Developer Course

Customization

T270 Workflow API
2022 R1

Revision: 8/12/2022

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: 2021 R2

Last Updated: 08/12/2022

Introduction

The *T270 Workflow API* training course shows how to create a workflow for a data entry form by using the workflow API of Acumatica Framework. The course describes in detail how to define and customize a workflow to change the state of an entity in Acumatica ERP.

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 through the customization tools. 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. (We recommend that you take these courses before completing the current course.)

After you complete all the lessons of the course, you will be familiar with the programming techniques for the definition of custom workflows and the customization of existing workflows 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 *Workflows in Acumatica ERP* and *Company Story and Application Requirements*.
2. Complete the lessons in all parts of the training guide.
3. In Partner University, take *T270 Certification Test: Workflow API*.

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

What Is in a Part?

The first part of the course explains how to create a workflow for a custom data entry form.

The second part of the course shows how to incorporate an existing workflow into a custom workflow.

The third part of the course describes the customization of a workflow of a predefined Acumatica ERP form.

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

What Is in a Lesson?

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

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

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

What Are the Documentation Resources?

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

Course Prerequisites

To complete this course, you should be familiar with the basic concepts of Acumatica Framework and Acumatica Customization Platform. 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 the *T220 Data Entry and Setup Forms* training course and are using the same environment for the current course, you can skip this step.

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

1. Make sure the environment that you are going to use for the training course conforms to the *System Requirements for Acumatica ERP 2022 R1*.
2. Make sure that the Web Server (IIS) features that are listed in *Configuring Web Server (IIS) Features* are turned on.
3. Install the Acuminator extension for Visual Studio.
4. Clone or download the customization project and the source code of the extension library from the *Help-and-Training-Examples* repository in Acumatica GitHub to a folder on your computer.
5. Install Acumatica ERP. On the Main Software Configuration page of the installation program, select the **Install Acumatica ERP** and **Install Debugger Tools** check boxes.

   If you have already installed Acumatica ERP without debugger tools, you should remove Acumatica ERP and install it again with the **Install Debugger Tools** check box selected. 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)` and `C:\Program Files` folders. (We recommend that you store the website folder outside of these folders to avoid an issue with permission to work in these folders when you perform customization of the website.)
      b. In the **Training Course** box, select the training course you are taking.

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

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

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

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

> If you cannot complete the instructions in this step for some reason, 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*.

If you have deployed the project using the wizard. and the Visual Studio project contains the following items, remove them:

- The `Workflows` folder with the `RSSVWorkOrderWorkflow.cs` file
- In the `RSSVWorkOrder.cs` file, the `Actions` region

Workflows in Acumatica ERP

A workflow is a depiction of the ways that the state of an entity created on a form changes as a result of a user invoking specific actions on the form. A workflow can be described as a state machine, with transitions indicating the movement of the entity through its processing in the system as the corresponding work is performed in the company. For example, a workflow can involve the changing of the status of an opportunity based on the buttons and commands a user invokes on the *Opportunities* (CR304000) form to reflect the progress made with the potential customer that represents an opportunity.

Defining or Customizing a Workflow

You can define or customize a workflow either by coding in an extension library or by using the Workflow Editor of the Customization Project Editor. For details on creating a workflow by using the Workflow Editor, see the *W150 Workflows* training course.

You can customize the predefined workflows of the forms that have them; the resulting customized workflows are also referred to as *inherited*. You can also create custom workflows that are not based on existing workflows.

You might need to use workflows if the movement of documents or entities in the company follows an established sequence of operations. By customizing predefined workflows to represent this sequence, you can automate the company's processes and speed the processing of documents or entities.

You can define a single workflow for the whole form or multiple workflows, one for each specific field value of the entity. You can configure the settings—such as field properties, conditions, and actions—for the whole form. For each of the workflows of the form, you configure the properties of actions and fields for every state the entity can have. These properties determine the appearance of the form when the entity has a particular state.

Using Conditions

Conditions can be used in the properties of actions and fields at the form level (that is, for all workflows of a particular form). At the workflow level, conditions can be used to determine whether transitions are performed. Also, conditions can be used to determine whether actions are performed automatically.

You can configure the action and field properties for a form and its workflows at the same time, depending on the conditions specified for a form when an entity has any state or has a particular state. The action properties indicate whether the action is visible and whether it is enabled. The field properties indicate whether the field is visible, whether it is enabled, and whether it is required. If at the workflow level or at the form level, a field is disabled, hidden, or required, it becomes disabled, hidden, or required, respectively, on the form. If an action is disabled or hidden at one of these levels, it becomes disabled or hidden, respectively, on the form.

Transition Steps in a Workflow

Transition steps in a workflow depend on whether a user performs a particular action or an event is raised in the code. An event can be raised when some change occurs on the current form to the current entity or on another form to another entity.

Transition Steps for an Action

In the following diagram, you can see the steps that the system performs during a transition to the target state when a user invokes an action on the UI of the current form.

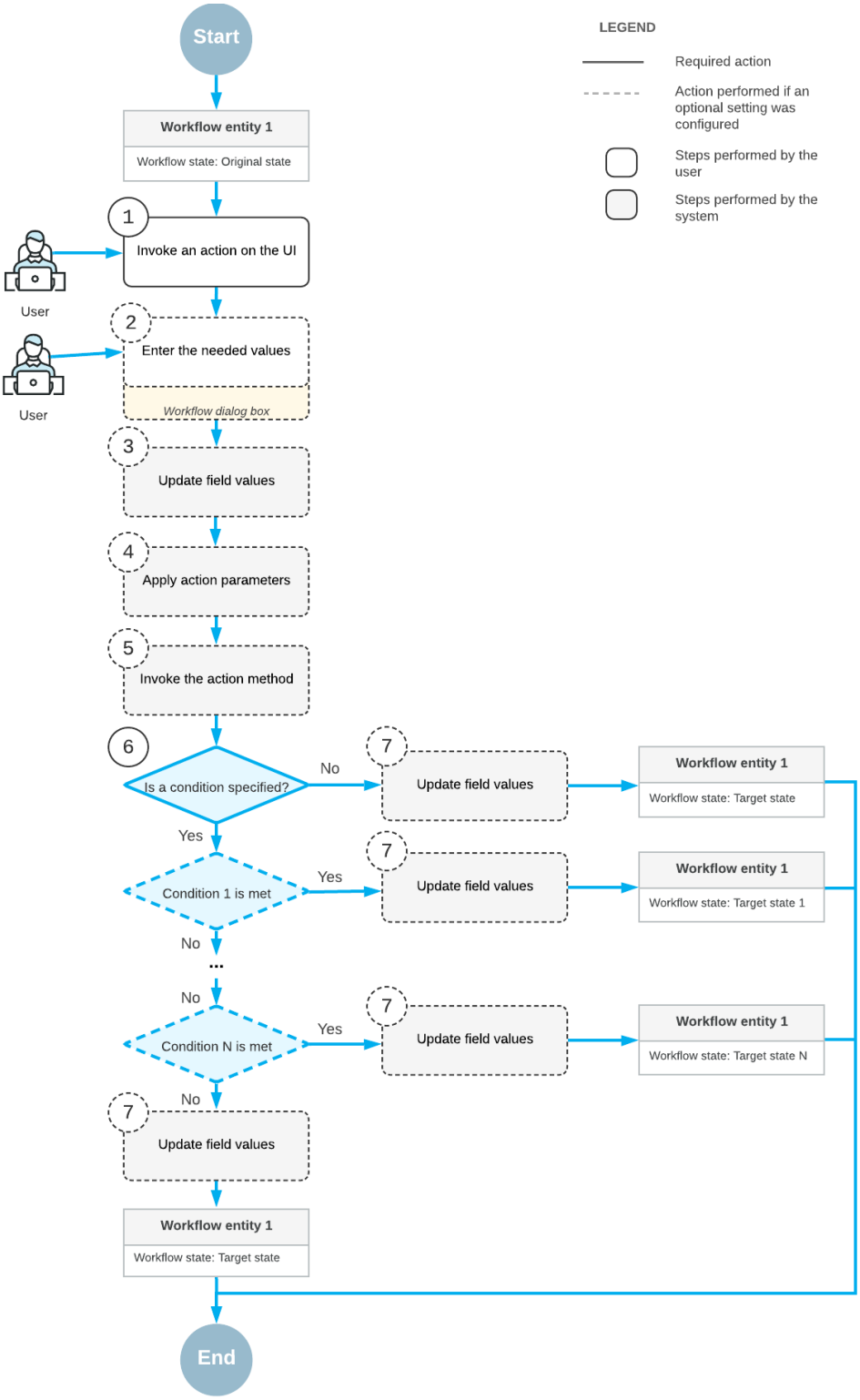


Figure: Steps in a transition for an action

First, a user invokes an action on the UI of the form (Item 1 in the diagram). If applicable, the dialog box you have specified for the action is displayed, and the user enters the needed values in this dialog box (Item 2). Then the system updates the fields you have specified with the values the user has entered in the dialog box for the action (Item 3). Optionally, if the action is defined in a graph, the system applies the properties that you have specified for the action (Item 4) and invokes the action method (Item 5); only actions defined in a graph can have a method to be

invoked and properties to be applied. If you have specified both a transition that the action triggers and a condition for this transition, the condition is then checked (Item 6).

> In each transition, you can check for only one condition. To check for multiple conditions, you have to define multiple transitions, one for each condition. However, if you need to create the same transition that is performed under different conditions, you cannot create multiple transitions with the same initial and target states. In that case, you should unite conditions with the OR operator.

If you have specified the fields to be updated after the transition, the system updates these fields (Item 7). The system changes the entity state to the target state you have specified for the transition.

Transition Steps for an Event Handler

In the following diagram, you can see the steps that the system performs during a transition to the target state when some change occurs on the current form to the current entity or on a different form to another entity. This raises an event in the code, which in turn invokes the event handler.

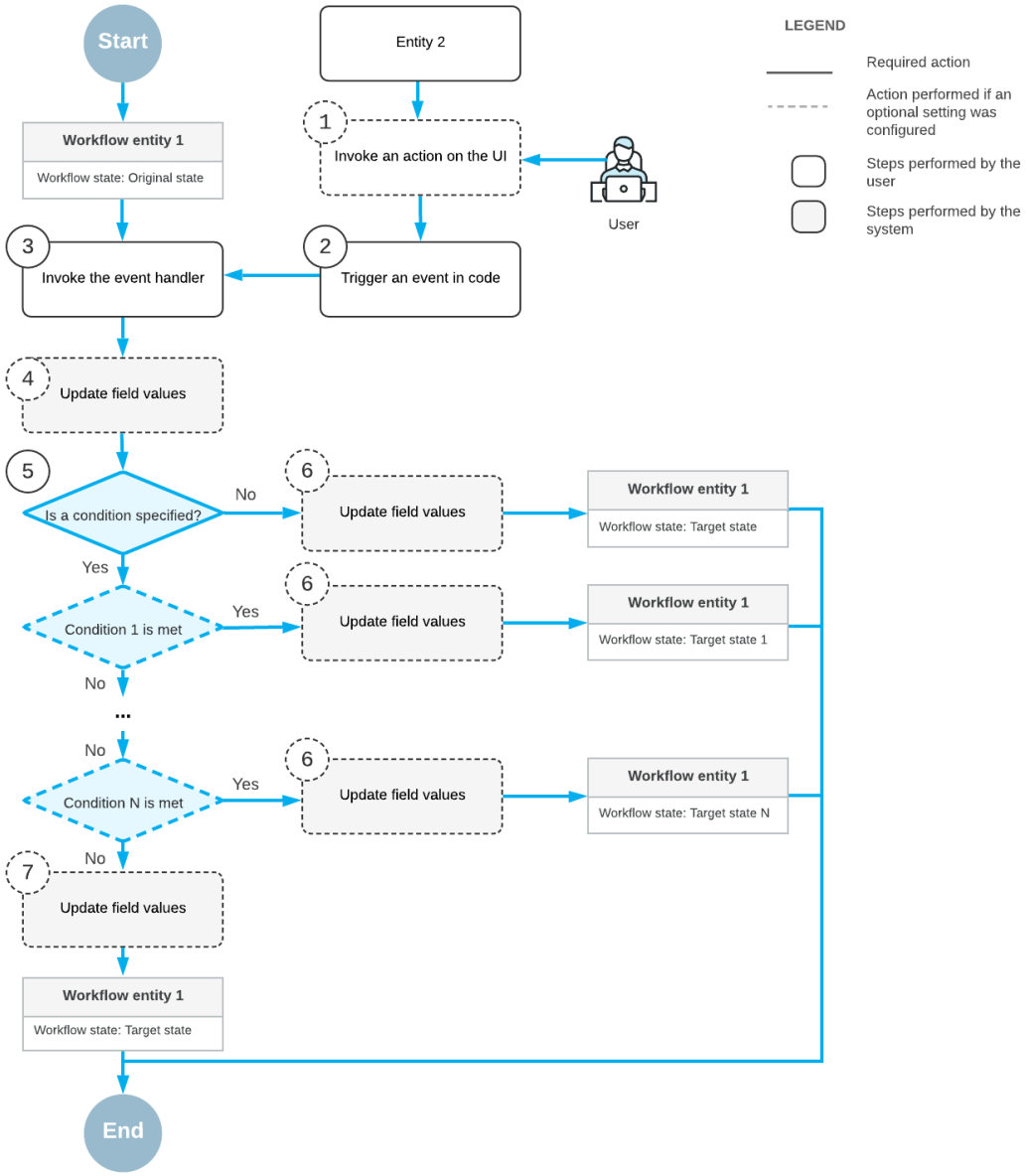


Figure: Steps in a transition for an event handler

First, a user invokes an action on the UI of a different form for some other entity (Item 1 in the diagram) or changes a value in a box on this form or another form. This triggers an event in the code (Item 2). Then the system invokes an event handler (Item 3) and updates any fields that you have specified for this event handler (Item 4). If you have specified a transition for this event handler and a condition for this transition, the condition is checked (Item 5).

In each transition, you can check for only one condition. To check for multiple conditions, you have to define multiple transitions, one for each condition. However, if you need to create the same transition that is performed under different conditions, you cannot create multiple transitions with the same initial and target states. In that case, you should unite conditions with the OR operator.

Optionally, the system updates the fields you have specified to be updated after the transition (Item 6). The system changes the original state of the entity to the target state you have specified for the transition.

Company Story and Application Requirements

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): The Smart Fix company uses this form to manage the list of the repair services the company provides.
- Serviced Devices (RS202000): On this form, Smart Fix company manages the list of the devices serviced by the company.

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 repair service the company provides. The Stock Items form has been customized to give users the ability to mark particular stock items as repair items—that is, items that are used for 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.

When you completed the steps of *Initial Configuration*, you have installed an instance of Acumatica ERP with the published customization project that contains the results of these courses.

In the *T270 Workflow API* course, you will implement a workflow on the Repair Work Orders form. The workflow will change the state of a document created on the form—that is, the status of the repair work order and the related properties of the fields and actions on the form. Also, you will customize the workflow on the *Invoices* (SO303000) form by adding an action that opens the Repair Work Orders form. You will make the action available on in one status of an invoice.

Types of Repair Work Orders

A repair work order may be created for the following types of services, which are defined on the Repair Services (RS201000) form:

- *Battery Replacement*
- *Liquid Damage*
- *Screen Repair*

As specified on the Repair Services form, the *Battery Replacement* service does not require a prepayment. The total cost of the order must be paid in full after the repair is completed.

The *Liquid Damage* service requires prepayment. The percent of the prepayment is specified on the Repair Work Order Preferences (RS101000) form.

In this course, you will implement the changing of statuses for the *Battery Replacement* and *Liquid Damage* services.

This training course does not cover the implementation of changing of statuses for the *Screen Repair* service. You can do this as a self-guided exercise.

Business Process Overview

The workflow to be used for repair work orders created on the Repair Work Orders (RS301000) form will differ slightly depending on the particular service being performed, which can be *Battery Replacement*, *Liquid Damage*, or *Screen Repair*. This topic describes how the workflow will work for a repair work order for the *Battery Replacement* and *Liquid Damage* service.

The Battery Replacement Service

When a user creates an order for the *Battery Replacement* service on the Repair Work Orders (RS301000) form, the order has the *On Hold* status. To change the order's status to *Ready for Assignment*, a user will click the **Remove Hold** button on the form toolbar. Then a user will click the **Assign** button and specify the assignee in the **Select Assignee** dialog box. The order's status will be changed to *Assigned*. When work on the order is completed, a user will complete the assigned order by clicking the **Complete** button on the **Labor** tab, which will give the order the *Completed* status. After that, a user creates an invoice for the order. When the invoice is fully paid, the system will assign the *Paid* status to the repair work order.

Thus, based on this workflow, a repair work order for the *Battery Replacement* service will be able to have the following statuses:

- *On Hold*

  A newly created order has this status by default.
- *Ready for Assignment*

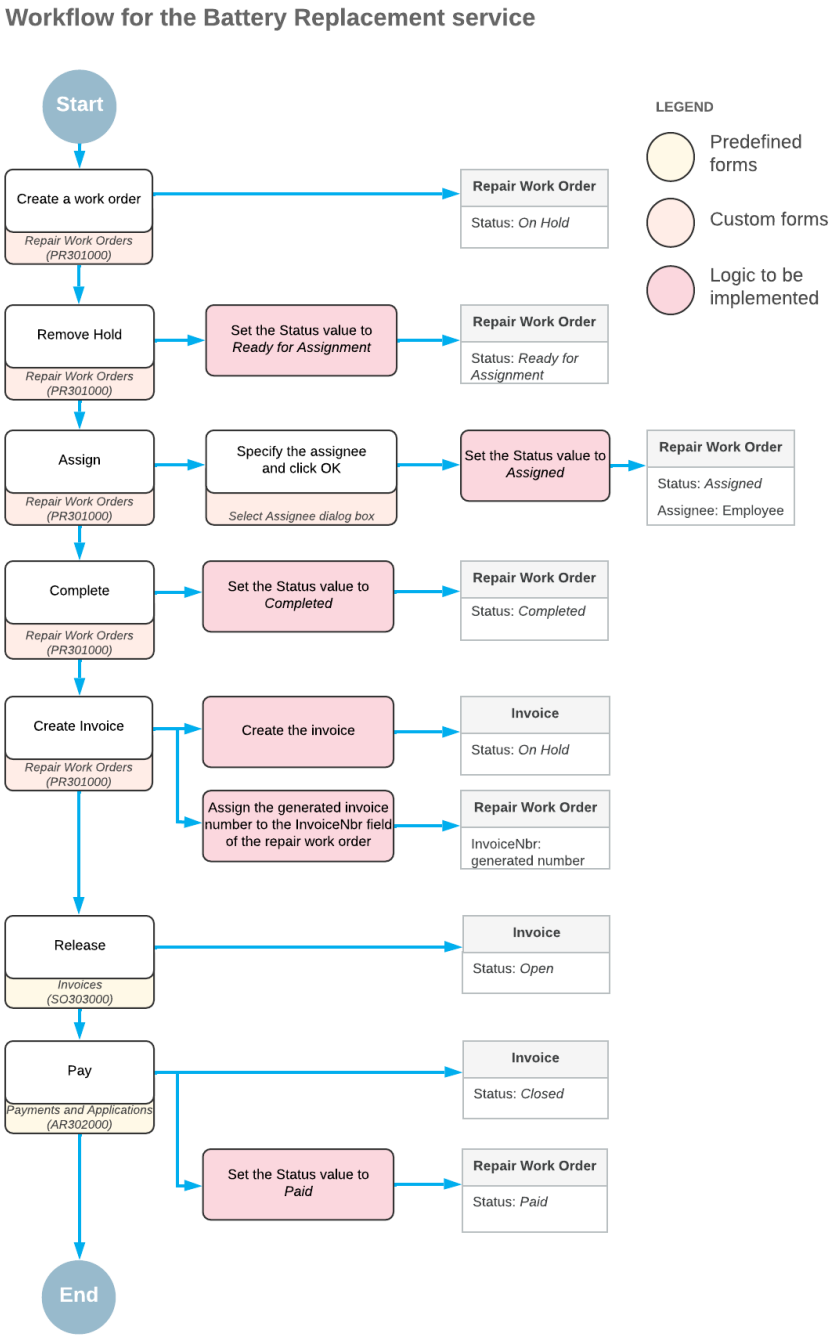
  This status will be assigned when a user clicks the **Remove Hold** button.
- *Assigned*

  This status will be assigned when a user clicks the **Assign** button.
- *Completed*

  This status will be assigned when a user clicks the **Complete** button on the **Labor** tab.
- *Paid*

  The system will assign this status to the order when the order is fully paid.

