Developer Course

Customization

T190 Quick Start in Customization 2022 R1

Revision: 4/14/2022

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: 04/14/2022

Introduction

The *T190 Quick Start in Customization* training course shows how to implement the most common scenarios of the customization of Acumatica ERP, such as the creation of custom fields, the validation of the value of a custom field, and the creation of Acumatica ERP entities by using actions. This course briefly describes Acumatica ERP customization techniques, which are described in greater detail in the following training courses: *T200 Maintenance Forms*, *T210 Customized Forms and Master-Detail Relationship*, *T220 Data Entry and Setup Forms*, *T230 Actions*, *T240 Processing Forms*, and *T250 Inquiry Forms*.

This 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 develop particular customization scenarios for a cell phone repair shop; these customization scenarios are fully performed in the training courses of the *T* series. We recommend that you take the courses in the *T* series after you complete this course to expand your understanding of Acumatica Framework and the Acumatica ERP customization tools.

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

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

How to Use This Course

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

1. Complete *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 *T190 Certification Test: Quick Start in Customization*.

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

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 Acumatica ERP user interface. Before you begin completing lessons, you should make sure you have the needed knowledge and background.

Required Knowledge and Background

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

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

Initial Configuration

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

Step 1: Preparing the Environment

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. The reinstallation of Acumatica ERP does not affect existing Acumatica ERP instances. For details, see *To Install the Acumatica ERP Tools*.

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

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

1. 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*.

Company Story and Customization Description

In this course, you will implement particular customization scenarios to support the cell phone repair shop of the Smart Fix company. The full implementation of the customization for the cell phone repair shop is described in the following training courses: *T200 Maintenance Forms*, *T210 Customized Forms and Master-Detail Relationship*, *T220 Data Entry and Setup Forms*, *T230 Actions*, *T240 Processing Forms*, and *T250 Inquiry Forms*.

> We recommend that you complete these training courses after you complete this course to have a better understanding of Acumatica Framework and Acumatica ERP customization tools.

The initial customization project that you will import and publish in the beginning of this course will contain the following new custom forms:

- The Repair Services (RS201000) maintenance form, which the Smart Fix company uses to manage the lists of repair services that the company provides
- The Serviced Devices (RS202000) maintenance form, which the Smart Fix company uses to manage the lists of devices that can be serviced
- The Services and Prices (RS203000) maintenance form, which provides users with the ability to define and maintain the price for each repair service provided by the Smart Fix company
- The Repair Work Orders (RS301000) data entry form, which is used to create and manage work orders for repairs
- The Repair Work Order Preferences (RS101000) setup form, which an administrative user uses to specify the company's preferences for the repair work orders

In this course, you will customize the Stock Items (IN202500) form of Acumatica ERP to mark particular stock items as repair items—that is, items that are used for repair services.

You will also implement the following scenarios as you complete this course:

- Update of a field value that depends on another field value on the Services and Prices custom maintenance form
- Validation of a field value on the Repair Work Orders custom data entry form
- Creation of an SO invoice for a repair work order on the Repair Work Orders form

Lesson 1: Creating a Customization Project

In this lesson, you will create a customization project for this training course, load the customization project prepared for the course, bind the customization project to the extension library (which contains the customization source code), and publish this project.

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

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

For details on customization projects, see *Customization Project*.

Lesson Objectives

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

- Create a customization project
- Load a customization project from a local folder
- Bind a customization project to an extension library
- Publish a customization project

Step 1.1: Creating a Customization Project

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

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

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

Related Links

- *To Create a New Project*

Step 1.2: Loading Items to the Customization Project

In this step, you will begin working in the Customization Project Editor and will load customization items from the customization project that has been prepared for this training course into the customization project created in the previous lesson. The customization project that has been prepared for this training course contains the changes that have been implemented for the customization of Acumatica ERP for the Smart Fix company. (These changes are described in *Company Story and Customization Description*.)

Loading the Items to the Customization Project

To load items form the customization project prepared for this training course, do the following:

1. On the Customization Projects (SM204505) form, click *PhoneRepairShop* in the table to open the customization project that you have created.
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\T190\SourceFiles\PhoneRepairShop` folder, which you have downloaded from Acumatica GitHub in *Initial Configuration*, and click **OK**.

The items of the customization project have been loaded, as shown in the following screenshot.

> The Customized Screens page can contain rows with empty **Screen ID** and **Title** because the customization project has not been published yet.

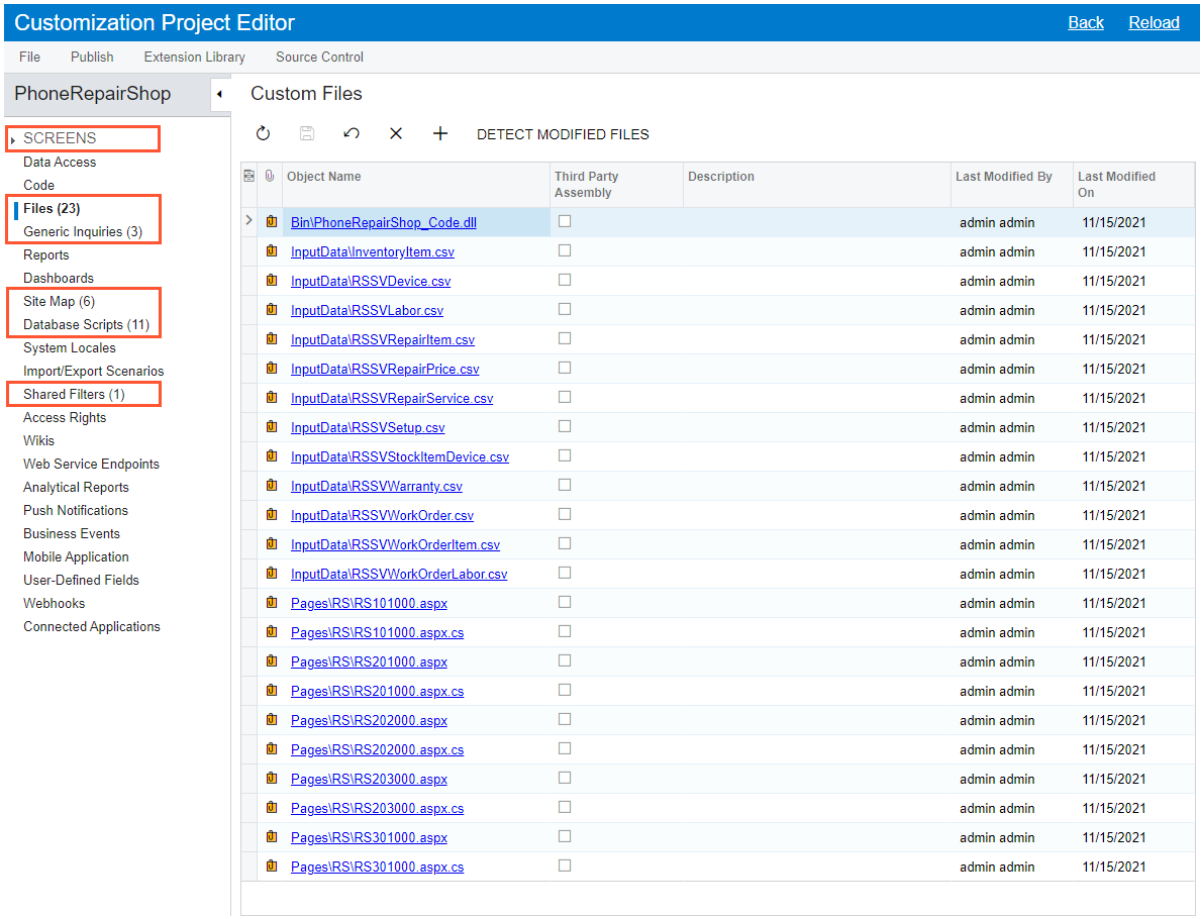


Figure: Items of the customization project

Related Links

- *To Update the Content of a Project from a Local Folder*

Step 1.3: Binding the Extension Library

In this step, you will bind the *PhoneRepairShop* customization project to the extension library that contains source code of customization items you have loaded to the customization project.

Binding the Extension Library

To bind the customization project to the source code of the extension library, do the following:

1. Copy the `Customization\T190\SourceFiles\PhoneRepairShop_Code` folder to the `App_Data\Projects` folder of the Acumatica ERP instance that you have prepared for this training course.

   > 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.

2. Open the Visual Studio solution and build the `PhoneRepairShop_Code` project.
3. Reopen the *PhoneRepairShop* project in the Customization Project Editor.
4. On the menu of the Customization Project Editor, click **Extension Library > Bind to Existing**.
5. In the dialog box that opens, specify the path to the `App_Data\Projects\PhoneRepairShop_Code` folder, and click **OK**.

Related Links

- *Extension Library (DLL) Versus Code in a Customization Project*

Step 1.4: Publishing the Customization Project

In this step, you will publish the *PhoneRepairShop* customization project to apply the changes in this project to the Acumatica ERP instance.

Publishing the Customization Project

To publish the project, do the following:

1. Open the *PhoneRepairShop* customization project in the Customization Project Editor.
2. Click **Files** on the left pane of the Customization Project Editor. The Custom Files page opens.
3. On the page toolbar, click **Detect Modified Files**.

   Because you have rebuilt the extension library in the `PhoneRepairShop_Code` Visual Studio project, the `Bin\PhoneRepairShop_Code.dll` file has been modified.
4. In the **Modified Files Detected** dialog box, which opens, make sure the **Selected** check box is selected for the `Bin\PhoneRepairShop_Code.dll` file, and click **Update Customization Project**.
5. Close the dialog box.
6. On the menu of the Customization Project Editor, click **Publish > Publish Current Project**.

   The **Compilation** panel opens, which shows the progress of the publication.

7. Close the **Compilation** panel when the publication has completed and the *Website updated* message is displayed.

Related Links

- *Publishing Customization Projects*

Step 1.5: Reviewing the Changes in Acumatica ERP

In this step, you will review the changes to the Acumatica ERP instance that have been applied as a result of the publication of the *PhoneRepairShop* customization project.

Reviewing the Changes

Review the changes as follows:

1. Open your Acumatica ERP instance for the training course.
2. On the main menu, notice the **Phone Repair Shop** workspace, (shown in the following screenshot), which had not been on the menu previously. The workspace contains the custom Acumatica ERP forms that have been developed for the Smart Fix company, as well as the Stock Items (IN202500) form, which will be modified as part of the customization in this course.

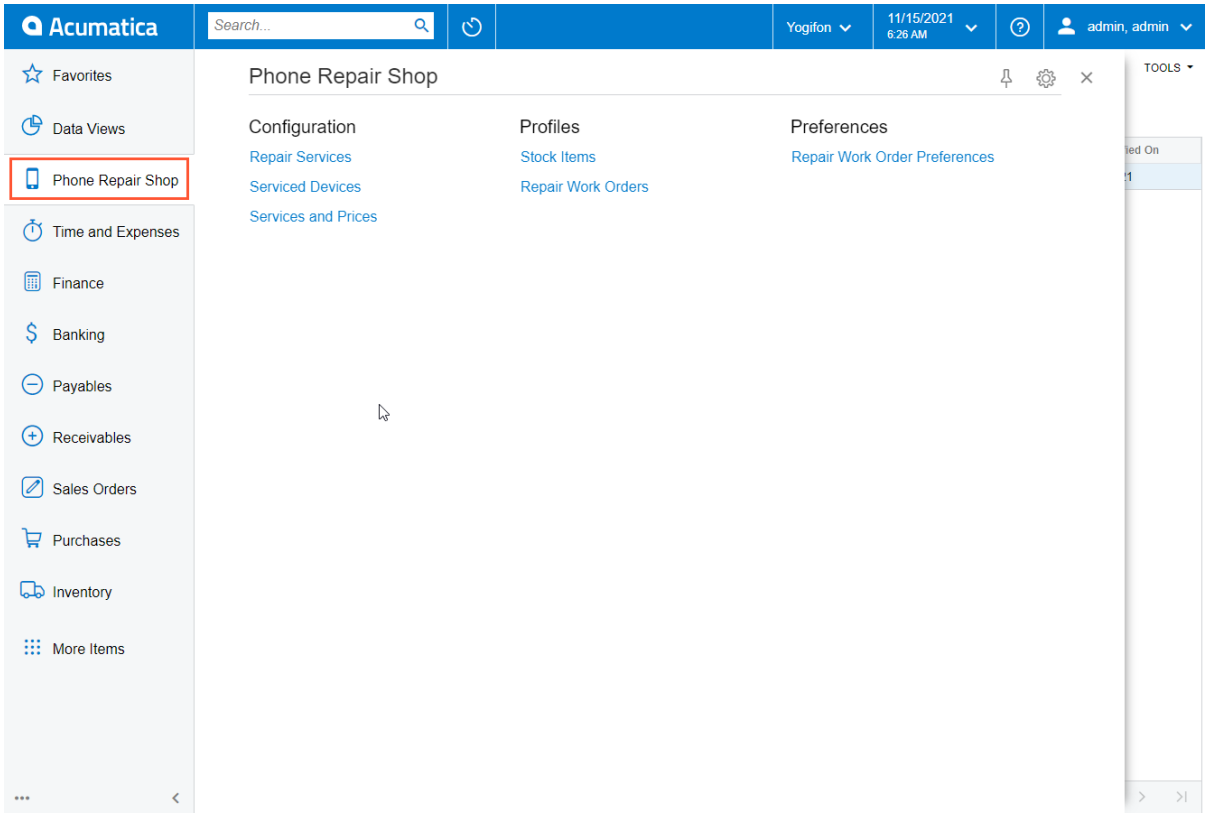


Figure: The Phone Repair Shop workspace

3. Open the Repair Services (RS201000) form, and review its content and functionality, which is shown in the following screenshot.

