(DisplayName = "Ext. Price", Enabled = false)]
        public virtual Decimal? ExtPrice { get; set; }
        public abstract class extPrice : PX.Data.BQL.BqlDecimal.Field<extPrice> { }
        #endregion
```

You can see the full example of the `RSSVLabor` DAC in the `CodeSnippets` folder for this course, which is available on Acumatica GitHub.

Graph

You need to define the data view for the **Labor** tab of the Services and Prices (RS203000) form, as shown in the following code.

```
    public SelectFrom<RSSVLabor>.
        Where<RSSVLabor.deviceID.
            IsEqual<RSSVRepairPrice.deviceID.FromCurrent>.
        And<RSSVLabor.serviceID.
            IsEqual<RSSVRepairPrice.serviceID.FromCurrent>>>.View
        Labor;
```

As with the `RepairItems` data view, the `Labor` detail data view must be defined after the `RepairPrices` master data view.

Definitions of Controls on the ASPX Page

When you define controls for the **Labor** tab of the Services and Prices (RS203000) form, you can use the following tips:

- Add a new `PXTabItem` control with a `PXGrid` control in it
- Specify the properties of the grid as follows:
  - `DataMember` (in the `PXGridLevel` control inside `PXGrid\Levels` in ASPX): `Labor`
  - `SkinID` (in the `PXGrid` control in ASPX): `Details`
  - `Enabled` in `AutoSize` (in the `AutoSize` control inside `PXGrid` in ASPX): `True`
  - `Width` (in the `PXGrid` control in ASPX): `100%`
- Add columns for the `InventoryID`, `InventoryID_InventoryItem_descr`, `DefaultPrice`, `Quantity`, and `ExtPrice` data fields.
- Set the `CommitChanges` property of the control for the `InventoryID` field to `True`.

Step 4.1.2: Calculating Field Values (with PXFormula)

You will now use the `PXFormula` attribute to implement calculations on the Services and Prices (RS203000) form. Because you are using the `PXFormula` attribute, you do not need to define any event handlers to implement these calculations.

In the `PXFormula` attribute, you can define conditional calculation of the field value. For details about conditional calculation, see *Calculation of Field Values* in the documentation.

Calculating Field Values

Do the following to implement the calculation of the field values:

1. Add the `PXFormula` attribute to the `BasePrice` field of the `RSSVRepairItem` DAC as shown in the following code.

```
#region BasePrice
[PXDBDecimal()]
[PXDefault(TypeCode.Decimal, "0.0")]
[PXUIField(DisplayName = "Price")]
[PXFormula(null,
    typeof(SumCalc<RSSVRepairPrice.price>))]
public virtual Decimal? BasePrice { get; set; }
public abstract class basePrice : PX.Data.BQL.BqlDecimal.Field<basePrice> { }
#endregion
```

   In this code, the `PXFormula` attribute calculates the `Price` value of the parent document as the sum of the `BasePrice` values of all detail records.

2. Add the `PXFormula` attribute to the `ExtPrice` field of the `RSSVLabor` DAC as shown in the following code.