The following diagram shows the planned workflow for a repair work order for the *Battery Replacement* service. The system actions that have been or will be implemented in the *PhoneRepairShop* customization project are shown in the middle column.



The Liquid Damage Service

When a user creates a repair work order for the *Liquid Damage* service on the Repair Work Orders (RS301000) form, the order has the *On Hold* status. To cause the system to change the order's status to *Pending Payment*, a user will click the **Remove Hold** button on the form toolbar. This status indicates that a user needs to create, release, and apply a prepayment.

An order for the *Liquid Damage* service requires prepayment, based on the **Requires Prepayment** check box being selected on the Repair Services (RS201000) form. The required percent to be prepaid has been specified in the **Prepayment Percent** box on the Repair Work Order Preferences (RS101000) form.

Thus, based on this workflow, a repair work order for the *Liquid Damage* service will be able to have the following statuses:

- *On Hold*

  A newly created order has this status by default.
- *Pending Payment*

  This status will be assigned when a user clicks the **Remove Hold** button on the form toolbar.
- *Ready for Assignment*

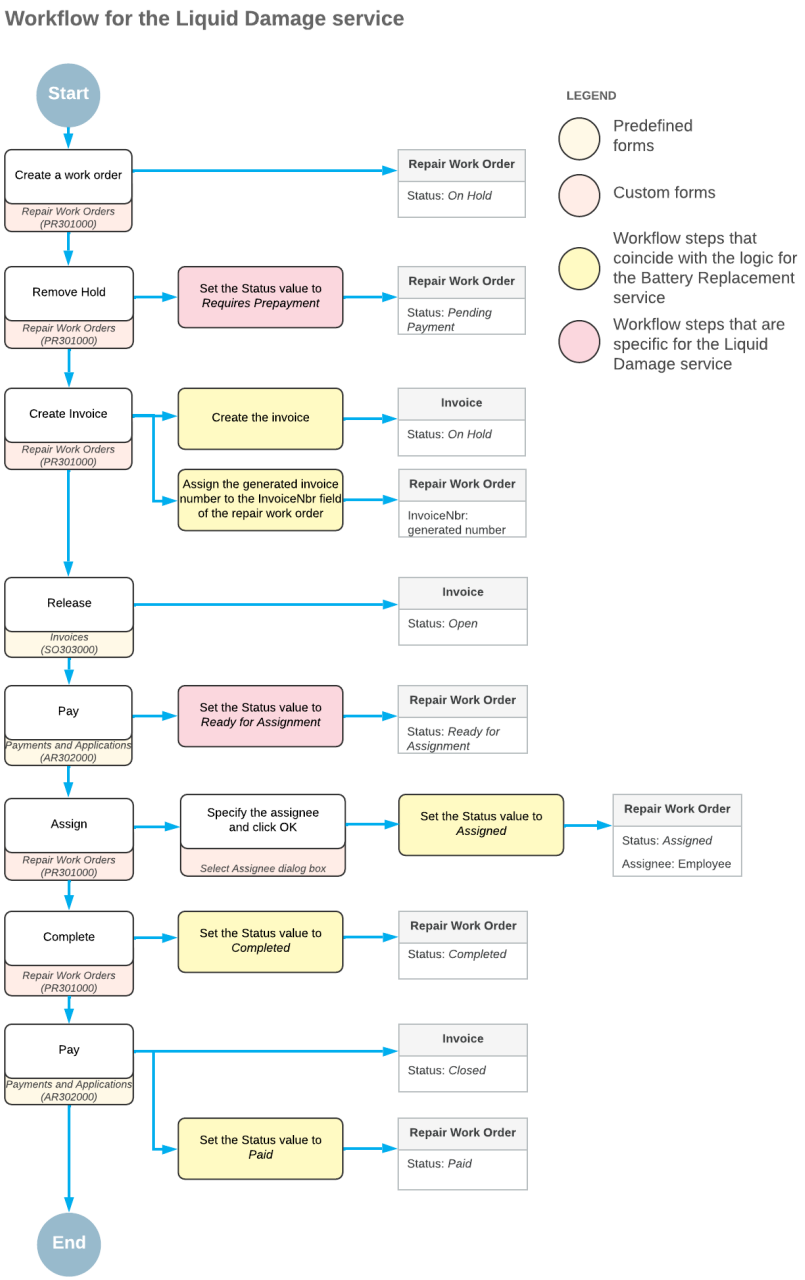
  This status will be assigned to an order when a prepayment for it has been created, released, and applied, and the prepayment percent is greater than or equal to the required prepayment percent.
- *Assigned*

  This status will be assigned when a user clicks the **Assign** button.
- *Completed*

  This status will be assigned when a user clicks the **Complete** button on the **Labor** tab.
- *Paid*

  The system will assign this status to the order when the order is fully paid. The payment will be applied to the same invoice to which the prepayment was applied.

The following diagram shows the planned workflow for a repair work order for the *Liquid Damage* service. The system actions that have been or will be implemented in the *PhoneRepairShop* customization project are shown in the middle column.



Customization Description

This topic describes changes that will be implemented as part of the customization for the Smart Fix company.

Custom Workflow for the Repair Work Orders Form

The review of the business processes has illustrated that the workflow for both services (one of which requires prepayment and one of which does not) can be united into a workflow, which is shown in the following diagram. In this course, you will implement this workflow for the Repair Work Orders (RS301000) form.

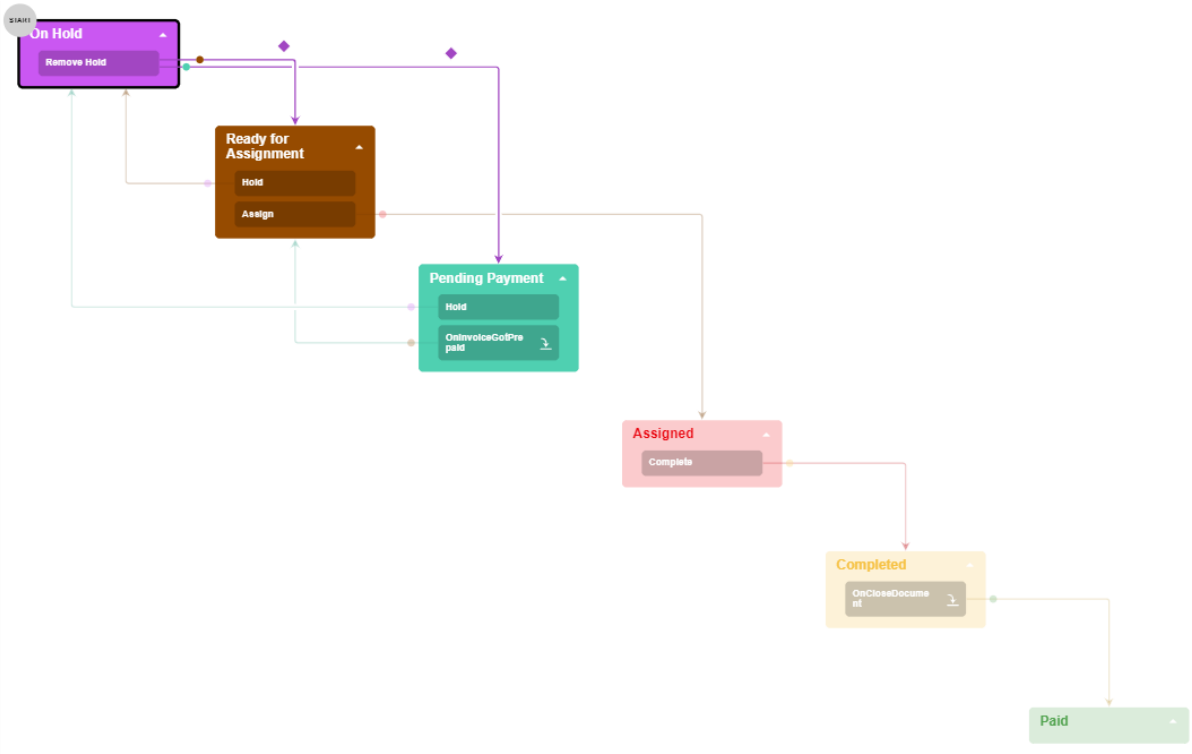


Figure: The Workflow on the Repair Work Orders form

In the workflow, you will implement the following items:

- The states of the workflow that correspond to the following statuses of a repair work order:
  - *On Hold*
  - *Pending Payment*
  - *Ready for Assignment*
  - *Assigned*
  - *Completed*
  - *Paid*
- The following actions, which trigger transitions of a repair work order:
  - **Remove Hold**, which triggers a transition from the *On Hold* status to *Pending Payment* or *Ready for Assignment* status
  - **Assign**, which triggers a transition from the *Ready for Assignment* status to the *Assigned* status
  - **Complete**, which triggers a transition from the *Assigned* status to the *Completed* status

  You will also define the **Create Invoice** action, which generates an invoice for the repair work order.
- The transitions between states of a repair work order
- A dialog box that is shown when a user clicks the **Assign** action
- The following event handlers, which trigger transitions for a repair work order:
  - `OnInvoiceGotPrepaid` event handler which triggers a transition from the *Pending Payment* status to the *Ready for Assignment* status
  - `OnCloseDocument` event handler which triggers a transition from the *Completed* status to the *Paid* status
- The conditions that determine to which state a document should transition from the `OnHold` state

You will also customize an existing Acumatica ERP graph to implement a transition in your custom workflow.

The resulting Repair Work Orders form will appear as shown in the following screenshot.

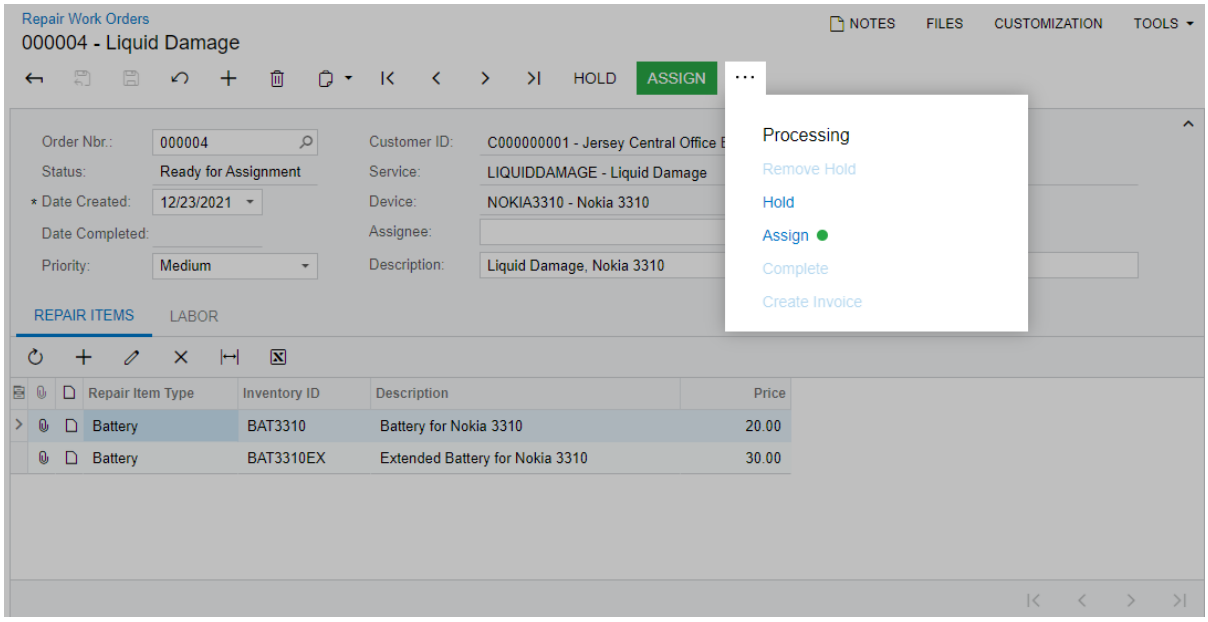


Figure: The Repair Work Orders form

Customized Workflow for the Invoices Form

To continue working on a repair work order after an invoice has been prepaid, a user needs an action that opens the corresponding repair work order from the *Invoices* (SO303000) form. The command corresponding to the action should be named **View Repair Work Order** and should be displayed on the More menu in the **Repair Work Orders** category, as shown in the following screenshot.

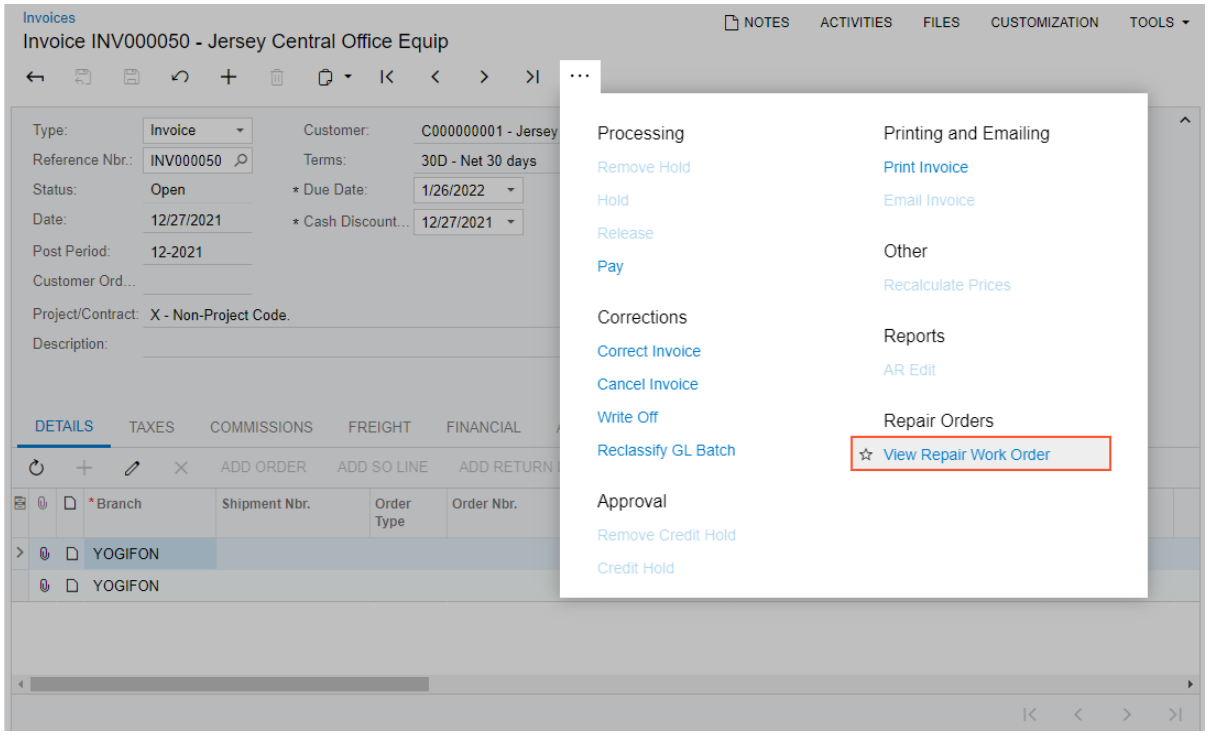


Figure: The View Repair Work Order action

To implement this task, you will extend the graph to define the action and customize the predefined workflow of the *Invoices* form. In the customized workflow, you will do the following:

- Define the workflow action
- Define the action category
- Add the action to the workflow state

Configuration of the Instance

Based on the description of the planned customization, you need to configure the PhoneRepairShop instance to be able to test the implemented logic.

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 on the *Invoices* (SO303000) form, 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.
3. Click **Enable** on the form toolbar.

If you test the **Create Invoice** action multiple times, you may run into an issue with not enough stock items in a warehouse. When an SO invoice is released, the quantity of stock items included in the invoice is checked, and if there are not enough stock items in a warehouse, an invoice cannot be created. To fix the issue, you can allow negative quantities for stock items.

To allow negative quantities for stock items, do the following:

1. 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.
2. On the **General** tab (**General** section), select the **Allow Negative Quantity** check box, as shown in the following screenshot.

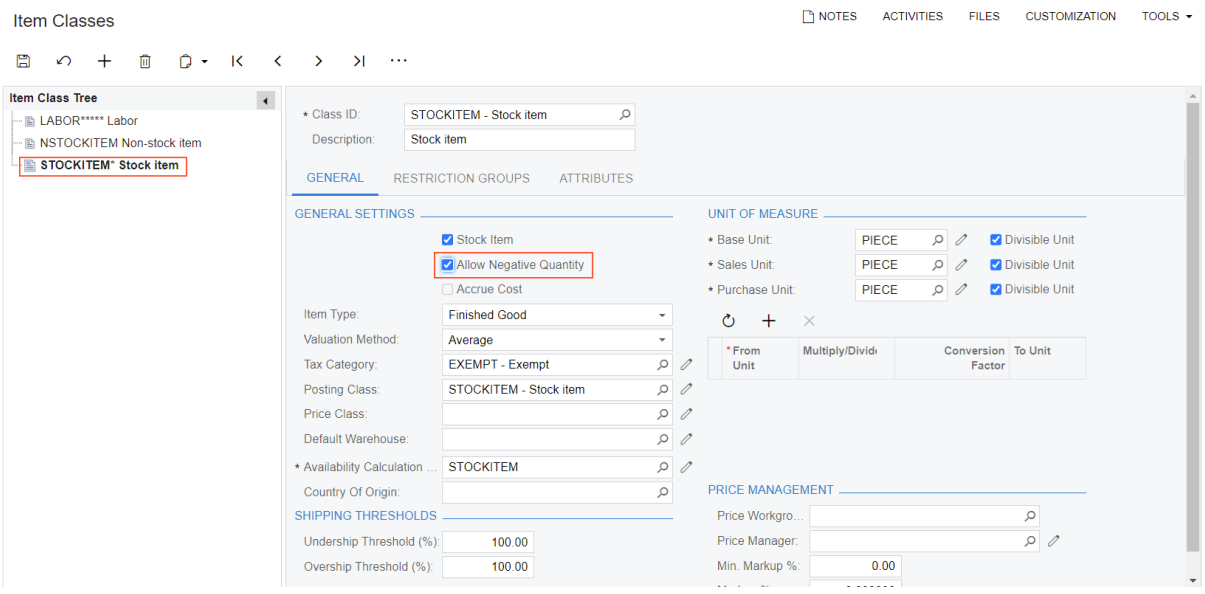


Figure: Item Classes form

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

Also, to skip entering values during release of an invoice, on the *Accounts Receivable Preferences*(AR101000) form, clear the **Validate Document Totals on Entry** and **Require Payment Reference on Entry** boxes.

Part 1: Creating a Custom Workflow

In this part, you will learn about the essential components of a workflow and implement the basic parts of a custom workflow for the Repair Work Orders (RS301000) form.

Lesson 1.1: Set Up the Basic Parts of the Workflow

In this lesson, you will learn how to set up the basic parts of a workflow. You will learn about screen configuration and perform the initial steps for implementing a workflow.

Learning Objectives

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

- Determine the set of states of a document in the workflow
- Define the set of states for the workflow
- Create the screen configuration method
- Enable workflow validations

Screen Configuration

Screen configuration is defined by the set of elements on a screen and the workflows in which these elements are used. To configure a screen, you can define any of the following:

- Conditions
- Dialog boxes
- States of an object that has been created on the screen
- Actions and their categories
- A workflow, for which you define the following:
  - Workflow states, which can include action definitions and field states
  - Workflow transitions, which can be triggered based on conditions
  - Events and event handlers

Every screen has an empty configuration by default. Therefore, when you implement a workflow, you edit the existing screen configuration. To edit the screen configuration, you override the `Configure` method in an extension of the graph that defines the business logic of the screen.

Inside the `Configure` method, you define a workflow by using the screen configuration API (which is also called the *workflow API*). The screen configuration API has a strongly typed syntax. It has many compiler-based validations, which reduce the likelihood of an error. The API is made mostly of fluent generic methods, which use LINQ expressions.

Step 1.1.1: Define the Workflow Class

In this step, you will define a class where the workflow will be implemented in later lessons.

A workflow is defined in an extension of the graph for which the workflow should be applied. For example, if you need to implement a workflow for the *Opportunities* (CR304000) form, whose business logic is defined in the `OpportunityMaint` graph, you need to create an extension of the `OpportunityMaint` graph.

