Developer Guide

Workflow API Guide
2025 R1

Acumatica
The Cloud ERP

Contents

Copyright **6**

Getting Started with Workflow API: General Information **7**

Company Story and Customization Description **9**

Business Process Overview 10

Customization Description 13

Preparing an Instance for Workflow Customization **17**

Test Instance for Workflow Customization: General Information 17

Test Instance for Workflow Customization: To Deploy a Test Instance 17

Test Instance for Workflow Customization: To Turn On Workflow Validation 19

Test Instance for Workflow Customization: To Configure the Instance 19

Preparing a Screen Configuration **22**

Screen Configuration: General Information 22

Screen Configuration: To Prepare a Screen Configuration for a Form Without a Predefined Workflow 24

Step 1: Defining the Workflow Class 25

Step 2: Defining the Set of States of the Workflow 26

Step 3: Overriding the Configure Method 27

Screen Configuration: To Prepare a Screen Configuration with a Predefined Workflow 28

Screen Configuration: Restrictions of the Configure Method 30

Defining Workflow States **33**

Workflow States: General Information 33

Workflow States: Configuration of Field States 35

Workflow States: Configuration of Actions 36

Workflow States: To Define a Workflow State 38

Workflow States: Tracking of Changes in States 40

Implementing Workflow Actions **41**

Workflow Actions: General Information 41

Workflow Actions: Configuration of Actions 43

Workflow Actions: Appearance of Buttons and Commands 44

Workflow Actions: Persisting Options 45

Workflow Actions: Mass Processing Actions 46

Workflow Actions: Action's Behavior 47

Workflow Actions: Opening of a Side Panel 48

Workflow Actions: Categories of the More Menu 50

Workflow Actions: To Implement a Simple Action 51

Step 1: Implementing the Action in the Graph ... 52
Step 2: Defining a Category for the Action ... 53
Step 3: Adding the Action to the Workflow ... 53
Workflow Actions: To Implement an Action with Field Assignments ... 54
Step 1: Adding and Configuring the Action ... 55
Step 2: Adding a Workflow State and a Transition ... 55
Step 3: Testing the Field Assignment ... 56
Workflow Actions: To Configure the Conditional Appearance of the Action ... 56
Step 1: Defining the Graph Action ... 57
Step 2: Configuring the Conditional Appearance of the Button ... 60
Step 3: Testing the Action ... 61
Workflow Actions: Order of Field Assignments ... 62
Implementing Transitions ... 64
Transitions: General Information ... 64
Transitions: To Implement a Transition Triggered by an Action ... 65
Transitions: Transition Groups ... 68
Transitions: To Implement a Group of Transitions ... 68
Step 1: Adding a Condition ... 69
Step 2: Adding the PendingPayment Workflow State (Self-Guided Exercise) ... 70
Step 3: Grouping Transitions and Adding Conditions to Transitions ... 70
Step 4 (Optional): Specifying the Order of Transitions ... 71
Step 5: Testing Transitions With Conditions ... 72
Defining Conditions ... 73
Conditions: General Information ... 73
Implementing Workflow Dialog Boxes ... 75
Workflow Dialog Boxes: General Information ... 75
Workflow Dialog Boxes: Configuration of Fields ... 76
Workflow Dialog Boxes: Configuration of the Layout ... 78
Workflow Dialog Boxes: To Implement a Transition with a Dialog Box ... 79
Step 1: Defining a Workflow Dialog Box ... 79
Step 2: Defining the Action That Opens the Workflow Dialog Box ... 80
Step 3: Defining the Workflow State and the Transition (Self-Guided Exercise) ... 81
Step 4: Testing the Assign Action ... 81
Configuring Field States ... 83
Field States: General Information ... 83
Field States: Levels of Configuration ... 84