```
#region ExtPrice
[PXDBDecimal()]
[PXDefault(TypeCode.Decimal, "0.0")]
[PXUIField(DisplayName = "Ext. Price", Enabled = false)]
[PXFormula(
    typeof(Mult<RSSVLabor.quantity, RSSVLabor.defaultPrice>),
    typeof(SumCalc<RSSVRepairPrice.price>))]
public virtual Decimal? ExtPrice { get; set; }
public abstract class extPrice : PX.Data.BQL.BqlDecimal.Field<extPrice> { }
#endregion
```

   In this code, the `PXFormula` attribute calculates the `ExtPrice` value as the product of the `Quantity` and `DefaultPrice` values of the same record, and calculates the `Price` value of the parent document as the sum of `ExtPrice` values of all detail records.

3. Build the project.
4. In the `RS203000.aspx` page, set the `CommitChanges` attribute to `True` for the `Quantity` and `DefaultPrice` grid columns of the `Labor` tab item and the `BasePrice` grid column of the `Repair Items` tab item.
5. Publish the customization project.

Testing the Calculations

Now you will test the calculations you have implemented. Open the Services and Prices (RS203000) form, and remove all records on the form because they have incorrect price values saved in the database.

Test the calculations as follows:

1. Add a record for the *Battery Replacement* service and the *Nokia 3310* device.
2. On the **Repair Items** tab, add new lines with the following values.

UI Element	First Line	Second Line
Repair Item Type	*Battery*	*Back Cover*
Required	Selected	Cleared
Inventory ID	*BAT3310*	*BCOV3310*
Default	Selected	Selected

   Once you have added the line, notice that the **Approximate Price** value is calculated in the Summary area.
3. On the **Labor** tab, add a row and specify the following settings:
   - **Inventory ID**: *CONSULT*
   - **Default Price**: 5
   - **Quantity**: 1

   Once you have finished editing the row, notice that the **Ext. Price** value has been calculated for the line. Also notice that the calculated value has been added to the **Approximate Price** value in the Summary area, as shown in the following screenshot.

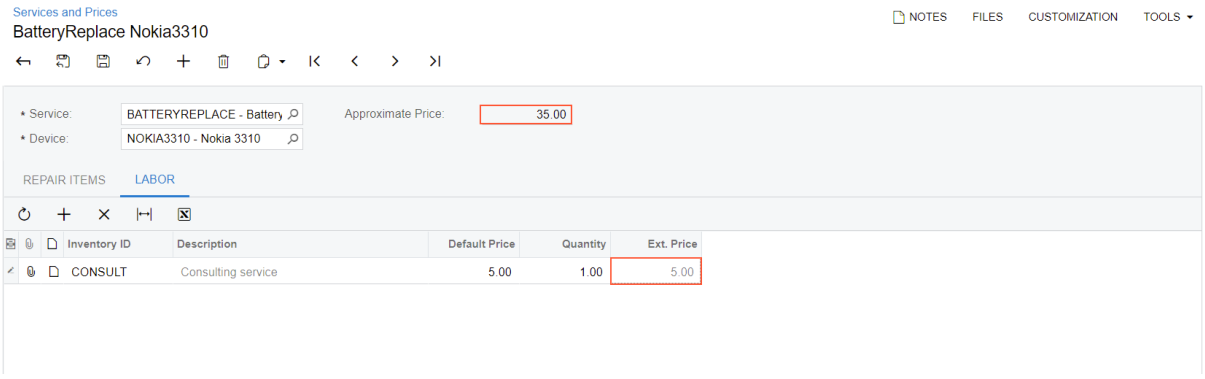


Figure: The calculation of the price

4. Save your changes.
5. Add another record for the *Liquid Damage* service and the *Nokia 3310* device.
6. On the **Repair Items** tab, add new lines with the following values.

UI Element	First Line	Second Line
Repair Item Type	*Battery*	*Battery*

UI Element	First Line	Second Line
Required	Cleared	Cleared
Inventory ID	*BAT3310*	*BAT3310EX*
Default	Selected	Cleared

7. Save your changes. Make sure the value of **Approximate Price** in the Summary area is 50.

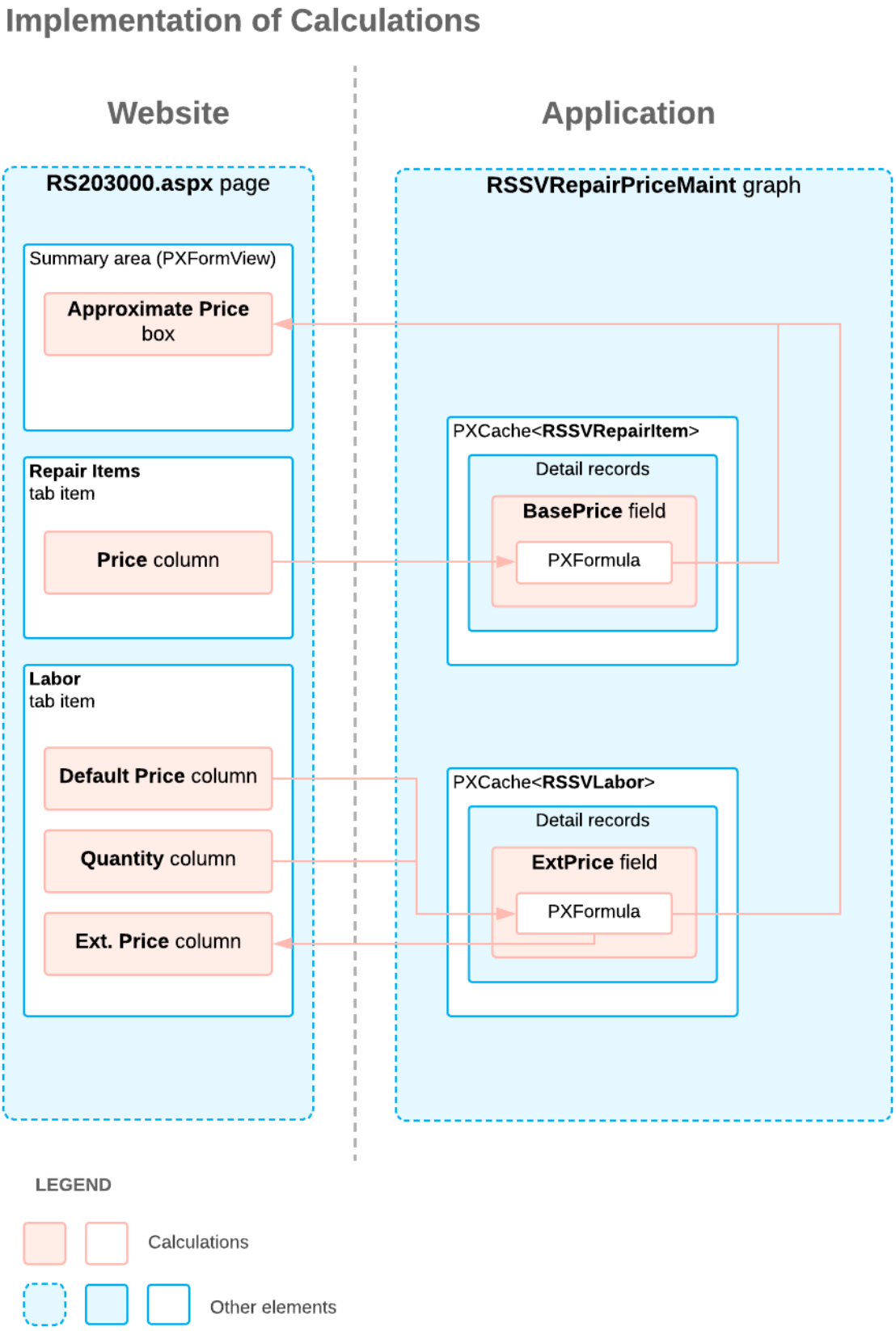
Related Links

- *PXFormulaAttribute Class*
- *Calculation of Field Values*

Lesson Summary

In this lesson, you have implemented the calculation of the value of the **Approximate Price** box on the Services and Prices (RS203000) form. You have used the `PXFormula` attribute to configure the calculation.

The implementation of the calculation is shown in the following diagram.


Implementation of Calculations
Website
Application
RS203000.aspx page
RSSVRepairPriceMaint graph
Summary area (PXFormView)