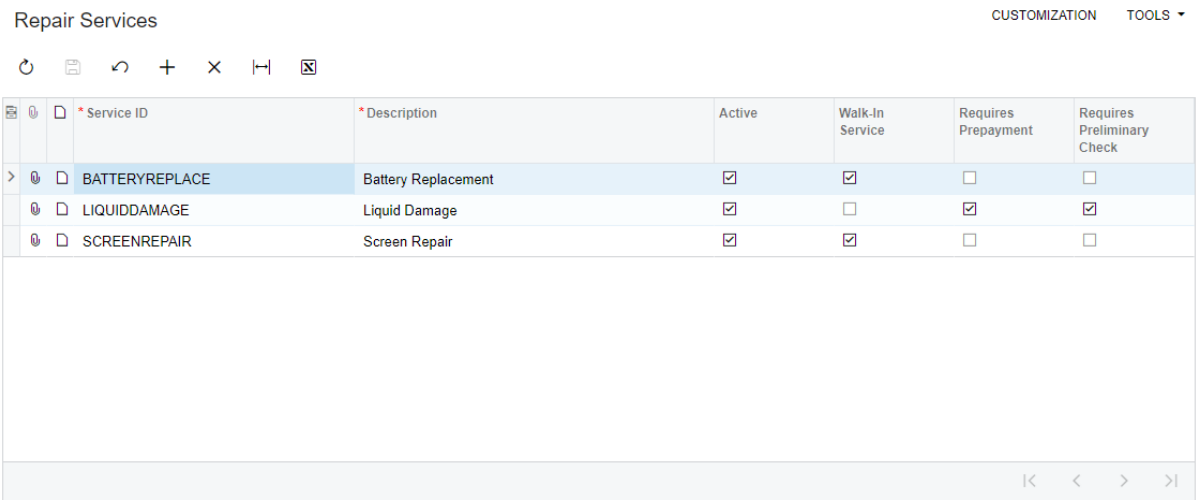


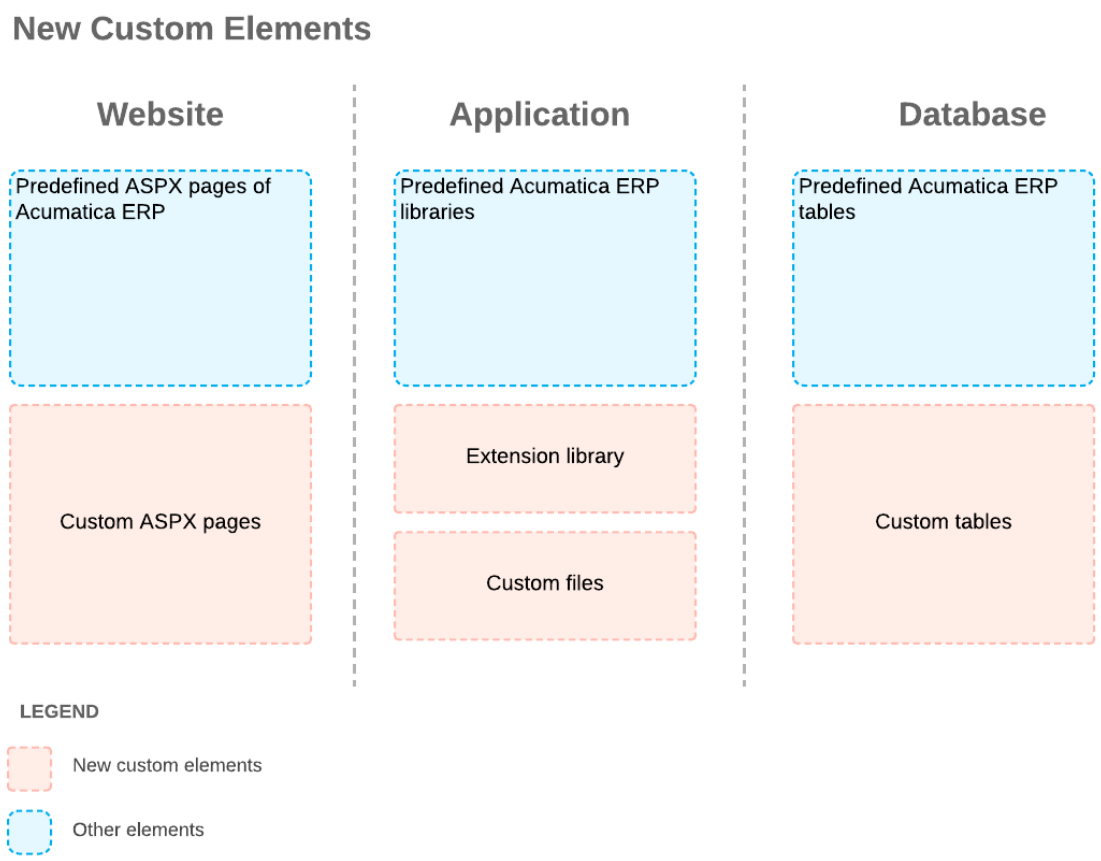
Figure: The Repair Services custom form

4. Open any other forms in the **Phone Repair Shop** workspace, and review their content and functionality.
5. In Microsoft SQL Server Management Studio, connect to the database of your Acumatica ERP instance for the training course. Find the database tables with names that start with *RSSV*. These are the custom tables that have been added during the publication of the customization project.
6. Open the Acumatica ERP instance folder in the file system. Notice the following files and folders:
   - `Pages\RS`: Contains the ASPX code of the custom forms. The forms have the *RS* prefix in their IDs; therefore, they are placed in the custom `RS` subfolder.
   - `InputData`: Contains CSV files with the data for the custom tables. This data is inserted in the database by the `InputData` customization plug-in, which is included in the customization project.
   - `CstPublished\pages_RS`: Contains the published code of the custom ASPX pages.
   - `Bin\PhoneRepairShop_Code.dll`: Contains the customization source code in an extension library.

Lesson Summary

In this lesson, you have learned how to create a customization project, load content to a customization project from a local folder, bind the project to an extension library, and publish the project.

The following diagram shows the changes that have been applied to the Acumatica ERP instance for the training course after the customization project has been published.



Review Questions

1. Which of the following actions will publish the customization project opened in the Customization Project Editor to the current Acumatica ERP instance?
   a. On the menu of the Customization Project Editor, you click **Publish > Publish Current Project**.
   b. On the menu of the Customization Project Editor, you click **Extension Library > Bind to Existing**.
   c. On the menu of the Customization Project Editor, you click **Source Control > Open Project from Folder**.

Answer Key

1. a

Additional Information: Customization Project Management

In this lesson, you have created and published a customization project. Other ways to manage customization projects is outside of the scope of this course, as is the management of different types of items in customization projects.

You can find examples of ways to work with customization projects and to include different types of items in a customization project in the following training courses:

- *T200 Maintenance Forms*
  - *Step 1.1.1: Create the Customization Project*
  - *Step 1.1.2: Add a Database Table Schema*
  - *Step 1.2.1: Use the New Screen Wizard to Create a Form Template*
  - *Step 1.3.3: Update the SiteMapNode Item*
  - *Step 1.8.1: Create an Extension Library*
  - *Step 2.3.3: Update the Files in the Customization Project*
  - *Step 2.4.2: Add the Site Map Item to the Customization Project*
  - *Step 2.4.3: Add the Form to the Screen Editor*
  - *Step 2.5.3: Save the Generic Inquiry to the Customization Project*
- *T210 Customized Forms and Master-Detail Relationship*
  - *Step 1.1.3: Viewing the Content of the Customization Project*
- *T220 Data Entry and Setup Forms*
  - *Step 1.1.5: Adding the Substitute Form with a Shared Filter to the Project*
- *T240 Processing Forms*
  - *Step 3.1.1: Including a Report in the Customization Project*
- *W140 Customization Projects*
  -
  -
  -
  -

Lesson 2: Creating 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).

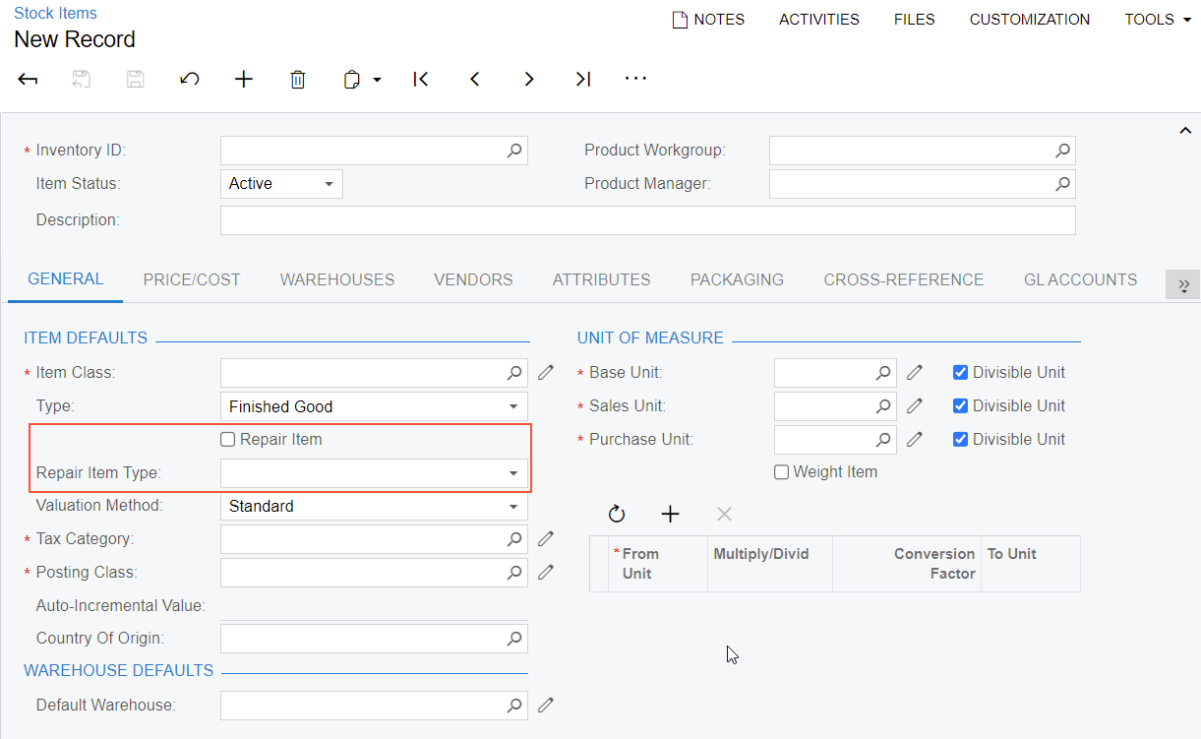


Figure: Custom elements to be added to the Stock Items form

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 2.1: 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 *Changes in the Application Code (C#)*.

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.

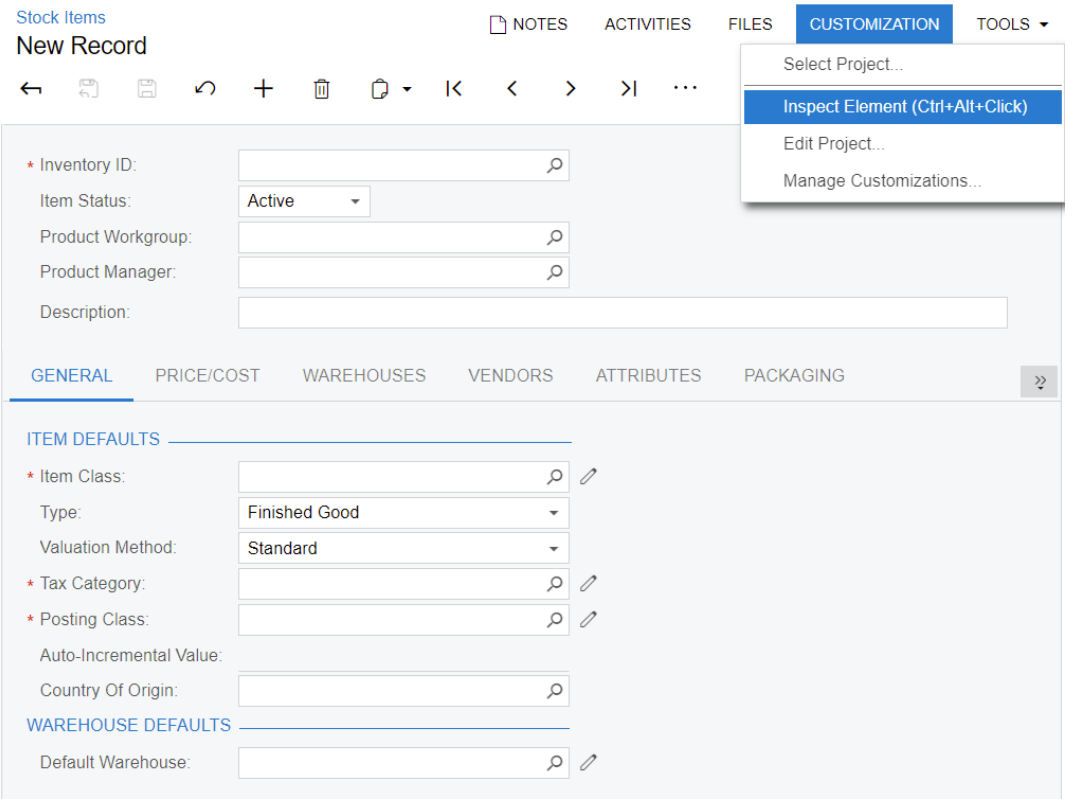


Figure: Customization menu

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.

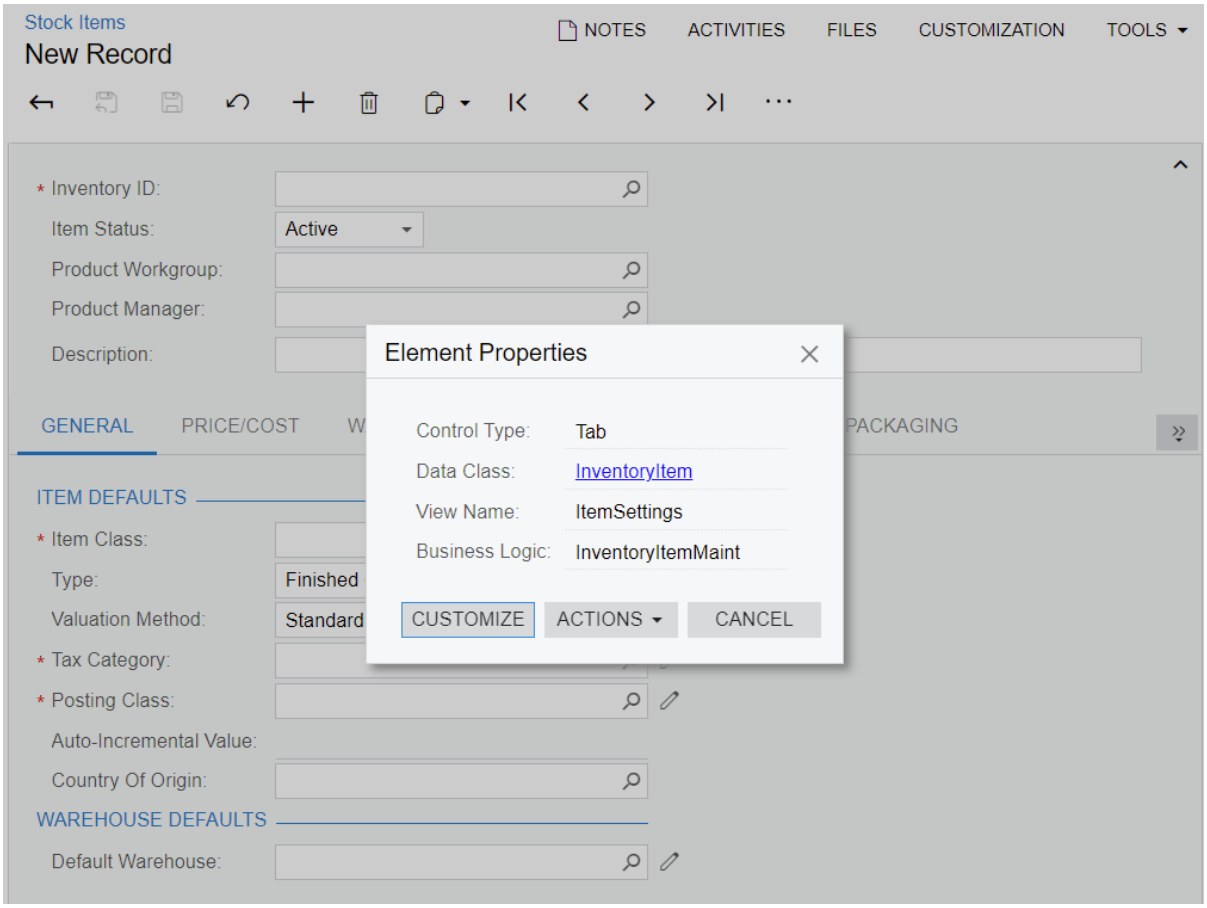


Figure: Element Properties dialog box

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.