Implementing Workflow Events..... 86
Workflow Events: General Information..... 86
Workflow Events: To Use an Existing Event..... 89
Step 1: Exploring the Acumatica ERP Source Code..... 90
Step 2: Preparing the Project for Debugging..... 90
Step 3: Exploring and Debugging the Code..... 92
Step 4: Defining the Paid Workflow State (Self-Guided Exercise)..... 94
Step 5: Defining the Workflow Event Handler..... 95
Step 6: Overriding the PerformPersist Method..... 96
Step 7: Testing the Transition..... 97
Workflow Events: To Create a Workflow Event..... 98
Step 1: Creating a Custom Field..... 99
Step 2: Deriving the Value of the Field..... 100
Step 3: Exploring the Acumatica ERP Source Code..... 101
Step 4: Declaring the Workflow Event..... 102
Step 5: Firing the Event..... 103
Step 6: Testing the Transition..... 105
Customizing a Predefined Workflow..... 107
Workflow Customization: General Information..... 107
Workflow Customization: To Add an Action to an Existing Workflow..... 108
Step 1: Adding a Graph Action..... 108
Step 2: Adding a Workflow Extension..... 109
Step 3: Defining a Workflow Action..... 110
Step 4: Testing the New Action..... 112
Implementing Composite States..... 113
Composite Workflow States: General Information..... 113
Composite Workflow States: To Update a Composite State..... 116
Step 1: Investigating the Source Code..... 117
Step 2: Extending the ARDocStatus class..... 118
Step 3: Adding the ApproveDiscount Action..... 118
Step 4: Adding the Skip Condition..... 119
Step 5: Adding the Postponed Workflow State to the Composite State..... 119
Step 6: Adding the Transition..... 120
Step 7: Testing the Transition..... 121
Defining Action Sequences..... 122
Action Sequences: General Info..... 122

Action Sequences: To Define an Action Sequence....124
Defining Workflows with a Workflow Identifying Field....129
Workflow-Identifying Fields: General Information.... 129
Workflow-Identifying Fields: To Add a Workflow for a Value of the Workflow-Identifying Field.... 130
Step 1: Adding the OrderType Field and Corresponding Box to the UI.... 132
Step 2: Specifying the Workflow-Identifying Field in the Workflow.... 134
Step 3: Adding a Workflow for the Specific Value of the Workflow-Identifying Field....135
Step 4: Testing the Workflow.... 137
Using Workflow-Identifying Fields of the Second Level.... 138
Workflow-Identifying Fields of the Second Level: General Information....138
Workflow-Identifying Fields of the Second Level: To Define a Workflow.... 140

Copyright

© 2025 Acumatica, Inc.

ALL RIGHTS RESERVED.

No part of this document may be reproduced, copied, or transmitted without the express prior consent of Acumatica, Inc.

3075 112th Avenue NE, Suite 200, Bellevue, WA 98004, USA

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: 2025 R1

Last Updated: 06/01/2025

Getting Started with Workflow API: General Information

A workflow is a depiction of the ways the state of a record created on a particular form changes as a result of specific user interactions on the form and other events. You can use Workflow API to design and tailor the workflows of forms to meet your company's needs.

Learning Objectives

In this topic, you will learn what workflows are and how you can use them.

Applicable Scenarios

You might need to create or customize workflows if the movement of records in the company follows an established sequence of operations. By customizing predefined workflows to represent this sequence, you can automate the company's processes, which can speed the processing of records. You may want to instead develop new workflows if the predefined workflows are not similar enough to the way the entity is processed in your company or the form does not already have a workflow.

Advantages of Workflows

Before workflows were introduced, similar logic was implemented in one of the following ways:

- By using automation steps.

  This functionality has been deprecated since 2023 R1.
- Through complex logic in graph code. Developers used a set of flags to track the record status and implemented event handlers, such as `RowSelected`. In the event handlers, they changed the properties of different elements of the screen, such as fields and actions, depending on the flags' values.

Workflows provide the following advantages over the previous approaches:

- **Shorter, more readable code:** You can use Workflow API to declaratively describe complicated logic that was previously in `RowSelected` event handlers. This makes it much easier to understand the structure and modify the logic—for example, to enable an action for a particular status.
- **Support for no-code customization:** When you implement logic by using Workflow API, users (customizers) can later view and customize it from the Customization Project Editor without coding.
- **Custom actions without changing graph code:** With Workflow API, you can define actions without changing the graph code. Such actions are created within the workflow. Users (customizers) can even create workflow actions in the Customization Project Editor without coding.
- **Conditions:** Workflows introduce conditions. You can change action and field properties dynamically depending on conditions.
- **No-code dialog boxes:** Worfklows introduce a type of dialog boxes that does not require additional code in the graph or on the frontend.