Approximate Price box
Repair Items tab item
Price column
Labor tab item
Default Price column
Quantity column
Ext. Price column
PXCache<RSSVRepairItem>
Detail records
BasePrice field
PXFormula
PXCache<RSSVLabor>
Detail records
ExtPrice field
PXFormula
LEGEND
Calculations
Other elements

Review Question

1. Suppose that on a custom Acumatica ERP form, you need to calculate the value of the **Total** box in the Summary area as a sum of the values in the **Full Price** column on the **Document Details** tab. Further suppose that each value in the **Full Price** column should be calculated as the value in the **Quantity** column multiplied by the value in the **Price** column. To which field would you assign the `PXFormula` attribute to calculate the value in the **Total** box?
   a. The DAC field that corresponds to the **Total** box
   b. The DAC field that corresponds to the **Full Price** column
   c. The DAC field that corresponds to the **Quantity** column
   d. The DAC field that corresponds to the **Price** column

Answer Key

1. b

Lesson 4.2: Inserting a Default Detail Record

In this lesson, you will add the **Warranty** tab to the Services and Prices (RS203000) form. This tab will display the list of warranties (which are contract templates in Acumatica ERP) for the particular service and device selected in the Summary area of the form. On this tab, the default warranty item is added each time a new service–device pair is defined on the form.

Description of Form Elements That Are Created in This Lesson

The **Warranty** tab has the **Contract ID** column, which contains the ID of the contract template. The tab also has the **Description**, **Duration**, **Duration Unit**, and **Contract Type** columns, which the system fills in with the read-only values of the corresponding fields of the selected contract template.

The **Warranty** tab with the default warranty item is shown in the following screenshot.

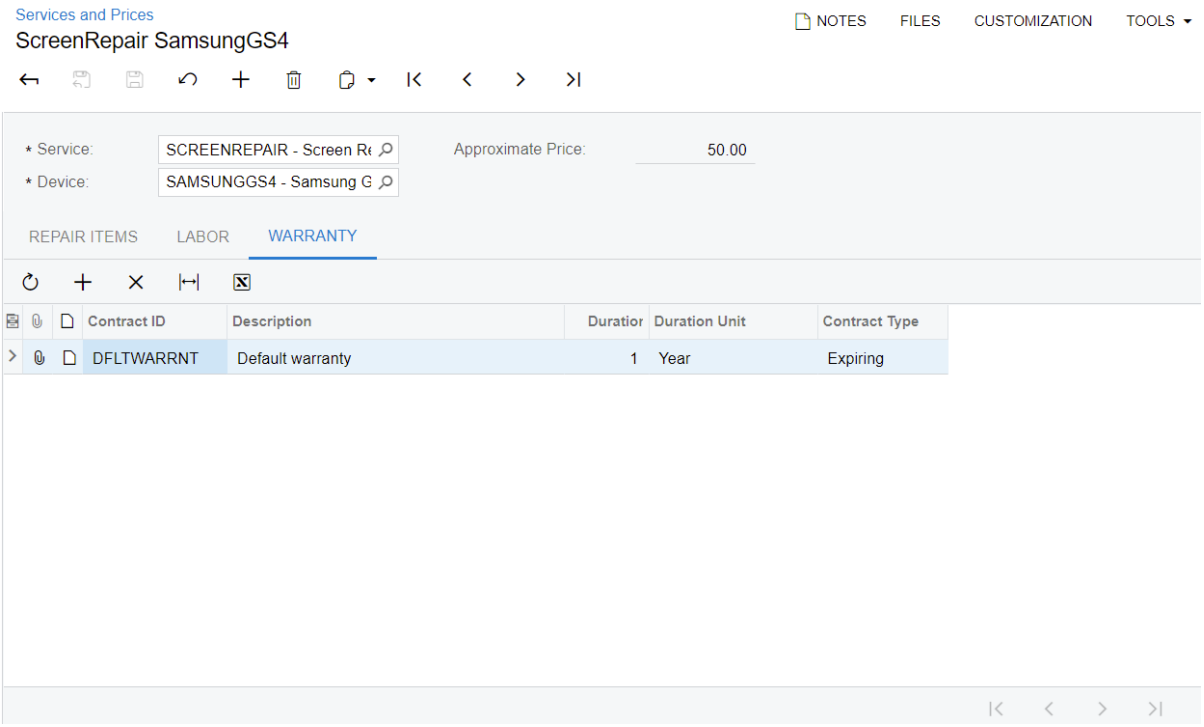
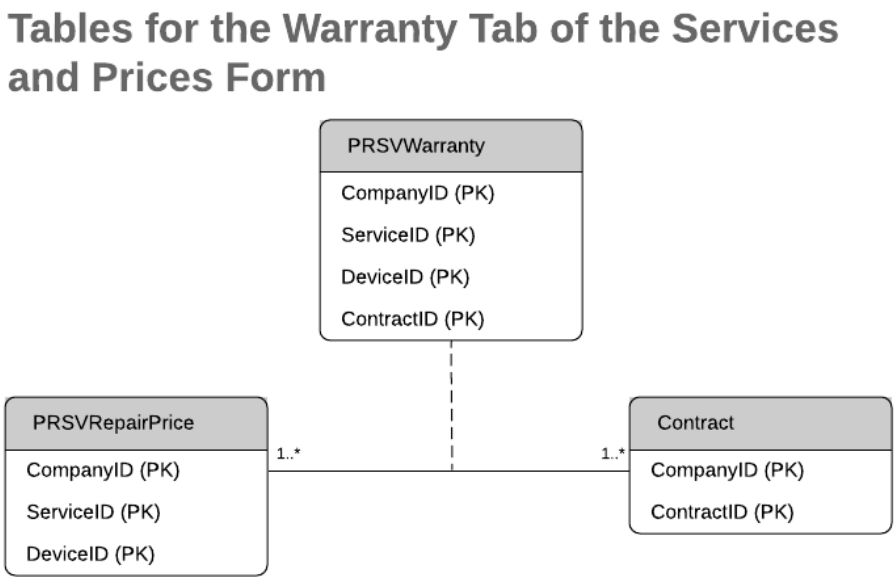


Figure: The Warranty tab

Relationships Between Database Tables

The repair prices for particular services and devices (`RSSVRepairPrice`) and the contract templates (`CT.ContractTemplate`, which are stored in the `Contract` database table) have a many-to-many relationship. The many-to-many links between records are stored in the separate `RSSVWarranty` custom table. The following diagram illustrates the relationships between the database tables that are used in this lesson.



Lesson Objectives

In this lesson, you will learn how to add a default detail record to the grid.

Step 4.2.1: Adding the Warranty Tab—Self-Guided Exercise

In this step, you will add the **Warranty** tab of the Services and Prices (RS203000) form on your own. Although this is a self-guided exercise, you can use the details and suggestions in this topic as you create the form. You have learned how to add a tab to a form in *Lesson 3.1: Adding a New Tab*.

DAC

You should define the `RSSVWarranty` DAC, and set up its fields and attributes as follows:

- For the DAC, define the following attribute.