To define a workflow class, do the following:

1. In the `PhoneRepairShop_Code` project, create the `Workflows` folder.
2. In the `Workflows` folder, add a new item based on the C# class template named `RSSVWorkOrderWorkflow.cs`.
3. In the `RSSVWorkOrderWorkflow.cs` file, define the workflow class, as the following code shows.

```
using PX.Data;
using PX.Data.WorkflowAPI;
using static PX.Data.WorkflowAPI.BoundedTo<PhoneRepairShop.RSSVWorkOrderEntry,
  PhoneRepairShop.RSSVWorkOrder>;

namespace PhoneRepairShop.Workflows
{
    public class RSSVWorkOrderWorkflow :
      PX.Data.PXGraphExtension<RSSVWorkOrderEntry>
    {

    }
}
```

4. Use Acuminator to suppress the *PX1016* error in a comment. In this course, for simplicity, the extension is always active.
5. Save your changes.

Step 1.1.2: Define the Set of States of the Workflow

The first step in creating a workflow is determining the set of states that an object can have in the workflow.

In this step, you will define the set of states for a document—in this case, a repair work order. The state is determined by one of the fields of the object, which is usually the `Status` field. Therefore, you need to define the set of states that corresponds to the set of statuses that a repair work order can have.

According to the customization description, which is available in *Company Story and Customization Description*, a repair work order can have the following statuses:

- *On Hold* (the `WorkOrderStatusConstants.OnHold` constant)
- *Ready for Assignment* (the `WorkOrderStatusConstants.ReadyForAssignment` constant)
- *Pending Payment* (the `WorkOrderStatusConstants.PendingPayment` constant)
- *Assigned* (the `WorkOrderStatusConstants.Assigned` constant)
- *Completed* (the `WorkOrderStatusConstants.Completed` constant)
- *Paid* (the `WorkOrderStatusConstants.Paid` constant)

Each state is defined by a set of a constant string value and a class derived from the `BqlString.Constant` class. The name of the class starts with a lowercase letter.

To create the set of states, do the following:

1. In the `Constants.cs` file, make sure the constants for the **Status** box are defined as shown in the following code. These values will be used to indicate states of the workflow.

```
//Constants for the statuses of repair work orders
```

```
public static class WorkOrderStatusConstants
{
public const string OnHold = "OH";
public const string PendingPayment = "PP";
public const string ReadyForAssignment = "RA";
public const string Assigned = "AS";
public const string Completed = "CM";
public const string Paid = "PD";
}
```

2. In the `RSSVWorkOrderWorkflow` class, define the `Constants` region and the public static class inside it, as the following code shows.

```
        #region Constants
        public static class States
        {
        }
        #endregion
```

3. In the `States` class, define the constant string values that correspond to the repair work order statuses, as the following code shows.

```
            public const string OnHold = WorkOrderStatusConstants.OnHold;
            public const string ReadyForAssignment =
              WorkOrderStatusConstants.ReadyForAssignment;
            public const string PendingPayment =
              WorkOrderStatusConstants.PendingPayment;
            public const string Assigned = WorkOrderStatusConstants.Assigned;
            public const string Completed = WorkOrderStatusConstants.Completed;
            public const string Paid = WorkOrderStatusConstants.Paid;
```

As the `Status` field value, you have used the enumeration you defined in *T220 Data Entry and Setup Forms*.

4. In the `States` class, define the classes for each state of the workflow, as the following code shows.

```
            public class onHold : PX.Data.BQL.BqlString.Constant<onHold>
            {
                public onHold() : base(OnHold) { }
            }

            public class readyForAssignment :
              PX.Data.BQL.BqlString.Constant<readyForAssignment>
            {
                public readyForAssignment() : base(ReadyForAssignment) { }
            }

            public class pendingPayment :
              PX.Data.BQL.BqlString.Constant<pendingPayment>
            {
                public pendingPayment() : base(PendingPayment) { }
            }

            public class assigned : PX.Data.BQL.BqlString.Constant<assigned>
            {
                public assigned() : base(Assigned) { }
            }

            public class completed : PX.Data.BQL.BqlString.Constant<completed>
```

```
{
    public completed() : base(Completed) { }
}

public class paid : PX.Data.BQL.BqlString.Constant<paid>
{
    public paid() : base(Paid) { }
}
```

5. Save your changes.

Step 1.1.3: Override the Screen Configuration Method

To implement a workflow, you will override the `Configure` method, which accepts the `PXScreenConfiguration` instance. Inside the `Configure` method, you will get the configuration context object, which you will later use to add the new screen configuration.

Do the following:

1. In the `RSSVWorkOrderWorkflow` class, override the `Configure` method, as the following code shows.

```
public override void Configure(PXScreenConfiguration config)
{
}
```

2. In the method, get the configuration context object, as the following code shows.

```
var context = config.GetScreenConfigurationContext<RSSVWorkOrderEntry,
    RSSVWorkOrder>();
```

   In the code above, you have gotten the current context of the screen configuration for the Repair Work Orders (RS301000) form by specifying the form graph and primary DAC as parameters of the `GetScreenConfigurationContext` method.

3. Add a template for defining the default workflow, as the following code shows.

```
context.AddScreenConfigurationFor(screen =>
    screen
    .StateIdentifierIs<RSSVWorkOrder.status>()
        .AddDefaultFlow(flow => ...));
```

   In the code above, you have added a new screen configuration for the Repair Work Orders form, specified the state identifier field, and started to add the default workflow.

   Because a workflow consists of states, you must specify a field that holds the state value (the *state identifier field*) for the workflow definition. You do this by calling the `StateIdentifierIs` method and providing the field name as a generic parameter.

   In the next lessons,, you will define a function inside the `AddDefaultFlow` method that specifies the settings of the default workflow.

4. Save your changes.

Now you are ready to add the new screen configuration —that is, the definition of the workflow.

Step 1.1.4: Enable Workflow Validation

On a form with an enabled workflow, to get detailed errors if a problem occurs with a workflow, you need to turn on the workflow validation as follows:

1. In the instance folder, open the `web.config` file.
2. In the `appSettings` tag of the file, find the `EnableWorkflowValidationOnStartup` key, and set its value to *True*, as the following code shows.

```
<add key="EnableWorkflowValidationOnStartup" value="True" />
```

Lesson Summary

In this lesson, you have prepared the base components of the workflow. You have learned how to create a workflow class, add a screen configuration, and define the set of states for the workflow.

Lesson 1.2: Implement a Simple Transition

According to the customization description, a user should be able to remove a repair work order from hold when the order has the *On Hold* status. The user will be able to do this by clicking the **Remove Hold** button on the form toolbar or in the More menu. The status of the repair work order should be changed to *Ready For Assignment* or *Pending Payment*, depending on whether the repair work order requires a prepayment.

In this lesson, you will learn how to implement a simple transition from the *Remove Hold* status to the *Ready for Assignment* status. This transition is performed without considering the prepayment requirements. For this transition, in the workflow, you will define the `onHold` and `readyForAssingment` states and the `ReleaseFromHold` action. You will implement the transition to the *Pending Payment* status in *Lesson 1.3: Implement a Group of Transitions*.

Learning Objectives

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

- Define and configure workflow actions
- Define a default category for workflow actions
- Define and configure workflow states
- Define a simple transition

About Transitions

In a workflow, all states are linked with transitions. Transitions define the changes of the lifecycle of a document. To implement a transition, you define the following elements of the screen configuration:

- The initial state of the transition
- The target state of the transition
- The action or event that triggers the transition

- The conditions that are checked before the transition is applied
- The transition itself

Also in the transition, you can specify field assignments that are executed if the current transition matches all criteria and is applied to the current document.

Defining the Transition from the On Hold to Ready for Assignment Status

To implement a transition from the *On Hold* status to the *Ready for Assignment* status, you need to define the following elements in the screen configuration:

- The `OnHold` state, which is the initial state of the transition

  The `OnHold` state is also the initial state of the whole workflow. You will define it as well.

- The `ReadyForAssignment` state, which is the target state of the transition
- The **Remove Hold** action, which is the element that triggers the transition
- The transition

The transition you implement in this lesson does not require a condition. You will define a condition for this transition in *Lesson 1.3: Implement a Group of Transitions*.

About Actions

For more information about actions, see *Configuration of Actions* and the *T230 Actions* training course.

Step 1.2.1: Implement the Remove Hold Action in the Graph

In this step, you will implement the `ReleaseFromHold` action to the `RSSVWorkOrderEntry` graph.

To add a workflow action, you should perform the following tasks:

1. Add the definition of the workflow action in a graph.

   You will add this definition in the current step.

2. Register this action configuration in the context. You do this by calling the `WithActions` method in the `AddScreenConfigurationFor` method.

   You will perform this item in *Step 1.2.2: Add the Remove Hold Action to the Workflow*.

Adding the ReleaseFromHold Action

To add the `ReleaseFromHold` action to the `RSSVWorkOrderEntry` graph, do the following:

1. In the `RSSVWorkOrderEntry` graph, add the `Actions` region.
2. In the `Actions` region, add the following code.

```
#region Actions
public PXAction<RSSVWorkOrder> ReleaseFromHold;
[PXButton(), PXUIField(DisplayName = "Remove Hold",
  MapEnableRights = PXCacheRights.Select,
  MapViewRights = PXCacheRights.Select)]
```

```
        protected virtual IEnumerable releaseFromHold(PXAdapter adapter)
           => adapter.Get();
        #endregion
```

In the code above, you have added a field of the `PXAction<>` type and the method that implements the action. The method is empty because the business logic behind the action will be described in the code of the workflow.

The action method is decorated with the `PXButton` attribute, where you specify the display name of the action and the cache access rights. For details on the `PXButton` attribute, see *PXButtonAttribute Class*.

3. Save your changes.

Step 1.2.2: Add the Remove Hold Action to the Workflow

In this step, you will add the **Remove Hold** action to the workflow and add a category for the associated **Remove Hold** command on the More menu.

Defining an Action in a Workflow

You add an action to the workflow by registering the action in the screen configuration as follows:

1. You call the `WithActions` method in the lambda expression passed to the `AddScreenConfigurationFor` method.
2. In the lambda expression for the `WithActions` method, you call the `Add` method as follows:
   - In the method, you specify the action member of the `PXAction<>` type that you defined in the graph.
   - Also you can specify in which category of the More menu the action should be displayed by calling the `WithCategory` method in a lambda expression.

The following code shows an example of an action definition.

```
context.AddScreenConfigurationFor(screen => screen
  ...
  .WithActions(actions =>
    actions.Add(g => g.PutOnHold, c => c
      .WithCategory(ProcessingCategory));
```

Defining a Category for the Action

Acumatica ERP provides a list of default categories for the More menu. You can add these categories to a custom workflow and add actions (which are associated with commands on the More menu) to these categories. The commands associated with actions related to changing the status of a document are usually displayed in the **Processing** category. To add the **Processing** category to your workflow, do the following:

1. In the `RSSVWorkOrderWorkflow` class, in the `Configure` method, define the **Processing** category as the following code shows.

```
            #region Categories
            var commonCategories = CommonActionCategories.Get(context);
            var processingCategory = commonCategories.Processing;
            #endregion
```

In the code above, you have obtained the list of default categories by calling the `CommonActionCategories.Get` method.

2. In the `Configure` method, locate the `AddDefaultFlow` method, which you have added in *Step 1.1.3: Override the Screen Configuration Method*.
3. For the screen parameter, call the `WithCategories` method after the `AddDefaultFlow` method as the following code shows.

```
context.AddScreenConfigurationFor(screen => screen
    .StateIdentifierIs<RSSVWorkOrder.status>()
        .AddDefaultFlow(flow => ...)
        .WithCategories(categories =>
        {
            categories.Add(processingCategory);
        })
    );
```

In the code above, in the lambda expression for the `WithCategories` method, youhave added the **Processing** category to the screen configuration by calling the `Add` method.

Defining the Remove Hold Action in the Workflow

To add the `ReleaseFromHold` action to the workflow, do the following:

1. In the `RSSVWorkOrderWorkflow` class, in the `Configure` method, locate the `AddDefaultFlow` method. (You have added this method in *Step 1.1.3: Override the Screen Configuration Method*.)
2. For the screen parameter, call the `WithActions` method after the `WithCategories` method, as the following code shows.

```
        .WithCategories(categories =>
        {
            categories.Add(processingCategory);
        })
        .WithActions(actions =>
        {                    });
```

3. In the lambda expression of the `WithActions` method, add the `ReleaseFromHold` action by calling the `Add` method, as the following code shows.

```
            actions.Add(g => g.ReleaseFromHold, c => c
                    .WithCategory(processingCategory));
```

In the code above, you have added to the screen configuration for the `ReleaseFromHold` action, which you have added in the `RSSVWorkOrder` graph in *Step 1.2.1: Implement the Remove Hold Action in the Graph*. By calling the `WithCategory` method, you have specified the category in which the command associated with the action is displayed on the More menu.

4. Save your changes.

Step 1.2.3: Add States to the Workflow

In this step, you will add the `OnHold` and `ReadyForAssignment` states to the workflow and configure the states.

Defining a State

Each workflow consists of one workflow state or multiple workflow states. Each state corresponds to a value of the state identifier field. Currently, only state identifier fields of the `String` type that have a predefined set of allowed values are supported. The values of the state identifier field are usually defined in the `PXStringList` attribute. To define the state identifier field, you need to call the `StateIdentifier` method for the new screen configuration and provide the field as the parameter, as you have done in *Step 1.1.3: Override the Screen Configuration Method*.

To define the set of states, you call the `WithFlowStates` method in the lambda expression specified for the `AddDefaultFlow` method and provide the set of states in the parameter as the following code shows. You define each state by calling the `Add` method and specifying the state identifier string as a generic parameter. As a parameter of the `Add` method, you specify a lambda expression where you can configure the state.

```
context.AddScreenConfigurationFor(screen =>
   screen.StateIdentifierIs<status>()
   .AddDefaultFlow(flow =>
      flow.WithFlowStates(fss =>
      {
         fss.Add<State.hold>(flowState => flowState.IsInitial());
         fss.Add<State.open>(flowState => { … });
         ...
      })
```

Each workflow must have one state, that is marked as the initial state. It provides the default value of state identifier field for the newly created document. You mark a state as initial by calling the `IsInitial` method, as shown in the code above.

Configuring States

Inside the lambda expression passed to the `Add` method, you configure the state. For each workflow state, you can do the following:

- Specify the properties that the specified DAC fields should have in this state. You do this by calling the `WithFieldStates` method in the lambda expression provided for the `Add` method.
- Specify which actions are available in this state and configure the appearance of these actions in this state, such as whether the action should be duplicated on a form toolbar. You do this by calling the `WithActions` method in the lambda expression provided for the `Add` method.
- Specify which event handlers are available in this state. You do this by calling the `WithEventHandlers` method in the lambda expression provided for the `Add` method.

Defining States in the Repair Work Orders Workflow

To add the `OnHold` and `ReadyForAssignment` states, do the following:

1. In the `RSSVWorkOrderWorkflow` class, in the `Configure` method, locate the `AddDefaultFlow` method. (You have added this method in *Step 1.1.3: Override the Screen Configuration Method*.)
2. Call the `WithFlowStates` method, as the following code shows.

```
                    .AddDefaultFlow(flow => flow
                        .WithFlowStates(fss =>
                        {                          }))
```

3. Inside the lambda expression of the `WithFlowStates` method, add the `OnHold` state by calling the `Add` method, as the following code shows.

```
                                fss.Add<States.onHold>(flowState =>
                                {
                                    return flowState
                                      .IsInitial()
                                      .WithActions(actions =>
                                      {
                                          actions.Add(g => g.ReleaseFromHold, a => a
                                          .IsDuplicatedInToolbar()
                                          .WithConnotation(ActionConnotation.Success));
                                      });
                                });
```

In the code above, you have added the `OnHold` state by specifying the `States.onHold` value as the generic parameter of the `Add` method. In the lambda expression passed to the `Add` method, you have done the following:

- Specified that the `OnHold` state is the initial state of the workflow by calling the `IsInitial` method.
- Specified the actions that are available in the `OnHold` state by calling the `WithActions` method and adding the action definition in the lambda expression. In this case, the `ReleaseFromHold` action is available in the `OnHold` state. By calling the `IsDuplicatedInToolbar` method, you have specified that the action should be shown as a button on the toolbar for this state.
- Specified the *Success* connotation for the action by calling the `WithConnotation` method. The button corresponding to this action will be highlighted in green when it is displayed on the form toolbar. For details, see *To Add a Connotation*.

4. After the `OnHold` state, define the `ReadyForAssignment` state, as the following code shows.

```
                                fss.Add<States.readyForAssignment>(flowState =>
                                {
                                    return flowState
                                      .WithFieldStates(states =>
                                      {
                                          states.AddField<RSSVWorkOrder.customerID>(state
=> state.IsDisabled());
                                          states.AddField<RSSVWorkOrder.serviceID>(state
=> state.IsDisabled());
                                          states.AddField<RSSVWorkOrder.deviceID>(state =>
state.IsDisabled());
                                      }); ;
                                });
```

In the code above, you have added the `ReadyForAssignment` state by specifying the `States.readyForAssignment` value as the generic parameter of the `Add` method. By calling the `WithFieldStates` method, you have specified the states of DAC fields in this states. In this case, you have specified that the `customerID`, `serviceID`, and `deviceID` fields of the `RSSVWorkOrder` DAC should be disabled in this state.

5. Save your changes.

The complete code added in this step should look as follows.

```
                    .AddDefaultFlow(flow => flow
                        .WithFlowStates(fss =>
                        {
                            fss.Add<States.onHold>(flowState =>
                            {
                                return flowState
                                  .IsInitial()
```

```
                                .WithActions(actions =>
                                {
                                    actions.Add(g => g.ReleaseFromHold, a => a
                                    .IsDuplicatedInToolbar()
                                    .WithConnotation(ActionConnotation.Success));
                                });
                            });
                            fss.Add<States.readyForAssignment>(flowState =>
                            {
                                return flowState
                                .WithFieldStates(states =>
                                {
                                    states.AddField<RSSVWorkOrder.customerID>(state =>
state.IsDisabled());
                                    states.AddField<RSSVWorkOrder.serviceID>(state =>
state.IsDisabled());
                                    states.AddField<RSSVWorkOrder.deviceID>(state =>
state.IsDisabled());
                                }); ;
                            });
                        }))
```

Step 1.2.4: Add the Transition to the Workflow

In this step, you will define the transition from the `OnHold` state to the `ReadyForAssignment` state in the workflow.

Defining a Transition

After you have defined all the entities required for a transition (the initial state, the target state, and the entity that triggers the transition), you need to add the transition definition to the workflow. You do this by calling the `WithTransitions` method in the lambda expression specified for the `AddDefaultFlow` method and providing the set of transitions in the parameter. You define each transition by calling the `Add` method. In the `Add` method, you specify a lambda expression in which you specify the following transition parameters:

- `From`: The initial state of the transition
- `To`: The target state of the transition
- `IsTriggeredOn`: The trigger of the transition

You can also specify a condition on which a transition is performed. You will do this in *Lesson 1.3: Implement a Group of Transitions*.

Defining a Transition in the Repair Work Orders Workflow

To define a transition from the `OnHold` state to the `ReadyForAssignment` state in the workflow, do the following:

1. In the `RSSVWorkOrderWorkflow` class, in the `Configure` method, locate the `AddDefaultFlow` method, which you have added in *Step 1.1.3: Override the Screen Configuration Method*.
2. Call the `WithTransitions` method, as the following code shows.

```
                    .AddDefaultFlow(flow => flow
                        .WithFlowStates(...)
                        .WithTransitions(transitions =>
```

```
                                {                                }))
```

3. Inside the lambda expression of the `WithTransitions` method, add the transition by calling the `Add` method, as the following code shows.

```
                                    transitions.Add(t => t.From<States.onHold>()
                                      .To<States.readyForAssignment>()
                                      .IsTriggeredOn(g => g.ReleaseFromHold));
```

   In the code above, you have specified the following:

   - The initial state of the workflow (which is `OnHold`) as a type parameter of the `From` method
   - The target state (which is `ReadyForAssignment`) as a type parameter of the `To` method
   - The entity that triggers the transition (which is the `ReleaseFromHold` action) in the `IsTriggeredOn` method

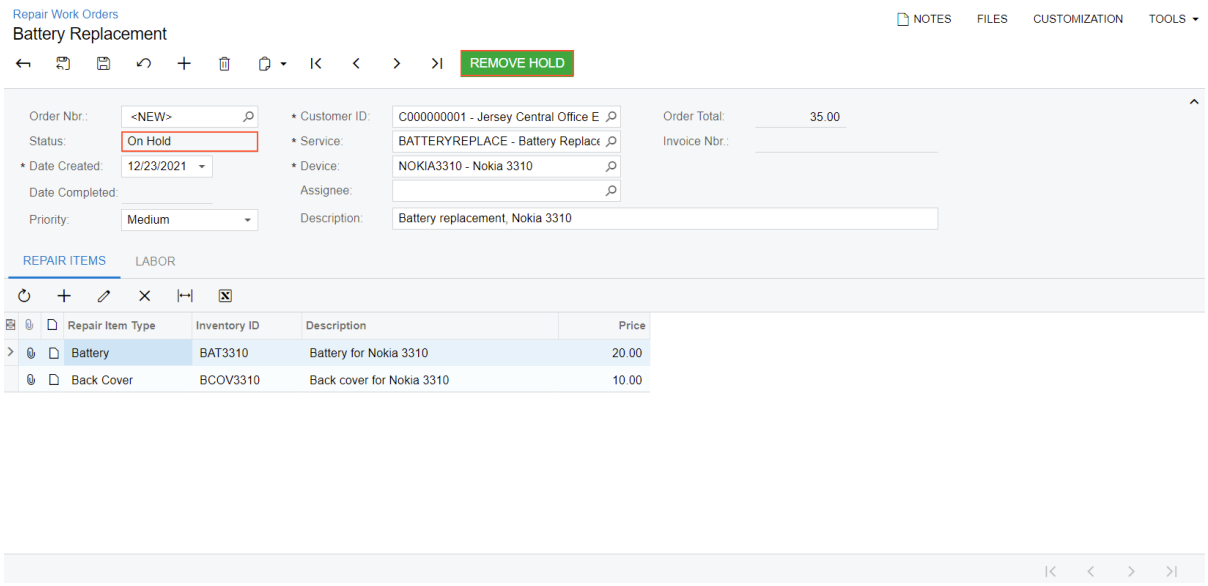
4. Save your changes.