Compared to automation steps, workflows are much easier to modify—for example, when adding a new state. Also, when a default workflow is updated, you can still apply customizations of the workflow unless they introduce breaking changes.

States and Transitions

A workflow can be described as a state machine, with transitions showing the movement of the record 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 user interactions on the *Opportunities* (CR304000) form to reflect the progress made with the potential customer that represents an opportunity.

For details on defining states, see *Defining Workflow States*. For details on defining transitions, see *Implementing Transitions*.

Actions and Fields

You can configure action and field properties for a form and its workflows at the same time, depending on the conditions specified for the form when a record is in any workflow state or in a particular workflow state. For details on actions and their properties, see *Implementing Workflow Actions*.

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. For details on defining and using conditions, see *Defining Conditions*.

Customization and Creation of Workflows

You can customize predefined workflows for forms that have them; the resulting customized workflows are sometimes referred to as *inherited* because they inherit all modifications of the predefined workflow. You can also create custom workflows that are not based on existing workflows. For details on customizing a predefined workflow, see *Customizing a Predefined Workflow*.

You can define a single workflow for the whole form or multiple workflows, one for a record with each value of the specific field that identifies the workflow state, such as the record type. 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 can specify the properties of actions and fields for every workflow state. These properties determine the appearance of the form when the record is in a particular workflow state.

Company Story and Customization Description

In the activities of this guide, you will develop a customization project to support the cell phone repair shop of the Smart Fix company. Parts of this customization were developed in the *T200 Maintenance Forms*, *T210 Customized Forms and Master-Detail Relationship*, and *T220 Data Entry and Setup Forms* training courses. The activities of this guide use the customization project that you can get as a result of completing these courses. You will deploy this project in *Test Instance for Workflow Customization: To Deploy a Test Instance*.

The *T200 Maintenance Forms* training course describes the creation of the following 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, the Smart Fix company manages the list of the devices serviced by the company.

The *T210 Customized Forms and Master-Detail Relationship* course covers the creation of another maintenance form, Services and Prices (RS203000), and the customization of the *Stock Items* (IN202500) form of Acumatica ERP. The Services and Prices (RS203000) form provides users with the ability to define and maintain the price for each repair service the company provides. The *Stock Items* (IN202500) 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, the Repair Work Orders (RS301000) data entry form, which is used to create and manage work orders for repairs, is created. The course also covers the creation of the Repair Work Order Preferences (RS101000) setup form, on which an administrative user specifies the company's preferences for the repair work orders.

In the activities of this guide, you will implement a workflow on the Repair Work Orders (RS301000) form. The workflow will change the state of a record 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. You will also customize the workflow on the *Invoices* (SO303000) form by doing the following:

- Adding an action that opens the Repair Work Orders form. You will make the action available for an invoice in one status.
- Adding the new workflow state, `Postponed`, to a composite state of the workflow, a transition from the `Postponed` workflow state, and an action that triggers the transition.

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 (RS201000) 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 the activities of this guide, you will implement the changing of the status for the *Battery Replacement* and *Liquid Damage* services.

The activities do not cover the implementation of the changing of the status 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 a repair work order for the *Battery Replacement* service on the Repair Work Orders (RS301000) form, the order will have 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, 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*