```
[PXCacheName("Warranty")]
```

- For the system fields, define the attributes as described in *Step 1.4.2: Configure the Attributes of the New DAC* in the *T200 Maintenance Forms* training course or in *Audit Fields*, *Concurrent Update Control (TStamp)*, and *Attachment of Additional Objects to Data Records (NoteID)* in the documentation.
- Mark the `ServiceID`, `DeviceID`, and `ContractID` fields as the key fields.
- Configure a master-detail relationship between the `RSSVRepairPrice` DAC and the `RSSVWarranty` DAC.
- For the `ContractID` field, use the `PXDimensionSelector` attribute as shown in the following code.

```
[PXDimensionSelector(ContractTemplateAttribute.DimensionName,
    typeof(Search<ContractTemplate.contractID,
        Where<ContractTemplate.baseType.IsEqual<
            CTPRType.contractTemplate>>>),
        typeof(ContractTemplate.contractCD),
    DescriptionField = typeof(ContractTemplate.description))]
```

> Add the `PX.Objects.CT` `using` directive to the file that contains the definition of the `RSSVWarranty` DAC to use the `ContractTemplateAttribute`, `ContractTemplate`, and `CTPRType` classes.

- For the `DefaultWarranty` field, use the attributes shown in the following code. Because you will use the `PXDefault` attribute with a constant as an argument specifying the default value for the field and the `PersistingCheck` property set to `PXPersistingCheck.Nothing`, the data field will have a default value and will not be required to be entered.

```
#region DefaultWarranty
[PXDBBool()]
[PXUIField(DisplayName = "Default Warranty")]
[PXDefault(false, PersistingCheck = PXPersistingCheck.Nothing)]
public virtual bool? DefaultWarranty { get; set; }
public abstract class defaultWarranty :
    PX.Data.BQL.BqlBool.Field<defaultWarranty> { }
#endregion
```

- Add the following unbound fields and add `PXFormula` attribute to them as the following code shows. This way you will be able to display values from the `ContractTemplate` DAC without joining it in a data view.