Step 1.2.5: Test the Transition

In this step, you will test the transition from the `OnHold` state to the `ReadyForAssignment` state. Do the following:

1. Rebuild the `PhoneRepairShop_Code` project.
2. In Acumatica ERP, open the Repair Work Orders (RS301000) form.
3. On the form, specify the following settings:
   - **Customer ID**: *C000000001*
   - **Service**: *Battery Replacement*
   - **Device**: *Nokia 3310*
   - **Description**: `Battery replacement, Nokia 3310`
4. Make sure that the document has the *On Hold* status and that the **Remove Hold** button is displayed on the form toolbar and highlighted in green, as shown in the following screenshot.

Figure: A new document on the Repair Work Orders form



> The More menu is not displayed because the workflow includes only one action and this action fits the form toolbar. If the only workflow action could not be displayed on the form toolbar, than the More menu would be displayed. You will test the More menu and see the configured category in *Exercise 1.1: Implement the PutOnHold Transition*.

5. Save the repair work order.

   The repair work order is assigned the *000003* number.

6. On the form toolbar, click **Remove Hold**.

7. Notice that the document status has been changed to *Ready for Assignment* and that the **Customer ID**, **Service ID**, and **Device ID** boxes are unavailable, as shown in the following screenshot.

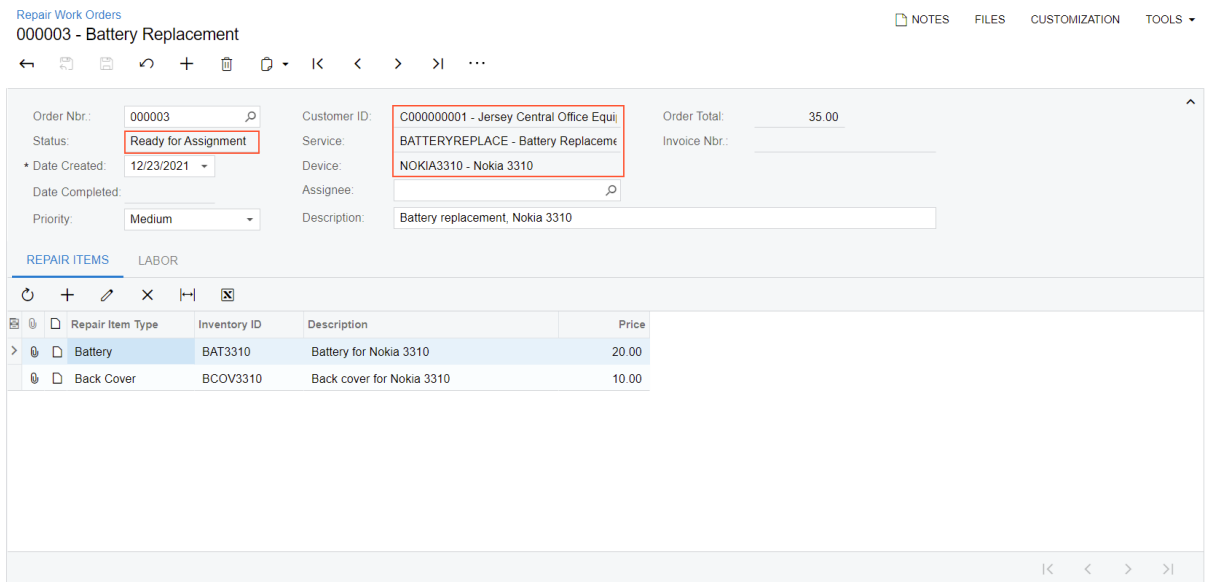


Figure: A document with the Ready for Assignment status

Lesson Summary

In this lesson, you have implemented the first transition for the workflow of the Repair Work Orders (RS301000) form. You have learned how to define workflow actions, workflow states, and a transition. You have also tested the transition.

Lesson 1.3: Implement a Group of Transitions

According to the application requirements, a repair work order will be able to have one of two statuses after the *On Hold* status, depending on whether the order requires prepayment. If a repair work order requires prepayment, it should be assigned the *Pending Prepayment* status when the **Remove Hold** button or command is clicked. If a repair work order does not require prepayment, it should be assigned the *Ready for Assignment* status when the **Remove Hold** button or command is clicked.

In the previous lesson, you have implemented an unconditional transition from the *On Hold* status to the *Ready for Assignment* status.

In this lesson, you will add a transition from the *On Hold* status to the *Pending Payment* status and modify the transition from the *On Hold* status to the *Ready for Assignment* status. You will also define the conditions on which these transitions are performed.

Learning Objectives

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

- Define a condition in a workflow
- Specify a condition for a transition
- Unite transitions into a group

About Conditions

A *condition* is a Boolean statement that is checked in a workflow. When the statement is true, a workflow element can change accordingly. You can use conditions in the following ways:

- To change Boolean properties of an action such as visibility and availability
- To change Boolean properties of fields such as visibility, requirement, and availability
- To check whether a transition should be performed

You declare a condition in a condition pack, that is, a class that inherits from the `Condition.Pack` class. In the class, you define a member of the `Condition` type by calling the `GetOrCreate` method. As a parameter, you provide a fluent BQL statement that uses the `FromBql` or `FromBqlType` method. An example of a condition definition is shown in the following code.

```
public class Conditions : Condition.Pack
{
  public Condition IsOnHold =>
    GetOrCreate(b => b.FromBql<hold.IsEqual<True>>());
  public Condition IsCompleted =>
    GetOrCreate(b => b.FromBql<completed.IsEqual<True>>());
  public Condition IsOnCreditHold =>
    GetOrCreate(b => b.FromBql<creditHold.IsEqual<True>>());
```

```
}
```

The `Condition.Pack` class provides the `GetOrCreate` method, which registers a condition definition under the name of the class property. Later, you can retrieve the condition through this property by using the `GetPack` method.

Conditions can be combined with AND, OR, and NOT C# operators to dynamically create and register new conditions.

Step 1.3.1: Add a Condition Pack

To implement a transition from the `OnHold` state to the `ReadyForAssignment` or `PendingPayment` state, based on the properties of the current repair work order, you need to define two conditions: one that is true when a repair work order requires prepayment, and another that is true when the document does not require prepayment. You will declare these conditions in this step. Each condition will be checked in the corresponding transition.

A repair work order requires prepayment when the service specified for the order requires prepayment. The service requires prepayment if the **Prepayment** check box is selected for the service on the Repair Services (RS201000) form. This means that to check whether a repair work order requires prepayment, you need to check the value of the `prepayment` property of the `RSSVRepairService` DAC record, which has the same `serviceID` value as the repair work order.

To declare the conditions, do the following:

1. In the `RSSVWorkOrderWorkflow` class, add the `Conditions` region.
2. In the `Conditions` region, declare a condition pack, as the following code shows.

```
public class Conditions : Condition.Pack
{        }
```

3. In the `Conditions` class, declare a condition that is true when the repair work order requires prepayment, as the following code shows.

```
public Condition RequiresPrepayment => GetOrCreate(b => b.FromBql<
  Where<Selector<RSSVWorkOrder.serviceID, RSSVRepairService.prepayment>,
  Equal<True>>>());
```

   In the code above, you have created a condition by calling the `GetOrCreate` method and providing a BQL statement in the lambda expression. In the BQL statement, you have selected the `RSSVRepairService` record by using the `RSSVWorkOrder.serviceID` value and checked whether the `RSSVRepairService.prepayment` value equals `true`.

4. After the `RequiresPrepayment` condition, declare a condition that is true when the repair work order does not require prepayment, as the following code shows.

```
public Condition DoesNotRequirePrepayment => GetOrCreate(b => b.FromBql<
  Where<Selector<RSSVWorkOrder.serviceID, RSSVRepairService.prepayment>,
  Equal<False>>>());
```

   In the code above, you have created the same condition as the previous one, except that you have checked whether the `RSSVRepairService.prepayment` value equals `false`.

5. In the `Configure` method, create an instance of the `Conditions` class, as the following code shows.

```
public override void Configure(PXScreenConfiguration config)
{
    var context = config.GetScreenConfigurationContext<RSSVWorkOrderEntry,
        RSSVWorkOrder>();
```

```
            var conditions = context.Conditions.GetPack<Conditions>();
            ...
        }
```

6. Save your changes.

Step 1.3.2: Add the Pending Payment State (Self-Guided Exercise)

To define a transition from the `OnHold` to `PendingPayment` state, you first need to define the `PendingPayment` state. In this step, you will define the `PendingPayment` state of the workflow as a self-guided exercise. You have learned how to add a state in *Step 1.2.3: Add States to the Workflow*.

While defining a state, you need to use the `States.pendingPayment` class, which you added in *Step 1.1.2: Define the Set of States of the Workflow*. In the `PendingPayment` state, make the **Customer ID**, **Service ID**, and **Device ID** boxes of the Repair Work Orders (RS301000) form disabled for the state.

Step 1.3.3: Group Transitions and Add Conditions to Transitions

In this step, you will group transitions and add a condition to each transition.

Grouping Transitions

You can group transitions by their initial states. In this case, you can structure your code and do not have to specify the initial state of the transition of the `From` parameter. You can group the transitions by calling the `AddGroupFrom` method in the lambda expression of the `WithTransitions` method and adding the transitions in the same lambda expression, as you did in *Step 1.2.4: Add the Transition to the Workflow*. An example is shown in the following code.

```
.WithTransitions(transitions =>
{
  transitions.AddGroupFrom<State.open>(ts => {
    ts.Add(t => t.To<State.hold>().IsTriggeredOn(g => g.putOnHold)
      .WithFieldAssignments(fas => fas.Add<hold>(true)));
    ts.Add(t => t.To<State.confirmed>().IsTriggeredOn(g => g.confirmShipmentAction));
  });
})
```

Specifying a Condition for a Transition

You specify a condition for a transition in the lambda expression provided for the `Add` method that adds the transition to the screen configuration by calling the `When` method and providing the condition name. The transition is performed when the provided condition is true. An example is shown in the following code.

```
.WithTransitions(transitions =>
{
  transitions.AddGroupFrom(initialState, ts =>
  {
    ...
    ts.Add(t => t
    .To<State.balanced>()
    .IsTriggeredOn(g => g.initializeState)
    .When(conditions.IsBalanced)
```

```
    }
}
```

Specifying Conditions and Grouping Transitions in the Repair Work Orders Workflow

To specify the conditions you defined in *Step 1.3.1: Add a Condition Pack* , do the following:

1. In the lambda expression provided for the `WithTransitions` method, define a group of transitions, as the following code shows.

```
                        .WithTransitions(transitions =>
                        {
                            transitions.AddGroupFrom<States.onHold>(ts =>
                            {
                            });
                        })
```

   In the code above, you have defined a group of transitions whose initial state is `OnHold`.

2. Move the definition of the transition that you added in *Step 1.2.4: Add the Transition to the Workflow* inside the group and remove the call of the `From` method because it is no longer necessary. The resulting code is shown below.

```
                            transitions.AddGroupFrom<States.onHold>(ts =>
                            {
                                ts.Add(t => t.To<States.readyForAssignment>()
                                    .IsTriggeredOn(g => g.ReleaseFromHold)
                    );
                            });
```

3. After the `IsTriggeredOn` method call, add the `When` method call, and specify the `DoesNotRequirePrepayment` condition, as the following code shows.

```
                                ts.Add(t => t.To<States.readyForAssignment>()
                                    .IsTriggeredOn(g => g.ReleaseFromHold)
                                    .When(conditions.DoesNotRequirePrepayment)
                                    );
```

4. After this transition, add the new transition from the `OnHold` state to the `PendingPayment` state, as the following code shows. As the condition, specify the `RequiresPrepayment` condition.

```
                                ts.Add(t => t.To<States.pendingPayment>()
                                    .IsTriggeredOn(g => g.ReleaseFromHold)
                                    .When(conditions.RequiresPrepayment));
```

5. Save your changes.

The full code of the section should look as follows.

```
                    .WithTransitions(transitions =>
                    {
                        transitions.AddGroupFrom<States.onHold>(ts =>
                        {
                            ts.Add(t => t.To<States.readyForAssignment>()
                                .IsTriggeredOn(g => g.ReleaseFromHold)
                                .When(conditions.DoesNotRequirePrepayment)
                                );
                            ts.Add(t => t.To<States.pendingPayment>()
```

```
                                        .IsTriggeredOn(g => g.ReleaseFromHold)
                                        .When(conditions.RequiresPrepayment));
                                });
                            })
```

Step 1.3.4: Test Transitions with Conditions

In this step, you will test transitions that depend on a condition. Do the following:

1. Rebuild the `PhoneRepairShop_Code` project.
2. In Acumatica ERP, open the Repair Work Orders (RS301000) form.
3. On the form, specify the following values:
   - **Customer ID**: *C000000001*
   - **Service**: *Liquid Damage*
   - **Device**: *Nokia 3310*
   - **Description**: `Liquid Damage, Nokia 3310`
4. On the form toolbar, click **Remove Hold**.
5. Notice that the repair work order status has been changed to *Pending Payment* (in contrast to what has been tested in *Step 1.2.5: Test the Transition*), because the service specified for the order requires prepayment. You can make sure that the service requires prepayment on the Repair Services (RS201000) form.

Lesson Summary

In this lesson, you have implemented a transition from the `OnHold` state to the `PendingPayment` state and added conditions for previously implemented transitions. You have learned how to put transitions into groups and declare condition packs.

Exercise 1.1: Implement the PutOnHold Transition

According to the customization description, a user should have the ability to put a repair work order on hold when it has the *Ready For Assignment* or *Pending Payment* status by clicking the **Hold** button on the form toolbar.

In this exercise, you will implement the following transitions, which are performed when the **Hold** button is clicked:

- The transition from the `ReadyForAssignment` state to the `OnHold` state
- The transition from the `PendingPayment` state to the `OnHold` state

Both transitions are performed without any conditions.

To implement these transitions, you need to do the following:

1. Declare the `PutOnHold` action (which is associated with the **Hold** button)in the `RepairWorkOrderEntry` graph.
2. Register the `PutOnHold` action in the screen configuration.
3. Register the `PutOnHold` action in the `ReadyForAssignment` and `PendingPayment` states.
4. Add a transition group for the `ReadyForAssignment` state, and add a transition to it.
5. Add a transition group for the `PendingPayment` state, and add a transition to it.

You do not need to add any new states to the workflow because only states that are already declared in the screen configuration are used in the new transitions.

Declaring the PutOnHold Action

To define the `PutOnHold` action in the `RepairWorkOrderEntry` graph, add the following code to the graph code.

```
public PXAction<RSSVWorkOrder> PutOnHold;
[PXButton, PXUIField(DisplayName = "Hold",
  MapEnableRights = PXCacheRights.Select,
  MapViewRights = PXCacheRights.Select)]
protected virtual IEnumerable putOnHold(PXAdapter adapter) => adapter.Get();
```

To register the action in the screen configuration, add the following code in the lambda expression provided for the `WithActions` method in the `RSSVWorkOrderWorkflow` class.

```
        actions.Add(g => g.PutOnHold, c => c
                .WithCategory(processingCategory));
```

Registering the PutOnHold Action in the States

To register the `PutOnHold` action in the `ReadyForAssignment` and `PendingPayment` states, do the following:

1. Locate the `Add` method for the `ReadyForAssignment` state. Call the `WithActions` method, and add the `PutOnHold` action, as the following code shows.

```
            fss.Add<States.readyForAssignment>(flowState =>
            {
                return flowState
                    .WithFieldStates(states =>
                    {
                        ...
                    })
                    .WithActions(actions =>
                    {
                        actions.Add(g => g.PutOnHold, a => a.IsDuplicatedInToolbar());
                    });
```

2. Locate the `Add` method for the `PendingPayment` state. Call the `WithActions` method, and add the `PutOnHold` action, as the following code shows.

```
            fss.Add<States.pendingPayment>(flowState =>
            {
                return flowState
                    .WithFieldStates(states =>
                    {
                        ...
                    })
                    .WithActions(actions =>
                    {
                        actions.Add(g => g.PutOnHold, a => a.IsDuplicatedInToolbar());
                    });
            });
```

Defining Transitions

To define the needed transitions, do the following in the `RepairWorkOrdersWorkflow` class:

- To define the transition from the `ReadyForAssignment` state to the `OnHold` state, add the following code in the lambda expression provided for the `WithTransitions` method.

```
transitions.AddGroupFrom<States.readyForAssignment>(ts =>
{
    ts.Add(t => t.To<States.onHold>().IsTriggeredOn(g => g.PutOnHold));
});
```

- To define the transition from the `PendingPayment` state to the `OnHold` state, add the following code after the code added in the previous instruction.

```
transitions.AddGroupFrom<States.pendingPayment>(ts =>
{
    ts.Add(t => t.To<States.onHold>().IsTriggeredOn(g => g.PutOnHold));
});
```

Testing the Transitions

To test the transitions you have implemented, do the following:

1. Rebuild the `PhoneRepairShop` project.
2. In Acumatica ERP, open the *000003* repair work order, which you created in *Step 1.2.5: Test the Transition* on the Repair Work Orders (RS301000) form. The document has the *Ready for Assignment* status.
3. Notice that the **Hold** button is displayed on the form toolbar. The **Remove Hold** and **Hold** commands are displayed on the More menu, as shown in the following screenshot.

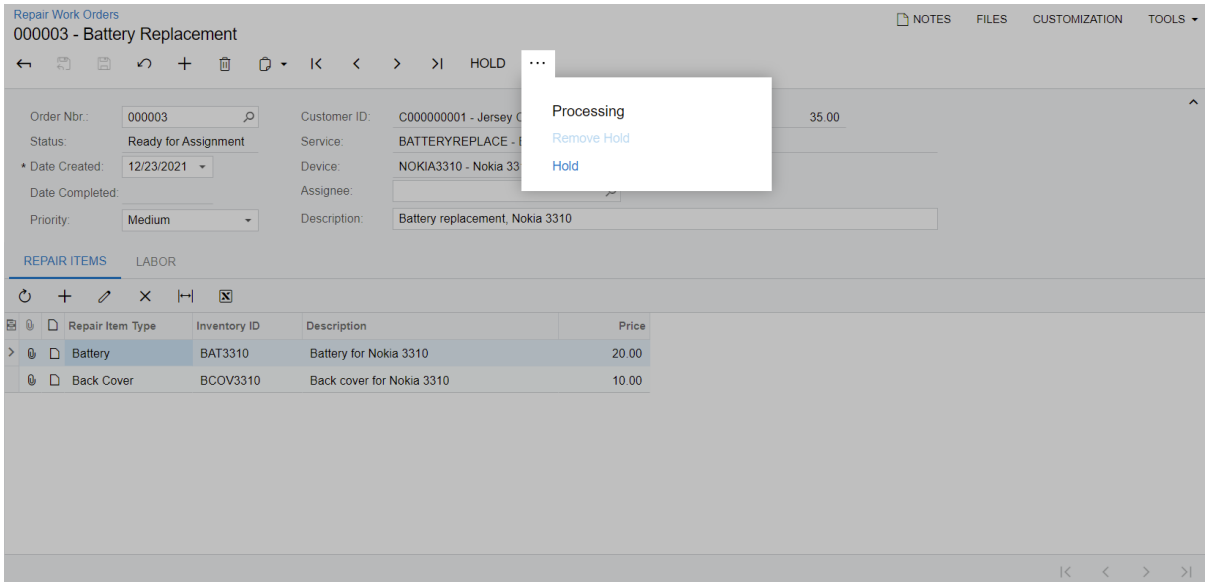


Figure: The More menu of the Repair Work Orders form

4. Click the **Hold** button.
5. Notice that the status of the document has changed to *On Hold*.

Lesson 1.4: Implement a Transition with a Dialog Box

According to the application requirements, a user should be able to assign a repair work order when it has the *Ready For Assignment* status. To do this, the user will click the **Assign** button on the form toolbar (or in the More menu) and enters the name of the assignee in the dialog box. When the user clicks **OK**, the repair work order will get the *Assigned* status.

