Developer Course

Customization

T230 Actions
2022 R2

Revision: 10/7/2022

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 R2

Last Updated: 10/07/2022

Introduction

The *T230 Actions* training course shows you how to define and customize various types of actions by using Acumatica Framework and the customization tools of Acumatica ERP.

Actions in the application code are associated with buttons on form toolbars and table toolbars and with commands on the More menu in Acumatica ERP.

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

The course is based on a set of examples that demonstrate the general approach to customizing Acumatica ERP. The course is designed to give you ideas about how to develop your own embedded applications through the customization tools. It demonstrates the implementation of actions and the customization of existing actions of Acumatica ERP, as well as the implementation of the commands and buttons on the UI that are associated with these actions. As you go through the course, you will continue the development of the customization for the cell phone repair shop, which was performed in the *T200 Maintenance Forms*, *T210 Customized Forms and Master-Detail Relationship*, and *T220 Data Entry and Setup Forms* training courses (which we recommend that you take before completing the current course).

After you complete all the lessons of the course, you will be familiar with the programming techniques used to define and customize actions in 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 the *Getting Started* section.
2. Complete the lessons in the training guide.
3. In Partner University, take *T230 ActionsCertification Test*.

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 the Documentation Resources Are

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.

Which License You Should Use

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

Course Prerequisites

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

Required Knowledge and Background

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

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

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

If you have completed any of the training courses of the *T* series and are using the same environment for the current course, you can skip this step.

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

1. Make sure the environment that you are going to use conforms to the *System Requirements for Acumatica ERP 2022 R2*.
2. Make sure that the Web Server (IIS) features that are listed in *Configuring Web Server (IIS) Features* are turned on.
3. Install the Acuminator extension for Visual Studio.
4. 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: Preparing the Needed Acumatica ERP Instance for the Training Course

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

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

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

3. Sign in to the new tenant by using the following credentials:
   - **Username**: `admin`
   - **Password**: `setup`

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

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

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

Step 3: Including of the Workflow Code in the Extension Library

The actions that you will implement in this course depend on the state of the repair work order, as described in *Company Story and Customization Description*. Changes of states of repair work orders can be implemented within a workflow. In this course, the code for the workflow is provided. (The implementation of this code is outside of the scope of this course.) You need to include the code that implements the workflow in the extension library so that you can use this code in the course.

To include the workflow code in your `PhoneRepairShop_Code` project, do the following:

1. In the `PhoneRepairShop_Code` project, create the `Workflow` folder.
2. Add the `RSSVWorkOrderWorkflow.cs` file (located in the `Customization/T230/SourceFiles/Workflow` folder, which you downloaded from the Acumatica GitHub) in the `Workflow` folder.

   The code contains the workflow definition, which is described in *Company Story and Customization Description*.
3. Copy the code from the `RSSVWorkOrderEntry.cs` file (located in the `Customization/T230/SourceFiles/Workflow` folder), and replace the contents of the `RSSVWorkOrderEntry.cs` file (located in the `PhoneRepairShop_Code` project).

   The code contains workflow actions and an event handler used in the workflow.

For more information on creating workflows, see the following sections of the Customization Guide:

- *Creating Workflows*
- *Defining a Workflow*

Getting Started

In this part of the course, you will review the company story and requirements to the customization that will be performed in this training course. You will also get an overview of the declaration of actions (which correspond to buttons on the form and table toolbars of Acumatica ERP forms and the corresponding More menu).

Company Story and Customization Description

In this course, you will continue the development to support the cell phone repair shop of the Smart Fix company; you began this development while completing the *T200 Maintenance Forms*, *T210 Customized Forms and Master-Detail Relationship*, and *T220 Data Entry and Setup Forms* training courses.

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

In the *T210 Customized Forms and Master-Detail Relationship* course, you have created another maintenance form, Services and Prices (RS203000), and customized the *Stock Items* (IN202500) form of Acumatica ERP. The Services and Prices form provides users with the ability to define and maintain the price for each provided repair service. The Stock Items form has been customized to mark particular stock items as repair items—that is, items that are used for the repair services.

In the *T220 Data Entry and Setup Forms* course, you have created the Repair Work Orders (RS301000) data entry form, which is used to create and manage work orders for repairs. You have also created the Repair Work Order Preferences (RS101000) setup form, which an administrative user uses to specify the company's preferences for the repair work orders.

The customization project that contains the results of these courses has been loaded and published in *Step 2: Preparing the Needed Acumatica ERP Instance for the Training Course*.

In the *T230 Actions* course, on the Repair Work Orders form, you will implement commands on the More menu and buttons on the form toolbar, as well as buttons on the table toolbars of the tabs of this form. You will also implement the underlying actions. By using these commands and buttons, users should be able to assign a work order to themselves, update the prices of repair and labor items, and create and open an invoice for the selected repair work order.

- Actions in the application code are associated with buttons on form toolbars and table toolbars and commands on the More menu in Acumatica ERP.
- Actions (and the associated buttons and commands) that change the status of a repair work order are implemented with the Workflow API and are not covered in this course.

Buttons and Commands on the Repair Work Orders Form

The following screenshots show how the Repair Work Orders (RS301000) form will look with the buttons and commands you will add in this course.

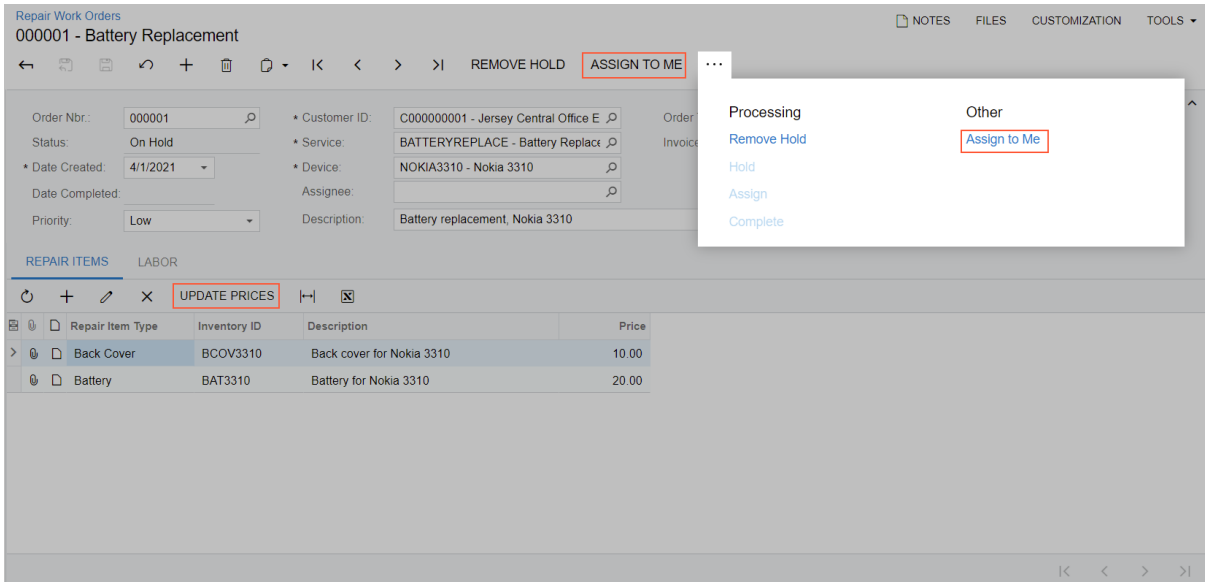


Figure: The Assign to Me command and button and the Update Prices button (on the Repair Items tab)