```
#region ContractDuration
[PXInt(MinValue = 1, MaxValue = 1000)]
[PXUIField(DisplayName = "Duration", Enabled = false)]
[PXFormula(typeof(
```

```
            Selector<RSSVWarranty.contractID, ContractTemplate.duration>))]
        public virtual int? ContractDuration { get; set; }
        public abstract class contractDuration :
            PX.Data.BQL.BqlInt.Field<contractDuration> { }
        #endregion

        #region ContractDurationType
        [PXString(1, IsFixed = true)]
        [PXUIField(DisplayName = "Duration Unit", Enabled = false)]
        [Contract.durationType.List]
        [PXFormula(typeof(
            Selector<RSSVWarranty.contractID, ContractTemplate.durationType>))]
        public virtual string ContractDurationType { get; set; }
        public abstract class contractDurationType :
            PX.Data.BQL.BqlString.Field<contractDurationType> { }
        #endregion

        #region ContractType
        [PXString(1, IsFixed = true)]
        [PXUIField(DisplayName = "Contract Type", Enabled = false)]
        [Contract.type.List]
        [PXFormula(typeof(
            Selector<RSSVWarranty.contractID, ContractTemplate.type>))]
        public virtual string ContractType { get; set; }
        public abstract class contractType :
            PX.Data.BQL.BqlString.Field<contractType> { }
        #endregion
```

You can see the full example of the `RSSVWarranty` DAC in the `CodeSnippets` folder for this course, which is available on Acumatica GitHub.

Graph

You need to define the data view for the **Warranty** tab of the Services and Prices (RS203000) form as follows:

```
    public SelectFrom<RSSVWarranty>.
        Where<RSSVWarranty.deviceID.
            IsEqual<RSSVRepairPrice.deviceID.FromCurrent>.
        And<RSSVWarranty.serviceID.
            IsEqual<RSSVRepairPrice.serviceID.FromCurrent>>>.
        OrderBy<RSSVWarranty.defaultWarranty.Desc>.View
        Warranty;
```

Definitions of Controls on the ASPX Page

When you define controls for the **Warranty** tab of the Services and Prices (RS203000) form, you can use the following tips:

- Add a new `PXTabItem` control with a `PXGrid` control in it
- Specify the properties of the grid as follows:
  - `DataMember` (in the `PXGridLevel` control inside `PXGrid\Levels` in ASPX): *Warranty*
  - `SkinID` (in the `PXGrid` control in ASPX): *Details*
  - `Enabled` in `AutoSize` (in the `AutoSize` control inside `PXGrid` in ASPX): `True`
  - `Width` (in the `PXGrid` control in ASPX): `100%`

- Add columns for the `ContractID`, `ContractID_description`, `ContractDuration`, `ContractDurationType`, and `ContractType` data fields.
- Set the `CommitChanges` property of the control for the `ContractID` field to `True`.

Step 4.2.2: Inserting a Default Detail Record (with RowInserted)

In this step, you will implement the automatic addition of a warranty, which the system adds by default with every new service–device pair on the Services and Prices (RS203000) form. A default warranty is a contract template with the `ContractCD` field equal to *DFLTWARRNT*, which is preconfigured for this training course. This contract template is added as a `RSSVWarranty` record, which is a detail record for a `RSSVRepairPrice` record, after a price is successfully inserted into the cache object.

Insertion of a Record

You will insert a new record into the cache of `RSSVRepairPrice` records in the `RowInserted` event handler.

The `Warranty` data view will select the detail records of the new service–device pair, because it is already the pair referenced by the `Current` property of the `RSSVRepairPrice` cache. The `Current` property is set to the inserted record right before the `RowInserted` event. Therefore, you can use the `Warranty` data view to check that it does not contain any records before insertion of the default record.

You will keep the original value of the `IsDirty` property of the `RSSVWarranty` cache to hide the fact that the handler performs modifications in the cache. Because of this setting, if a user adds a new service–device pair and tries to leave the form before entering any data, no confirmation will be requested from the user. For more information about insertion of data records, see *Insertion of a Data Record*.

Instructions for Inserting a Default Detail Record

Do the following to insert the default detail data record:

1. In the `RSSVRepairPriceMaint` graph, add the fluent BQL constant that defines the string with the ID of the contract with the default warranty.

```
#region Constant
//The fluent BQL constant for the free warranty that is inserted by default
public const string DefaultWarrantyConstant = "DFLTWARRNT";
public class defaultWarranty : PX.Data.BQL.BqlString.Constant<defaultWarranty>
{
    public defaultWarranty()
        : base(DefaultWarrantyConstant)
    {
    }
}
#endregion
```

2. Add the following `using` directive.

```
using PX.Objects.CT;
```

3. Add the data view for the default warranty, as the following code shows.

```
//The view for the default warranty
public SelectFrom<ContractTemplate>.
    Where<ContractTemplate.contractCD.IsEqual<defaultWarranty>>.
    View DefaultWarranty;
```

4. Add the following `RowInserted` event handler to the `RSSVRepairPriceMaint` graph.