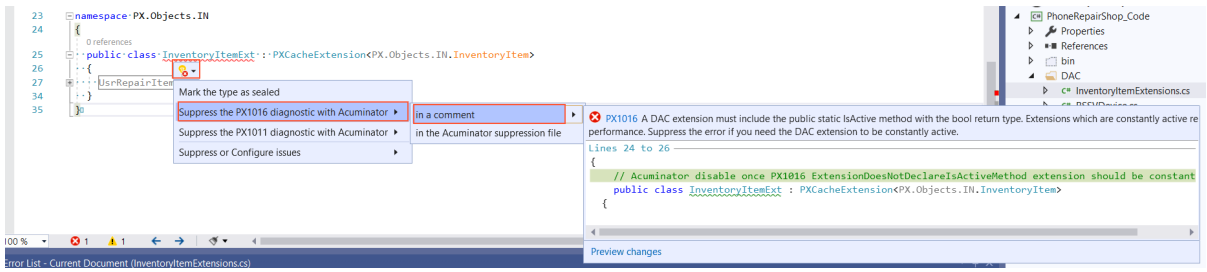


Figure: Suppression of the error in a comment

   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.

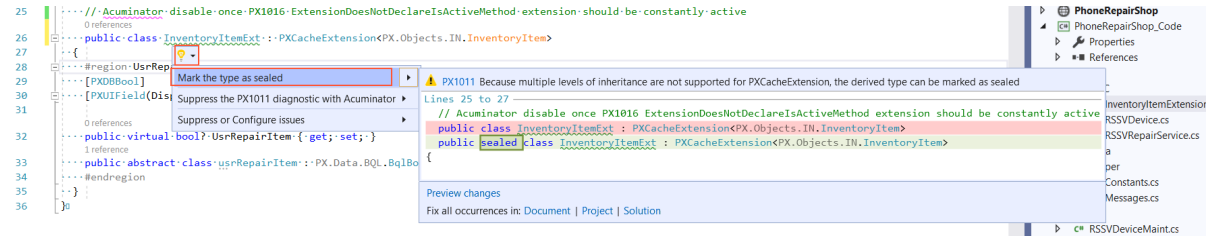


Figure: Fix of the warning

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 2.2: 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.

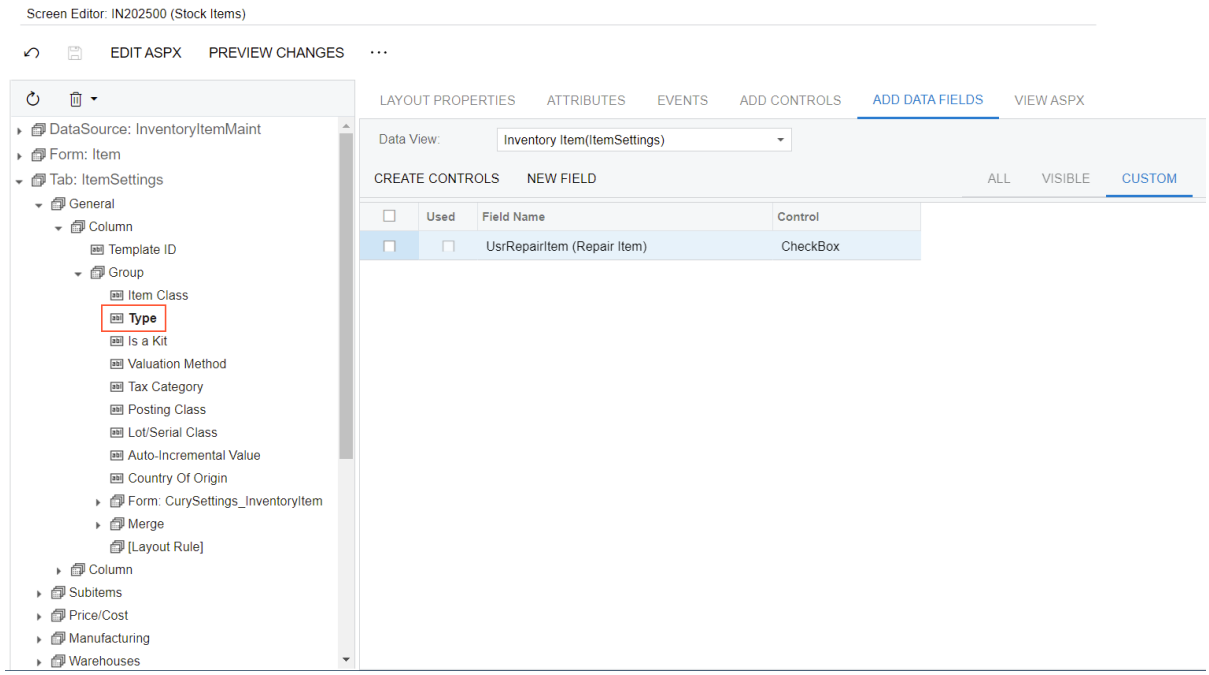


Figure: The Type node in the control tree

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.

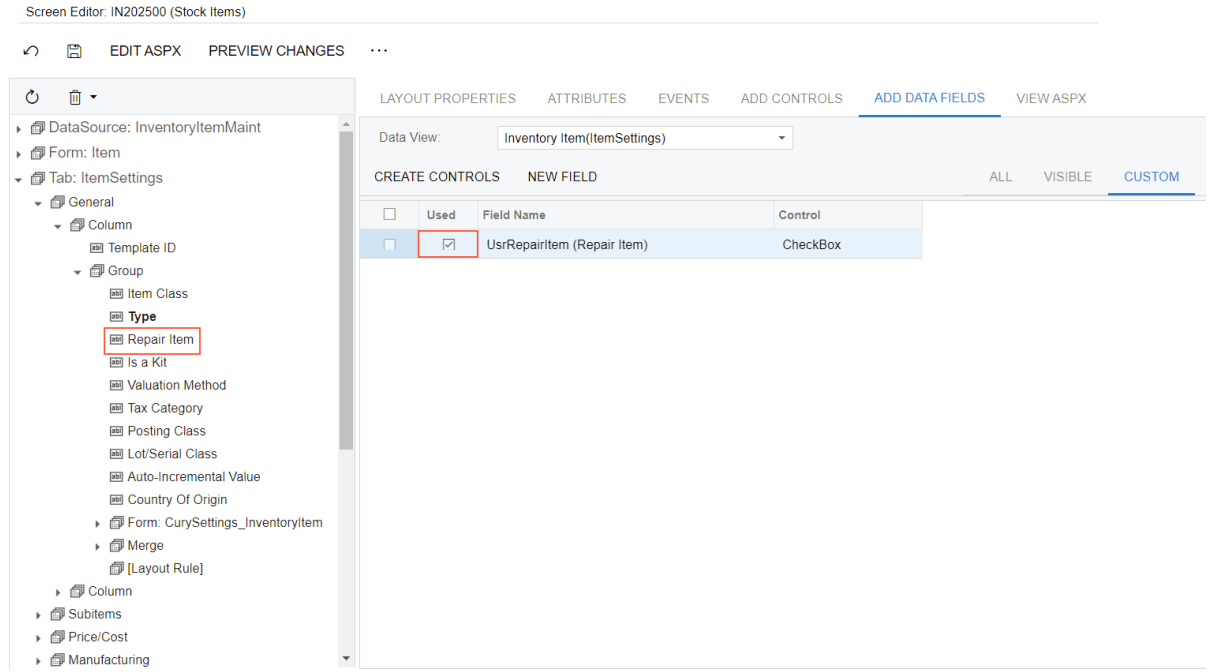


Figure: The added control

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. 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; 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.

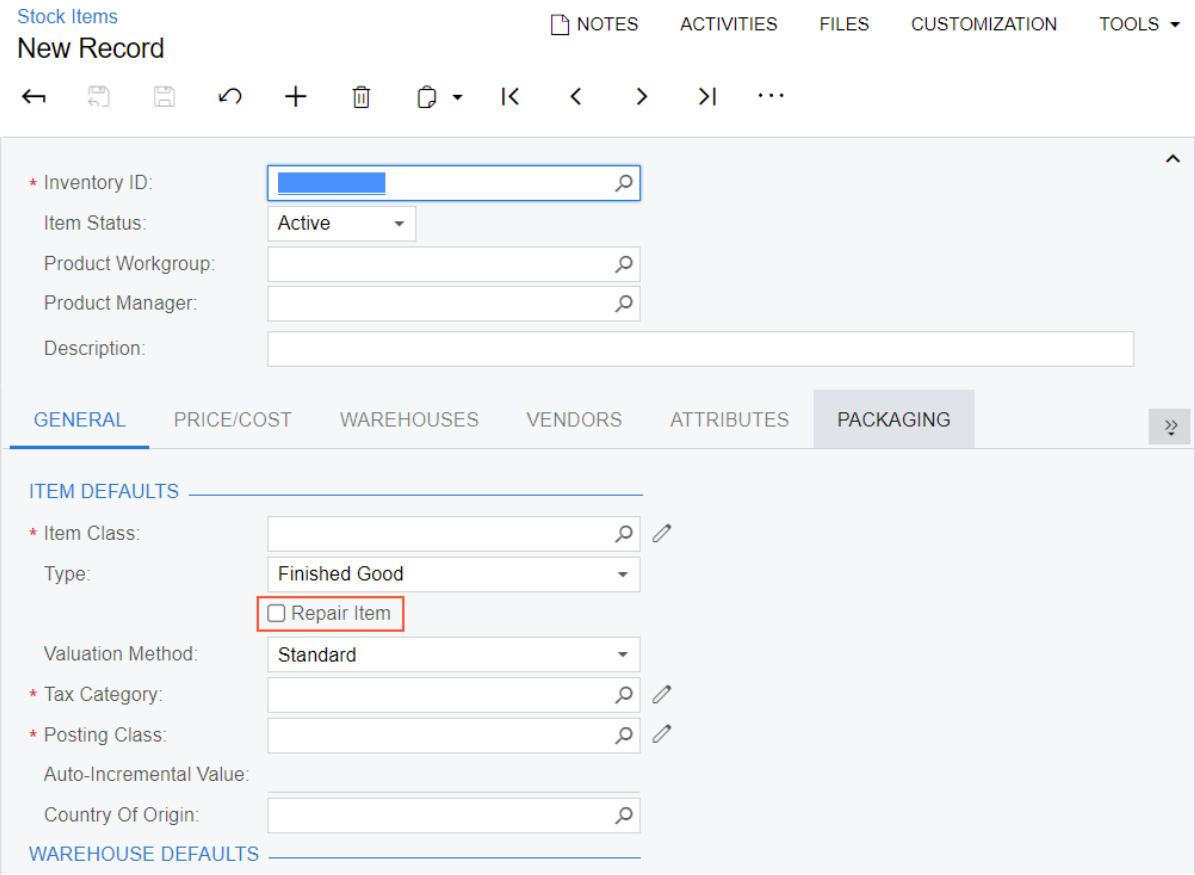


Figure: The Repair Item check box

Related Links

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

Step 2.3: 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 2.4: 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.

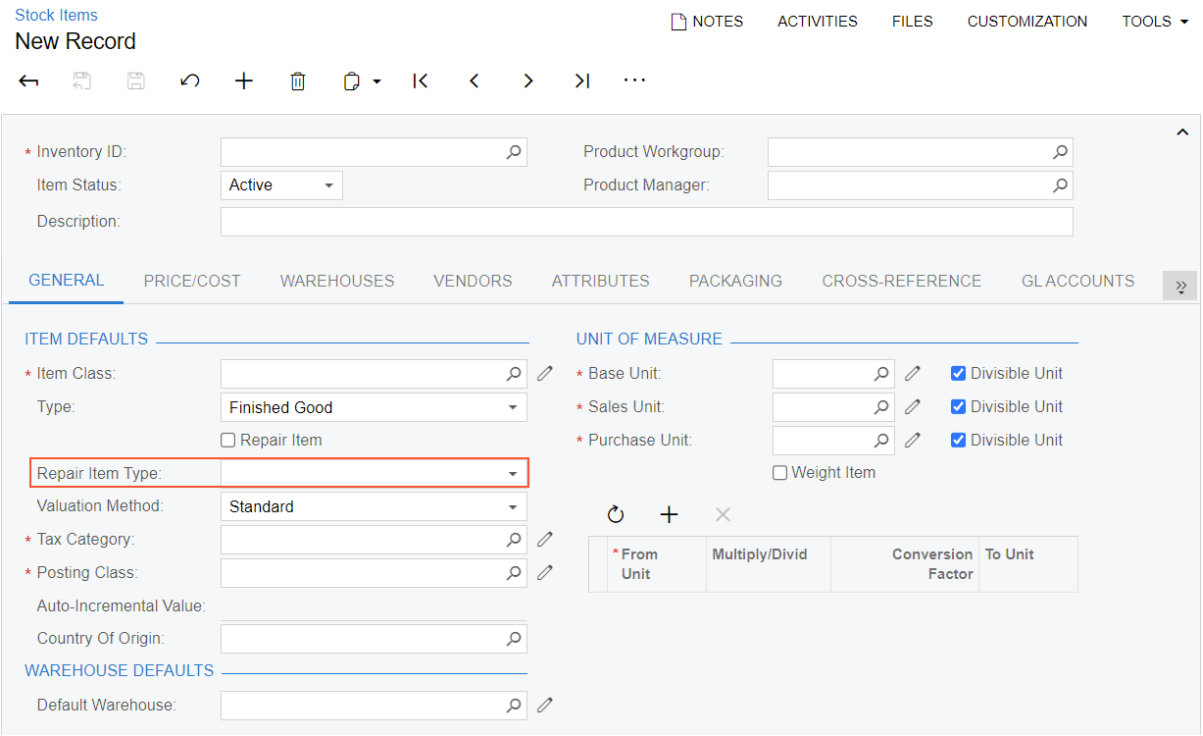


Figure: The Repair Item Type box

Related Links

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

Step 2.5: 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.