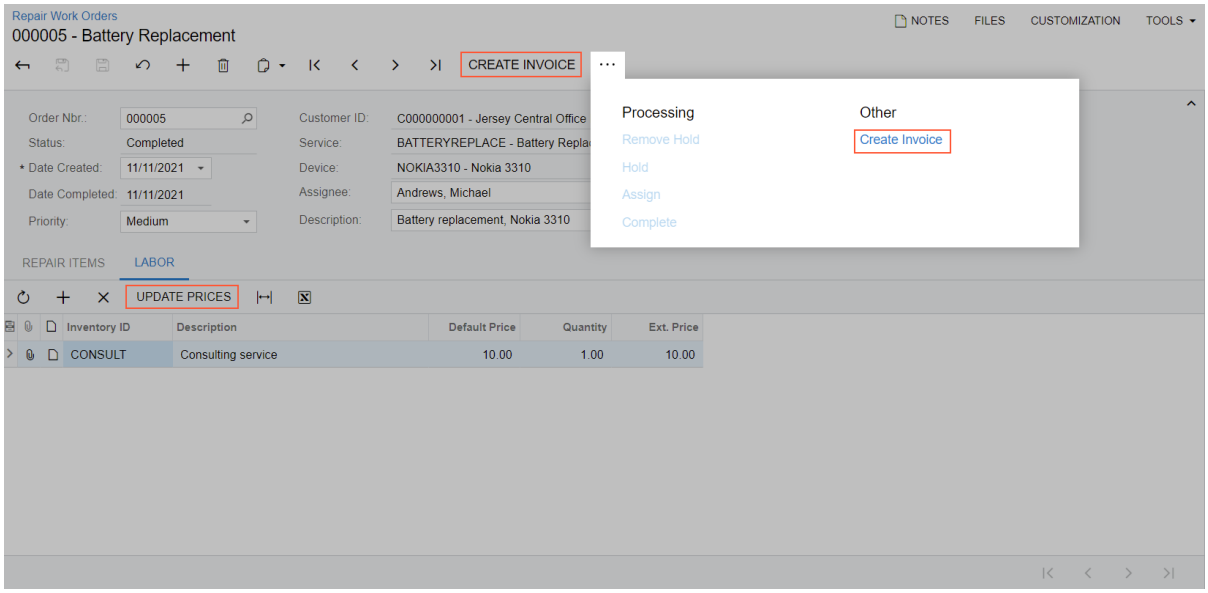


Figure: The Update Prices button (on the Labor tab) and the Create Invoice button and command

You will add the following buttons and commands to the form and implement the underlying actions:

- The **Assign to Me** button on the form toolbar and command on the More menu

  By clicking this button or command, a user can assign a repair work order to themselves—that is, to cause the system to insert their username into the **Assignee** box. The button and command should be available until the repair work order status is changed to *Assigned*—that is, when the repair work order has the *On Hold* or *Ready for Assignment* status.
- The **Update Prices** button on the table toolbar of the **Repair Items** and **Labor** tabs

  If the price of a repair item or labor item has been changed on the Services and Prices (RS203000) form, a user should be able to update this item's price on the Repair Work Orders form: on the **Repair Items** tab for a repair item or the **Labor** tab for a labor item. The **Update Prices** button should be available until an invoice has been created for a repair work order.
- The **Create Invoice** button on the form toolbar and command on the More menu

When a repair work order is completed, a user should be able to create an invoice for it. That is, the button and command should be available when a repair work order has the *Completed* status.

For the form, you will also implement a link in the **Invoice Nbr.** box, which a user can click to open a new tab with the corresponding invoice selected on the *Invoices* (SO303000) form.

Statuses of a Repair Work Order

A repair work order can have different statuses, and some buttons and commands that you will implement in this course may be visible and available depending on the status of the repair work order. In this course, the workflow engine is used to change the state of an entity. You have included the code of the workflow in the extension library in *Step 3: Including of the Workflow Code in the Extension Library*.

The workflow to be used for repair work orders created on the Repair Work Orders (RS301000) form differs slightly depending on the particular service being performed, which can be *Battery Replacement*, *Liquid Damage*, or *Screen Repair*. The following diagram shows this workflow.

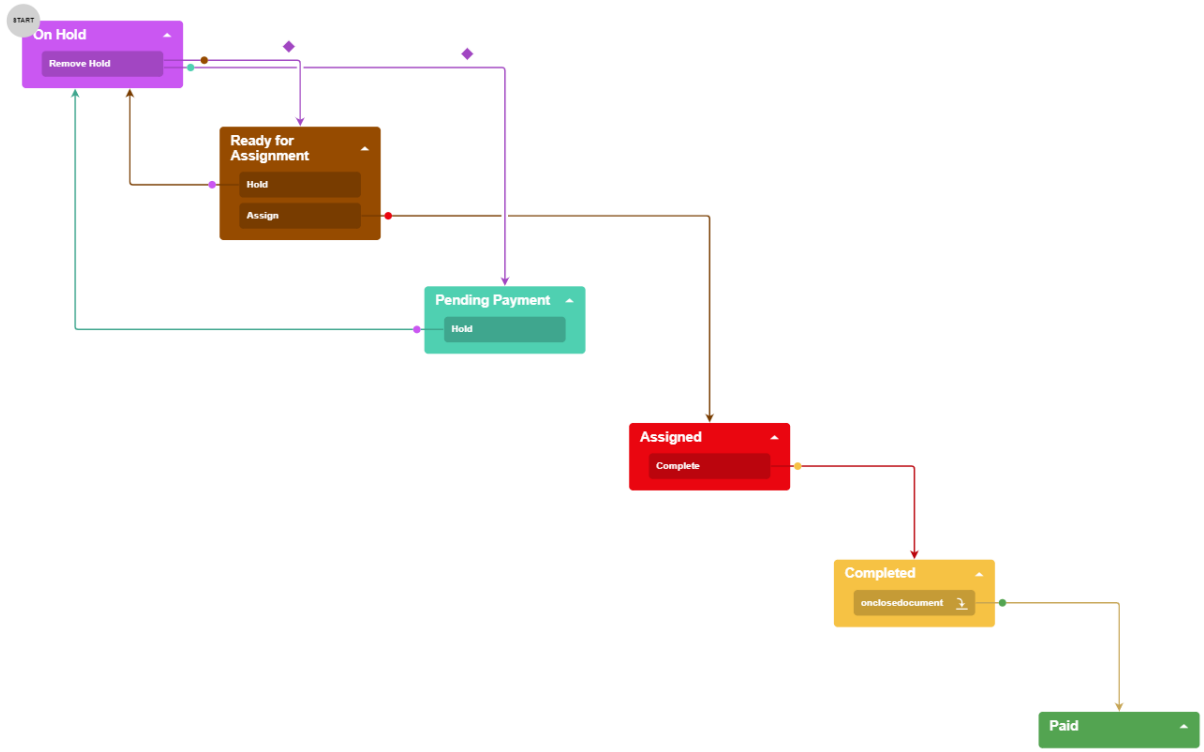


Figure: The workflow of a repair work order

The workflow code provides the *On Hold*, *Ready for Assignment*, *Pending Payment*, *Assigned*, *Completed*, and *Paid* statuses and the following commands (for which buttons may also be shown on the form toolbar, depending on the status):

- **Remove Hold**, which switches the status from *On Hold* to *Ready for Assignment* or *Pending Payment*, depending on whether the repair work order requires prepayment
- **Hold**, which switches the status from *Ready for Assignment* or *Pending Payment* to *On Hold*
- **Assign**, which switches the status from *Ready for Assignment* to *Assigned*
- **Complete** which switches the status from *Assigned* to *Completed*

The workflow also contains an event handler, which switches the status from *Completed* to *Paid* when the invoice is fully paid.

Declaration of Actions in Acumatica Framework

Actions are methods that can be invoked from outside of the graph, from the UI, or through the web service API. Actions are associated with buttons or commands (or both) on the user interface.

Action Declaration in a Graph

The declaration of an action in a graph consists of the following:

- A field of the `PXAction<>` type, which is declared as follows.

```
public PXAction<Shipment> CancelShipment;
```

  In the `PXAction<>` type parameter, you should specify the main DAC of the primary data view. Otherwise, the button corresponding to the action cannot be displayed on the form toolbar (and the corresponding command cannot be displayed on the More menu).
- A method that implements the action; this method has the `PXButton` and `PXUIField` attributes. This method has the following forms of declaration:
  - Without parameters and returning `void`: This standard form of declaration is shown in the following code example.

```
[PXButton]
[PXUIField(DisplayName = "Cancel Shipment")]
protected virtual void cancelShipment()
{
    ...
}
```

    This type of declaration is used for an action that is executed synchronously and is not called from a processing form.
  - With a parameter of the `PXAdapter` type and returning `IEnumerable`: You can see an example of this form of declaration in the following code.

```
[PXButton]
[PXUIField(DisplayName = "Release")]
protected virtual IEnumerable release(PXAdapter adapter)
{
    ...
    return adapter.Get();
}
```

    This type of declaration should be used when the action initiates a background operation or is called from a processing form.

The field and the method should have the same name, differing only in the capitalization of the first letter.

A graph includes the `Actions` collection of all `PXAction<>` objects defined in the graph.