```
//Insert the default detail record.
protected virtual void _(Events.RowInserted<RSSVRepairPrice> e)
{
    if (Warranty.Select().Count == 0)
    {
        bool oldDirty = Warranty.Cache.IsDirty;
        // Retrieve the default warranty.
        Contract defaultWarranty = (Contract)DefaultWarranty.Select();
        if (defaultWarranty != null)
        {
            RSSVWarranty line = new RSSVWarranty();
            line.ContractID = defaultWarranty.ContractID;
            line.DefaultWarranty = true;
            // Insert the data record into
            // the cache of the Warranty data view.
            Warranty.Insert(line);
            // Clear the flag that indicates in the UI whether the cache
            // contains changes.
            Warranty.Cache.IsDirty = oldDirty;
        }
    }
}
```

   In this code, you ensure that there are no `RSSVWarranty` records for the new `RSSVRepairPrice` record, and you insert a new `RSSVWarranty` record with the default warranty. You retrieve the default warranty through the `DefaultWaranty` data view and use it to set the fields of the new warranty line.

5. Rebuild the project.

Instructions for Testing the Logic

To test the logic you have implemented, open the Services and Prices (RS203000) form, and do the following:

1. Create a record, and in the Summary area, specify the *Screen Repair* service and the *Samsung Galaxy S4* device.
2. On the **Warranty** tab, ensure that the default warranty has been added automatically for the new service–device pair.
3. Navigate away from the page without saving the new pair. No dialog box should appear asking you to confirm that you want to leave the page.

Related Links

- *Insertion of a Data Record*
- *Cancellation of Attribute Event Handlers*
- *PXCache Class*
- *RowInserted Event*
- *FieldDefaulting Event*
- *Constants in Fluent BQL*

Step 4.2.3: Adding UI Representation Logic (with RowSelected and RowDeleting)

You will now add UI presentation logic for `RSSVWarranty` detail data records.

You will use the `RowDeleting` event handler to prevent removal of the default line. In this event handler, you will throw `PXException` if a user tries to delete the default line. The exception message will not be displayed if the line is deleted along with the parent item.

You will use the `RowSelected` event handler to make the default line unavailable for editing. In this event handler, you will invoke the `SetEnabled()` method, which doesn't specify the field, so the method makes unavailable or available for editing all fields for the given data record. Here you will make unavailable for editing all fields of the warranty line corresponding to the default warranty; you will make available for editing all fields for all other warranty lines.

Preventing the Removal of the Default Line

To prevent a user from deleting the default line on the Services and Prices (RS203000) form, do the following:

1. In the `Helper\Messages.cs` file, add the following constant for the error message.

```
public static class Messages
{
    ...
    public const string DefaultWarrantyCanNotBeDeleted =
        "The default warranty cannot be deleted.";
}
```

2. Implement the `RowDeleting` event handler of the `RSSVWarranty` DAC in the `RSSVRepairPriceMaint` graph as the following code shows.

```
    //Prevent removal of the default warranty.
    protected virtual void _(Events.RowDeleting<RSSVWarranty> e)
    {
        if (e.Row.DefaultWarranty != true) return;

        if (e.ExternalCall && RepairPrices.Current != null &&
            RepairPrices.Cache.GetStatus(RepairPrices.Current) !=
            PXEntryStatus.Deleted)
        {
            throw new PXException(Messages.DefaultWarrantyCanNotBeDeleted);
        }
    }
```

Making the Default Line Unavailable for Editing

To make the default line unavailable for editing, do the following:

1. Add the following `RowSelected` event handler of the `RSSVWarranty` DAC to the `RSSVRepairPriceMaint` graph.