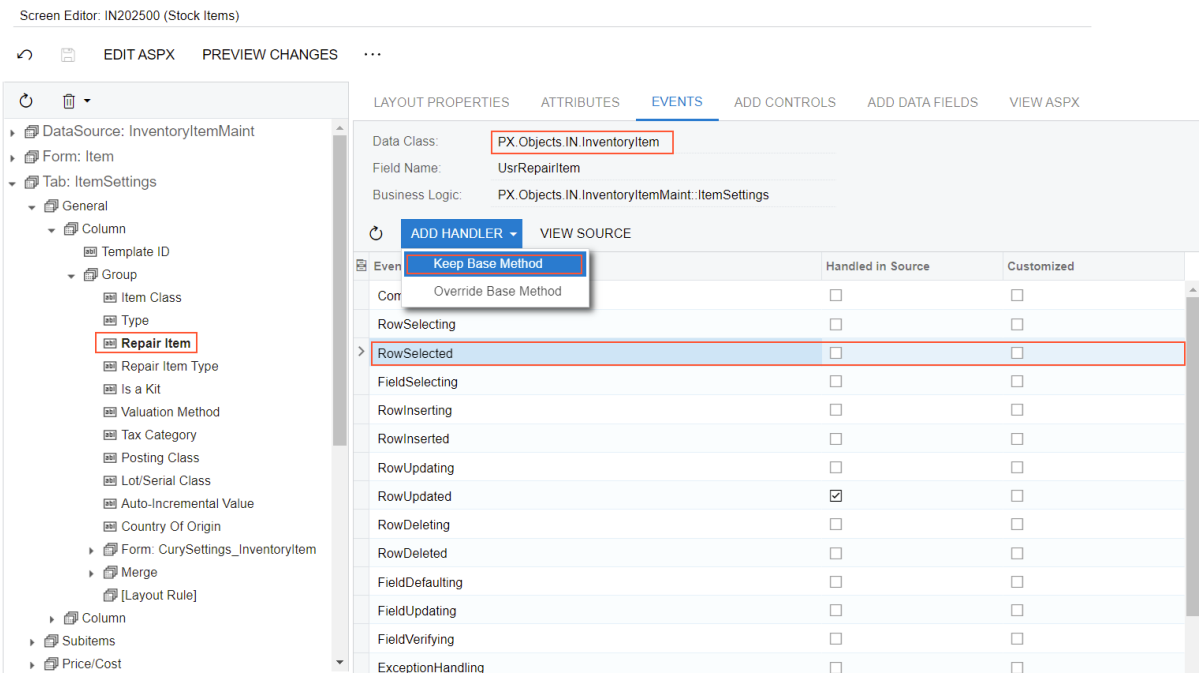


Figure: The generation of the event handler

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. Update the customization project with the changes you have made in this lesson, and publish the project.
5. Open the Screen Editor for the *Stock Items* (IN202500) form.
6. Set the `CommitChanges` property to `True` for the `UsrRepairItem` data field, as the following screenshot shows.

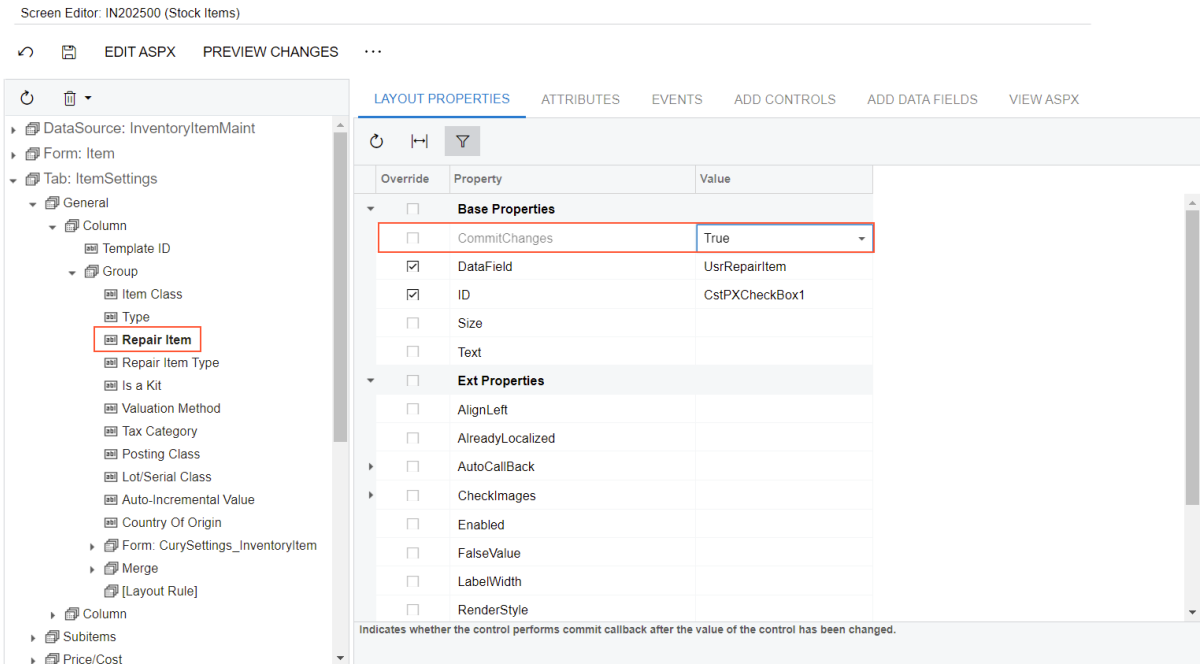


Figure: The CommitChanges property

7. Click **Save** to save the changes to the customization project.

8. Publish the customization 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 2.6: Testing the Customized Form

Now you will modify the *BAT3310*, *BAT3310EX*, and *BCOV3310* stock item records to indicate that they are repair items. To do this, perform the following actions:

1. On the Stock Items (IN202500) form, select the *BAT3310* stock item.
2. 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.
3. In the **Item Defaults** section of the **General** tab, specify the following settings:
   - **Repair Item**: Selected

     Notice that once you select the check box, the **Repair Item Type** box becomes available.
   - **Repair Item Type**: *Battery*
4. On the form toolbar, click **Save** to save the record in the database.
5. Repeat the previous instructions to modify the *BAT3310EX* and *BCOV3310* stock item records as specified in the following table.

UI Element (Location)	First Modified Record	Second Modified Record
Inventory ID (Summary area)	`BAT3310EX`	`BCOV3310`
Repair Item (**Item Defaults** section of the **General** tab)	Selected	Selected
Repair Item Type (**Item Defaults** section of the **General** tab)	*Battery*	*Back Cover*

Lesson Summary

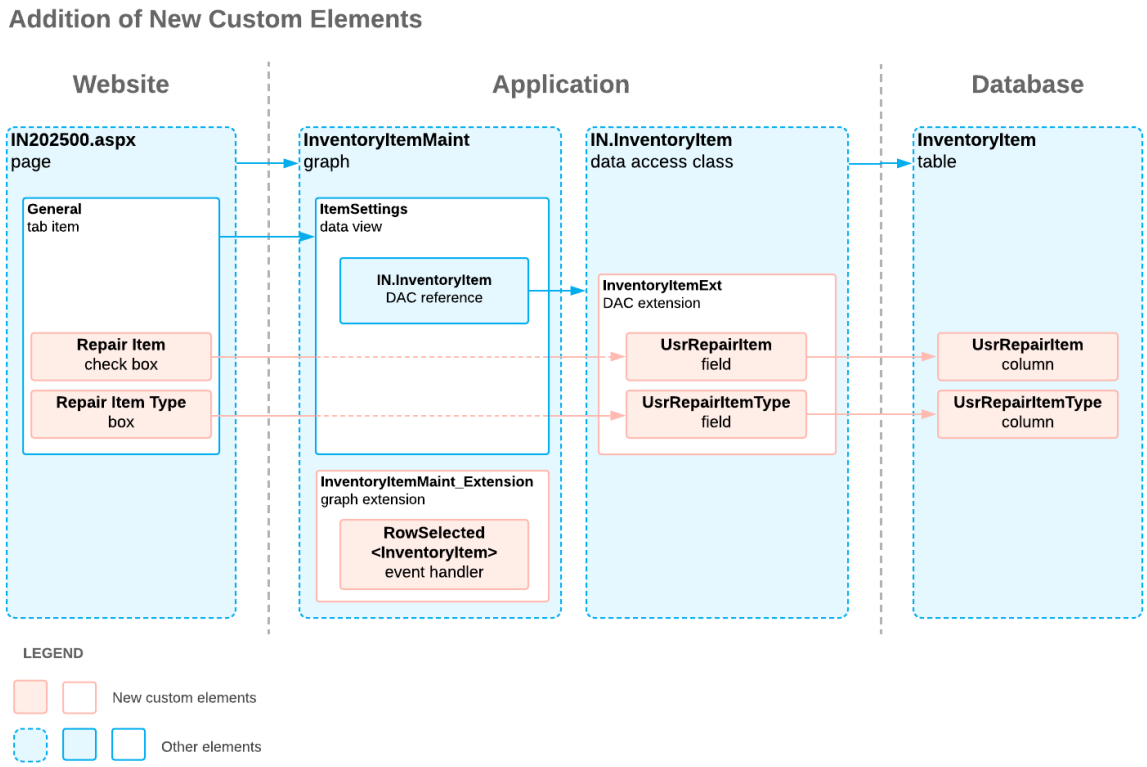
In this lesson, you have learned how to create a control so that you can display on a form a custom field bound to the database. To implement this customization, you have learned how to add the necessary modifications to a customization project and how to publish the project to apply the changes to the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

Additional Information: Custom Elements

In this lesson, you have learned how to create a custom check box and drop-down list on an Acumatica ERP form. The creation of other custom elements, such as tabs, is outside of the scope of this course.

You can find information about the creation of custom elements, along with examples, in the following lessons of the *T210 Customized Forms and Master-Detail Relationship* training course:

- *Lesson 1.1: Adding Custom Fields*
- *Lesson 3.1: Adding a New Tab*

Lesson 3: Implementing the Update and Validation of Field Values

In this lesson, you will modify the business logic of the Services and Prices (RS203000) and Repair Work Orders (RS301000) forms.

The **Repair Items** tab of the Services and Prices form lists the repair items that are used to perform this particular service on the specific device. When a user adds a repair item to the list, the user selects the inventory ID (in the **Inventory ID** column) that corresponds to the needed repair item. To display consistent data on the form, for a particular row, if a value is selected in the **Inventory ID** column, the values in the **Repair Item Type** and **Price** columns must be changed to the repair item type and base price (respectively) of the selected stock item, as specified on the Stock Items (IN202500) form.

The Repair Work Orders form is a data entry form on which users create and manage work orders for repairs. The system uses the data defined on the Services and Prices form for this particular service on the specified device to automatically fill in the default values of particular elements on the Repair Work Orders form and validate the values of particular elements. On the **Labor** tab of the form, the user entering the order lists the services to be provided and their quantities. For each row on the **Labor** tab, the value in the **Quantity** column must satisfy the following conditions:

- The value must be greater than or equal to 0.
- The value must be greater than or equal to the value in the **Quantity** column specified for the corresponding record on the **Labor** tab of the Services and Prices form (that is, the record that has the same inventory ID, service ID, and device ID on the Services and Prices form as the current row on the **Labor** tab of the Repair Work Orders form).

Lesson Objectives

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

- Update the fields of a data record on update of a field of this record
- Validate the value of a field that does not depend on the values of other fields of the same record

Step 3.1: Updating Fields of a Record on Update of a Field of This Record (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 3.2: Validating an Independent Field Value (with FieldVerifying)

In this step, you will implement the validation of the value in the **Quantity** column on the **Labor** tab of the Repair Work Orders (RS301000) form. For each row on this tab, the value in the **Quantity** column must be greater than or equal to 0. The value also must be greater than or equal to the value specified for the corresponding record on the **Labor** tab of the Services and Prices (RS203000) form (that is, the record that has the same inventory ID, service ID, and device ID on the Services and Prices form as the current row on the **Labor** tab of the Repair Work Orders form). Thus, a nonnegative quantity must be specified for each row, and the value specified for the labor on the **Labor** tab of the Services and Prices form (for the same service and device as those selected on the Repair Work Orders form) will function as a minimum quantity.

You will implement the `FieldVerifying` event handler for the `Quantity` field of the `RSSVWorkOrderLabor` DAC. This event handler is intended for field validation that is independent of other fields in the same data record. For details about the validation of independent field values, see *Validation of Field Values*.

In the event handler, you will do the following:

- When the new value in the **Quantity** column is negative, you will throw an exception (by using `PXSetPropertyException`) to cancel the assignment of the new value to the `Quantity` field.

- When the value is not negative but is smaller than the default quantity specified on the Services and Prices form (in the `RSSVLabor.Quantity` field), you will attach the exception to the field by using the `RaiseExceptionHandling` method and exit the method normally. This method will display a warning for the validated data field but will not raise an exception, so that the method finishes normally and `e.NewValue` is set.

  To attach a warning to the control, you will specify `PXErrorLevel.Warning` in the `PXSetPropertyException` constructor.

  > ⚠ `RaiseExceptionHandling`, which is used to prevent the saving of a record or to display an error or warning on the form, cannot be invoked on a `PXCache` instance in the following event handlers: `FieldDefaulting`, `FieldSelecting`, `RowSelecting`, and `RowPersisted`. For details, see *`RaiseExceptionHandling`* in the API Reference.

To select the default data record from the `RSSVLabor` DAC, you will configure a fluent BQL query with three required parameters. In the `Select()` method that executes the query, as the parameters, you will pass the values of `RSSVWorkOrder.ServiceID`, `RSSVWorkOrder.DeviceID`, and `RSSVWorkOrderLabor.InventoryID` from the row for which the event is triggered. To use parameters in a fluent BQL query, you need to add the `PX.Data.BQL using` directive to the code. For details about the parameters in fluent BQL, see *Parameters in Fluent BQL*.