Callback on the Action

When a user invokes an action through a button or command on the UI, the page sends a request to the server side of the application (that is, it executes the callback). By default, for a button or command, the callback is always executed—that is, the `CommitChanges` property of the `PXButton` attribute is `true`. If you do not need the form

to send the recent changes made on the form, set the `CommitChanges` property of the `PXButton` attribute to `false` as follows.

```
[PXButton(CommitChanges = false)]
```

The `CommitChanges` property must be always set to `true` for the actions that cause changes to be saved to the database.

Types of Actions

You typically use actions for the following purposes:

- To redirect a user to a specific form or report.
- To modify or validate data records and save changes to the database.
- To start a background operation, which is executed on a separate thread.

Changing of Appearance of Buttons and Commands on the UI at Runtime

You can adjust the appearance of a button or command (or both, if applicable) on the UI that is associated with an action at runtime (for example, to make the button or command unavailable or hidden). To do this, you should use the methods of the `PXAction<>` class, as the following code example shows.

```
// Disabling the CancelShipment action
CancelShipment.SetEnabled(false);
```

You do not use the static methods of the `PXUIField` attribute, because these methods work only with the attribute copies stored in `PXCache` objects.

Related Links

- *Action*

Lesson 1: Defining an Action and the Associated Button on the Form Toolbar

On the Repair Work Orders (RS300000) form, users should be able to assign the selected repair work order to themselves.

To implement this scenario, you will create the `AssignToMe` action, the associated **Assign to Me** button on the form toolbar, and the command with the same name on the More menu. The button and command will be visible only if the repair work order has the *On Hold* or *Ready for Assignment* status and available if the **Assignee** box value does not already contain the current user's information.

When the user clicks the button or command, the contact associated with the current user, which is specified on the *Users* (SM201010) form, will be inserted into the **Assignee** box. (If the user has selected an assignee before clicking the button or command, it will be overridden.) The clicking of the button or command does not affect the status of the repair work order.

Lesson Objectives

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

- Create an action, the associated button on the form toolbar, and the equivalent command on the More menu
- Configure the availability and visibility of the button and command depending on field values on the form

Step 1.1: Implementing the AssignToMe Action

In this step, you will implement the `AssignToMe` action. The associated button will be placed on the form toolbar; the equivalent command on the More menu (under the default **Other** category).

Commands on the More menu are listed under categories. You can change the category of the command by setting the value of the `Category` property of the `PXButton` attribute.

To implement the `AssignToMe` action, do the following:

1. In Visual Studio, open the `RSSVWorkOrderEntry.cs` file.
2. In the `Actions` region of the `RSSVWorkOrderEntry` class, insert the following code.

```
public PXAction<RSSVWorkOrder> AssignToMe;
[PXButton]
[PXUIField(DisplayName = "Assign to Me", Enabled = true)]
protected virtual void assignToMe()
{
    // Get the current order from the cache.
    RSSVWorkOrder row = WorkOrders.Current;

    // Assign the contact ID associated with the current user
    row.Assignee = PXAccess.GetContactID();

    // Update the data record in the cache.
    WorkOrders.Update(row);

    // Trigger the Save action to save changes in the database.
```

```
            Actions.PressSave();
        }
```

> There is no need to set the `CommitChanges` property of the `PXButton` attribute to `True` because `CommitChanges` is `True` by default for `PXButton`.

3. Save your changes.

Step 1.2: Specifying the Availability and Visibility of the Assign to Me Button and Command (with RowSelected)

The **Assign to Me** button on the form toolbar (and the equivalent command on the More menu) should be visible for only orders with the *Ready for Assignment* status and available if the **Assignee** box does not already contain the employee name of the user who is currently signed in. To satisfy these conditions, you should specify the `Enabled` and `Visible` properties of the `AssignToMe` action as follows:

1. In Visual Studio, open the `RSSVWorkOrderEntry.cs` file.
2. Add the handler for the `RowSelected` event for the `RSSVWorkOrder` DAC, as shown in the following code.

```
        // Manage visibility and availability of the actions.
        protected virtual void _(Events.RowSelected<RSSVWorkOrder> e)
        {
            RSSVWorkOrder row = e.Row;
            if (row == null) return;}
```

3. In the `_(Events.RowSelected<RSSVWorkOrder> e)` event handler, add the following code to the end of the method.

```
            AssignToMe.SetEnabled((row.Status ==
WorkOrderStatusConstants.ReadyForAssignment ||
                row.Status == WorkOrderStatusConstants.OnHold) &&
                WorkOrders.Cache.GetStatus(row) != PXEntryStatus.Inserted);
```

   In the code above, you disable the **Assign to Me** button and command until the repair work order is saved in the database by checking the `WorkOrders` object status in the `PXCache`.

4. In the `_(Events.RowSelected<RSSVWorkOrder> e)` event handler, add the following code to the end of the method.

```
            AssignToMe.SetVisible(row.Assignee != PXAccess.GetContactID());
```

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

Related Links

- *To Hide or Show an Action*

Step 1.3: Testing the Assign to Me Button and Associated Action

To test the `AssignToMe` action and the **Assign to Me** button, do the following:

1. In Acumatica ERP, open the Repair Work Orders (RS301000) form.
2. Create a work order by specifying the following values:
   - **Customer ID**: *C000000001*
   - **Service**: *Battery Replacement*
   - **Device**: *Nokia 3310*
   - **Description**: `Battery replacement, Nokia 3310`
3. Save your changes.

   The *000003* repair work order is created.

   Notice that the **Assign to Me** button is displayed on the form toolbar, as shown in the following screenshot.

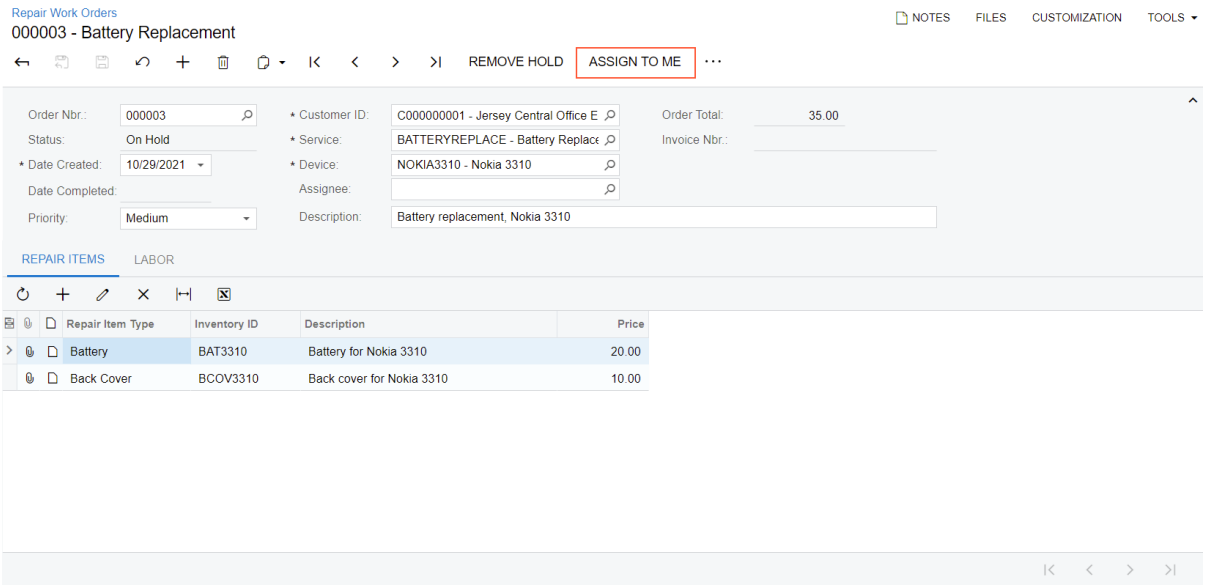


Figure: The Repair Work Orders form

4. On the form toolbar, click the **Assign to Me** button.

   In the **Assignee** box, notice that the employee name associated with the current user is now specified; the employee associated with the user is specified in the **Linked Entity** box of the *Users* (SM201010) form. Also, notice that the **Assign to Me** button is no longer displayed on the form toolbar.