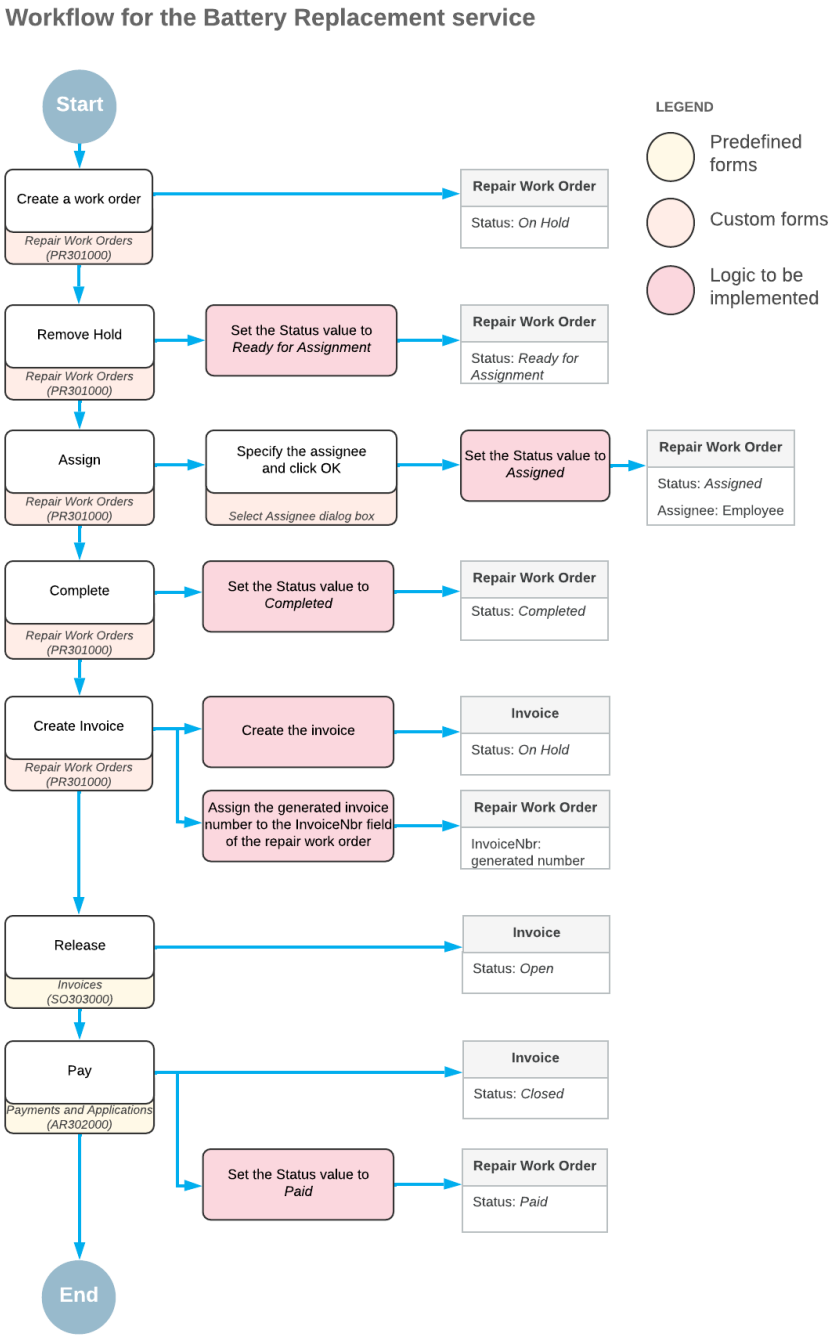
  This status will be assigned when a user clicks the **Remove Hold** button.
- *Assigned*

  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.

> A repair work order for the *Liquid Damage* service requires prepayment if the **Requires Prepayment** check box is selected on the Repair Services (RS201000) form. The required percent to be prepaid has been specified in the **Prepayment Percent** box of the Repair Work Order Preferences (RS101000) form.

To create and apply the prepayment, a user should first create an invoice for the repair work order, and then create, release, and apply the prepayment on the *Payments and Applications* (AR302000) form; the repair work order's status will be changed to *Ready for Assignment*. Then a user will assign the order to an employee, and the order's status will be changed to *Assigned*.