Validating the Value of the Quantity Field

To validate the value of the `Quantity` field, do the following:

1. In the `Messages.cs` file, add the following constants to the `Messages` class.

```
public const string QuantityCannotBeNegative =
  "The value in the Quantity column cannot be negative.";
public const string QuantityToSmall = @"The value in the Quantity column
  has been corrected to the minimum possible value.";
```

2. In the `RSSVWorkOrderEntry.cs` file, add the following `using` directive (if it has not been added yet).

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

3. Add the following `FieldVerifying` event handler to the `RSSVWorkOrderEntry` graph.

```
//Validate that Quantity is greater than or equal to 0 and
//correct the value to the default if the value is less than the default.
protected virtual void _(Events.FieldVerifying<RSSVWorkOrderLabor,
  RSSVWorkOrderLabor.quantity> e)
{
    if (e.Row == null || e.NewValue == null) return;

    if ((decimal)e.NewValue < 0)
    {
        //Throwing an exception to cancel the assignment of the new value to the field
        throw new PXSetPropertyException(Messages.QuantityCannotBeNegative);
    }

    var workOrder = WorkOrders.Current;
    if (workOrder != null)
    {
        //Retrieving the default labor item related to the work order labor
        RSSVLabor labor = SelectFrom<RSSVLabor>.
            Where<RSSVLabor.serviceID.IsEqual<@P.AsInt>.
                And<RSSVLabor.deviceID.IsEqual<@P.AsInt>>.
```

```
                And<RSSVLabor.inventoryID.IsEqual<@P.AsInt>>>
            .View.Select(this, workOrder.ServiceID, workOrder.DeviceID,
e.Row.InventoryID);
        if (labor != null && (decimal)e.NewValue < labor.Quantity)
        {
            //Correcting the LineQty value
            e.NewValue = labor.Quantity;
            //Raising the ExceptionHandling event for the Quantity field
            //to attach the exception object to the field
            e.Cache.RaiseExceptionHandling<RSSVWorkOrderLabor.quantity>(e.Row,
e.NewValue,
                new PXSetPropertyException(Messages.QuantityToSmall,
PXErrorLevel.Warning));
        }
    }
}
```

4. Rebuild the project.
5. In the Screen Editor or in the ASPX code in Visual Studio, make sure that `CommitChanges` is set to `True` for the `Quantity` field in the grid of the **Labor** tab on the Repair Work Orders (RS301000) form.
6. Save your changes on the page.
7. Publish the customization project.

Testing the Validation

To check the validation, on the Repair Work Orders (RS301000) form, do the following:

1. Select the work order with the *000001* order number.
2. On the **Labor** tab, in the row for the *CONSULT* labor item, change the value in the **Quantity** column to `-1` and press Enter. Make sure the error is displayed, as shown in the following screenshot.

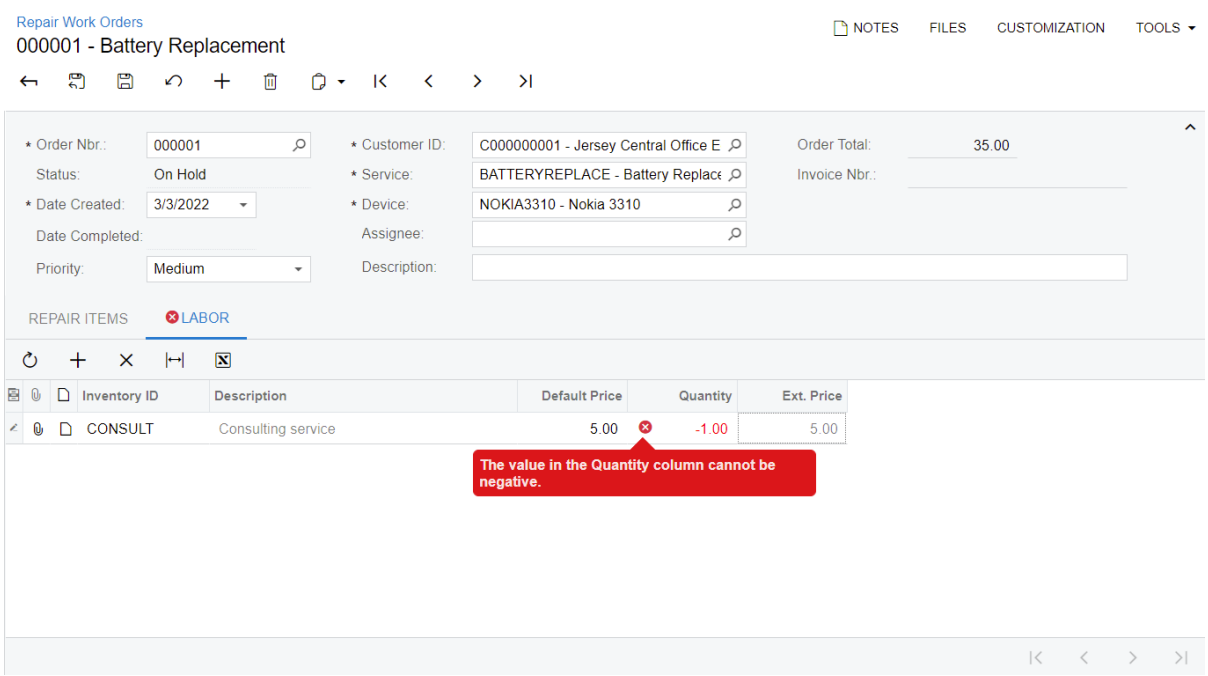


Figure: The error for a negative value

3. Change the value to `0.5`, which is smaller than the default value of 1. Make sure that the warning message is generated on the control and the value is corrected to 1, as shown in the following screenshot.

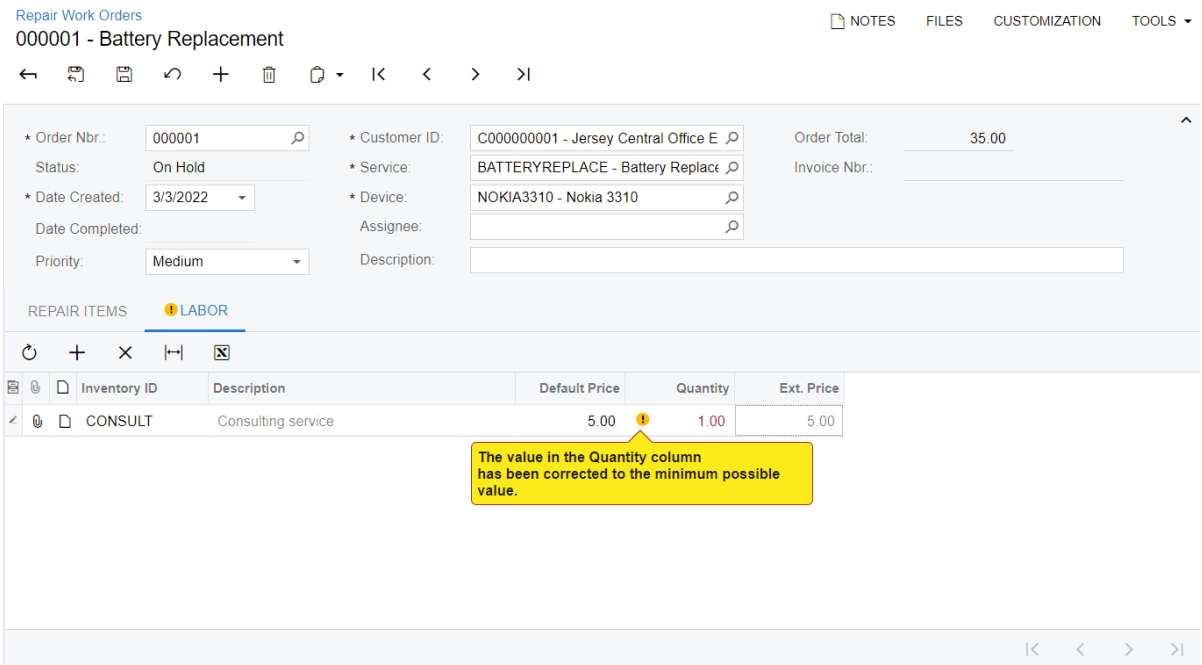


Figure: The warning message

4. Change the value to `2`. Make sure no warning or error is displayed.
5. Save the changes.

Related Links

- *Validation of Field Values*
- *Validation of a Data Record*
- *Parameters in Fluent BQL*
- *PXCache.RaiseExceptionHandling Method*

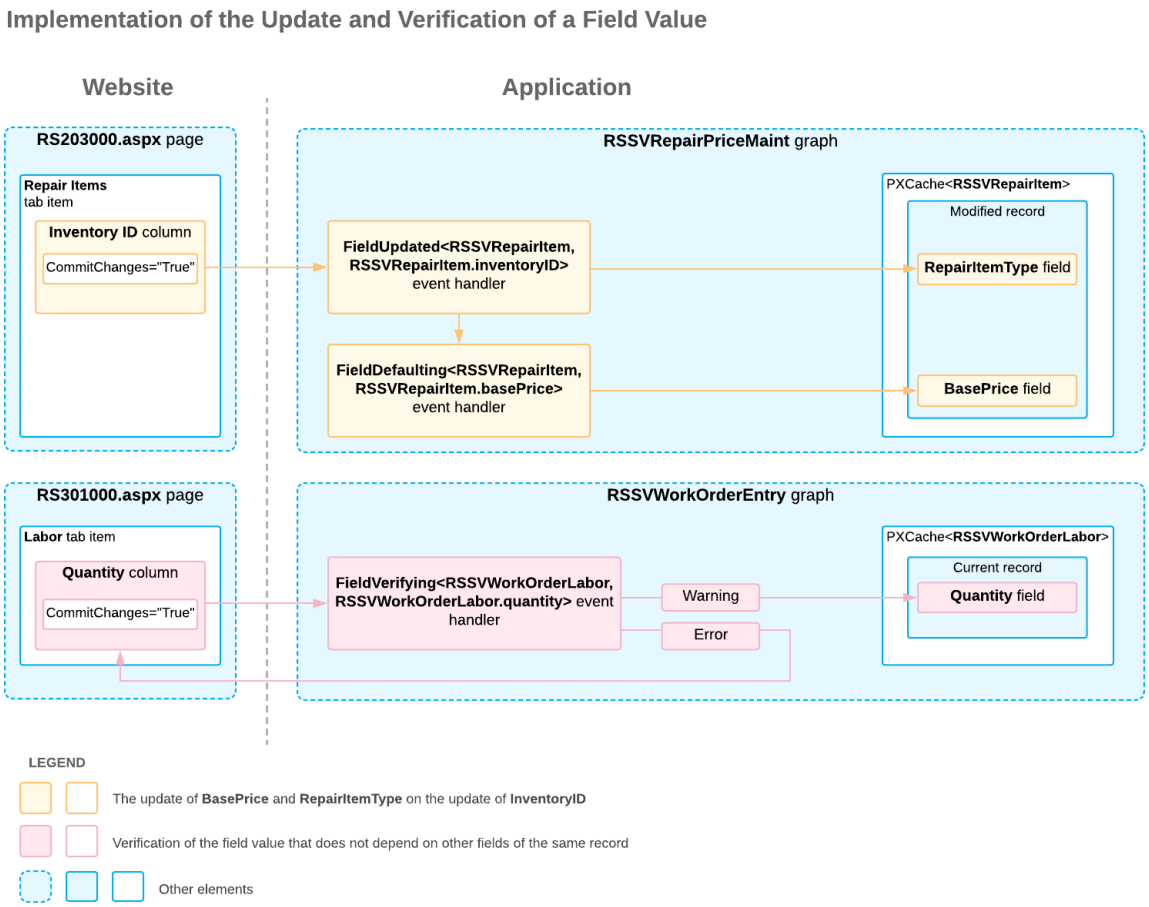
Lesson Summary

In this lesson, you have defined the business logic scenarios on the **Repair Items** tab of the Services and Prices (RS203000) form and on the **Labor** tab of the Repair Work Orders (RS301000) form.

You have used the `FieldUpdated` and `FieldDefaulting` event handlers to modify the values of a detail record on update of the **Inventory ID** column of this detail record. In the `FieldUpdated` event handler, you have used the `PXSelectorAttribute.Select<>()` method to obtain the stock item record with the inventory ID selected in the updated field.

To verify the value of a field that does not depend on other fields of the same record, you have used the `FieldVerifying` event handler. In this event handler, you have thrown an exception by using `PXSetPropertyException` to display an error and cancel the assignment of the new value. To display a warning, you have attached the exception to the field by using the `RaiseExceptionHandling` method.

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



Review Questions

1. Which of the following objects would you use to throw an exception to cancel the assignment of a new value to a field?
   a. `e.Cancel` of the `FieldVerifying` event handler
   b. `PXSetPropertyException`
   c. `RaiseExceptionHandling`
2. Which event handler should be used to validate an independent field value?
   a. `FieldDefaulting`
   b. `FieldSelecting`
   c. `FieldVerifying`
   d. `FieldUpdated`
   e. `RowSelected`
3. Which event handler is used to update a value of a dependent field within a particular data record?
   a. `FieldDefaulting`
   b. `FieldSelecting`
   c. `FieldVerifying`

   d. `FieldUpdated`
   e. `RowSelected`
4. How would you specify a required integer parameter in a fluent BQL query?
   a. `@P.AsInt`
   b. `Argument.AsInt`
   c. `@P`
   d. `Argument`

Answer Key

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

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*

Additional Information: Use of Event Handlers

In this lesson, you have learned in which situations you can use the `FieldUpdated` and `FieldVerifying` event handlers.

Although the use of other event handlers is outside of the scope of this course, you can find information about how to use other event handlers, along with examples, in the following training courses:

- *T200 Maintenance Forms*
  - *Step 1.6.1: Add an Event Handler in the Customization Project Editor*
  - *Step 1.9.1: Add an Event Handler in Visual Studio*
- *T210 Customized Forms and Master-Detail Relationship*
  - *Step 1.1.6: Making the Custom Field Conditionally Available (with RowSelected)*
  - *Step 2.2.2: Updating Fields of the Same Record on Update of a Field (with FieldUpdated and FieldDefaulting)*
  - *Step 2.2.3: Updating a Field of Another Record on Update of a Field (with RowUpdated)*
  - *Step 3.1.4: Hiding the Tab from the Form (with RowSelected)*
  - *Step 4.2.2: Inserting a Default Detail Record (with RowInserted)*
  - *Step 4.2.3: Adding UI Representation Logic (with RowSelected and RowDeleting)*
- *T220 Data Entry and Setup Forms*
  - *Step 1.2.1: Creating a Work Order from a Template (with RowUpdated)*
  - *Step 1.2.2: Updating Fields of the Same Record on Update of a Field (with FieldUpdated and FieldDefaulting) —Self-Guided Exercise*
  - *Step 1.3.1: Validating an Independent Field Value (with FieldVerifying)*
  - *Step 1.3.2: Validating Dependent Fields of Records (with RowUpdating)*
- *T230 Actions*
  - *Step 1.2: Specifying the Availability and Visibility of the Assign to Me Button and Command (with RowSelected)*
  - *Step 3.3: Specify the Availability of the Create Invoice Button and Command*
- *T240 Processing Forms*
  - *Step 2.2.4: Defining the External Presentation of Field Values (in FieldSelecting)*

Lesson 4: Creating an Acumatica ERP Entity Corresponding to a Custom Entity

For a repair work order to be billed and then paid, a user needs to create an invoice for the order. In this lesson, for the Repair Work Orders (RS301000) form, you will implement the `CreateInvoice` action, which initiates the creation of an invoice. You will also create the associated **Create Invoice** button on the form toolbar and the equivalent command on the More menu (under **Other**).