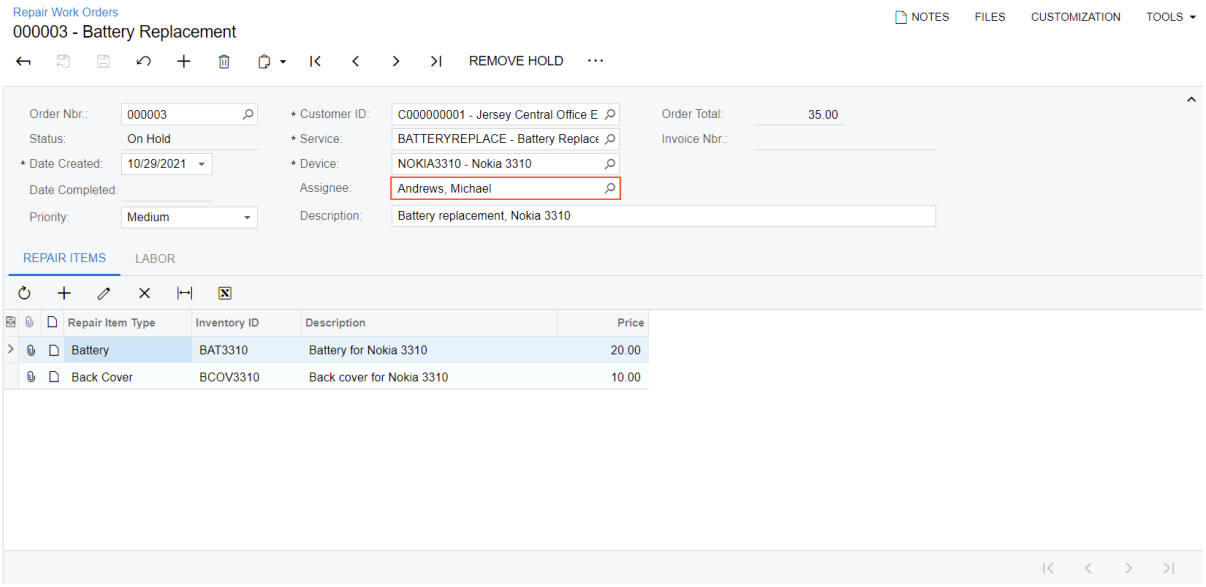


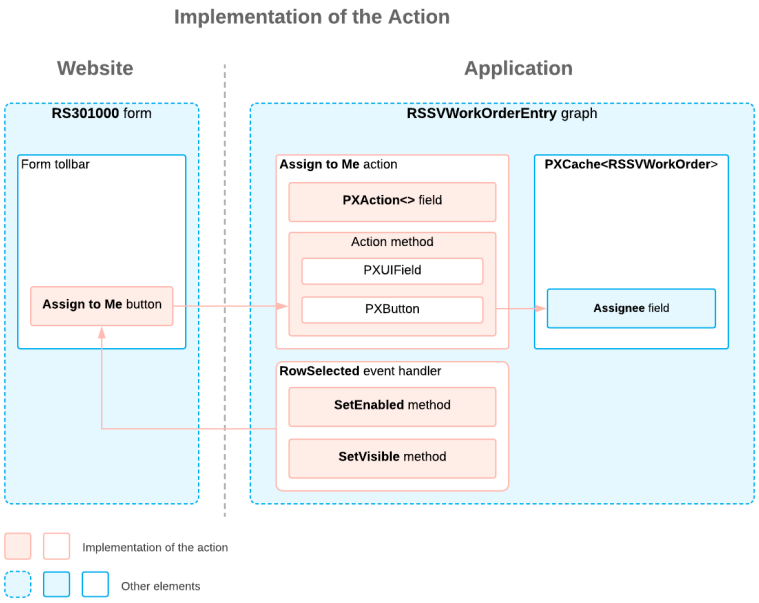
Figure: The modified Repair Work Orders form

Lesson Summary

In this lesson, you have learned how to define an action in Acumatica Framework and display the corresponding button on the form toolbar and command on the More menu. You have implemented the action, configured the availability of the button and command, and tested the action and button.

To define an action, you have added the field of the `PXAction<>` type and an action method with the `PXButton` and `PXUIField` attributes to the `RSSVWorkOrderEntry` graph. To configure the availability and visibility of the corresponding command, you have used the `SetEnabled` and `SetVisible` methods in the `RowSelected` event handler of the `RSSVWorkOrderEntry` graph.

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



Lesson 2: Defining Actions and the Associated Buttons on the Table Toolbar

On the Repair Work Orders (RS300000) form, a user should be able to update the base prices of repair items on the **Repair Items** tab and the default prices of labor items on the **Labor** tab.

To implement this scenario, you will create two actions and the associated **Update Prices** buttons on the table toolbar of the respective tab. These buttons should be available only if no invoice has been created for the repair work order selected on the form.

Lesson Objectives

In this lesson, you will learn how to specify the location of a button associated with an action on a form.

Step 2.1: Implementing the UpdateItemPrices Action

In this step, you will implement the `UpdateItemPrices` action in the `RSSVWorkOrderEntry` graph.

You will specify the availability of the button related to the action in the `RowSelected` event handler of the same graph by using the `SetEnabled()` method of `PXAction<>`. You will hide the button from the form toolbar and the corresponding command from the More menu by setting the `DisplayOnMainToolbar` property of the `PXButton` attribute to `false`.

In the `RSSVWorkOrderEntry` graph, do the following:

1. In the `Actions` region, add the `UpdateItemPrices` action, as the following code shows.

```
public PXAction<RSSVWorkOrder> UpdateItemPrices;
[PXButton(DisplayOnMainToolbar = false)]
[PXUIField(DisplayName = "Update Prices", Enabled = true)]
protected virtual void updateItemPrices()
{
    var order = WorkOrders.Current;
    if (order.ServiceID != null && order.DeviceID != null)
    {
        var repairItems = RepairItems.Select();
        foreach (RSSVWorkOrderItem item in repairItems)
        {
            RSSVRepairItem origItem = SelectFrom<RSSVRepairItem>.
                    Where<RSSVRepairItem.inventoryID.IsEqual<@P.AsInt>>.View.
                    Select(this, item.InventoryID);
            if (origItem != null)
            {
                item.BasePrice = origItem.BasePrice;
                RepairItems.Update(item);
            }
        }

        Actions.PressSave();
    }
}
```

In the action method, you select a repair item specified on the Services and Prices (RS203000) form and assign its price to a repair item specified on the Repair Work Orders (RS301000) form. (In the *T220 Data Entry*

and Setup Forms training course, a similar scenario is implemented in the `RowUpdated` event handler of the `RSSVWorkOrderEntry` graph.)

In the action method, you do not need to update the `RSSVWorkOrder.OrderTotal` field, which contains the total price of the work order. The value is updated automatically because the `PXFormula` attribute is specified for the `RSSVWorkOrderItem.BasePrice` field.

2. In the `RowSelected` event handler, specify that the button associated with the action should be available only when no invoice has been created for the order, as the following code shows.

```
UpdateItemPrices.SetEnabled(WorkOrders.Current.InvoiceNbr == null);
```

3. Rebuild the project.

Step 2.2: Specifying the Location of the Update Prices Button

To configure the location of a button on the form, you should define the `PXToolBarButton` element in the desired location of the form's ASPX file.

Thus, to place the **Update Prices** button on the table toolbar of the **Repair Items** tab, you should modify the ASPX page of the Repair Work Orders (RS301000) form. Do the following:

1. Open the `RS301000.aspx` file.

The file is located in the `Pages/RS` folder of the instance.

2. Add the following code in the `<px:PXTabItem Text="Repair Items">` tag after the `Levels` closing tag.

```
<ActionBar>
 <CustomItems>
  <px:PXToolBarButton Text="UpdateItemPrices">
   <AutoCallBack Command="UpdateItemPrices" Target="ds" />
  </px:PXToolBarButton>
 </CustomItems>
</ActionBar>
```

In this code, you define an action bar on the table toolbar of the **Repair Items** tab, and add a button to it.

3. Save your changes.
4. In the Customization Project Editor, update the `RS301000.aspx` file, and publish the customization project.

Step 2.3: Adding the Update Prices Button on the Labor Tab (Self-Guided Exercise)

In this exercise, you will implement the `UpdateLaborPrices` action in the `RSSVWorkOrderEntry` graph. You will configure the availability of the button associated with the action on the form in the `RowSelected` event handler of the same graph, hide the button from the form toolbar (and the corresponding command from the More menu), and define the location of the button on the **Labor** tab of the Repair Work Order (RS301000) form.