When work on the order is completed, a user can complete the assigned order by clicking the **Complete** button on the **Labor** tab, which will cause the order to be assigned the *Completed* status. After that, a user will apply the rest of the payment to the invoice created 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 *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*

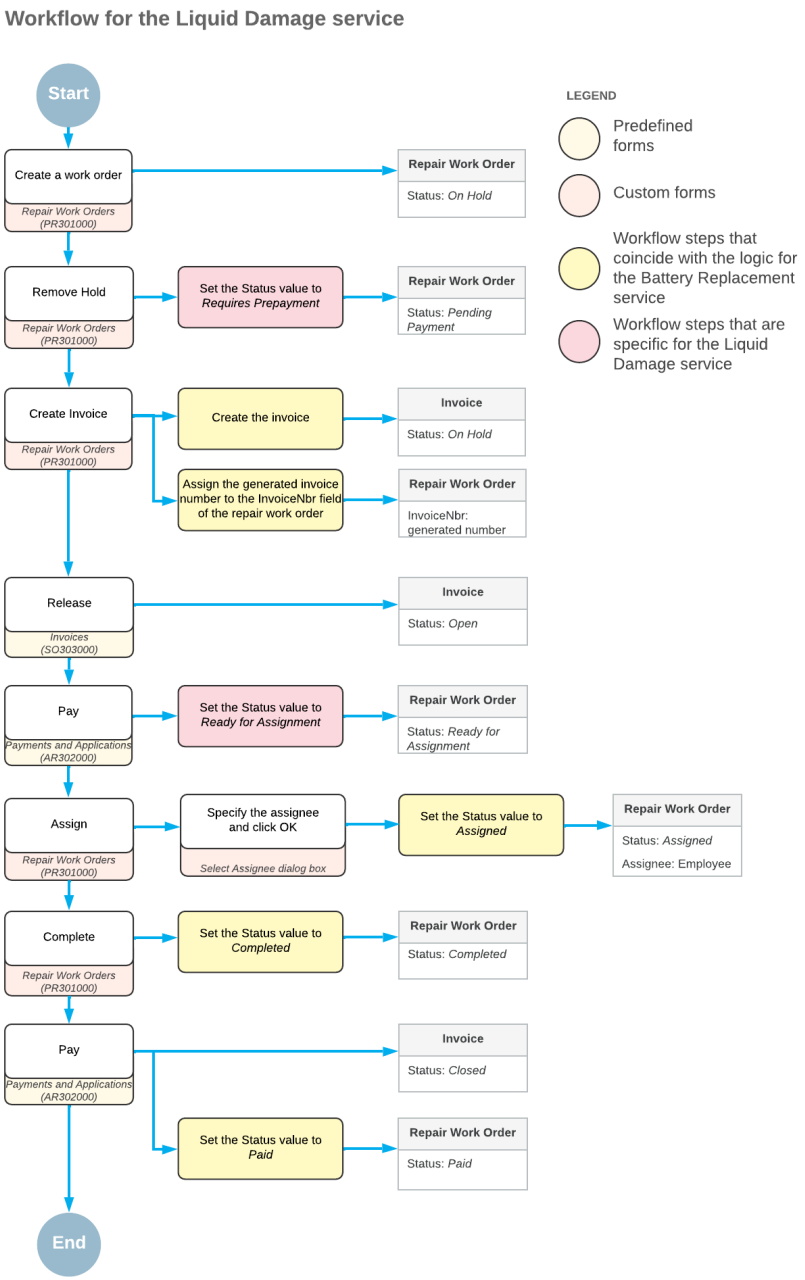
  This status will be assigned to an order when a prepayment for it has been created, released, and applied, and the prepayment percent is greater than or equal to the required prepayment percent.
- *Assigned*

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

Custom Workflow for the Repair Work Orders Form

A review of the company's 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 single workflow, which is shown in the following diagram. In the activities of this guide, you will implement this workflow for the Repair Work Orders (RS301000) form.