In this lesson, you will learn how to define a dialog box in a workflow. You will define the `Assign` action, register it in a workflow, and specify a dialog box for it. You will also define the `Assigned` state and the transition from the `ReadyForAssignment` state to the `Assigned` state.

Learning Objectives

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

- Define a dialog box in a workflow
- Specify a dialog box for an action

About Dialog Boxes

A *dialog box* is a screen configuration block that can be displayed to a user to provide particular field values when a user clicks a button that corresponds to an action. A dialog box is referred to in code as a `form`.

Defining a Dialog Box

A dialog box can contain the following types of fields:

- A custom check box field, which is defined with the `WithCheckBoxField` method. An example is shown in the following code.

```
fields.Add("InspectionPassed", field => field.WithCheckBoxField());
```

- A custom combo box field, which is defined with the `WithComboBoxField` method. The list of combo box values is defined with the `ComboBoxValues` method. An example is shown in the following code.

```
fields.Add("InspectionType", field =>
  field.WithComboBoxField().ComboBoxValues(("M", "Manual"),("A", "Auto")));
```

- A field that copies its state from the specified DAC field. It is defined with the `WithSchemaOf` method. An example is shown in the following code.

```
fields.Add("Reason", field => field.WithSchemaOf<RequestForInformation.reason>();
```

To define a dialog box, you should do the following:

1. You define the dialog box in the `Configure` method by using the `context.Forms.Create` method. In the method parameters, you provide the internal name of the dialog box and the configuration of the dialog box. In the configuration, you provide a lambda expression where you specify the title of the dialog box in the `Prompt` method and the fields that should be displayed in the dialog box in the `WithFields` method. The following code shows an example of a dialog box definition.

```
public override void Configure(PXScreenConfiguration config)
{
```

```
  ...
  var formClose = context.Forms.Create("FormClose",
    form => form.Prompt("Select Reason").WithFields(fields => {
      fields.Add("Reason", field => { … });
  }));
}
```

In the code above, the `FormClose` dialog box with the **Select Reason** title is declared.

2. In the `WithFields` method, you add fields by using the `Add` method, as described above.
3. You register the dialog box in the screen configuration by specifying the dialog box name in the `WithForms` method. An example is shown in the following code.

```
.AddDefaultFlow(flow => flow
  ...
  .WithForms(forms => forms.Add(formClose)))
```

By default, a dialog box has the **OK** and **Cancel** buttons.

Linking a Dialog Box to an Action

To display a dialog box when a user clicks a button that corresponds to an action, you need to specify this dialog box in the action configuration. You do this by calling the `WithForm` method inside the lambda expression provided for the `Add` method.

To insert data specified by a user in a dialog box field to a DAC field, you need to call the `WithFieldAssignments` method. In the lambda expression provided for the method, you add the DAC field by calling the `Add` method. In the lambda expression for the `Add` method, you call the `SetFromFormField` method and provide the dialog box name and the dialog box field name as parameters.

An example of an action with a dialog box is shown in the following code.

```
actions.Add(g => g.close, c => c
  .WithForm(formClose)
  .WithFieldAssignments(fields =>
  {
    fields.Add<RequestForInformation.reason>(f =>
      f.SetFromFormField(formClose, "Reason"));
  }));
```

In the code above, the `RequestForInformation.reason` field is updated with the value specified in the `Reason` field of the `formClose` dialog box.

The original code of the action is executed only after the dialog box is closed and field assignments are applied. If an action triggers a transition, the transition is performed only if a user clicks **OK** in the dialog box.

Step 1.4.1: Define a Dialog Box

In this step, you will define and configure the **Select Assignee** dialog box.

Do the following:

1. In the `Configure` method of the `RSSVWorkOrderWorkflow` class, define the **Select Assignee** dialog box, as the following code shows.

```
public override void Configure(PXScreenConfiguration config)
{
    var context = config.GetScreenConfigurationContext<RSSVWorkOrderEntry,
 RSSVWorkOrder>();

    var formAssign = context.Forms.Create("FormAssign", form =>
        form.Prompt("Select Assignee").WithFields(fields =>
        {        }));
    ...
}
```

In the code above, you have declared the `FormAssign` dialog box with the **Select Assignee** title.

2. Inside the lambda expression for the `WithFields` method, add the `Assignee` field that is displayed in the dialog box.

```
            fields.Add("Assignee", field => field
                .WithSchemaOf<RSSVWorkOrder.assignee>()
                .IsRequired()
                .Prompt("Assignee"));
```

In the code above, you have added the `Assignee` field to the dialog box. You have copied the field configuration from the `RSSVWorkOrder.assignee` field by specifying this DAC field in the `WithSchemaOf` method. You have specified that the field is required by calling the `IsRequired` method, and you have specified the UI name of the field in the `Prompt` method.

3. Register the dialog box in the screen configuration by calling the `WithForms` method, as the following code shows.

```
    context.AddScreenConfigurationFor(screen => screen
        ...
        .WithForms(forms => forms.Add(formAssign))
        );
```

Step 1.4.2: Define the Assign Action

In this step, you will add the `Assign` action to the `RSSVWorkOrderEntry` graph and register the action in the screen configuration. Do the following:

1. In the `RSSVWorkOrderEntry` graph, define the `Assign` action, as the following code shows.

```
public PXAction<RSSVWorkOrder> Assign;
[PXButton]
[PXUIField(DisplayName = "Assign", Enabled = false)]
protected virtual IEnumerable assign(PXAdapter adapter) => adapter.Get();
```

2. In the `RSSVWorkOrderWorkflow` class, in the lambda expression for the `WithActions` method, add the `Assign` action, as the following code shows.

```
.WithActions(actions =>
{
    ...
    actions.Add(g => g.Assign, c => c
        .WithCategory(processingCategory)
        .WithForm(formAssign)
        .WithFieldAssignments(fields => {
```

```
            fields.Add<RSSVWorkOrder.assignee>(f =>
                f.SetFromFormField(formAssign, "Assignee"));
        }));
})
```

In the code above, you have added the `Assign` action to the screen configuration. You have specified the name of the dialog box for the action in the `WithForm` method and the DAC field that should be updated from the dialog box in the `WithFieldAssignments` method.

3. Save your changes.

Step 1.4.3: Define the Assigned State and the Transition (Self-Guided Exercise)

In this step, you will define the `Assigned` state and the transition from the `ReadyForAssignment` state to the `Assigned` state on your own. You have learned how to define a state in *Step 1.2.3: Add States to the Workflow*.

While defining a state, you need to use the `States.assigned` class, which you added in *Step 1.1.2: Define the Set of States of the Workflow*. In the `Assigned` state, make the **Customer ID**, **Service ID**, and **Device ID** boxes disabled for the state. Also, do not forget to add the `Assign` action to the `ReadyForAssignment` state, display it on the form toolbar, and specify the *Success* connotation for it.

You have learned how to define a transition in *Step 1.3.3: Group Transitions and Add Conditions to Transitions*. The transition should be defined inside a group and is performed without any conditions.

Step 1.4.4: Test the Assign Button

In this step, you will test the **Assign** button (and the associated action) and the **Select Assignee** dialog box. Do the following:

1. Rebuild the `PhoneRepairShop_Code` project.
2. In Acumatica ERP, open the *000003* repair work order, which you have created in *Step 1.2.5: Test the Transition*, on the Repair Work Orders (RS301000) form.

   The repair work order should have the *On Hold* status because of the testing in *Exercise 1.1: Implement the PutOnHold Transition*.
3. On the form toolbar, click **Remove Hold**.

   The status of the repair work order changes to *Ready for Assignment*. The **Assign** button is displayed on the form toolbar, as shown in the following screenshot.

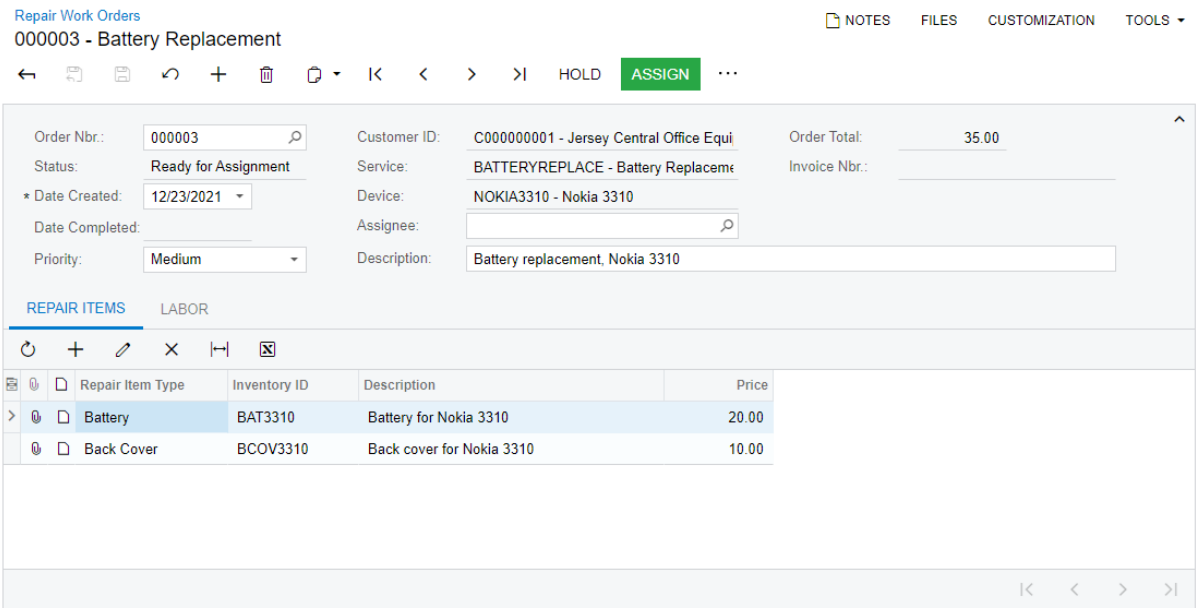


Figure: The Assign button

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

   The **Select Assignee** dialog box appears, as shown in the following screenshot.

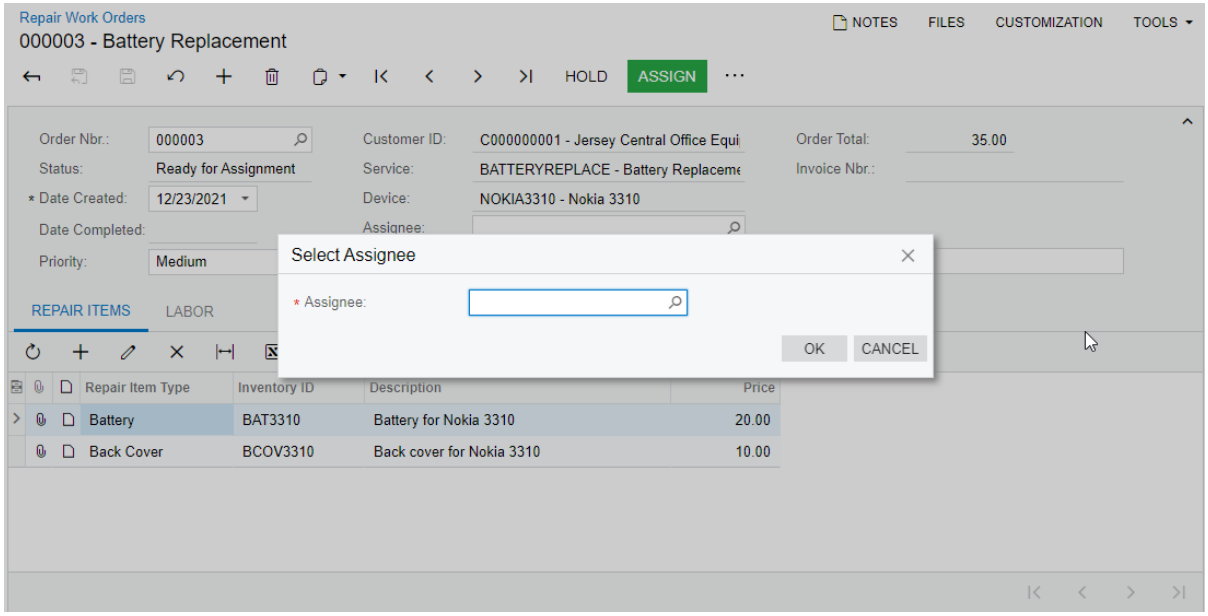


Figure: The Select Assignee dialog box

5. In the **Select Assignee** dialog box, select *Andrews, Michael*.
6. Click **OK**.
7. Notice that the document status has been changed to *Assigned* and the specified assignee is displayed in the **Assignee** box.

Lesson Summary

In this lesson, you have implemented the `Assign` action, a transition from the `ReadyForAssignment` state to the `Assigned` state, and a dialog box that is shown when a user clicks the **Assign** button on the form toolbar.

Lesson 1.5: Implement a Transition with Field Assignments

According to the application requirements, a user should be able to mark a repair work order as completed when it has the *Assigned* status. To do that, a user should be able to click the **Complete** button, which will be displayed on the table toolbar of the **Labor** tab. The action will cause the date of the completion to be inserted in the **Date Completed** box of the Repair Work Orders (RS301000) form, and the repair work order to be assigned the *Completed* status.

In this lesson, you will define the `Complete` action, the `Completed` state, and the transition from the `Assigned` to `Completed` state. You will learn how to define the assignment of the **Date Completed** box in the transition. You will also learn how to define the location of the **Complete** button, as well as the equivalent command on the More menu.

Learning Objectives

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

- Define a DAC field assignment with the workflow API methods
- Define the location of the button and command associated with a workflow action

Step 1.5.1: Add the State, the Action, and the Transition

In this step, you will define the `Completed` state, the `Complete` action, and the transition from the `Assigned` state to the `Completed` state.

Do the following:

1. In the `RSSVWorkOrderEntry` graph, declare the `Complete` action, as the following code shows.

```
public PXAction<RSSVWorkOrder> Complete;
[PXButton]
[PXUIField(DisplayName = "Complete", Enabled = false)]
protected virtual IEnumerable complete(PXAdapter adapter) => adapter.Get();
```

2. In the `RSSVWorkOrderWorkflow` class, add the `Complete` action in the lambda expression for the `WithActions` method, as the following code shows.

```
        actions.Add(g => g.Complete, c => c
            .WithCategory(processingCategory)
            .WithFieldAssignments(fas => fas.Add<RSSVWorkOrder.dateCompleted>(f =>
            f.SetFromToday())));
```

In the code above, you have defined the `Complete` action, whose associated command will be displayed in the **Processing** category of the More menu. Also, in the `WithFieldAssignments` method, by calling the `SetFromToday` method, you have specified that the `RSSVWorkOrder.dateCompleted` field is assigned the current business date.

3. To make the `Complete` action available in the `Assigned` state, add the action to the configuration of the `Assigned` state in the `WithFlowStates` method, as the following code shows.

```
fss.Add<States.assigned>(flowState =>
{
    return flowState
        ...
        .WithActions(actions =>
        {
            actions.Add(g => g.Complete, a => a
                .IsDuplicatedInToolbar()
                .WithConnotation(ActionConnotation.Success));
        });
});
```

4. In the lambda expression for the `WithFlowStates` method, add the definition of the `Completed` state, as the following code shows.

```
fss.Add<States.completed>(flowState =>
{
    return flowState
        .WithFieldStates(states =>
        {
            states.AddField<RSSVWorkOrder.customerID>(state =>
                state.IsDisabled());
            states.AddField<RSSVWorkOrder.serviceID>(state =>
                state.IsDisabled());
            states.AddField<RSSVWorkOrder.deviceID>(state =>
                state.IsDisabled());
        });
});
```

   In the state configuration, you have made the `Customer ID`, `Service ID`, and `Device ID` fields disabled.

5. In the `WithTransitions` method, define the transition from the `Assigned` to `Completed` state, as the following code shows.

```
transitions.AddGroupFrom<States.assigned>(ts =>
{
    ts.Add(t => t.To<States.completed>().IsTriggeredOn(g =>
        g.Complete));
});
```

6. Rebuild the `PhoneRepairShop_Code` project.

Step 1.5.2: Configure the Button Location

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.

In this step, to place the **Complete** button on the table toolbar of the **Labor** tab, you will 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="Labor">` tag, after the `Levels` closing tag.

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

   In this code, you have defined an action bar on the table toolbar of the **Labor** tab, and added 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 1.5.3: Test the Complete Button

In this step, you will make sure that the **Complete** button appears on the form toolbar and that the underlying action works as designed. To test the **Complete** button, do the following:

1. In Acumatica ERP, open the *000003* repair work order, which you have created in *Step 1.2.5: Test the Transition* on the Repair Work Orders (RS301000) form.

   The repair work order should have the *Assigned* status.

2. Go to the **Labor** tab. Notice that the **Complete** button is located on the table toolbar, as shown in the following screenshot.

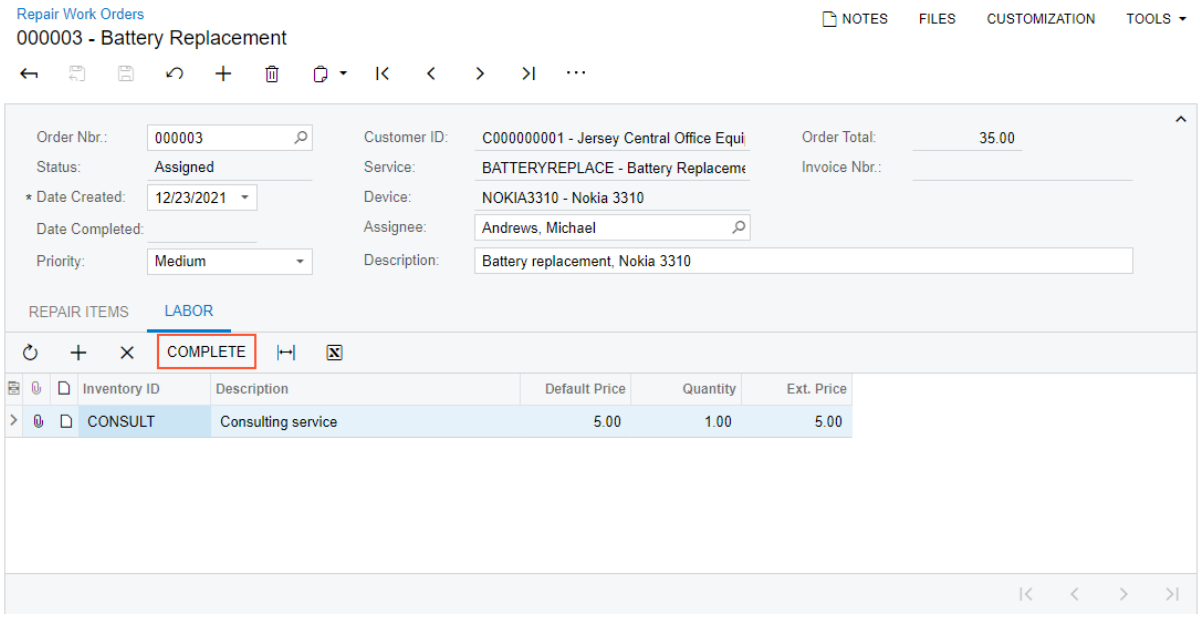


Figure: The Complete button

3. On the table toolbar, click **Complete**. Notice that the document status has been changed to *Completed*.

Lesson Summary

In this lesson, you have implemented the `Complete` action, as well as its associated button on the table toolbar and command on the More menu. You have also defined the transition from the `Assigned` state to the `Completed` state.

Exercise 1.2: Configure the Conditional Appearance of a Button and Command

For a repair work order to be billed and then paid, a user needs to create an invoice for the order. An invoice can be created for a repair work order only if it has the *Pending Payment* or *Completed* status and no invoice has been created for this repair work order. In this exercise, you will implement the **Create Invoice** action and configure the conditional availability of the associated button on the form toolbar and the equivalent command on the More menu.