In the action method, you should select a labor item specified on the Services and Prices (RS203000) form and assign its price to a labor item specified on the Repair Work Orders (RS301000) form. A similar scenario is implemented in the `RowUpdated` event handler of the `RSSVWorkOrderEntry` graph and described in the *T220 Data Entry and Setup Forms* training course.

In the `RowSelected` event handler, make the `UpdateLaborPrices` action enabled when no invoice has been created for the order.

To perform this step, use the information you have learned in *Step 2.1: Implementing the UpdateItemPrices Action* and *Step 2.2: Specifying the Location of the Update Prices Button*.

> You can find the code changes made in this step in the `Customization\T230\CodeSnippets\Step2.3` folder, which you have downloaded from Acumatica GitHub.

Step 2.4: Testing the Update Prices Buttons and Associated Actions

To test the buttons (and the underlying actions) implemented in this lesson, do the following:

1. Modify the original repair and labor items by doing the following:
   a. Open the Service and Prices (RS203000) form, and specify the following settings:
      - **Service**: *Battery Replacement*
      - **Device**: *Nokia 3310*
   b. On the **Repair Items** tab, in the row with the *BAT3310* inventory ID, enter `25` in the **Price** column.
   c. On the **Labor** tab, in the row with the *CONSULT* inventory ID, enter `10` in the **Default Price** column.
   d. Save your changes.
2. Update the prices by doing the following:
   a. Open the Repair Work Orders (RS301000) form, and select the *000003* repair work order you created in *Step 1.3: Testing the Assign to Me Button and Associated Action*. On the **Repair Items** tab, make sure the **Update Prices** button is displayed, as shown in the following screenshot.

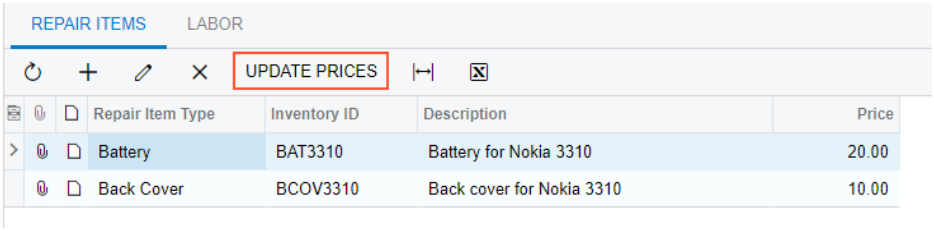


Figure: The Update Prices button on the Repair Items tab

   b. On the **Repair Items** tab, notice the old price of the repair item with the *BAT3310* inventory ID.
   c. On the table toolbar, click **Update Prices**.

      Notice that the item price has been updated and the **Order Total** value in the Summary area of the form has been updated.
   d. On the **Labor** tab, notice the old price of the labor item with the *CONSULT* inventory ID.
   e. On the table toolbar, click **Update Prices**.

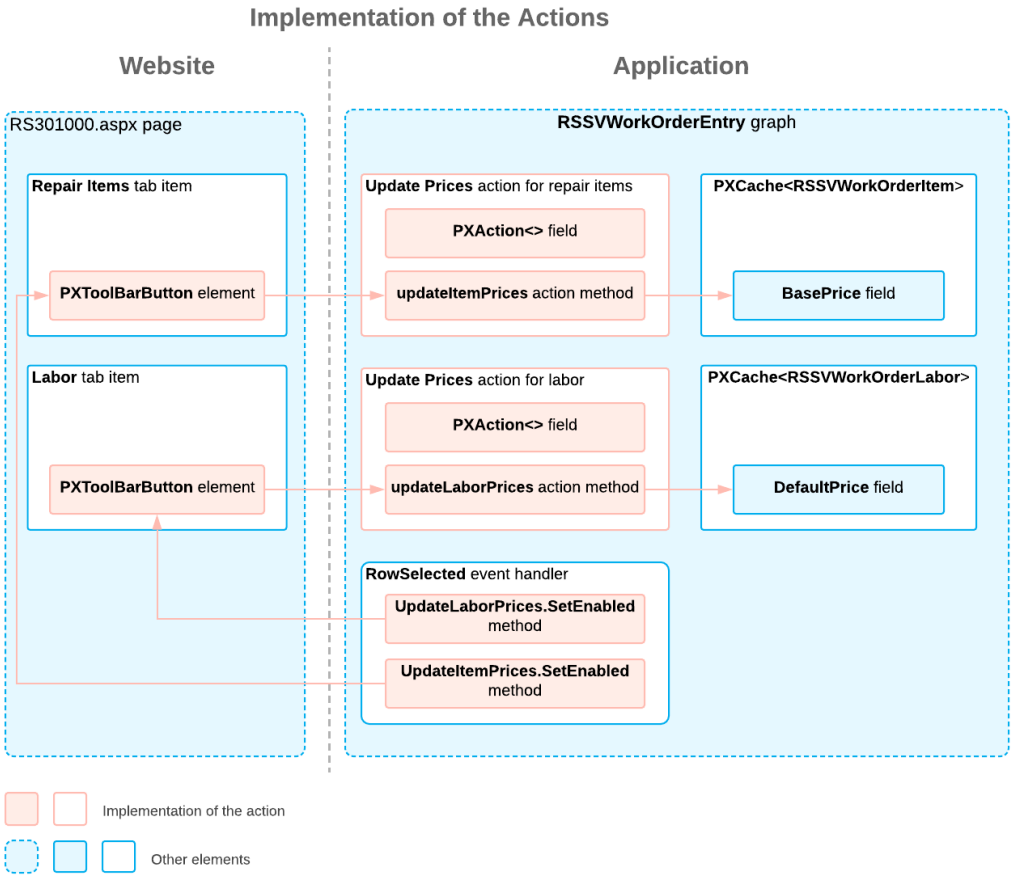
      Notice that the item price has been updated and the **Order Total** value in the Summary area of the form has been updated.

Lesson Summary

In this lesson, you have defined the `UpdateItemPrices` and `UpdateLaborPrices` actions and learned how to place the associated buttons on the table toolbar of a tab. To configure the button's location to be on a table toolbar, you have done the following:

- Hidden the buttons from the form toolbar and the equivalent commands from the More menu by setting the `DisplayOnMainToolbar` property of the `PXButton` attribute to `false`
- In the ASPX file of the form, added the `ActionBar`, `CustomItem`, `PXToolBarButton`, and `AutoCallBack` elements to the `PXGrid` section of the `PXTabItem` element

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



Lesson 3: Implementing an Asynchronous Operation

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.

Before You Proceed

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 (under **Inventory and Order Management**), 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 Settings** section), select the **Allow Negative Quantity** check box, as shown in the following screenshot.

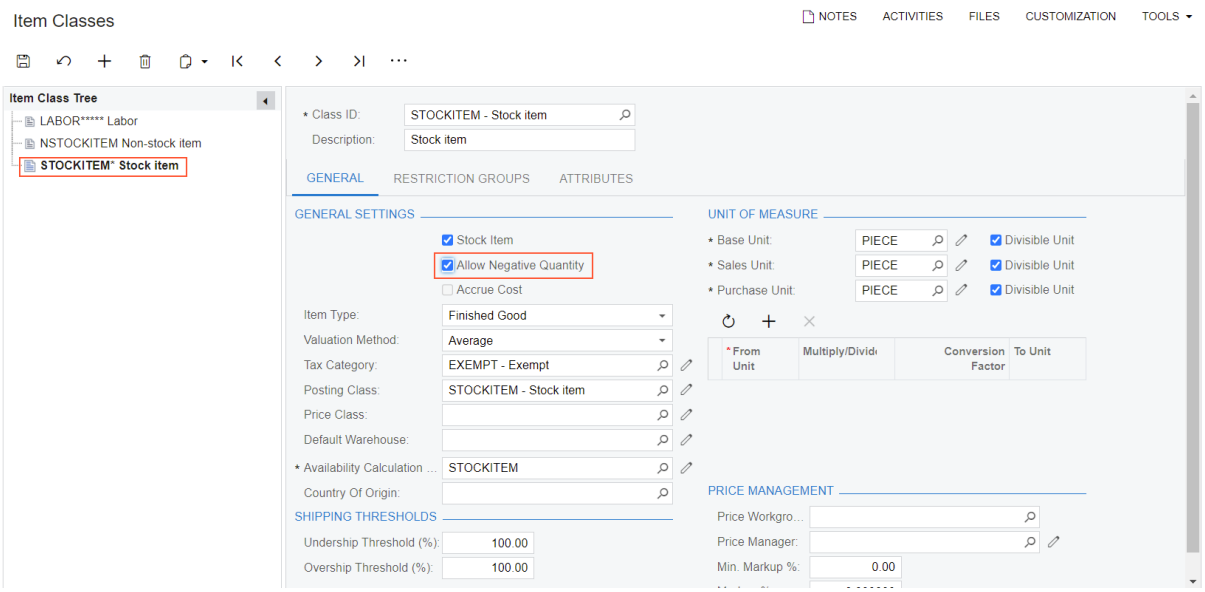


Figure: Item Classes form

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

Using the PXLongOperation.StartOperation() Method

To make the system invoke a method on a separate thread, you can use the `PXLongOperation.StartOperation()` method. Within the method that you pass to `StartOperation()`, you call the method you want to be run asynchronously, as the following code shows.

```
PXLongOperation.StartOperation(this, delegate ()
{
  InvoiceOrder(graphCopy); });
});
```