Creating an invoice might be a time-consuming operation, so it needs to be performed asynchronously. To perform asynchronous operations, Acumatica Framework provides the `PXLongOperation` class, which you will learn how to use in this lesson.

Lesson Objectives

In this lesson, you will learn how to implement an asynchronous operation by using the `PXLongOperation` class.

Step 4.1: Performing Preliminary Steps

In the Smart Fix company, there are no shipments or sales orders associated with repair work orders. Thus, you need to enable the *Advanced SO Invoices* feature on the *Enable/Disable Features* (CS100000) form so that during the creation of an SO invoice, stock items can be added directly to the SO invoice without sales orders and shipments being processed.

To turn on the feature, do the following:

1. On the form toolbar of the *Enable/Disable Features* (CS100000) form, click **Modify**.
2. Select the **Advanced SO Invoices** check box, and click **Enable** on the form toolbar.
3. On the *Item Classes* (IN201000) form, in the **Item Class Tree**, select *STOCKITEM*. All the stock items used in this lesson belong to this class.
4. On the **General** tab (**General** section), select the **Allow Negative Quantity** check box, as shown in the following screenshot.

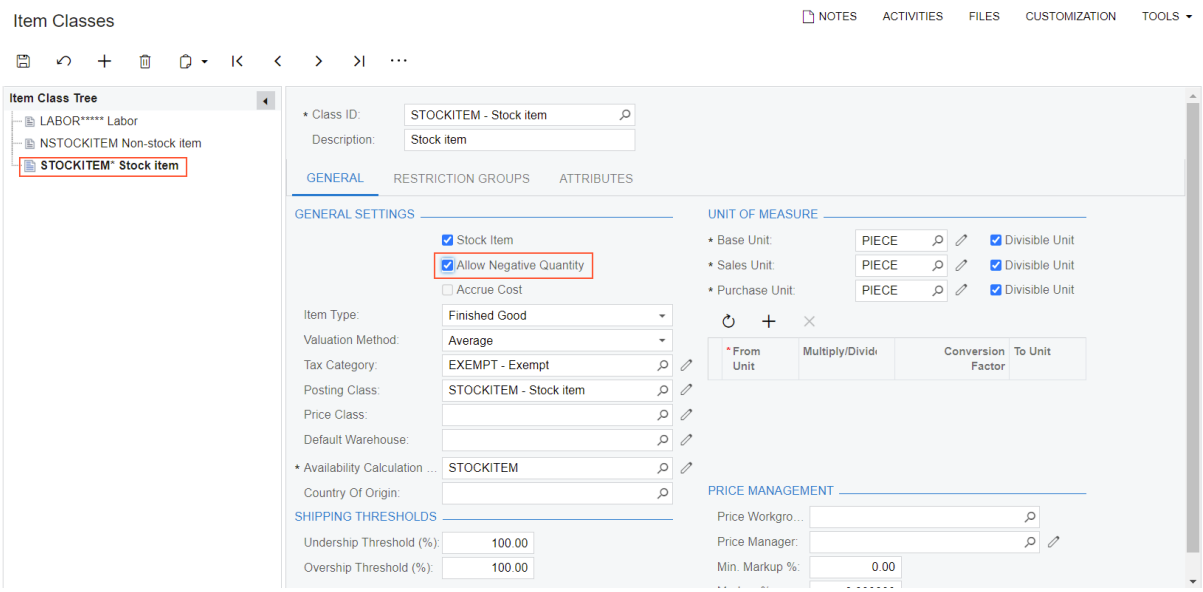


Figure: Item Classes form

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

Step 4.2: Defining the Logic of Creating an SO Invoice

You should define the method in which an invoice is created, and then you can call this method in the `PXLongOperation.StartOperation` method.

You will use multiple graphs in the method that creates an invoice. To save all changes from multiple graphs to the database, you will use a single `PXTransactionScope` object. It gives you the ability to avoid incomplete data being saved to the database if an error occurs in the middle of the method.

> In Acumatica ERP, there are two types of invoices that can be created for a customer: SO and AR. An SO invoice, which can include stock items, is an extension of an AR invoice, which cannot include stock items. Repair work orders usually have stock items; therefore, you will implement the creation of an SO invoice. However, regardless of your implementation, if an order does not include stock items, the system will create an AR invoice, and if an order includes stock items, the system will create an SO invoice.

To define the method in which the SO invoice is created, do the following:

1. Add the following `using` directives to the `RSSVWorkOrderEntry.cs` file (if they have not been added yet).

```
using PX.Objects.SO;
using PX.Objects.AR;
using System.Collections;
using System.Collections.Generic;
```

> Instead of adding the `using` directives manually, you can add them with the help of the Quick Actions and Refactorings feature of Visual Studio after you define the method in the next instruction.

2. Add the following static method, `CreateInvoice`, to the `RSSVWorkOrderEntry` graph. The `CreateInvoice` method creates the SO invoice for the current work order.

```
        private static void CreateInvoice(RSSVWorkOrder workOrder)
        {
            using (var ts = new PXTransactionScope())
            {
                // Create an instance of the SOInvoiceEntry graph.
                var invoiceEntry = PXGraph.CreateInstance<SOInvoiceEntry>();
                // Initialize the summary of the invoice.
                var doc = new ARInvoice()
                {
                    DocType = ARDocType.Invoice
                };
                doc = invoiceEntry.Document.Insert(doc);
                doc.CustomerID = workOrder.CustomerID;
                invoiceEntry.Document.Update(doc);

                // Create an instance of the RSSVWorkOrderEntry graph.
                var workOrderEntry = PXGraph.CreateInstance<RSSVWorkOrderEntry>();
                workOrderEntry.WorkOrders.Current = workOrder;

                // Add the lines associated with the repair items
                // (from the Repair Items tab).
                foreach (RSSVWorkOrderItem line in
workOrderEntry.RepairItems.Select())
                {
                    var repairTran = invoiceEntry.Transactions.Insert();
                    repairTran.InventoryID = line.InventoryID;
                    repairTran.Qty = 1;
                    repairTran.CuryUnitPrice = line.BasePrice;
                    invoiceEntry.Transactions.Update(repairTran);
                }
                // Add the lines associated with labor (from the Labor tab).
                foreach (RSSVWorkOrderLabor line in workOrderEntry.Labor.Select())
                {
                    var laborTran = invoiceEntry.Transactions.Insert();
                    laborTran.InventoryID = line.InventoryID;
                    laborTran.Qty = line.Quantity;
                    laborTran.CuryUnitPrice = line.DefaultPrice;
                    laborTran.CuryExtPrice = line.ExtPrice;
                    invoiceEntry.Transactions.Update(laborTran);
                }

                // Save the invoice to the database.
                invoiceEntry.Actions.PressSave();

                // Assign the generated invoice number and save the changes.
                workOrder.InvoiceNbr = invoiceEntry.Document.Current.RefNbr;
                workOrderEntry.WorkOrders.Update(workOrder);
                workOrderEntry.Actions.PressSave();

                ts.Complete();
            }
        }
```

In the method above, you first create an instance of the `SOInvoiceEntry` graph. This graph works with SO invoices.

> To instantiate graphs from code, use the `PXGraph.CreateInstance<T>()` method. Do not use the graph constructor `new T()`, because in this case, no extensions or overrides of the graph are initialized.

You then initialize the summary of the invoice by using the `ARInvoice` class. You assign a value to the `CustomerID` field, which is required to create an invoice. After that you update the `ARInvoice` instance in cache.

Then you create an instance of the `RSSVWorkOrderEntry` graph, which you need to get access to the current work order and to save the generated invoice number to the current work order.

After creating all needed graph instances, you select the repair and labor items specified on the Repair Work Orders form by using the instance of the `RSSVWorkOrderEntry` graph. Then you add lines to the invoice by adding instances of the `ARTran` class: the lines associated with repair items, followed by the lines associated with labor items.

To save the created invoice in the database, you call the `PressSave()` method of the `SOInvoiceEntry` graph.

After you have created an invoice, you save the number of the generated invoice to the work order and update its value in the cache. Then you save the changes to the database by invoking the `Actions.PressSave()` method.

At the end of the method, you complete the transaction.

Step 4.3: Defining the Create Invoice Action

In the `RSSVWorkOrderEntry` graph, define the `CreateInvoiceAction` action, which adds the **Create Invoice** command to the More menu (under **Other**), adds the button with the same name on the form toolbar, and invokes the `PXLongOperation.StartOperation` method, as shown in the following code.

> To perform a background operation, an action method needs to have a parameter of the `PXAdapter` type and return `IEnumerable`.

```
public PXAction<RSSVWorkOrder> CreateInvoiceAction;
[PXButton]
[PXUIField(DisplayName = "Create Invoice", Enabled = true)]
protected virtual IEnumerable createInvoiceAction(PXAdapter adapter)
{
    // Populate a local list variable.
    List<RSSVWorkOrder> list = new List<RSSVWorkOrder>();
    foreach (RSSVWorkOrder order in adapter.Get<RSSVWorkOrder>())
    {
        list.Add(order);
    }

    // Trigger the Save action to save changes in the database.
    Actions.PressSave();

    var workOrder = WorkOrders.Current;
    PXLongOperation.StartOperation(this, delegate () {
        CreateInvoice(workOrder);
    });

    // Return the local list variable.
```

```
            return list;
        }
```

In the `createInvoiceAction` method, you compose a list of work orders by using the `adapter.Get` method, and invoke the `Actions.PressSave` action. Because the return of the `adapter.Get` method does not include data that has not been saved on the form, by calling the `PressSave` method, you update the `workOrders` in the composed list.

Then you use the `PXLongOperation.StartOperation()` method to create an invoice. Within the method that you pass to `StartOperation()`, you invoke the `CreateInvoice` method, which creates the invoice for the current work order.

> ⚠ Inside the delegate method of the `StartOperation` method, you cannot use members of the current graph.

Finally, you return the list of work orders.

Related Links

- *PXLongOperation*

Step 4.4: Defining the Visibility and Availability of the Create Invoice Action

According to the workflow for a repair work order, a user should be able to create an invoice only after the work order has been completed. This means that the **Create Invoice** button and command should be visible for only a work order with the *Completed* status. Only one invoice can be created for a single work order, so after a user has clicked **Create Invoice** and the invoice has been created successfully, the button and command should become unavailable.

You configure the availability and visibility of the **Create Invoice** command in the `RowSelected` event handler of the `RSSVWorkOrderEntry` graph. To configure the visibility of the command, you use the `SetVisible` method. To configure the availability of the command, you use the `SetEnabled` method.

To configure the command as described above, do the following:

1. Add the following code to the `_(Events.RowSelected<RSSVWorkOrder> e)` method of the `RSSVWorkOrderEntry` class.

```
        CreateInvoiceAction.SetVisible(
            WorkOrders.Current.Status == WorkOrderStatusConstants.Completed);
        CreateInvoiceAction.SetEnabled(WorkOrders.Current.InvoiceNbr == null &&
            WorkOrders.Current.Status == WorkOrderStatusConstants.Completed);
```

2. To apply changes in the `RSSVWorkOrderEntry` class, rebuild the Visual Studio project.

Step 4.5: Testing the Create Invoice Action

To test the **Create Invoice** button and the corresponding action, do the following:

1. In Acumatica ERP, open the Repair Work Orders (RS301000) form.
2. Open the *000001* repair work order.

   Creation of an invoice is available for completed work orders. To change the status of the order to *Completed*, do the following:

   a. On the form toolbar, click **Assign**.

The status of the work order is changed to *Assigned*.

b. On the form toolbar, click **Complete**.

Notice that the **Create Invoice** button is displayed and available on the form toolbar; this is because the status of the work order is *Completed*.

3. On the form toolbar, click **Create Invoice**.

A notification appears indicating processing, as shown in the following screenshot.

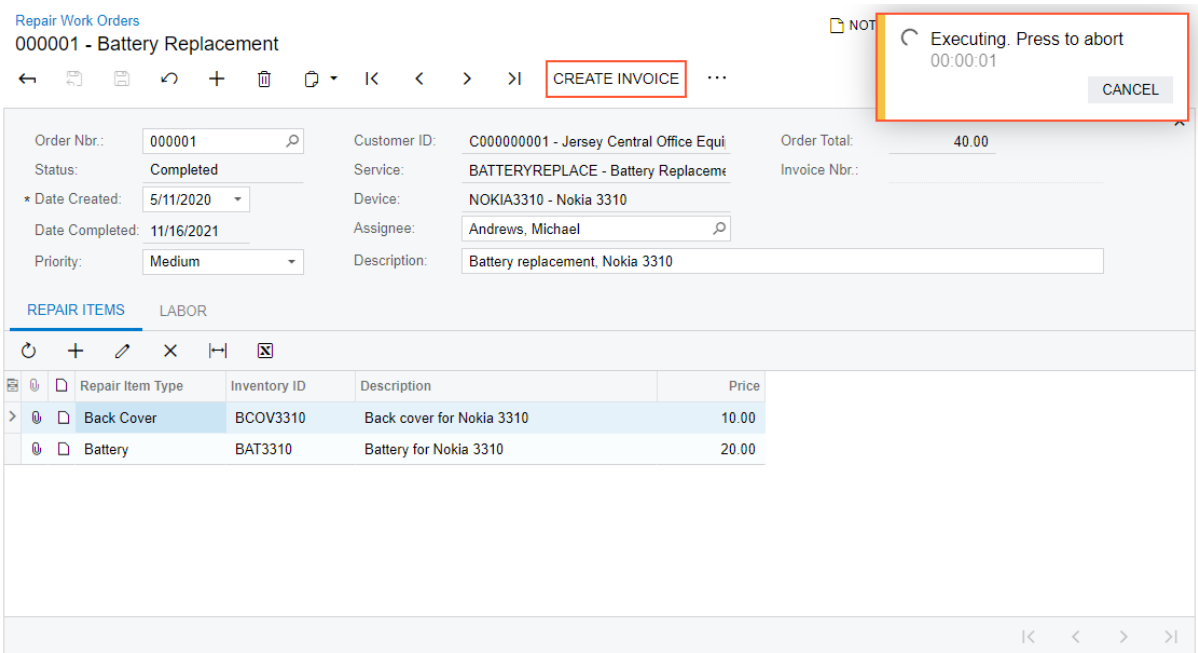


Figure: Creation of an SO invoice

When the process is complete, the number of the created invoice is displayed in the **Invoice Nbr.** box, as shown in the following screenshot.