Define the Create Invoice Action in the Graph

To define the **Create Invoice** action in the `RSSVWorkOrderEntry` graph, do the following:

1. In the `RSSVWorkOrderEntry` graph, add the following code:

The **Create Invoice** action performs an asynchronous operation. For details on implementing actions, see the *T230 Actions* training course.

```
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();
 }
}

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;
}
```

2. Add the following `using` directives to the `RSSVWorkOrderEntry` graph.

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

Configure the Conditional Appearance of the Button and Command

The **Create Invoice** button on the form toolbar and the equivalent command on the More menu should be available only if the repair work order has the *Pending Payment* or *Completed* status and if no invoice for the order has been created yet (that is, if the `RSSSVWorkOrder.InvoiceNbr` field is null). You can configure the availability by combining two features of the workflow API as follows:

1. Configure a condition that checks whether the `InvoiceNbr` field is not null, and disable the action when the condition is true by using the `IsDisabledWhen` method. Do the following:
   a. In the `Conditions` class, define the condition, as the following code shows.

```
public Condition DisableCreateInvoice => GetOrCreate(b => b.FromBql<
    Where<RSSVWorkOrder.invoiceNbr.IsNotNull>>());
```

   b. In the `WithActions` method of the screen configuration, register the **Create Invoice** action, as the following code shows.

```
actions.Add(g => g.CreateInvoiceAction, c => c
    .WithCategory(processingCategory)
    .IsDisabledWhen(conditions.DisableCreateInvoice));
```

   By using the `IsDisabledWhen` method, you have specified that the action is disabled when the provided condition is true.
2. Add the action to only the states where it is available (that is, the `PendingPayment` and `Completed` states), as the following code shows.

```
.WithActions(actions =>
{
    actions.Add(g => g.CreateInvoiceAction, a => a
    .IsDuplicatedInToolbar()
    .WithConnotation(ActionConnotation.Success));
});
```

Test the Create Invoice Action

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

1. Rebuild the `PhoneRepairShop_Code` project.
2. In Acumatica ERP, open the *000003* repair work order on the Repair Work Orders (RS301000) form.

   The repair work order should have the *Completed* status.
3. On the form toolbar, click the **Create Invoice** button.

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

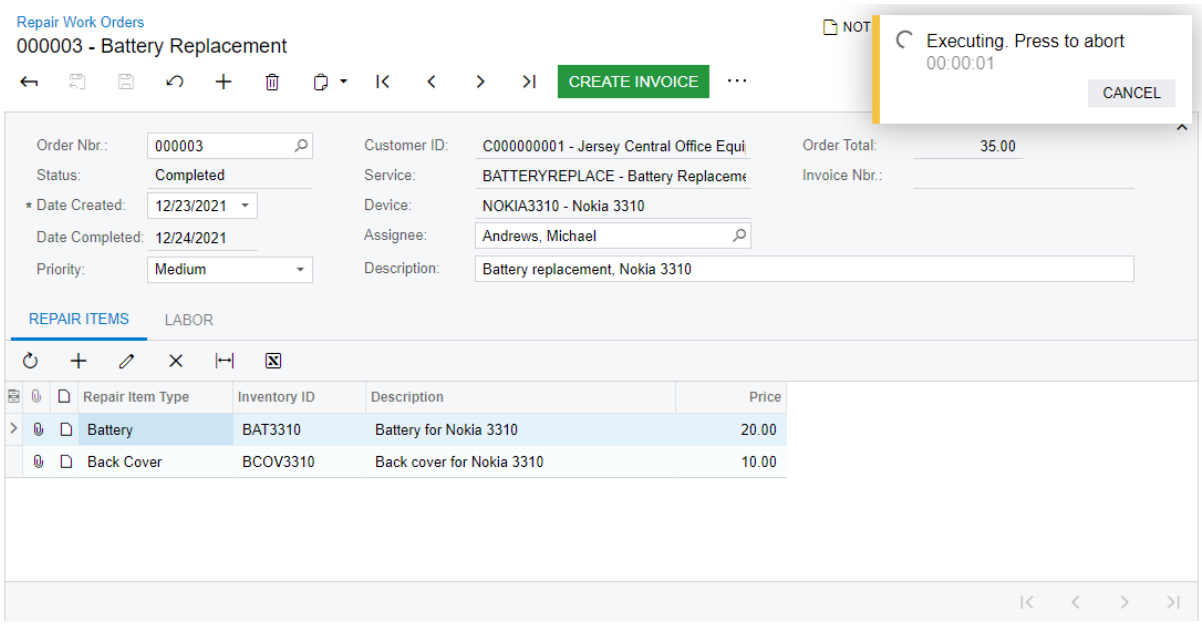


Figure: Creation of an invoice

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

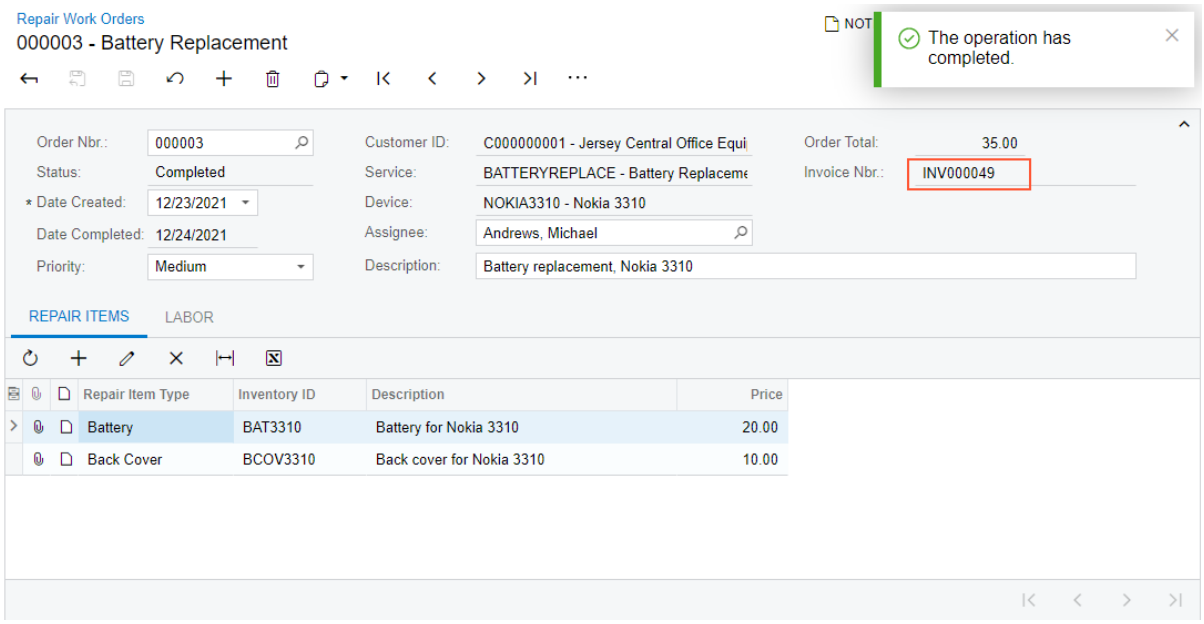


Figure: The invoice number of the repair work order

Part 1 Summary

In the part of the course, you have learned how to define a screen configuration. You have learned how to define workflow states, actions, dialog boxes, and transitions. You have also learned how to configure these parts of the screen configuration.

Part 2: Incorporating an Existing Workflow into a Custom Workflow

In Acumatica ERP, most of the predefined data entry forms have workflows defined. If the workflow of a custom form is dependent on transitions that happen on an existing Acumatica ERP form, you can implement the interaction between workflows by using events. In this part, you will learn how to use existing events and create your own events in a custom workflow.

About Workflow Events

A *workflow event* is an event that can be used to trigger a transition. Workflow events introduce cross-graph interactions and are similar to .NET events.

To define a workflow event, you need to define the following entities:

- The event itself.

  An event is an object in code that corresponds to a DAC and has a name; it can be defined anywhere. However, we strongly recommend that you place it in a nested class in a DAC the event relates to. To define a new event, you need to create a new class that inherits from the `PXEntityEvent.Container` class. As the type parameter of the `PXEntityEvent` class, you specify the name of the primary DAC of the form where the event is fired. In the class, you create the event as a field of the `PXEntityEvent` type.

  This class has one or two generic parameters. The first parameter must be the same one that you used in the class declaration. This parameter shows that this event happens on records of the specified DAC. Also, you can use the second parameter to provide additional information about the event that occurred. Usually, the second parameter holds an additional DAC record that corresponds to the event.

  An example of an event declaration is shown in the following code. Among other events, the `InvoiceLinked` event is declared. This event shows that the invoice, which is defined as the second type parameter, is linked with an `SOOrderShipment` record.

```
public partial class SOOrderShipment : IBqlTable{
  public class Events : PXEntityEvent<SOOrderShipment>.Container<Events>{
    public PXEntityEvent<SOOrderShipment, SOShipment> ShipmentLinked;
    public PXEntityEvent<SOOrderShipment, SOShipment> ShipmentUnlinked;
    public PXEntityEvent<SOOrderShipment, SOInvoice> InvoiceLinked;
    public PXEntityEvent<SOOrderShipment, SOInvoice> InvoiceUnlinked;
  }
}
```

- An event handler.

  You can define an event handler as a member of a graph or a graph extension. You do this by declaring a member of the `PXWorkflowEventHandler` type, as the following code example shows.

```
public PXWorkflowEventHandler<SOShipment, SOOrderShipment, SOInvoice> OnInvoiceLinked;
public PXWorkflowEventHandler<SOShipment, SOInvoice> OnInvoiceReleased;
```

  In the first type parameter, you specify the primary DAC to indicate that this handler is used to manipulate the records of the primary DAC. In the second type parameter, you specify the DAC where the event for this handler is defined. The third parameter is optional and is used to further restrict the handler to only those events that provide additional information or a record.

- The binding of the event handler to an event.

  To bind the event handler to an event, in the screen configuration, you call the `WithHandlers` method, and add the handler by using the `Add` method, as the following code example shows.

```
.WithHandlers(handlers =>
{
  handlers.Add(handler =>
  {
    return handler
      .WithTargetOf<SOOrder>()
      .OfEntityEvent<SOOrder.Events>(e => e.ShipmentCreationFailed) //
PXEntityEvent<SOOrder>
      .Is(g => g.OnShipmentCreationFailed) // PXWorkflowEventHandler<SOOrder, SOOrder
 >
      .UsesTargetAsPrimaryEntity()
      .DisplayName("Shipment Creation Failed");
  });
})
```

In the `OfEntityEvent` method, you select the workflow event declared in the DAC. The event must match the type specified in the `WithTargetOf` method and the type of the current screen configuration context. In the `Is` method, you specify the event handler that matches the event signature. By calling the `UsesTargetAsPrimaryEntity` method, you specify that the workflow should use the record on which the event is fired. For a cross-graph execution, you need to call the `UsesPrimaryEntityGetter` method instead and select the DAC object that should be used to get the source document for the event.

- The registration of an event handler in a workflow state where the corresponding event should be fired.

  To register an event handler in a workflow state, you call the `WithEventHandlers` method in the state definition, as the following code example shows.

```
flowStates.Add<State.completed>(flowState => {
  return flowState
    ...
    .WithEventHandlers(handlers =>
    {
      handlers.Add(g => g.OnInvoiceUnlinked);
      handlers.Add(g => g.OnInvoiceCancelled);
    })
});
```

Firing a Workflow Event

After you have declared a workflow event, you need to fire it. To fire a workflow event, you select an event from the list of events declared in the class derived from the `PXEntityEvent` class and call the `FireOn` method. In the `FireOn` method parameters, you provide the graph that will be used as a host for event execution, and instances of DACs that were declared in the `PXEntityEvent`. An example of firing the `InvoiceLinked` is shown in the following code.

```
public static SOOrderShipment LinkInvoice(this SOOrderShipment self,
    SOInvoice invoice, PXGraph graph)
{
  if (self is null || invoice is null)
    return self;

  self.InvoiceType = invoice.DocType;
  self.InvoiceNbr = invoice.RefNbr;
  self = (SOOrderShipment)graph.Caches<SOOrderShipment>().Update(self);

  SOOrderShipment.Events
    .Select(e => e.InvoiceLinked)  //PXEntityEvent<SOOrderShipment, SOInvoice>
```

```
    .FireOn(graph, self, invoice);

  return self;
}
```

In the code above, you select the `InvoicedLinked` event from the list of `SOOrderShipment` events and call the `FireOn` method. As a parameter, you provide an instance of the graph that will be used as a host for event execution, and instances of `SOOrderShipment` and `SOInvoice` DACs that were linked with each other. The signature of the `InvoiceLinked` event restricts the `FireOn` method to an overload that has this exact parameters.

Triggering a Transition by an Event

To define a transition that is triggered by an event, you specify the event handler name in the `IsTriggeredOn` method of the transition definition, as the following code shows.

```
transitions.AddGroupFrom<States.completed>(ts =>
{
  ts.Add(t => t.To<States.linked>().IsTriggeredOn(g => g.OnInvoiceLinked));
});
```

Lesson 2.1: Use an Existing Event in a Custom Workflow

According to the customization description, a repair work order should be assigned the *Paid* status when the invoice created for the order is fully paid—that is, when the invoice document is assigned the *Closed* status. In this lesson, to change the status of the repair work order from *Completed* to *Paid*, you will use an existing event that is fired when the invoice document is assigned the *Closed* status.

Lesson Objectives

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

- Explore the code of the predefined workflow to find the event
- Create a custom event handler
- Bind the event handler to an event
- Register an event handler in a particular state
- Implement a transition triggered by an event
- Override the Persist method

Step 2.1.1: Explore the Acumatica ERP Source Code

To learn whether you can use the code of the workflow in your customization, you need first to explore this code. In this step, you will explore the source code of Acumatica ERP and search for the event that you can use.

Locating the Release Action

To change the status of a repair work order whose invoice has been paid, you need to use an event that is fired when the invoice is fully paid and closed. The change of the invoice's status occurs as a result of the release of a payment. Therefore, to find the event, you should first learn the code location of the `Release` action on the *Payments and Applications* (AR302000) form by doing the following:

1. In Acumatica ERP, open the *Payments and Applications* form.
2. Inspect the **Release** command on the More menu: Press Ctrl + Alt, and click **Release**.
3. In the **Element Properties** dialog box, which opens, learn the name of the graph where the action is defined: `ARPaymentEntry`.
4. Click **Actions > View Business Logic Source**.

   The Source Code browser opens.
5. In the **Graph Name** box, learn the full name of the graph: `PX.Objects.AR.ARPaymentEntry`.

   You will need this name later to find the graph in the source code.

To find the event that you can use in the custom workflow, it is helpful to debug the Acumatica ERP source code with breakpoints and see which breakpoint is hit in which scenario.

Preparing the Project for Debugging

To prepare the `PhoneRepairShop_Code` Visual Studio project for the debugging of the 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 are using 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** check box 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** check box during installation, you should remove Acumatica ERP and install it again with the **Install Debugger Tools** check box selected. For details, see *To Install the Acumatica ERP Tools*.

   > 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.

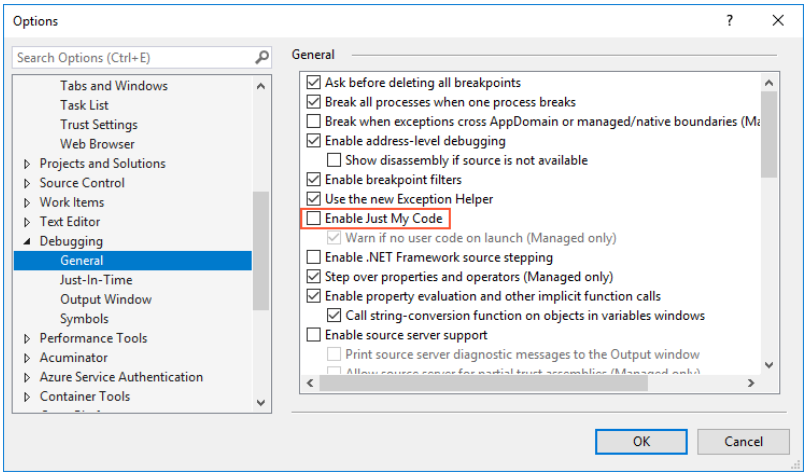


Figure: The cleared the Enable Just My Code check box

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, such as `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 **Release** action—that is, the `IEnumerable Release(PXAdapter adapter)` method. The code should look as shown in the following screenshot.

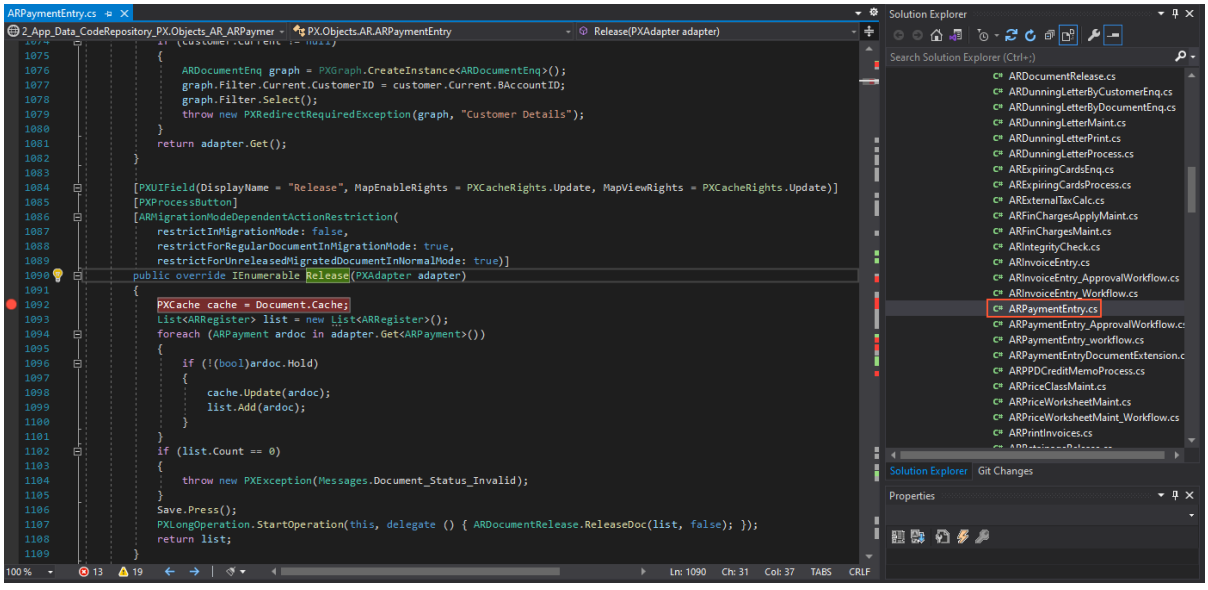


Figure: The source code of the Release action

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

7. Attach the Visual Studio debugger to a running 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.

Step 2.1.2: Explore and Debug the Code

Now that you have prepared the `PhoneRepairShop_Code` project for debugging in Visual Studio, you can continue exploring the code of the `Release` action to find the event you should use in the custom workflow. Do the following:

1. In the code of the `Release` method, find the call of the `ARDocumentRelease.ReleaseDoc` method.

   The `ReleaseDoc` static method of the `ARDocumentRelease` static class is used to process the list of records in an invoice.

   > You may notice that the `ReleaseDoc` operation starts within the `StartOperation` static method of the `PXLongOperation` static class to be executed in a background thread.

2. Go to the definition of the `ARDocumentRelease.ReleaseDoc` method, which is in the `ARDocumentRelease.cs` file.

   Notice that for the created instance, the `ReleaseDoc` method invokes the `ARReleaseProcess.ReleaseDocProc` static method, which is applied for each document in the list of the documents to be released.

3. Go to the definition of the `ARReleaseProcess.ReleaseDocProc` method.

   Notice that the `ARReleaseProcess.ReleaseDocProc` method receives the document as a record of the `ARRegister` data access class. During the document processing, the method creates a work copy of the record, specifies the fields of the record in the copy, and then restores the record in the cache from the completely ready copy. Inside the method, you can find the call to the `ProcessPayment` method.

4. Go to the definition of the `ARReleaseProcess.ProcessPayment` method, and explore its code. The method processes the release of a payment.

   Continue exploring the methods that are called inside the `ARReleaseProcess.ProcessPayments` method. You can find the `ARReleaseProcess.CloseInvoiceAndClearBalances` method. The call hierarchy is shown in the following screenshot.