The `PXLongOperation.StartOperation()` method creates a separate thread and executes the specified delegate asynchronously on this thread. The method passed into `PXLongOperation.StartOperation()` matches the following delegate type, which has no input parameters.

```
delegate void PXToggleAsyncDelegate();
```

The anonymous method definition `(delegate())` is used to shorten the code in the example.

Inside the `delegate()` method, you should not use members of the current graph, because this would lead to synchronous execution of the method. Instead, use a copy of the graph, which you can create by using the `var graphCopy = this;` statement.

Related Links

- *PXLongOperation*
- *Asynchronous Execution*

Step 3.1: Defining the Logic Used to Create an 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 3.2: Defining the CreateInvoiceAction 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.

Step 3.3: Specifying the Availability of the Create Invoice Button and Command

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 3.4: Testing the Create Invoice Button and Associated Action

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

1. In Acumatica ERP, open the Repair Work Orders (RS301000) form.
2. Open the *000003* repair work order.

   Notice that the **Create Invoice** command is not displayed on the More menu because the status of the repair work order is not *Completed*.
3. Change the status of the repair work order to *Completed* by clicking the following workflow commands on the form toolbar:
   a. **Remove Hold**. The status of the repair work order is changed to *Ready for Assignment*.
   b. **Assign**. The status of the repair work order is changed to *Assigned*.
   c. **Complete**. The status of the repair work order is changed to *Completed*.

   These commands are defined as a part of the workflow in the `RSSVWorkOrderWorkflow.cs` and `RSSVWorkOrderEntry.cs` files.

   The **Create Invoice** button is displayed and available on the form toolbar because the status of the work order is *Completed*.
4. On the form toolbar, click **Create Invoice**.