```
    //Make the default warranty unavailable for editing.
    protected virtual void _(Events.RowSelected<RSSVWarranty> e)
    {
        RSSVWarranty line = e.Row;
        if (line == null) return;
        PXUIFieldAttribute.SetEnabled<RSSVWarranty.contractID>(e.Cache,
            line, line.DefaultWarranty != true);
    }
```

2. Rebuild the project.

3. Publish the customization project to include the latest changes in the project.

Testing the Logic

To make sure that the event handlers work as expected, refresh the Services and Prices (RS203000) form, and do the following:

1. In the Summary area, create a record for the *Screen Repair* service and the *Samsung Galaxy S4* device.
2. On the **Warranty** tab, notice that all columns in the line for the default warranty are disabled: You can't edit or copy a value in any column.
3. Click the row with the default warranty and click **Delete Row** on the table toolbar. The error is displayed for the row as shown in the following screenshot.

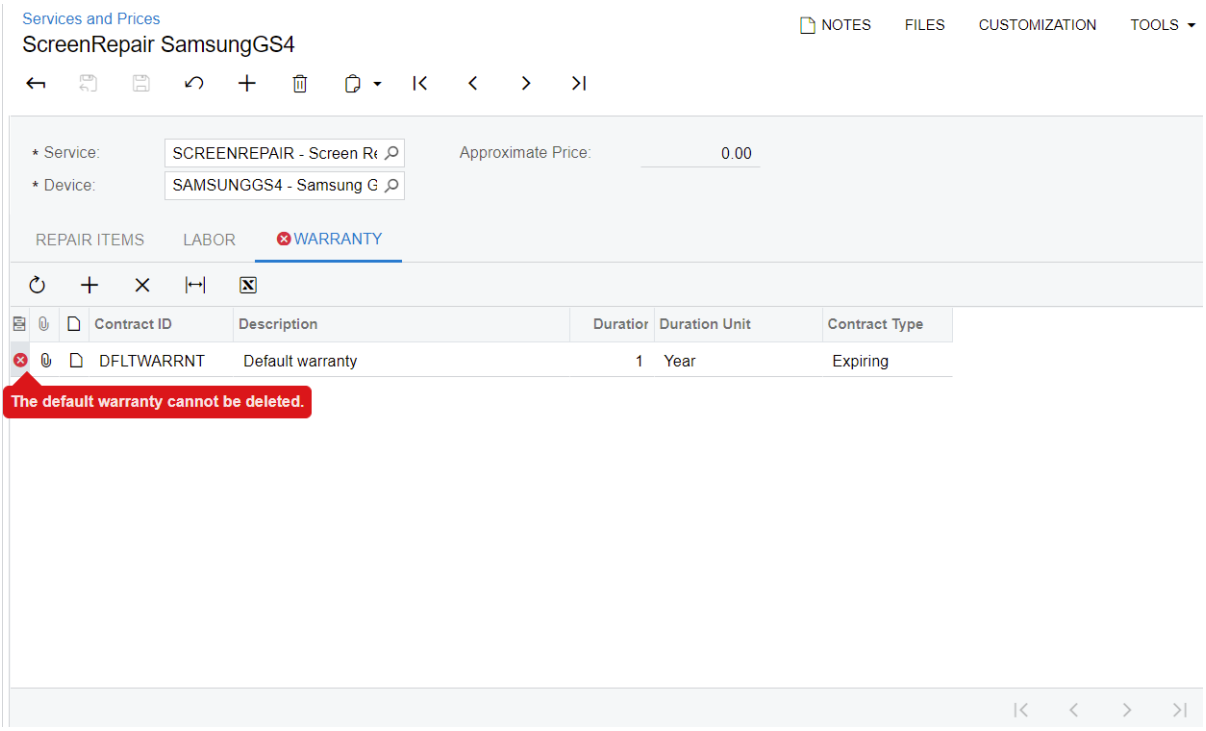


Figure: The error on the form

4. On the **Repair Items** tab, add a row with the following settings:
   - **Repair Item Type**: *Screen*
   - **Required**: Selected
   - **Inventory ID**: *SCRSGS4*
5. Save the changes.

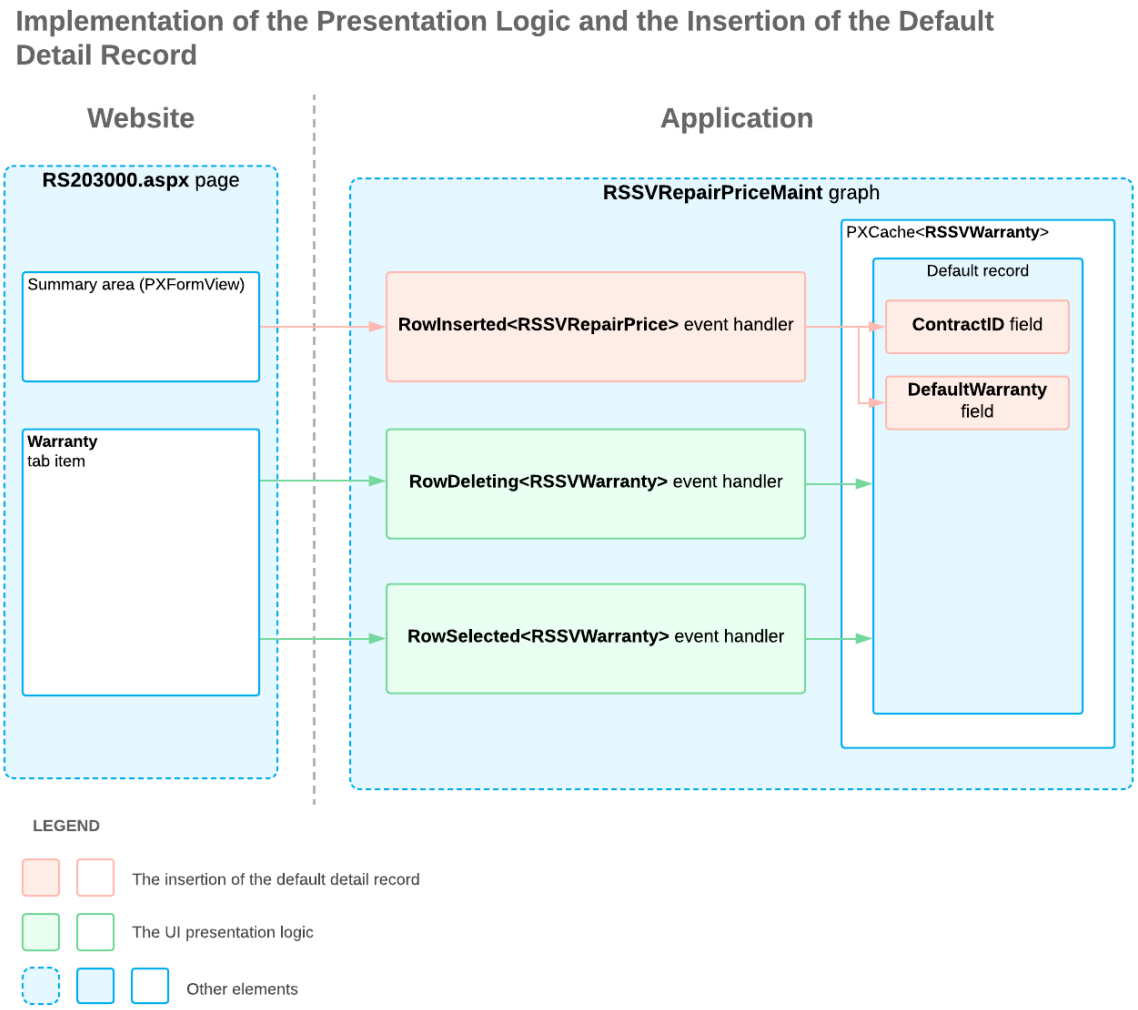
Related Links

- *RowSelected Event*

Lesson Summary

In this lesson, you have learned how to add a default detail record to the grid. To add a default record, you have used the `RowInserted` event handler for the parent DAC. You have also defined the UI presentation of the default detail record in the `RowSelected` and `RowDeleting` event handlers for the child DAC.

The implementation of the insertion of the default detail record and of the UI presentation logic is shown in the following diagram.



Review Questions

1. In which of the following cases would you use the `IsDirty` property of `PXCache` in an event handler?
   a. To hide the fact that the handler does modifications in the cache
   b. To hide the fact that the UI presentation logic has been changed
   c. To update the detail lines after a line has been changed
2. In which of the following event handlers would you implement the insertion of a default detail data record?

a. `RowSelected` for the parent DAC
b. `RowSelected` for the child DAC
c. `RowInserted` for the parent DAC
d. `RowInserted` for the child DAC

Answer Key

1. a
2. c

Appendix: Use of Event Handlers

This topic lists the scenarios in which particular event handlers have been used in this course.

Table: Use of Event Handlers

Event	Scenario	Examples in the Guide
`FieldUpdated`	Update of the fields of a data record on update of a field of this record	*Step 2.2.2: Updating Fields of the Same Record on Update of a Field (with FieldUpdated and FieldDefaulting)*
`FieldDefaulting`	Setting of a default value of a field that depends on other field values	*Step 2.2.2: Updating Fields of the Same Record on Update of a Field (with FieldUpdated and FieldDefaulting)*
`RowDeleting`	Prevention of the removal of a data record at runtime	*Step 4.2.3: Adding UI Representation Logic (with RowSelected and RowDeleting)*
`RowInserted`	Insertion of a detail record	*Step 4.2.2: Inserting a Default Detail Record (with RowInserted)*
`RowSelected`	Configuration of the UI logic: • Making a box or tab available or unavailable at runtime • Making a row unavailable for editing	• *Step 1.1.6: Making the Custom Field Conditionally Available (with RowSelected)* • *Step 3.1.4: Hiding the Tab from the Form (with RowSelected)* • *Step 4.2.3: Adding UI Representation Logic (with RowSelected and RowDeleting)*