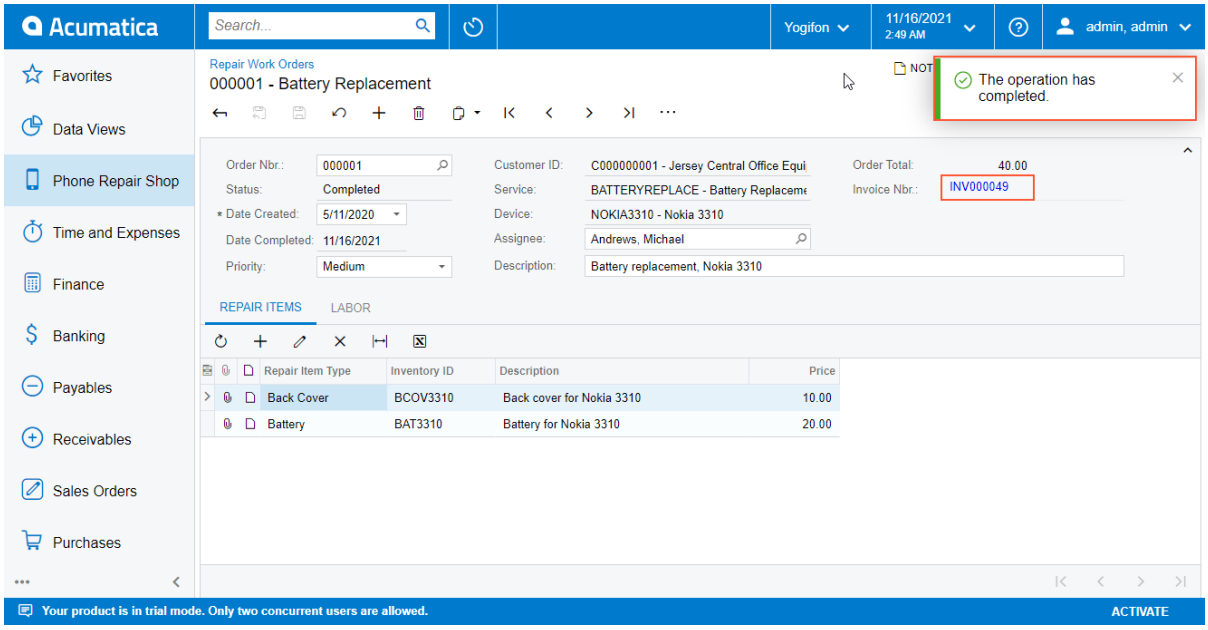


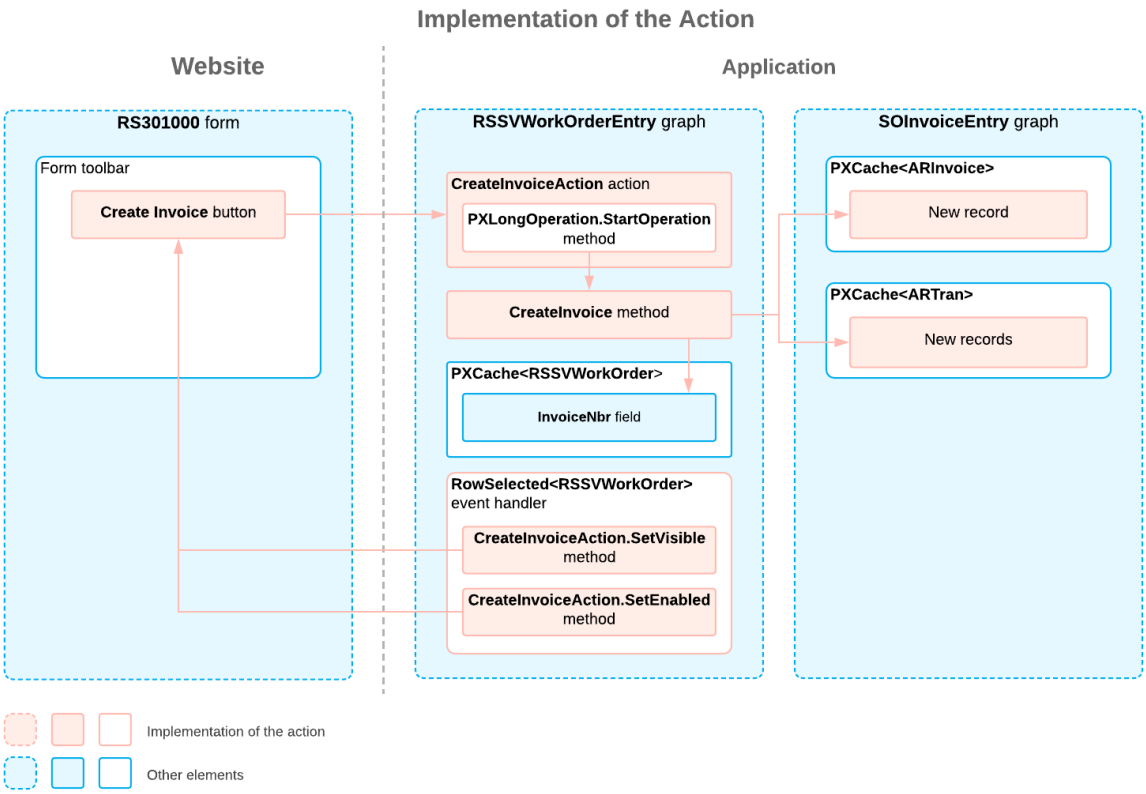
Figure: Update of the Invoice Nbr box

Lesson Summary

In this lesson, you have learned how to initiate an asynchronous operation inside an action method by using the `PXLongOperation` class. Also, you have implemented the creation of an SO invoice based on a repair work order by doing the following in the `RSSVWorkOrderEntry` graph:

- Defining the static `CreateInvoice` method, which creates an instance of the `SOInvoiceEntry` graph
- Defining the **Create Invoice** button on the form toolbar and the command with the same name on the More menu; the underlying action initiates the asynchronous execution of the `CreateInvoice` method by using the `PXLongOperation` class
- Specifying the availability of the `Create Invoice` action in the `RowSelected` event handler so that only a single invoice can be created for a repair work order

The following diagram shows the changes that you have made in this lesson.



Review Questions

1. What attribute do you use to set up a button that is used to initiate a command on the user interface?
   a. `PXButton`
   b. `PXAction`
   c. `PXUIField`
2. How do you configure the visibility of an action at run time?

a. By configuring the ASPX page
b. By calling the `SetVisible` method
c. By setting the `Visible` property value to *True*

Answer Key

1. a
2. b

Additional Information: Actions

In this lesson, you have learned how to implement an action that creates an Acumatica ERP entity (an SO invoice) that corresponds to a custom entity (a repair work order). The implementation of other kinds of actions is outside of the scope of this course but may be useful to some readers. You can find information about how to implement other kinds of actions, along with examples, in the following training courses:

- *T230 Actions*
  - *Lesson 1: Define an Action and the Associated Button on the Form Toolbar*
  - *Lesson 2: Define Actions and the Associated Buttons on the Table Toolbar*
  - *Lesson 3: Implement an Asynchronous Operation*
  - *Lesson 4: Define a Link to an Acumatica ERP Entity*
- *T240 Processing Forms*
  - *Step 1.1.2: Changing the Processing Action*
  - *Step 2.2.3: Implementing the Assignment Operation*
  - *Step 3.1.2: Adding Redirection to a Report at the End of the Processing Delegate*

Lesson 5: Deriving the Value of a Custom Field from Another Entity

In Acumatica ERP, a user can create a payment for an invoice by using the **Pay** action on the *Invoices* (SO303000) form. When the payment is created, it is opened on the *Payments and Applications* (AR302000) form.

The workflow for the *Battery Replacement* service involves one payment, which is made upon completion of the work. Conversely, the workflow for the *Liquid Damage* service involves both a prepayment before the repair work is assigned and a final payment after the work is complete.

For the creation of the prepayment, you now need to have the default prepayment percentage for the payment displayed on the *Payments and Applications* form to facilitate the entry of the prepayment amount, and to make it possible for a user to change that percentage for the current payment. To do that, you need to derive the value of the **Prepayment Percent** element on the Repair Work Order Preferences (RS101000) form and assign it to the corresponding custom field of the *Payments and Applications* form.

You will perform these tasks in this lesson.

Lesson Objectives

In this lesson, you will learn how to do derive the value for a custom field from another form.

Step 5.1: Adding a Custom Field to the Payments and Applications Form—Self-Guided Exercise

To display and modify the prepayment percentage on the *Payments and Applications* (AR302000) form, you need to add the **Prepayment Percent** box to the form. The process of adding a custom field to the form has been shown in *Lesson 1.1: Adding Custom Fields* of the *T210 Customized Forms and Master-Detail Relationship* training course. Complete the following general tasks:

1. By using the Element Inspector, learn the name of the DAC and graph that define the Summary area of the *Payments and Applications* form. In this case, the DAC is `ARPayment` and the graph is `ARPaymentEntry`. You need the DAC name to know which database table and DAC to extend. You will need the graph name in the next step.
2. Add a column named `UsrPrepaymentPercent` to the `ARPayment` table with the same parameters as are specified for the `PrepaymentPercent` field of the `RSSVSetup` table. The data type of the column is `decimal(9, 6)`.
3. Create an extension of the `ARPayment` DAC, and add the `UsrPrepaymentPercent` field to the extension.

   > If you create a DAC extension by using the Customization Project Editor, it creates an extension of the base DAC. So in this case, the system will create an extension of the `ARRegister` DAC because the `ARRegister` DAC is the base DAC for the `ARPayment` DAC.

   The code for the field is shown below.

```
#region PrepaymentPercent
[PXDBDecimal()]
[PXDefault(TypeCode.Decimal, "0.0", PersistingCheck = PXPersistingCheck.Nothing)]
[PXUIField(DisplayName = "Prepayment Percent")]
public Decimal? UsrPrepaymentPercent { get; set; }
```

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

4. Add a box for the `UsrPrepaymentPercent` field to the Summary area of the *Payments and Applications* form. In the Screen Editor, the location of the element appears as shown in the following screenshot.

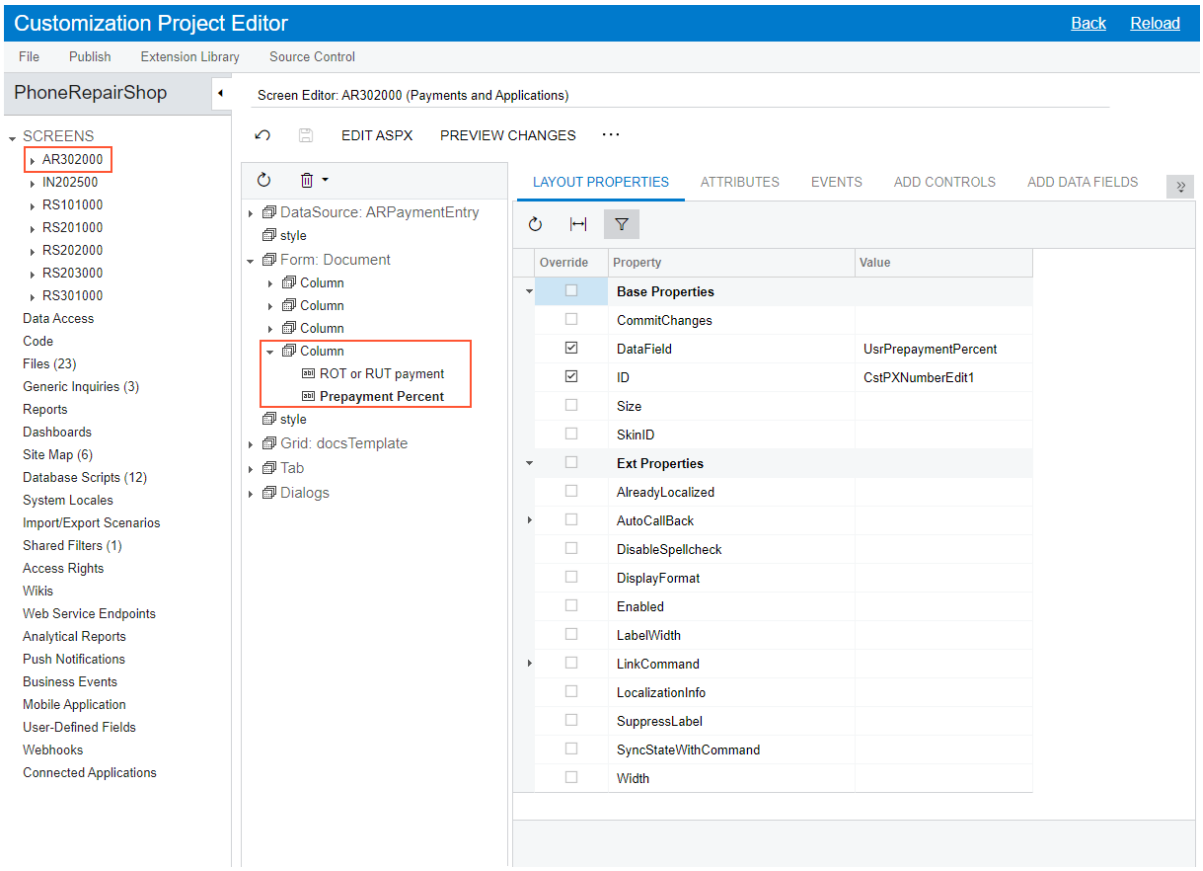


Figure: Prepayment Percent element

5. Correct the width for the **Prepayment Percent** label. To do this, in the `Column` element that is the parent to the `Prepayment Percent` element, for the `LabelsWidth` property, specify the *M* value.
6. Publish the customization project.

Step 5.2: Deriving the Default Value of the PrepaymentPercent Field

You can implement the deriving of a field value from the `RSSVSetup` DAC and the copying of it to the `ARPayment` DAC by doing one of the following:

- Using the `FieldDefaulting` event
- Using the `PXDefault` attribute

To populate the `UsrPrepaymentPercent` field of the `ARPayment` extension when a payment is created, you can use the `FieldDefaulting` event. Do the following:

1. Create an extension of the `ARPaymentEntry` graph, as shown in the following code.

   You learned the name of the graph to extend in Instruction 1 of the previous step.

```
namespace PX.Objects.AR
```

```
{
  public class ARPaymentEntry_Extension : PXGraphExtension<ARPaymentEntry>
  {
  }
}
```

2. Add the following `using` directives.

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

3. Use Acuminator to suppress the *PX1016* error in a comment. In this course, for simplicity, the extension is always active.
4. Define the `FieldDefaulting` event handler for the `UsrPrepaymentPercent` field of the `ARPayment` extension, as shown in the following code.

```
public virtual void _(Events.FieldDefaulting<ARPayment,
                      ARPaymentExt.usrPrepaymentPercent> e)
{
  ARPayment payment = (ARPayment)e.Row;
  RSSVSetup setupRecord = SelectFrom<RSSVSetup>.View.Select(Base);
  if (setupRecord != null)
  {
      e.NewValue = setupRecord.PrepaymentPercent;
  }
}
```

In the code above, you have selected the record with the repair work order preferences and assigned the `PrepaymentPercent` field value to the `UsrPrepaymentPercent` field of the `ARPayment` DAC. You have checked for the `null` value of the `setupRecord` so that the `NullReferenceException` exception is not thrown if the data on the form has not been filled in yet.