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

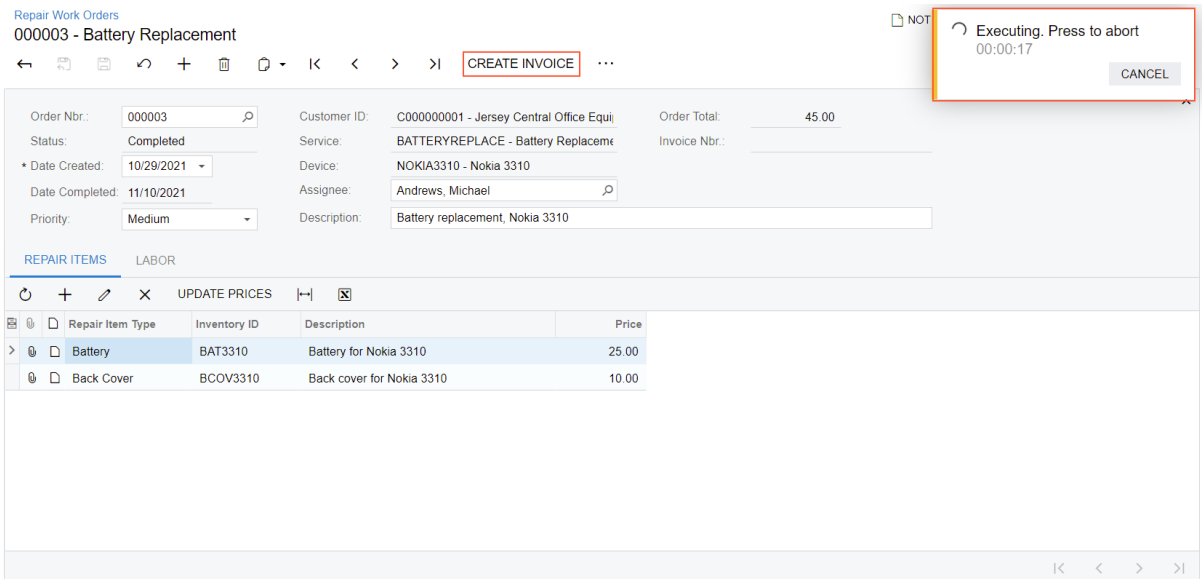


Figure: Creation of an invoice

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

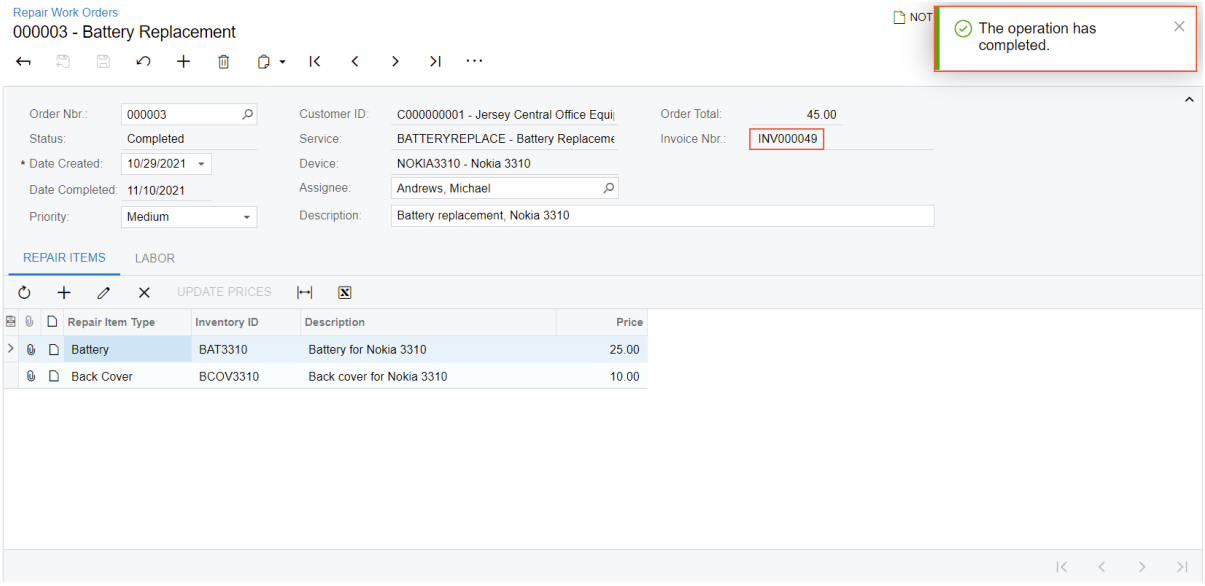


Figure: Update of the Invoice Nbr box

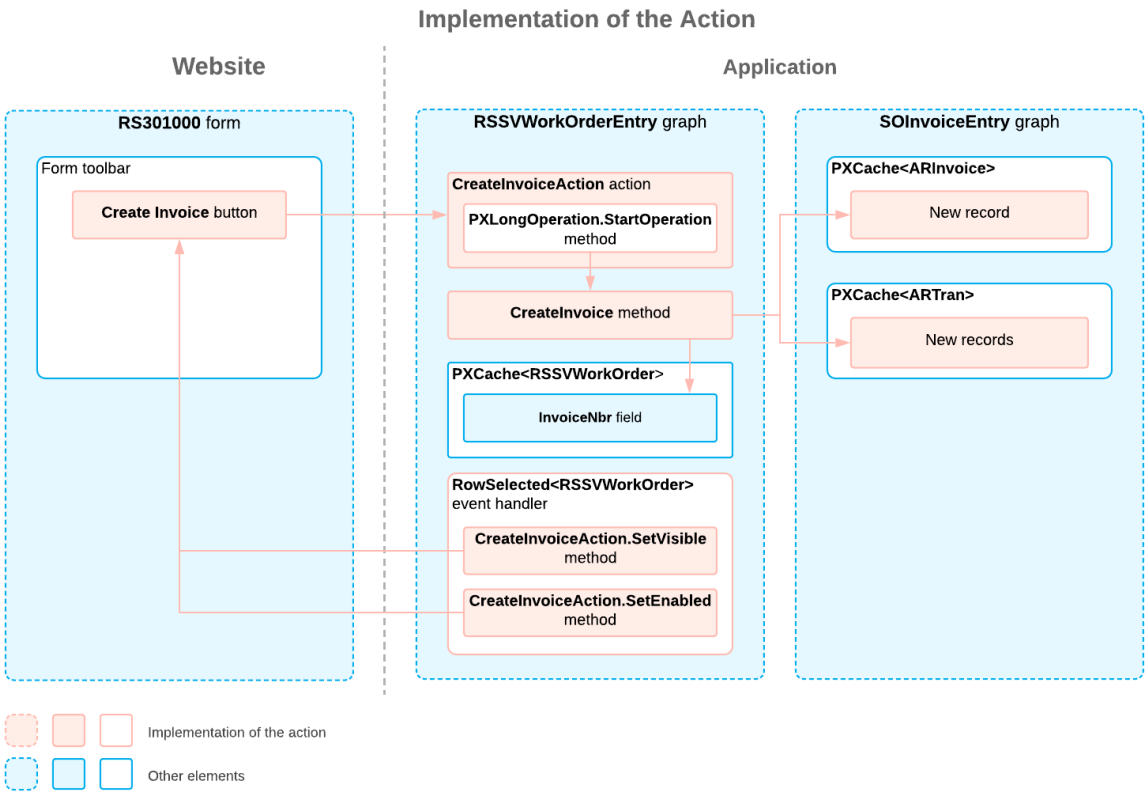
Lesson Summary

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.



Lesson 4: Defining a Link to an Acumatica ERP Entity

Once an invoice has been created, a user can find it on the *Invoices* (SO303000) form. However, it is easier to open an invoice from the corresponding repair work order on the Repair Work Orders (RS301000) form.
In this lesson, you will modify the **Invoice Nbr.** element (and the `InvoiceNbr` field) on the Repair Work Orders form so that the invoice number is displayed as a link to the *Invoices* form with the corresponding invoice opened, as shown in the following screenshot. (While clicking a link is not an action per se, a user clicks the link to open the form and view the invoice. Therefore, the behavior is similar.)

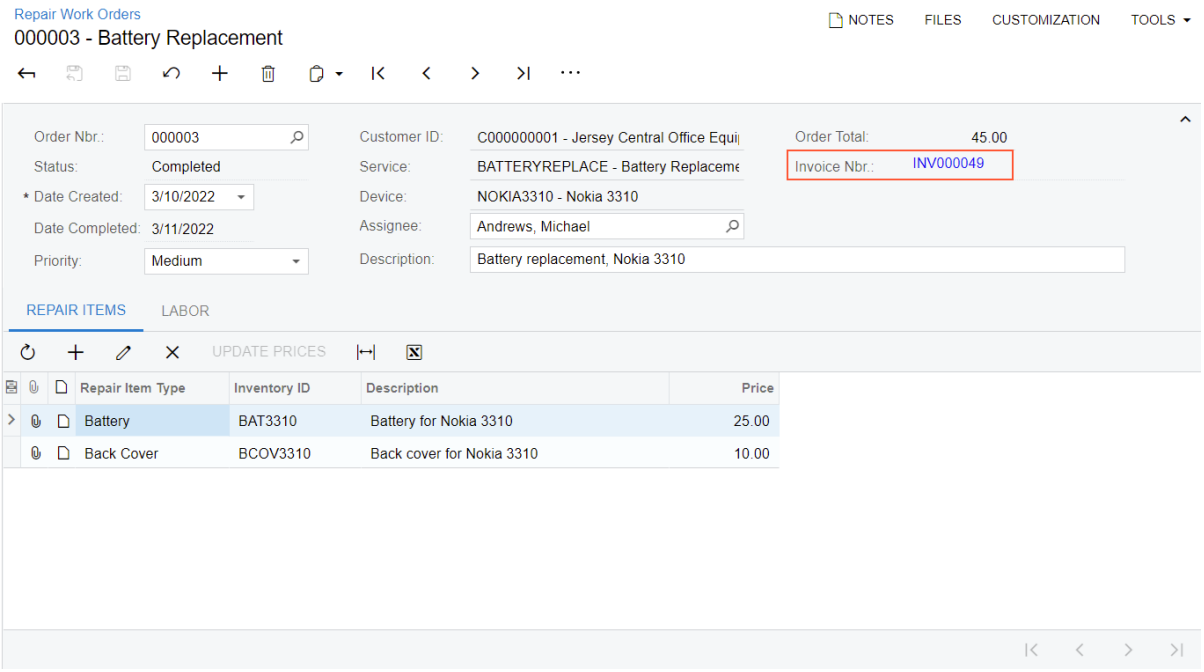


Figure: The link to open the related invoice on the Repair Work Orders form

To define this link, you will add the `PXSelector` attribute for the `InvoiceNbr` field of the `RSSVWorkOrder` DAC and configure an element for the `InvoiceNbr` field in the `RS301000.aspx` file.

Lesson Objectives

As you complete this lesson, you will learn how to define a link to an Acumatica ERP entity on a form.

Step 4.1: Configuring the RSSVWorkOrders DAC

Originally, the `InvoiceNbr` field of the `RSSVWorkOrder` DAC was configured as a plain text field. For the field to be a hyperlink, do the following in the `RSSVWorkOrder.cs` file:

1. Add the following `using` directives.

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

2. Add the following `PXSelector` attribute to the `InvoiceNbr` field definition.

```
        [PXSelector(typeof(SearchFor<SOInvoice.refNbr>.
```

```
            Where<SOInvoice.docType.IsEqual<ARDocType.invoice>>))]
```

With this attribute, the field holds the link to the invoice (which is stored in the `refNbr` field of the `SOInvoice` DAC) while displaying the invoice number.

3. Rebuild the `PhoneRepairShop_Code` project.

Related Links

- *PXSelectorAttribute*
- *Search and Select Commands and Data Views in Fluent BQL*
- *SearchFor<TField> Class*

Step 4.2: Configuring the RS301000.aspx File

After you have added the `PXSelector` attribute for the `InvoiceNbr` field, you need to configure the corresponding element in the ASPX file, as follows:

1. In the `RS301000.aspx` file, replace the original `PXTextEdit` element generated for the `InvoiceNbr` field with the following `PXSelector` element.

```
<px:PXSelector ID="edInvoiceNbr" runat="server"
 DataField="InvoiceNbr" Enabled="False" AllowEdit="True" />
```

The `AllowEdit` property set to *true* indicates that the selector should appear as a hyperlink.

2. Update the files in the *PhoneRepairShop* customization project and publish the project.

Step 4.3: Testing the Link

To test the link you have configured, do the following:

1. In Acumatica ERP, open the Repair Work Orders (RS301000) form.
2. Open the order for which you created an invoice in the previous lesson.

   Notice a hyperlink in the **Invoice Nbr.** box, as shown in the following screenshot.