`RowUpdated`	Update of the field of a data record on update of a field of another record	*Step 2.2.3: Updating a Field of Another Record on Update of a Field (with RowUpdated)*

Appendix: Reference Implementation

You can find the reference implementation of the customization described in this course in the `Customization\T210` folder of the *Help-and-Training-Examples* repository in Acumatica GitHub.

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

If for some reason you cannot complete the instructions in *Step 2: Preparing the Needed Acumatica ERP Instance for the Training Course* in Initial Configuration, you can create an Acumatica ERP instance as described in this topic and manually publish the needed customization project as described in *Appendix: Publishing the Required Customization Project*.

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

1. To deploy a new application instance, open the Acumatica ERP Configuration Wizard, and do the following:
   a. On the Database Configuration page, type the name of the database: `PhoneRepairShop`.
   b. On the Tenant Setup page, set up a tenant with the *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 a 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 perform customization of the website.

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

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

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

Appendix: Publishing the Required Customization Project

> If for some reason you cannot complete the instructions in *Step 2: Preparing the Needed Acumatica ERP Instance for the Training Course* in Initial Configuration, you can create an Acumatica ERP instance as described in *Appendix: Deploying the Needed Acumatica ERP Instance for the Training Course* and manually publish the needed customization project as described in this topic.

Load the customization project with the results of the *T200 Maintenance Forms* training course and publish this project as follows:

1. On the Customization Projects (SM204505) form, create a project with the name *PhoneRepairShop* and open it.
2. On the menu of the Customization Project Editor, click **Source Control > Open Project from Folder**.
3. In the dialog box that opens, specify the path to the `Customization\T200\PhoneRepairShop` folder, which you have downloaded from Acumatica GitHub, and click **OK**.
4. Bind the customization project to the source code of the extension library as follows:
   a. Copy the `Customization\T200\PhoneRepairShop_Code` folder to the `App_Data\Projects` folder of the website.

      > By default, the system uses the `App_Data\Projects` folder of the website as the parent folder for the solution projects of extension libraries.
      >
      > If the website folder is outside of the `C:\Program Files (x86)` and `C:\Program Files` folders, we recommend that you use the `App_Data\Projects` folder for the project of the extension library.

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

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

The published customization project contains all changes to Acumatica ERP website and database that have been performed during the completion of the *T200 Maintenance Forms* training course. This project also contains the customization plug-in that fills in the tables created in the *T200 Maintenance Forms* training course with the custom data entered in this training course. For details about the customization plug-ins, see *To Add a Customization Plug-In to a Project*. The creation of customization plug-ins is outside of the scope of this course.