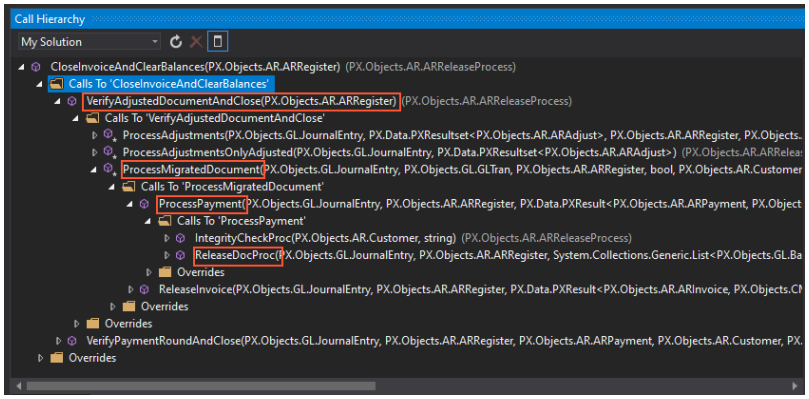


Figure: The call hierarchy for the CloseInvoiceAndClearBalances method

Notice the following line in the code of the `CloseInvoiceAndClearBalances` method.

```
RaiseInvoiceEvent(ardoc, ARInvoice.Events.Select(ev => ev.CloseDocument));
```

If you go to the definition of the `RaiseInvoiceEvent` method, you can find that the `CloseDocument` event is fired in this method, as shown in the following line of the method.

```
invEvent.FireOn(this, (ARInvoice)doc);
```

5. To make sure that `CloseDocument` is the event that changes the invoice status to *Closed*, do the following:
   a. Navigate to the `ARInvoiceEntry_Workflow.cs` file, which contains the definition of the workflow for the *Invoices* (SO303000) form.
   b. In the `WithHandlers` method, locate the handler for the `CloseDocument` event. The handler's name is `OnCloseDocument`.
   c. Locate the transition from the `Open` state to the `Closed` state. Notice that the transition is triggered by the `OnCloseDocument` event handler, as shown in the following screenshot.

```
transitions.AddGroupFrom<State.open>(ts =>
{
    ts.Add(t => t
        .To<State.open>()
        .IsTriggeredOn(g=>g.emailInvoice)
        .WithFieldAssignments(fass => fass.Add<emailed>(v => v.SetFromValue(true))));
    ts.Add(t => t
        .To<State.closed>()
        .IsTriggeredOn(g => g.OnReleaseDocument)
        .When(conditions.IsClosed));
    ts.Add(t => t
        .To<State.closed>()
        .IsTriggeredOn(g => g.OnCloseDocument));
    ts.Add(t => t
        .To<State.canceled>()
        .IsTriggeredOn(g => g.OnCancelDocument));
});
```

Figure: Transitions from the Open state

6. To make sure that the firing of the `CloseDocument` event in the `ARReleaseProcess.CloseInvoiceAndClearBalances` method is indeed what happens when an invoice is fully paid, do the following:
   a. Add a breakpoint to the line that fires the `OnCloseDocument` event of the `ARReleaseProcess.CloseInvoiceAndClearBalances` method.
   b. Prepare a payment to debug by doing the following:
      a. Open the invoice you created *Exercise 1.2: Configure the Conditional Appearance of a Button and Command* by searching for its number on the *Invoices* form. The invoice should have the *INV000049* number.

b. On the form toolbar of the *Invoices* form, click **Remove Hold**.

   The invoice is assigned the *Balanced* status.

c. On the form toolbar, click **Release**. The invoice is released.

d. On the More menu, click **Pay**.

   The *Payments and Applications* (AR302000) form opens.

c. On the form toolbar of the *Payments and Applications* form, click **Remove Hold**, and then click **Release**.

d. In Visual Studio, notice that a breakpoint in the `ARReleaseProcess.CloseInvoiceAndClearBalances` method is hit when an invoice is closed.

e. Stop debugging.

Step 2.1.3: Define the Paid State (Self-Guided Exercise)

To define a transition from the `Completed` state to the `Paid` state, you first need to define the `Paid` state. In this step, you will define the `Paid` state of the workflow as a self-guided exercise. You have learned how to add a state in *Step 1.2.3: Add States to the Workflow*.

While defining a state, you need to use the `States.paid` class, which you added in *Step 1.1.2: Define the Set of States of the Workflow*. In the `Paid` state, make the **Customer ID**, **Service ID**, and **Device ID** boxes disabled for the state.

Step 2.1.4: Define the Event Handler

In this step, you will define an event handler for the `ARInvoiceEntry.OnCloseDocument` event. Do the following:

1. In the `RSSVWorkOrderEntry` graph, define an event handler, as the following code shows.

```
#region Workflow Event Handlers
public PXWorkflowEventHandler<RSSVWorkOrder, ARInvoice> OnCloseDocument;
#endregion
```

   In the first type parameter of the handler, you have specified the primary DAC of the form where the handler is used. As the second type parameter, you have specified the primary DAC of the form where the event is fired.

   > Add the `using` directive for the `PX.Data.WorkflowAPI` namespace if it has not been added earlier.

2. In the `RSSVWorkOrderWorkflow` class, bind the `OnCloseDocument` event handler to the `CloseDocument` event in the screen configuration: After the `AddDefaultFlow` method, call the `WithHandlers` method. In the method, add the `OnCloseDocument`, as the following code shows.

```
                    .WithHandlers(handlers =>
                    {
                        handlers.Add(handler => handler
                            .WithTargetOf<ARInvoice>()
                            .OfEntityEvent<ARInvoice.Events>(e => e.CloseDocument)
                            .Is(g => g.OnCloseDocument)
                            .UsesPrimaryEntityGetter<
```

```
                            SelectFrom<RSSVWorkOrder>.

Where<RSSVWorkOrder.invoiceNbr.IsEqual<ARRegister.refNbr.FromCurrent>>>());
                    })
```

In the code above, you have added a handler for an event that changes the state of the invoice. In the `UsesPrimaryEntityGetter` method, you have selected the repair work order whose status should be updated by the number of the invoice that has been closed.

Make sure you have added the `using` directives, which are shown in the following code.

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

3. In the definition of the `Completed` state, add the event handler, as the following code shows.

```
                            fss.Add<States.completed>(flowState =>
                            {
                                return flowState
                                    .WithFieldStates(...)
                                    .WithActions(...)
                                    .WithEventHandlers(handlers =>
                                    {
                                        handlers.Add(g => g.OnCloseDocument);
                                    });
                            });
```

4. In the lambda expression for the `WithTransitions` method, add the transition from the `Completed` state to the `Paid` state, as the following code shows.

```
                        .WithTransitions(transitions =>
                        {
                            ...
                            transitions.AddGroupFrom<States.completed>(ts =>
                            {
                                ts.Add(t => t.To<States.paid>().IsTriggeredOn(g =>
g.OnCloseDocument));
                            });
                        })
```

5. Save your changes.

Step 2.1.5: Override the Persist Method

The `Persist` method of any graph saves changes to the database. In this lesson, you are customizing an existing wokflow to update another entity. For the changes to be saved to the database, you need to override the `Persist` method of the form where you are customizing the workflow and specify which changes should be saved to the database.

In this step, you will override the `Persist` method of the `ARReleaseProcess` graph so that the changes to a repair work order that are triggered by the event on the *Invoices* (SO303000) form are saved to the database. Do the following:

1. In the `PhoneRepairShop_Code` project, create a file named `ARReleaseProcess.cs` based on a C# template.

2. In the `ARReleaseProcess.cs` file, declare the `ARReleaseProcess` graph extension, as the following code shows.

```
public class ARReleaseProcess_Extension : PXGraphExtension<ARReleaseProcess>
{
}
```

> Add the `using` directives shown in the following code.
>
> ```
> using PX.Data;
> using PX.Objects.AR;
> ```

3. Use Acuminator to suppress the *PX1016* error in a comment. In this course, for simplicity, the extension is always active.
4. In the `ARReleaseProcess_Extension`, add the following code.

```
public SelectFrom<RSSVWorkOrder>.View UpdWorkOrder;

[PXOverride]
public virtual void Persist(Action baseMethod)
{
    using (PXTransactionScope ts = new PXTransactionScope())
    {
        baseMethod();
        UpdWorkOrder.Cache.Persist(PXDBOperation.Update);
        ts.Complete(Base);
    }
    UpdWorkOrder.Cache.Persisted(false);
}
```

> Add the `PX.Data.BQL.Fluent` namespace in the `using` directive.

In the code above, you have declared a view for repair work orders, and overridden the `Persist` method. In the overridden method, you have done the following: opened a transaction that saves changes to the database, called the base method, saved your changes in `PXCache` to the database, and completed the transaction. Finally, you have called the `Persisted` method to mark the saving of changes to the database as completed. For details, see *Persisted Method*.

5. Save your changes.

Step 2.1.6: Test the Transition

To test the transition from the `Completed` state to the `Paid` state, do the following:

1. Rebuild the `PhoneRepairShop_Code` project.
2. In Acumatica ERP, on the Repair Work Orders (RS301000) form, create a repair work order with the following settings:
   - **Customer ID**: *C000000001*
   - **Service**: *Battery Replacement*
   - **Device**: *Nokia 3310*
   - **Description**: `Battery replacement, Nokia 3310`

3. Save the record.

   The repair work order is assigned the *000005* number.

4. Create an invoice for the order as follows:

   a. On the form toolbar, click **Remove Hold** and then **Assign**.

   b. In the **Select Assignee** dialog box, select *Andrews, Michael* in the **Assignee** box, and click **OK**.

   c. On the More menu, click **Complete**.

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

   Notice the invoice number in the **Invoice Nbr.** box.

5. On the *Invoices* (SO303000) form, open the invoice created in the previous instruction.

6. On the form toolbar, click **Remove Hold** and then **Release**.

   The invoice is assigned the *Open* status.

7. On the More menu, click **Pay**.

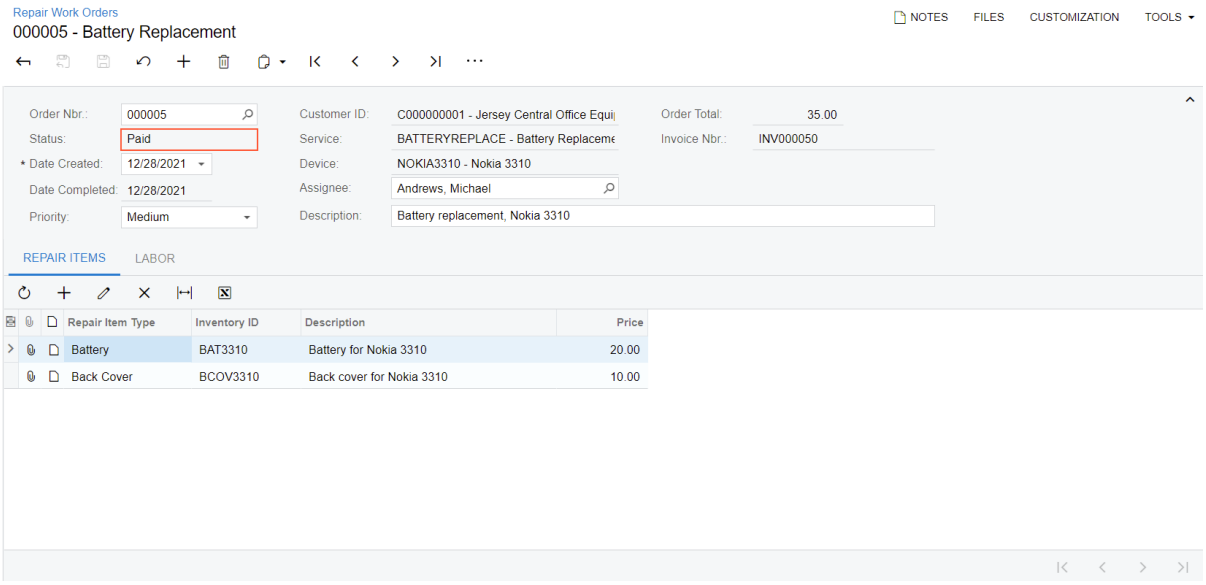
   The *Payments and Applications* (AR302000) form opens.

8. On the form toolbar, click **Remove Hold** and then **Release**.

   The invoice is now fully paid.

9. On the Repair Work Orders form, open the *000005* repair work order. Make sure it has the *Paid* status, as shown in the following screenshot.

Figure: The repair work order with the Paid status



Lesson Summary

In this lesson, you have used the `OnCloseDocument` event to trigger the transition from the `Completed` state to the `Paid` state. You have implemented the `Paid` state, created a custom event handler, and registered the event handler in the screen configuration. You have also overridden the `Persist` method to save changes to the database.

Lesson 2.2: Create a New Event

According to the customization requirements, a repair work order should be assigned the *Ready for Assignment* status when the invoice created for the order is prepaid in the specified percent.

For a repair work order to be prepaid, a user will first create an invoice for this order and then apply a prepayment to the invoice. If the percent of the applied prepayment is greater than or equal to the required prepayment percentage, which is specified on the *Payments and Applications* (AR302000) form, the system will change the status of the order from *Pending Payment* to *Ready for Assignment* to indicate that the prepayment is applied.

To implement this status change, you will define a new event and fire it when an invoice is prepaid. You will define the event handler, register it in the screen configuration for your custom workflow, and configure a transition.

Learning Objectives

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

- Derive the value for a custom field from another form
- Create a custom event
- Override a method to fire the event

Step 2.2.1: Create a Custom Field 

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 create a custom field in this step and derive the field value in the next step.

Adding a Custom Field to the Payment and Applications Form

Step 2.2.2: Derive the Value of the 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.

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 2.2.3: Explore the Source Code

In this step, you will explore the course code of *Invoices* (SO303000) and *Payments and Applications* (AR302000) forms to learn the field names and methods you need to use in order to determine where and when to fire the event that triggers the transition from the `PendingPayment` to `ReadyForAssignment` state.

Learning Field Names

To determine whether an invoice has been prepaid and whether the event that triggers the transition from the `PendingPayment` state to the `ReadyForAssignment` state should be fired, you need to calculate the percentage of the invoice amount that has been prepaid. To calculate this percentage, you need the outstanding amount of the invoice and the original amount of the invoice. These values are displayed in the **Balance** and **Amount** boxes, respectively, of the *Invoices* (SO303000) form. Thus, to calculate the percentage, you should learn the field names of these boxes by doing the following:

1. Open the *Invoices* form.
2. To open the Element Inspector for the **Balance** box, press Ctrl + Alt and click the box.

   In the **Element Properties** dialog box, which opens, notice that the **Balance** box is defined by the `CuryDocBal` field of the `ARInvoice` data access class.
3. To open the Element Inspector for the **Amount** box, press Ctrl + Alt and click the box.

   In the **Element Properties** dialog box, which opens, notice that the **Amount** box is defined by the `CuryOrigDocAmt` field of the `ARInvoice` data access class.

   If you do not see the **Amount** box on the form, you need to select the **Validate Document Totals on Entry** check box on the *Accounts Receivable Preferences*(AR101000) form and then perform this instruction.

Exploring the Source Code of the Release Method

To fire the event that triggers the transition from the `PendingPayment` to `ReadyForAssignment` state, you need to first override the method where the outstanding amount of the invoice (which you just explored) is calculated and updated. This recalculation is performed on release of the prepayment that is applied to an invoice. Thus, to find the method to override, you need to explore the code of the action that is associated with the **Release** button on the *Payments and Applications* (AR302000) form.

You have already started exploring the methods invoked in the code of the `Release` action in *Step 2.1.2: Explore and Debug the Code*. You might have noticed that the amounts of the invoice, including the `CuryDocBal` field value, are recalculated in the `UpdateBalances` method of the `ARReleaseProcess` class. To make sure this is the right method to override, add a breakpoint to the `UpdateBalances` method, run the application in debug mode, and trace how the `CuryDocBal` value is changed in the method.

As a result of this debugging, you can see that the `UpdateBalances` method is the method you should override to calculate the prepaid percentage of the invoice and to fire the event that triggers the transition from the `PendingPayment` state to the `ReadyForAssignment` state, if necessary.

Step 2.2.4: Declare the Event

In this step, you will declare the event that triggers the transition from the `PendingPayment` state to the `ReadyForAssignment` state, define an event handler for this event, and register the event handler in the screen configuration. You will also define the transition triggered by the event.

Do the following:

1. In the `RSSVWorkOrder` DAC, declare the class that contains the custom event, as the following code shows.

```
public class MyEvents : PXEntityEvent<ARRegister>.Container<MyEvents>
{
    public PXEntityEvent<ARRegister> InvoiceGotPrepaid;
}
```

Make sure that you added the `PX.Data.WorkflowAPI` `using` directive.

2. In the `RSSVWorkOrderEntry` graph, declare an event handler for the `InvoiceGotPrepaid` event, as the following code shows.

```
public PXWorkflowEventHandler<RSSVWorkOrder, ARRegister> OnInvoiceGotPrepaid;
```

3. In the `RSSVWorkOrderWorkflow` class, bind the `OnInvoiceGotPrepaid` event handler to the `InvoiceGotPrepaid` event in the screen configuration: In the lambda expression for the `WithHandlers` method, add the handler, as the following code shows.

```
.WithHandlers(handlers =>
{
    ...
    handlers.Add(handler => handler
        .WithTargetOf<ARRegister>()
        .OfEntityEvent<RSSVWorkOrder.MyEvents>(e => e.InvoiceGotPrepaid)
        .Is(g => g.OnInvoiceGotPrepaid)
        .UsesPrimaryEntityGetter<
            SelectFrom<RSSVWorkOrder>.
            Where<RSSVWorkOrder.invoiceNbr
            .IsEqual<ARRegister.refNbr.FromCurrent>>>());
})
```

4. In the definition of the `PendingPayment` state, add the event handler, as the following code shows.

```
fss.Add<States.pendingPayment>(flowState =>
{
    return flowState
        .WithFieldStates(...)
        .WithActions(...)
        .WithEventHandlers(handlers =>
        {
            handlers.Add(g => g.OnInvoiceGotPrepaid);
        });
});
```

5. In the lambda expression for the `WithTransitions` method, declare the transition from the `PendingPayment` state to the `ReadyForAssignment` state, as the following code shows.

```
transitions.AddGroupFrom<States.pendingPayment>(ts =>
{
    ...
    ts.Add(t => t.To<States.readyForAssignment>()
        .IsTriggeredOn(g => g.OnInvoiceGotPrepaid));
});
```

6. Save your changes.

Step 2.2.5: Fire the Event

As you learned in *Step 2.2.3: Explore the Source Code*, you need to override the `UpdateBalances` method to fire the event that triggers the transition from the `PendingPayment` state to the `ReadyForAssignment` state. However, you should first calculate the prepaid percentage value based on the values of the **Balance** and **Amount** boxes of an invoice on the *Invoices* (SO303000) form. These values have been calculated in the base `UpdateBalances` method.

You can work with the fields of the `ARRegister adjddoc` parameter instead of selecting the corresponding `ARInvoice` parameter because `ARRegister` is the base class for the `ARInvoice` class and contains the same essential fields.

The `ARRegister` entity has been modified in the base method and its value has been updated in `PXCache`. But the value passed as the parameter to the overridden method has not been updated. Thus, in the overridden method, you should use the cached value. To get the cached value of the entity, you need to use the `Locate` method. For details, see *Locate(Object) Method*.

Firing the Event

Add the following code to the `ARReleaseProcess_Extension` class of the `PhoneRepairShop_Code` project.

```
public delegate void UpdateBalancesDelegate(ARAdjust adj,
    ARRegister adjddoc, ARTran adjdtran);
[PXOverride]
public virtual void UpdateBalances(ARAdjust adj,
    ARRegister adjddoc, ARTran adjdtran,
    UpdateBalancesDelegate baseMethod)
{
    baseMethod(adj, adjddoc, adjdtran);

    ARRegister ardoc = adjddoc;
    ARRegister cached = (ARRegister)Base.ARDocument.Cache.Locate(ardoc);
    if (cached != null)
    {
        ardoc = cached;
    }

    RSSVWorkOrder order = SelectFrom<RSSVWorkOrder>.
        Where<RSSVWorkOrder.invoiceNbr.
        IsEqual<ARRegister.refNbr.FromCurrent>>
        .View.SelectSingleBound(Base, new[] { ardoc });
```

```
            if (order != null &&
                order.Status == WorkOrderStatusConstants.PendingPayment)
            {
                var payment = SelectFrom<ARPayment>.
                    Where<ARPayment.docType.
                        IsEqual<ARAdjust.adjgDocType.FromCurrent>.
                        And<ARPayment.refNbr.
                        IsEqual<ARAdjust.adjgRefNbr.FromCurrent>>>
                    .View.SelectSingleBound(Base, new[] { ardoc });

                if (payment != null)
                {
                    var paidPercent = (ardoc.CuryOrigDocAmt - ardoc.CuryDocBal) * 100
                        / ardoc.CuryOrigDocAmt;
                    var paymentExt = PXCache<ARPayment>.
                        GetExtension<ARPaymentExt>(payment);
                    if (paidPercent >= paymentExt.UsrPrepaymentPercent)
                    {
                        RSSVWorkOrder.MyEvents
                          .Select(e => e.InvoiceGotPrepaid)
                          .FireOn(Base, ardoc);
                        // No need to call the Persist method.
                    }
                }
            }
        }
```

In the code above, first you have called the base method, and then you have obtained the cached value of the invoice. You select the repair work order with the same invoice number as the invoice passed as a parameter. Then you select the prepayment and its extension (which contains the prepayment percent) and calculate the prepaid percentage for invoice. If the prepaid percentage is greater than the required percentage, you fire the `InvoiceGotPrepaid` event.