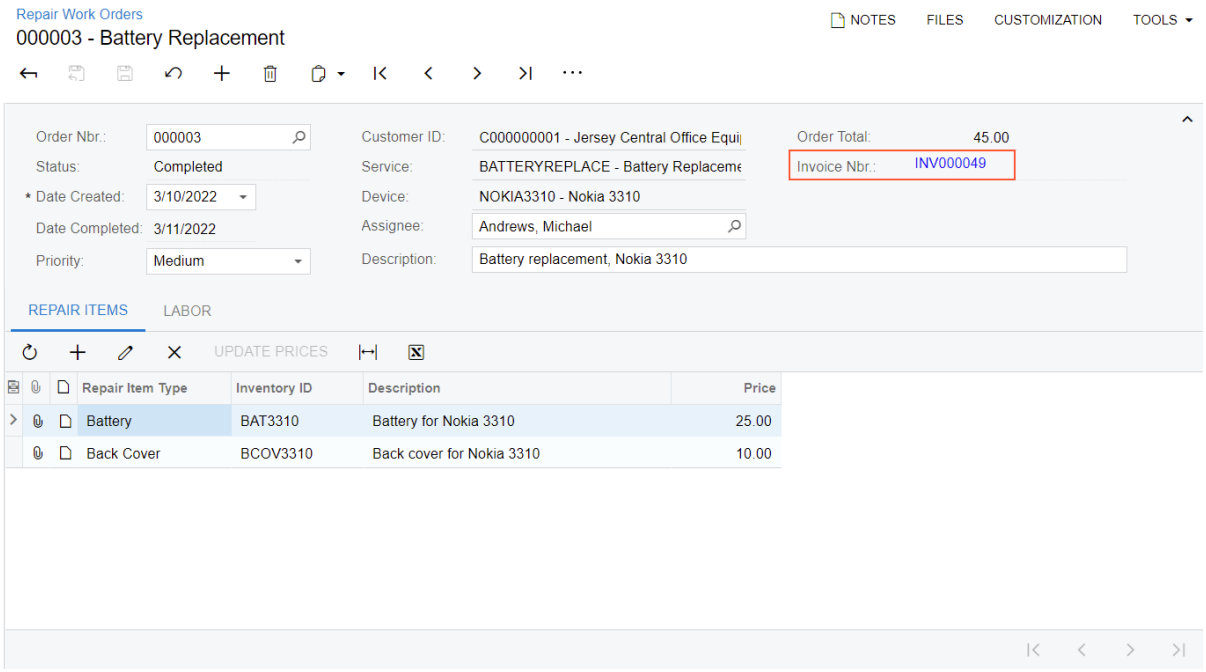


Figure: The Invoice Nbr. box with a link

3. Click the number of the invoice in the link.

   The *Invoices* (SO303000) form opens in a new window displaying the invoice created for the opened repair work order, as shown in the following screenshot.

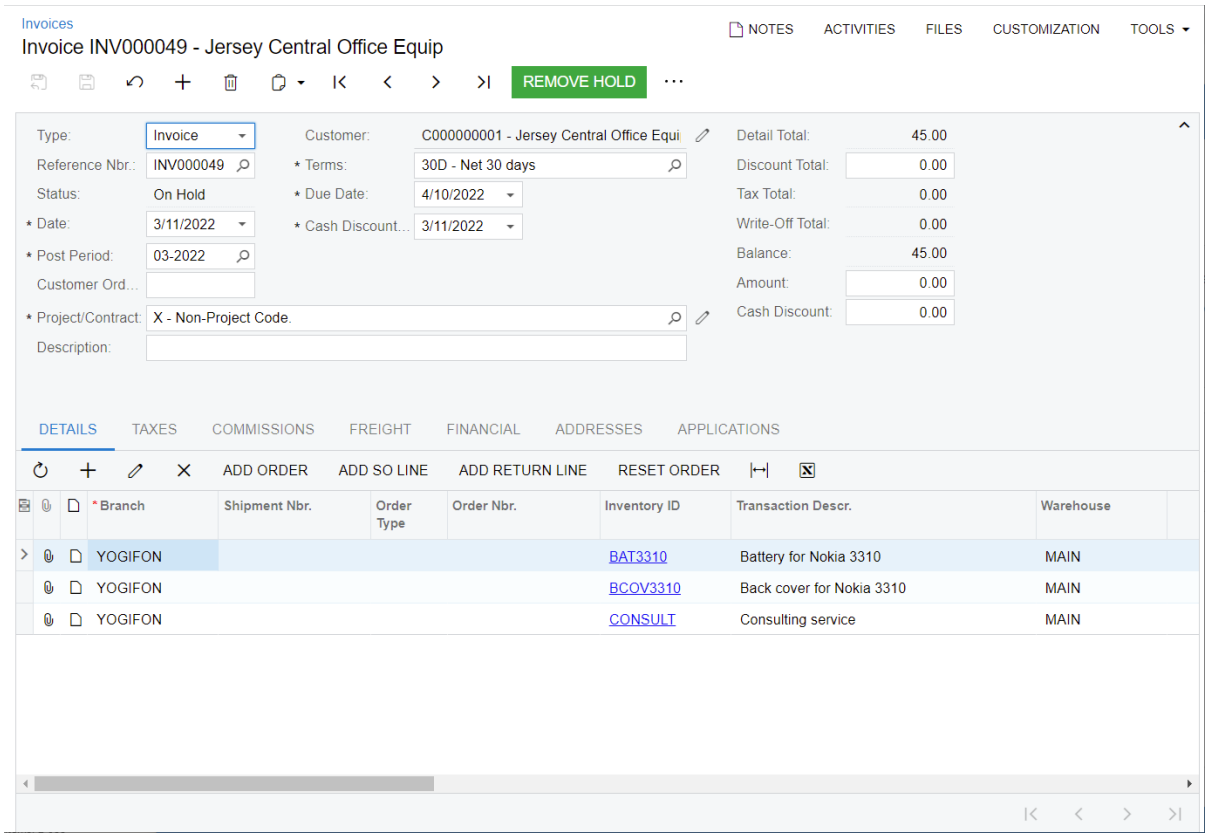


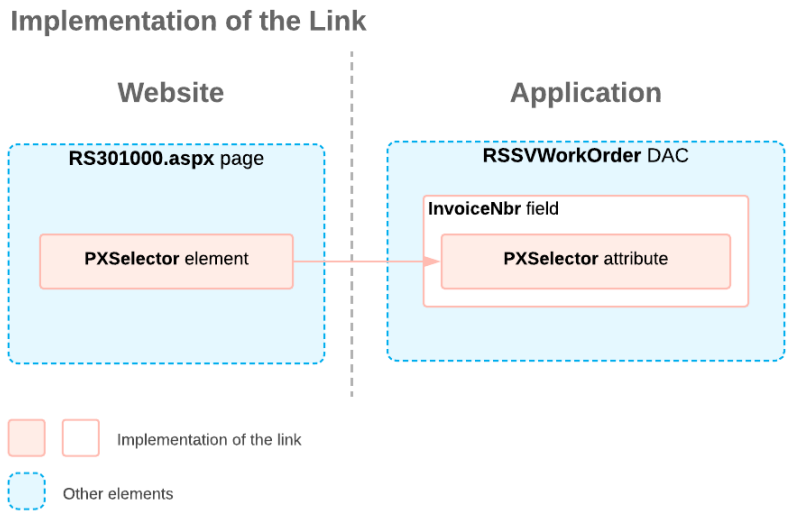
Figure: The Invoices form

> If an invoice was created for a repair work order with only non-stock items, by default, an AR invoice is created and opened in the *Invoices and Memos* (AR301000) form. The rest of the workflow is the same for both types of invoices.

Lesson Summary

In this lesson, you have configured the **Invoice Nbr.** box to contain a hyperlink that leads to the *Invoices* (SO303000) form. For this purpose, you have added the `PXSelector` attribute to the `InvoiceNbr` field of the `RSSVWorkOrder` DAC and configured the `PXSelector` element in the `RS301000.aspx` file.

The following diagram shows the changes that you have performed 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 location of a button or command on the form?
   a. By using the `SetVisible` and `SetEnabled` methods
   b. By configuring the attributes of the corresponding action
   c. By configuring the ASPX code of the page where the command is displayed
3. Select all of the items that you should use to initiate an asynchronous operation in an action.
   a. Call the `PXLongOperation.StartOperation()` method
   b. Pass the delegate of the `PXToggleAsyncDelegate` type
   c. Stop the asynchronous operation manually
4. Which attribute should you use for an action to redirect the user to a different page?
   a. `PXButton`
   b. `PXLink`
   c. `PXSelector`

Answer Keys

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

Appendix: Reference Implementation

You can find the reference implementation of the customization described in this course in the `Customization\T230` folder of the *Help-and-Training-Examples* repository in Acumatica GitHub.

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

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

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

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

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

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

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

Appendix: Publishing the Required Customization Project

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

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

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

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

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

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

The published customization project contains all changes to the Acumatica ERP website and database that have been performed in the *T200 Maintenance Forms*, *T210 Customized Forms and Master-Detail Relationship*, and *T220 Data Entry and Setup Forms* training courses. This project also contains the customization plug-ins that fill in the tables created in the *T200 Maintenance Forms*, *T210 Customized Forms and Master-Detail Relationship*, and *T220 Data Entry and Setup Forms* training courses with the custom data entered in these training courses. For details about the customization plug-ins, see *To Add a Customization Plug-In to a Project*. (The creation of customization plug-ins is outside of the scope of this course.)