> In the event handler, specify `ARRegisterExt.usrPrepaymentPercent` instead of `ARPaymentExt.usrPrepaymentPercent`, if the `usrPrepaymentPercent` field belongs to the `ARRegisterExt` DAC extension.

Another way to derive the default value is to use the `PXDefault` attribute, which performs the same logic. If you use this approach, the `PXDefault` attribute for the `UsrPrepaymentPercent` field should look as follows.

```
[PXDefault(typeof(Select<RSSVSetup>), SourceField = typeof(RSSVSetup.prepaymentPercent),
        PersistingCheck = PXPersistingCheck.Nothing)]
```

This approach provides the following advantages:

- You do not need to create a graph extension.
- Your logic is written in declarative style.

> You need to specify the `SourceField` parameter if the field names are not identical.

Step 5.3: Testing the Deriving of the Field Value

To test that the value of the **Prepayment Percent** box on the Payments and Applications (AR302000) form is correctly specified for payments, do the following:

1. On the Invoices (SO303000) form, open the *INV000049* invoice, which you created in *Step 4.5: Testing the Create Invoice Action*
2. Release the invoice by doing the following:
   a. Type `40` in the **Amount** box of the Summary area.
   b. On the form toolbar, click **Remove Hold**.
   c. Release the invoice by clicking **Release** on the form toolbar.
3. On the More menu (under **Processing**), click **Pay**.

   The Payments and Applications (AR302000) form opens. In the Summary area, notice that the **Prepayment Percent** box has the *10.00* value (see the following screenshot), which has been copied from the Repair Work Order Preferences (RS101000) form.

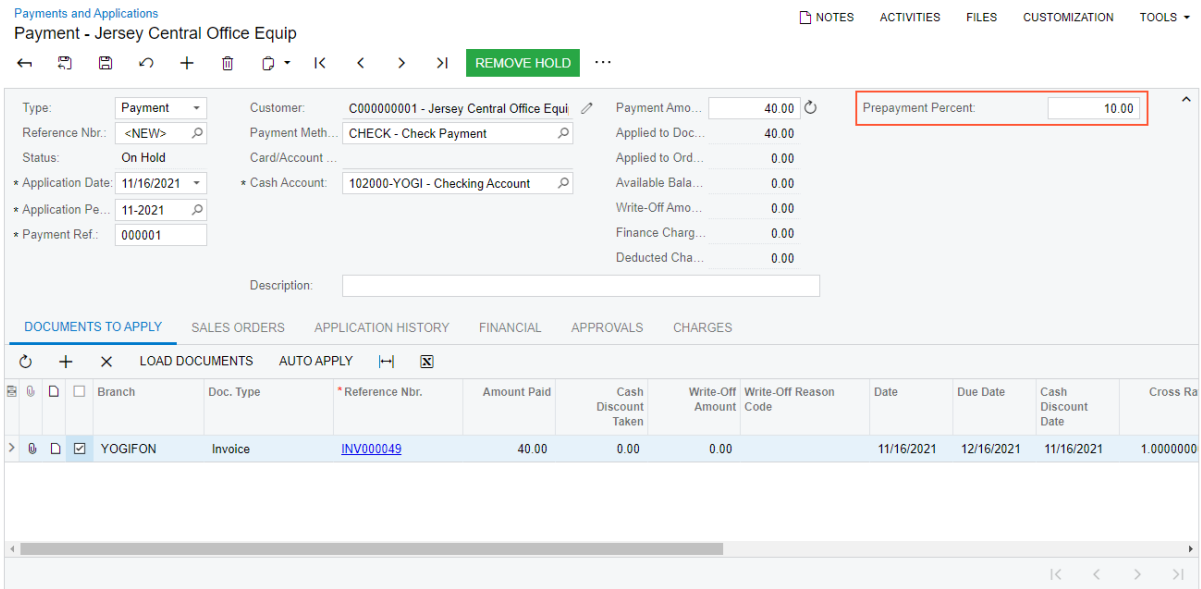


   Figure: The Prepayment Percent box

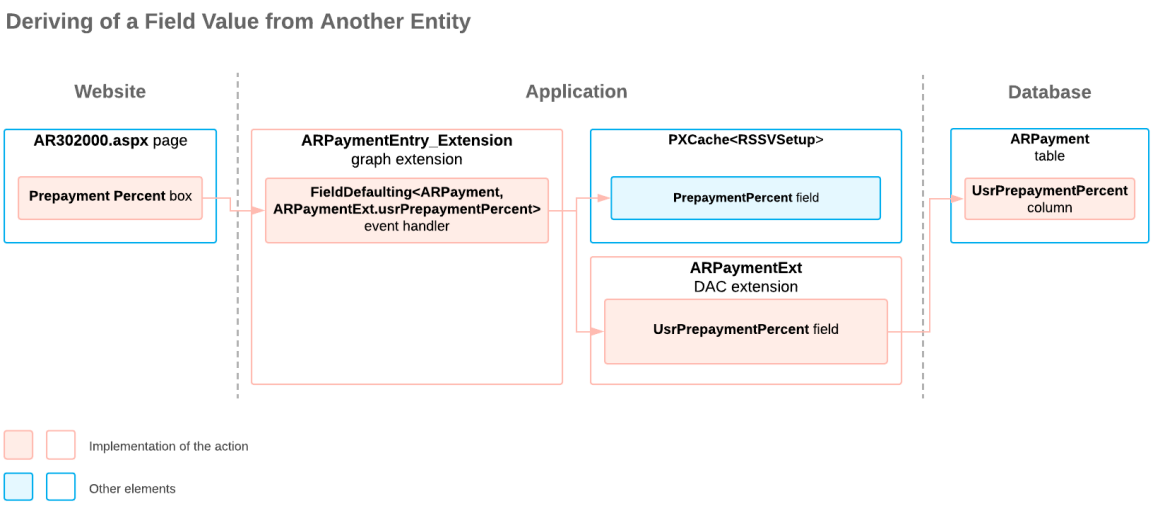
4. Save the payment.

Lesson Summary

In this lesson, you created a custom field on the Payments and Applications (AR302000) form and learned how to assign its default value, which is derived from another entity. To assign a default value for a custom entity, you have done the following:

1. Defined the extension of a graph in which the field is initialized
2. In the graph extension, defined the `FieldDefaulting` event handler for the custom field

The following diagram shows the changes that you have performed in this lesson.



Review Question

1. Which ways can you use to derive the value of a custom field from a custom entity to an Acumatica ERP entity? Select all applicable ways.
   a. Use the `PXDefault` attribute
   b. Use the `FieldDefaulting` event handler
   c. Use the `RowSelected` event handler

Answer Key

1. a, b

Lesson 6: Debugging Customization Code

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

You can also debug the source code of Acumatica ERP to find out the method that you need to override to modify the business logic of Acumatica ERP. For example, you may need to customize the release process of a payment. In this lesson, you will explore the code of the payment release process through debugging.

Lesson Objectives

As you complete this lesson, you will learn how to debug the source code of Acumatica ERP.

Step 6.1: Debugging the Acumatica ERP Source Code

Acumatica ERP has open source code, which you can easily view with the Source Code browser.

However, in order to find a method that you need to override, it is helpful to debug Acumatica ERP source code with breakpoints and see which breakpoint is hit in which scenario. To prepare the `PhoneRepairShop_Code` Visual Studio project for the debugging of Acumatica ERP code, you should do the following:

1. Make sure the Acumatica program database (PDB) files are located in the `Bin` folder of the Acumatica ERP instance folder that you use for the training course (for example, in `PhoneRepairShop\Bin`).

   The PDB files are copied to the `Files\Bin` folder of the Acumatica ERP installation folder (such as `C:\Program Files\Acumatica ERP\Files\Bin`) during the installation process if the **Install Debugger Tools** option is selected in the Acumatica ERP Installation wizard. When you create a new instance or update an existing one, the PDB files are copied to the `Bin` folder of the instance. If you haven't selected the **Install Debugger Tools** option during installation, you should remove Acumatica ERP and install it again with the **Install Debugger Tools** option selected. For details, see *To Install the Acumatica ERP Tools*.

   > A PDB file holds debugging and project state information that allows incremental linking of a debug configuration of your program. In general, a PDB file contains the link between compiler instructions and some lines in source code.

2. Configure the `web.config` file of the instance by doing the following:
   a. In the file system, open in the text editor the `web.config` file, which is located in the root folder of the *PhoneRepairShop* instance.
   b. In the `<system.web>` tag of the file, locate the `<compilation>` element.
   c. Set the debug attribute of the element to *True*, as shown in the following code.

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

   d. Save your changes.
3. Configure the `PhoneRepairShop_Code` project for debugging by doing the following:
   a. In Visual Studio, open the `PhoneRepairShop_Code` solution, which includes both the `PhoneRepairShop_Code` project and the *PhoneRepairShop* website.
   b. In the main menu, select **Tools > Options**.

c. In the **Debugging > General** section, clear the **Enable Just My Code** check box, as shown in the following screenshot.

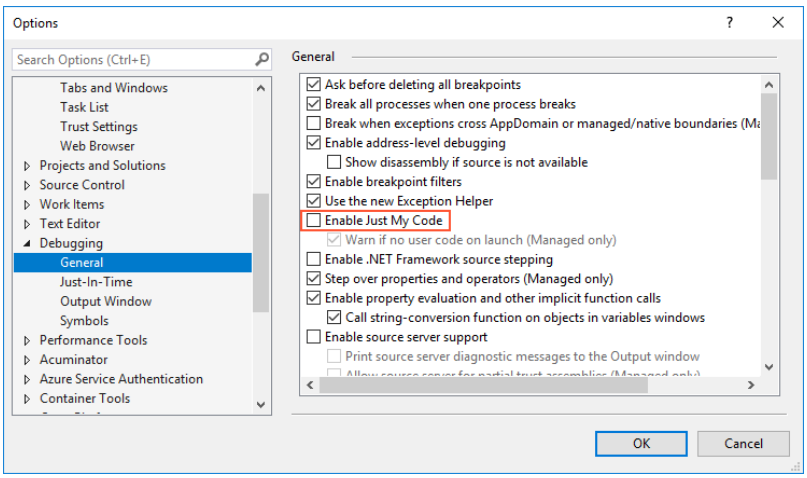


Figure: Clearing the Enable Just My Code check box

d. In the **Debugging > Symbols** section, in the **Symbols file (.pdb) locations** list, add the path to the location of the PDB files that you discovered in Instruction 1 of this step. See the following example.

```
C:\Training\PhoneRepairShop\Bin
```

e. Click **OK**.

4. Open the Acumatica ERP source code files. For the *PhoneRepairShop* instance, all files are located in the `PhoneRepairShop/App_Data/CodeRepository` folder.

5. To view the source code of the `Release` action of the Payments and Applications (AR302000) form, open the `PX.Objects.AR.ARPaymentEntry` graph: In the Solution Explorer, select **PhoneRepairShop > App_Data > CodeRepository > PX.Objects > AR > ARPaymentEntry.cs**, and go to the definition of the `IEnumerable Release(PXAdapter adapter)` method. The code should look as shown in the following screenshot.

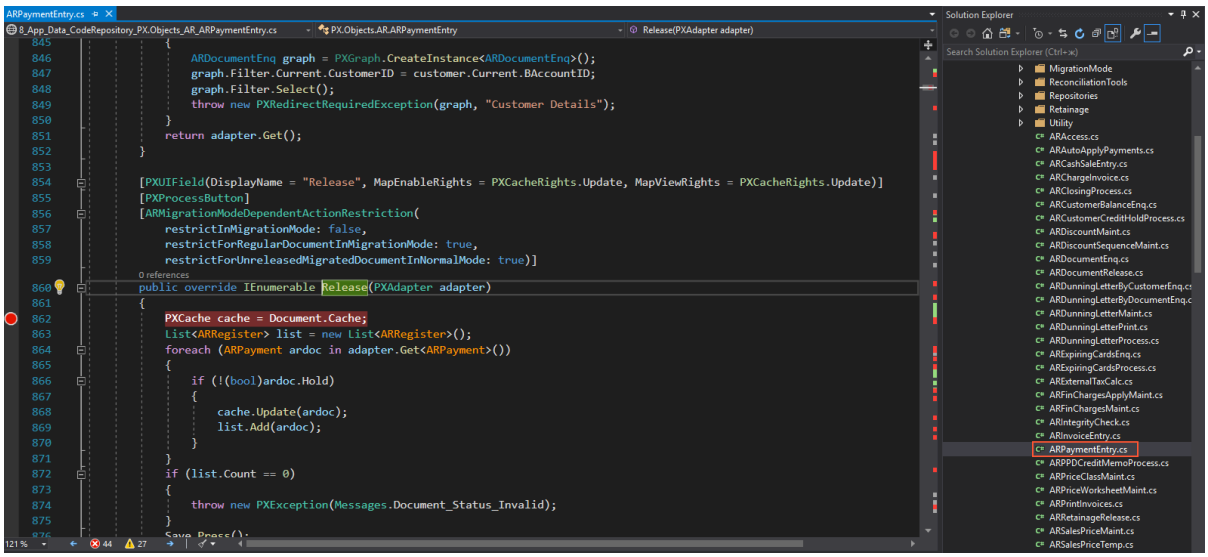


Figure: Viewing the source code of the Release action

6. Add a breakpoint inside the `Release` method, as shown in the screenshot above.

7. Attach the Visual Studio debugger to the `w3wp.exe` process.

8. Start debugging by doing the following:
   a. In Acumatica ERP, open the Payments and Applications form.
   b. Create a payment.
   c. On the form toolbar, click the **Release** button.

      Wait until the breakpoint is hit.
   d. In Visual Studio, view the debug information for the `Release` method.

> If an invoice was created for a repair work order with only non-stock items, by default, an AR invoice is created instead of the SO invoice. There is no difference for the release of a payment for AR and SO invoices, so you don't need to customize the closing of AR invoices as well.

Related Links

- *To View and Debug Acumatica ERP Source Code*

Lesson Summary

In this lesson, you have learned how to debug the code of Acumatica ERP by using program database (PDB) files.

Review Question

1. Where are the Acumatica ERP PDB files located?
   a. On the Partner Portal
   b. In the `Files/Bin` folder of the Acumatica ERP installation folder
   c. In the `Bin` folder of the customization project

Answer Key

1. b

Additional Information: the Debugging of Customization Code

In this lesson, you have learned how to debug the code of Acumatica ERP.

The debugging of customization code that is created in the Customization Project Editor and is located in the `App_RuntimeCode` folder is outside of the scope of this course. You can find information about how to debug this code along with an example in *Lesson 1.7: Debug the Customization Code* of the *T200 Maintenance Forms* training course.