Both SO and AR payments and SO and AR invoices should be selected from the `ARPayment` and `ARInvoice` tables. To avoid selecting the document of the wrong type, when you select payments or invoices, you should check not only the `refNbr` of the document, but also its `docType`.

At the end of the method, there is no need to call the `Persist` method, because it is called at the end of the release process.

Step 2.2.6: Test the Transition

In this step, you will test the transition from the `PendingPayment` state to the `ReadyForAssignment` state. Do the following:

1. Rebuild the `PhoneRepairShop_Code` project.
2. In Acumatica ERP, open the *000004* repair work order. It should have the *Pending Payment* status.
3. On the form toolbar, click **Create Invoice**.

   In the **Invoice Nbr.** box, note the number of the invoice created for the repair work order.
4. On the *Invoices* (SO303000) form, open the invoice created in the previous instruction.
5. On the form toolbar, click **Remove Hold** and then **Release**.

   The invoice gets the *Open* status.

6. On the More menu, click **Pay**.

   The *Payments and Applications* (AR302000) form opens. Note that the **Prepayment Percent** box has the *10.00* value, which has been copied from the Repair Work Order Preferences (RS101000) form.

7. On the *Payments and Applications* form, change the existing values to the following values, as shown in the following screenshot:

   - **Prepayment Percent**: `5.00`
   - **Documents to Apply > Amount Paid**: `3.00`

   > As you have noticed previously, the total amount of the invoice is $50.00. The paid amount ($3.00) is greater than the amount ($2.50) corresponding to the prepayment percent you specified on the form (which is 5%). Thus, $3 is enough to prepay the work order.

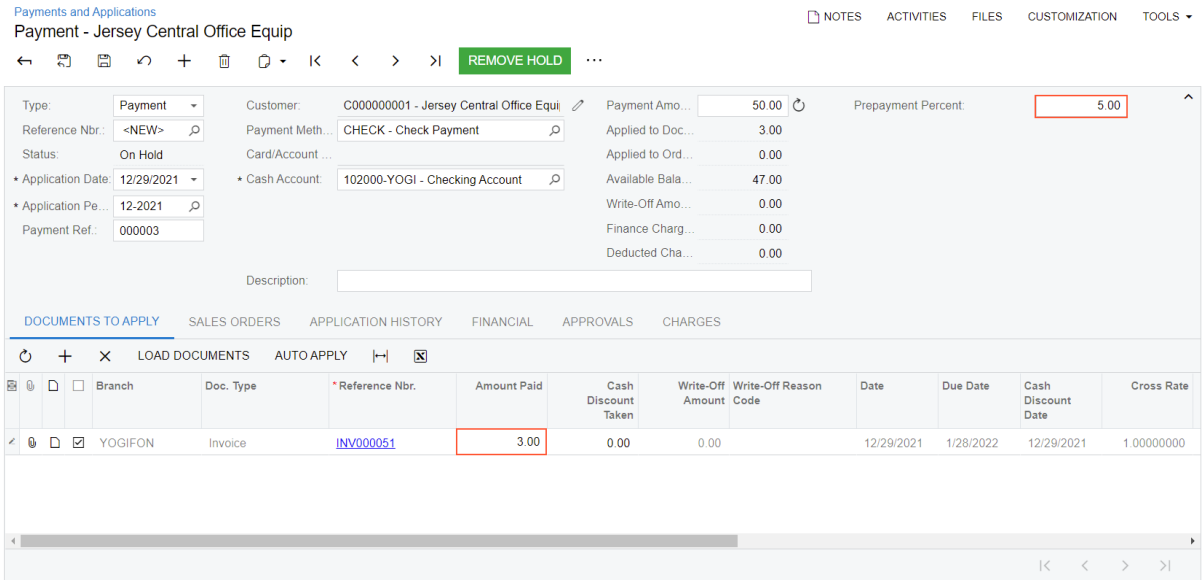


Figure: The prepayment for the created invoice

8. On the form toolbar, click **Remove Hold** and then **Release**.

   The prepayment is applied.

9. On the Repair Work Orders form, open the *000004* repair work order.

   Notice that the status of the created order has changed to *Ready for Assignment*, as shown in the following screenshot.

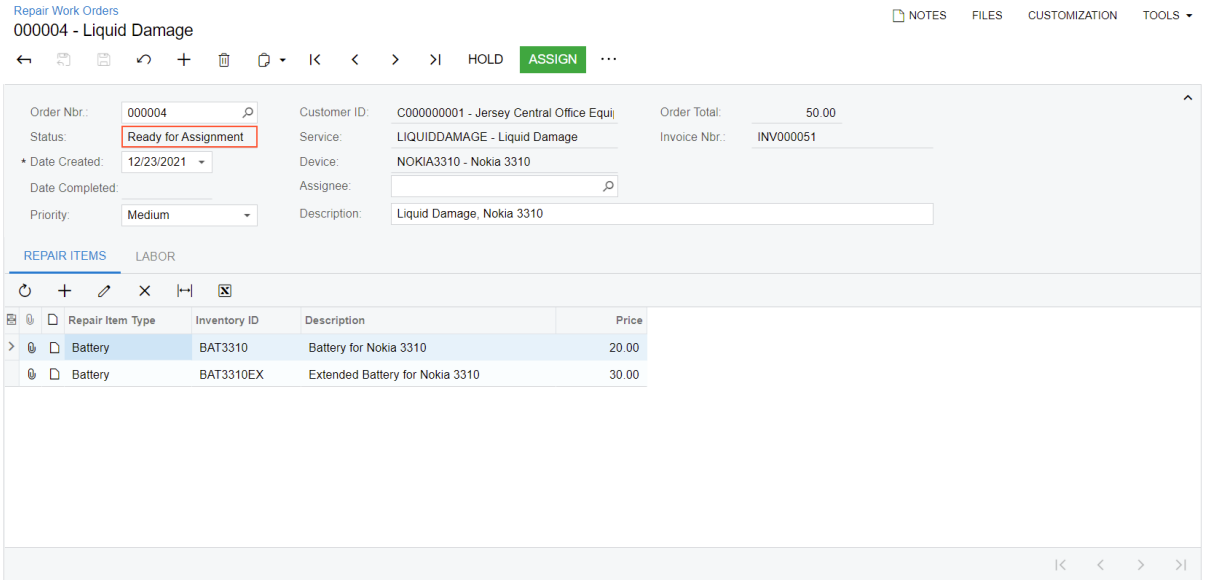


Figure: The repair work order with the Ready for Assignment status

Lesson Summary

In this lesson, you have learned how to create a custom workflow event. You have implemented a new event, defined an event handler for it, and fired the event in an overridden method. You have also learned how to derive values from one entity and copy them to another entity.

Part 2 Summary

In this part, you have learned how to define a custom workflow event and fire it and how to use existing workflow events in a custom workflow. You have also learned how to derive a value of a custom field from another entity and how to explore the source code of Acumatica ERP.

You have completed the implementation of the workflow for the Repair Work Orders (RS301000) form. To verify that all elements of the workflow have been implemented, you can open the state diagram for the Repair Work Orders form and compare it to the one shown in *Customization Description*.

Part 3: Customizing an Existing Workflow

Not only can you create your own workflows and use predefined workflows in your custom workflow, but you can also customize predefined workflows. In this part, you will learn how to customize the predefined workflow on the *Invoices* (SO303000) form.

About Customization of Workflows

Using workflow API, you can not only create your own workflows but also customize workflows defined in source code of Acumatica ERP. The workflow API provides a set of methods to update and remove workflows and elements of a workflow such as states and transitions.

In previous parts of the course, you have used the `AddScreenConfigurationFor` and `AddDefaultFlow` methods to add a workflow to a form, and `Add` methods to add actions, states, transitions, and other elements of the screen configuration to the workflow. For each of these `Add` methods, the workflow API provides the `Update` method for updating the element and the `Delete` method for deleting the element from the workflow. Similarly, for the `AddScreenConfigurationFor` and `AddDefaultFlow` methods, the workflow API provides the `UpdateScreenConfigurationFor` and `UpdateDefaultFlow` methods that allow you to update the existing screen configuration and the existing default workflow, respectively. To access the elements of the screen configuration, you use the same methods as the methods that you use while adding the workflow, such as `WithFlowStates`, `WithActions`, `WithTransitions`.

An example of an updated workflow is shown in the following code.

```
context.UpdateScreenConfigurationFor(screen =>
{
  return screen
    .UpdateDefaultFlow(InjectApprovalWorkflow)
    .WithActions(actions =>
    {
      actions.Add(approve);
      actions.Add(reject);
      actions.Update(
        g => g.putOnHold,
          a => a.WithFieldAssignments(fas =>
          {
            fas.Add<ARRegister.approved>(f => f.SetFromValue(false));
            fas.Add<ARRegister.rejected>(f => f.SetFromValue(false));
          }));
      actions.Update(
        g => g.releaseFromCreditHold,
        a => (BoundedTo<ARInvoiceEntry, ARInvoice>.
          ActionDefinition.ConfiguratorAction)a.InFolder(approvalCategory, reject));
    });
});
```

Lesson 3.1: Customize the Workflow for Invoices

To quickly continue working on a repair work order after an invoice has been prepaid, a user needs to be able to invoke a command that opens the corresponding repair work order from the *Invoices* (SO303000) form. In this

lesson, you will customize the workflow on the *Invoices* form to add the **View Repair Work Order** command and its underlying action, which opens the repair work order.

Lesson Objectives

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

- Update an existing screen configuration
- Define a new action category
- Define a workflow action based on a graph action

Step 3.1.1: Add the Graph Action

In this step, you will create an extension of a graph of the *Invoices* (SO303000) form. In the extension, you will add an action that opens a repair work order. Do the following:

1. Learn the name of the graph where the action should be added:
   a. On the *Invoices* form, click Ctrl + Alt, and click the Summary area of the form.
   b. In the **Element Properties** dialog box, notice the name in the **Business Logic** box: `SOInvoiceEntry`.
2. In the `PhoneRepairShop_Code` project, create an extension of the `SOInvoiceEntry` graph, as the following code shows.

```
namespace PX.Objects.SO
{
    public class SOInvoiceEntry_Extension : PXGraphExtension<SOInvoiceEntry>
    {
    }
}
```

Use Acuminator to suppress the *PX1016* error in a comment. In this course, for simplicity, the extension is always active.

3. In the graph extension, implement the **View Repair Work Order** action, as the following code shows.

```
        public PXAction<ARInvoice> ViewOrder;
        [PXButton, PXUIField(DisplayName = "View Repair Work Order")]
        protected virtual IEnumerable viewOrder(PXAdapter adapter)
        {
            var orderEntry = PXGraph.CreateInstance<RSSVWorkOrderEntry>();
            var order = orderEntry.WorkOrders.Search<RSSVWorkOrder.invoiceNbr>(
                Base.Document.Current.RefNbr);
            if (order == null)
                return adapter.Get();

            orderEntry.WorkOrders.Current = order;
            throw new PXRedirectRequiredException(orderEntry, true,
                nameof(ViewOrder))
            {
                Mode = PXBaseRedirectException.WindowMode.NewWindow
            };
        }
```

4. Add the following `using` directives.

```
using PhoneRepairShop;
using PX.Data;
using PX.Objects.AR;
using System.Collections;
```

5. Save your changes.

Step 3.1.2: Add the Workflow Extension

In this step, you will create an extension for the workflow defined for the *Invoices* (SO303000) form. Do the following:

1. Find the name of the class that defines the workflow for the *Invoices* form. To do this, explore the contents of the `PX.Objects/SO/Workflow` folder of the Acumatica ERP source code. There you can find the class you need to extend: `SOInvoiceEntry_Workflow`, which is an extension of the `SOInvoiceEntry` graph.

   In the code of the `SOInvoiceEntry_Workflow` class, notice that the states of the workflow are defined in the `ARDocStatus` class. You will need this class in the next lesson.

2. In the `Workflows` folder of the `PhoneRepairShop_Code` project, create the `SOInvoiceRepairOrder_Workflow.cs` file.

3. In the `SOInvoiceRepairOrder_Workflow.cs` file, define the `SOInvoiceRepairOrder_Workflow` class, as the following code shows.

```
public class SOInvoiceOrder_Workflow : PXGraphExtension<SOInvoiceEntry_Workflow,
    SOInvoiceEntry>
{
    public override void Configure(PXScreenConfiguration config)
    {
        Configure(config.GetScreenConfigurationContext<SOInvoiceEntry, ARInvoice>());
    }

    protected virtual void Configure(WorkflowContext<SOInvoiceEntry,
        ARInvoice> context)
    {
    }
}
```

   In the code above, you have defined an extension of the `SOInvoiceEntry_Workflow` class. As a second parameter of the extension, you have specified the graph of the *Invoices* form. In the extension, you have overridden the `Configure(PXScreenConfiguration)` method, which initializes the screen configuration, and declared the `Configure(WorkflowContext)` method where you will update the workflow.

4. Make sure that you have added the needed `using` directives.

5. Save your changes.

Step 3.1.3: Define the Workflow Action

The **View Repair Work Order** command should be displayed in its own category on the More menu of the *Invoices* (SO301000) form. The action is displayed when the invoice is prepaid—that is, when the invoice has the *Open*

status. In this step, you will define a custom category, add the action to the workflow, and make it available in the `Open` state of the workflow.

Do the following:

1. In the `SOInvoiceRepairOrder_Workflow.cs` file, define the class that contains information for custom categories, as the following code shows.

```
public static class ActionCategories
{
    public const string RepairCategoryID = "Repair Orders Category";

    [PXLocalizable]
    public static class DisplayNames
    {
        public const string RepairOrders = "Repair Orders";
    }
}
```

   In the code above, you have defined a category with the **Repair Orders** display name and the string identifier for this category.

   > Add the `PX.Common using` directive to use the `PXLocalizable` attribute.

2. In the `Configure(WorkflowContext)` method, add the definition of the **Repair Orders** category and the **View Repair Work Order** action, as the following code shows.

```
var repairCategory = context.Categories.CreateNew(
    ActionCategories.RepairCategoryID,
    category => category.DisplayName(
    ActionCategories.DisplayNames.RepairOrders));

var viewOrder = context.ActionDefinitions
  .CreateExisting<SOInvoiceEntry_Extension>(g => g.ViewOrder,
    a => a.WithCategory(repairCategory));
```

   In the code above, you have created a workflow definition of the **Repair Orders** category by using the `Categories.CreateNew` method and a workflow definition of the **View Repair Work Order** action by using the `ActionDefinitions.CreateExisting` method. As the type parameter of the `CreateExisting` method, you specify the extension where the action is defined.

3. After the definitions of the category and the action, update the default workflow, as the following code shows.

```
context.UpdateScreenConfigurationFor(screen => screen
    .UpdateDefaultFlow(flow =>
    {
        return flow
            .WithFlowStates(flowStates =>
            {
                flowStates.Update<ARDocStatus.open>(flowState =>
                {
                    return flowState.WithActions(actions =>
                        actions.Add(viewOrder));
                });
            });
    })
    .WithCategories(categories =>
```

```
                {
                    categories.Add(repairCategory);
                })
                .WithActions(actions =>
                {
                    actions.Add(viewOrder);
                })
            );
```

In the code above, you have updated the screen configuration by calling the `UpdateScreenConfigurationFor` method. In the lambda expression for the method, you have updated the default workflow by calling the `UpdateDefaultFlow` method. In the lambda expression for the `UpdateDefaultFlow` method, you have added the **View Repair Work Order** action to the `Open` state of the workflow.

In the `WithCategories` method, you have added the **Repair Orders** category to the screen configuration. In the `WithActions` method, you have added the **View Repair Work Order** action to the screen configuration.

4. Save your changes.

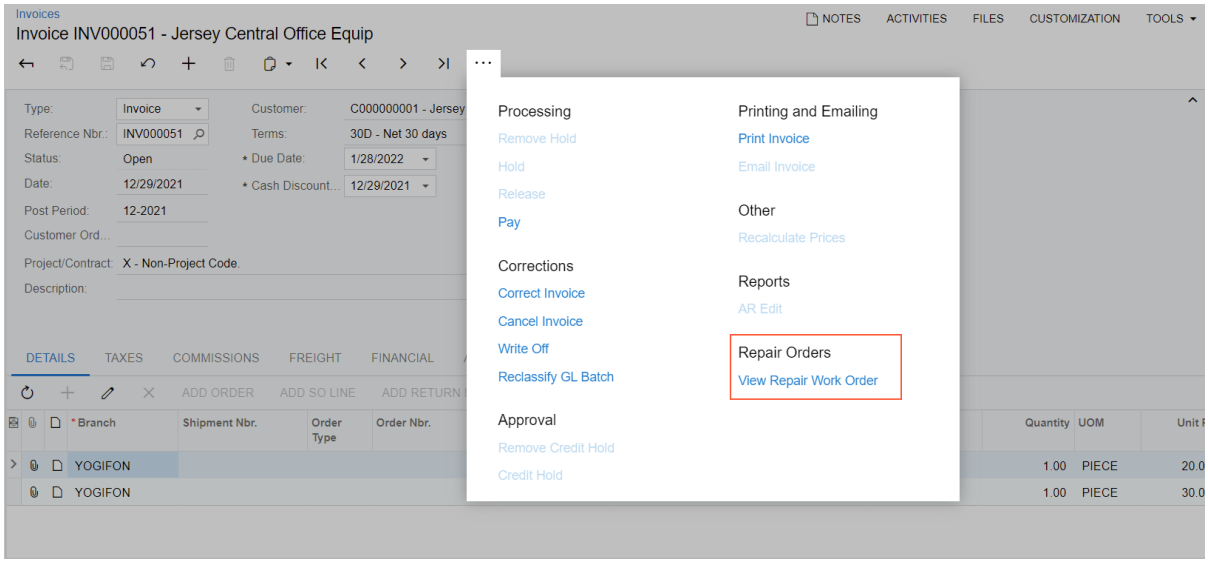
Step 3.1.4: Test the Customized Workflow

In this step, you will test the **View Repair Work Order** command. Do the following:

1. Rebuild the `PhoneRepairShop_Code` project.
2. In Acumatica ERP, on the *Invoices* (SO303000) form, open the invoice that you created for the *000004* repair work order in *Step 2.2.6: Test the Transition*.

   The invoice has the *Open* status.
3. On the More menu, find the **View Repair Work Order** command. Note that it is displayed under the **Repair Orders** category, as shown in the following screenshot.

   Figure: The More menu of the Invoices form



4. Click the **View Repair Work Order** command.

   In a new window, the Repair Work Orders (RS301000) form opens with the *000004* repair work order.

5. Return to the *Invoices* form.
6. Select the *INV000049* invoice. Notice that the **View Repair Work Order** command is unavailable on the More menu because this invoice has the *Closed* status.

Lesson Summary

In this lesson, you have learned how to customize an existing workflow. You have implemented a new action and a new action category, and added them to the customized workflow.

Appendix: Reference Implementation

You can find the reference implementation of the customization described in this course in the `Customization\T270` 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 *I100* data inserted by specifying the following settings:
      - **Login Tenant Name**: `MyTenant`
      - **New**: Selected
      - **Insert Data**: *I100*
      - **Parent Tenant ID**: *1*
      - **Visible**: Selected
   c. On the **Instance Configuration** page, in the **Local Path of the Instance** box, select a folder that is outside of the `C:\Program Files (x86)` or `C:\Program Files` folder. We recommend that you store the website folder outside of these folders to avoid an issue with permission to work in these folders when you perform customization of the website.

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

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

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

Appendix: Publishing the Required Customization Project

> If for some reason you cannot complete the instructions in *Step 2: Preparing the Needed Acumatica ERP Instance for the Training Course*, 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)` and `C:\Program Files` folders, we recommend that you use the `App_Data\Projects` folder for the project of the extension library.

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