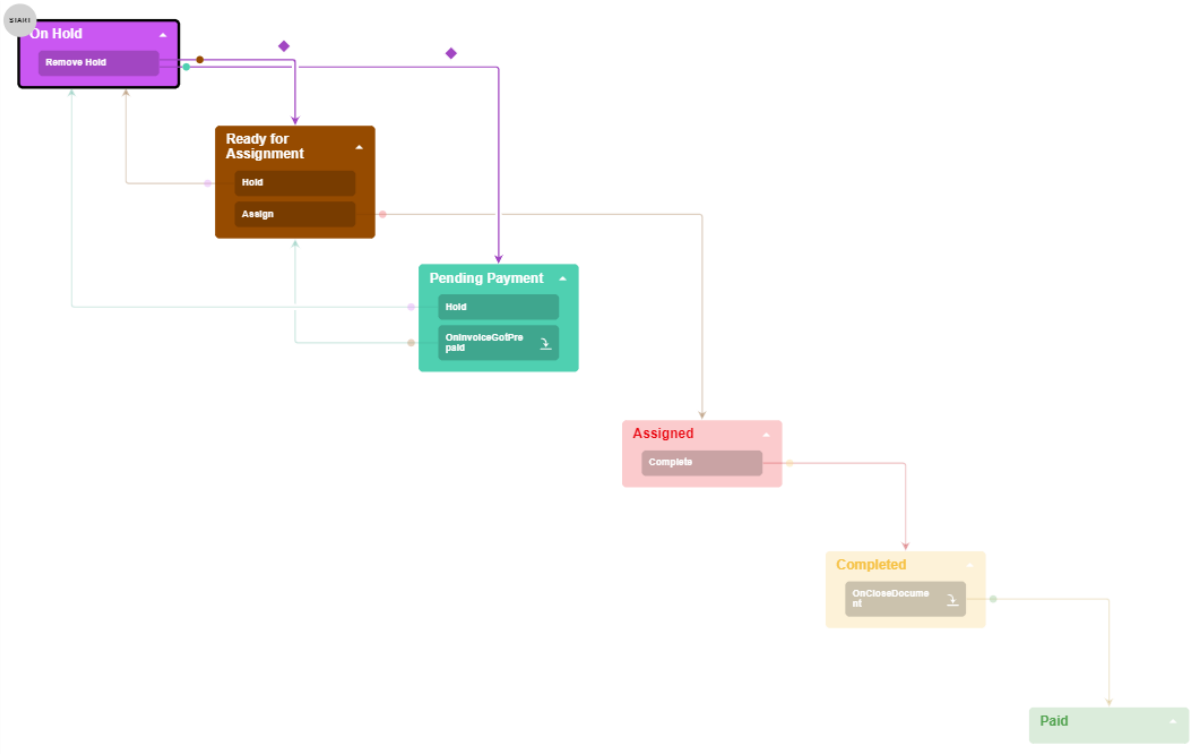


Figure: The workflow on the Repair Work Orders form

In the workflow, you will implement the following items:

- The following states of the workflow, which correspond to the noted statuses of a repair work order:
  - `WorkOrderStatusConstants.OnHold` (*On Hold*)
  - `WorkOrderStatusConstants.PendingPayment` (*Pending Payment*)
  - `WorkOrderStatusConstants.ReadyForAssignment` (*Ready for Assignment*)
  - `WorkOrderStatusConstants.Assigned` (*Assigned*)
  - `WorkOrderStatusConstants.Completed` (*Completed*)
  - `WorkOrderStatusConstants.Paid` (*Paid*)
- The following actions, which trigger the transitions of a repair work order and correspond to the noted command and button on the UI:
  - `ReleaseFromHold` (**Remove Hold**), which triggers a transition from the `OnHold` state to the `PendingPayment` or `ReadyForAssignment` state
  - `Assign` (**Assign**), which triggers a transition from the `ReadyForAssignment` state to the `Assigned` state
  - `Complete` (**Complete**), which triggers a transition from the `Assigned` state to the `Completed` status

  You will also define the `CreateInvoice` action, which generates an invoice for the repair work order.
- The transitions between states of the workflow
- A dialog box that is shown when a user clicks the `Assign` action
- The following workflow event handlers, which trigger transitions for a repair work order:
  - `OnInvoiceGotPrepaid`, which triggers a transition from the `PendingPayment` workflow state to the `ReadyForAssignment` workflow state
  - `OnCloseDocument`, which triggers a transition from the `Completed` workflow state to the `Paid` workflow state
- The conditions that determine for a record created on the form to which workflow state the system should transit from the `OnHold` workflow state

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

The resulting Repair Work Orders (RS301000) form will appear as shown in the following screenshot.

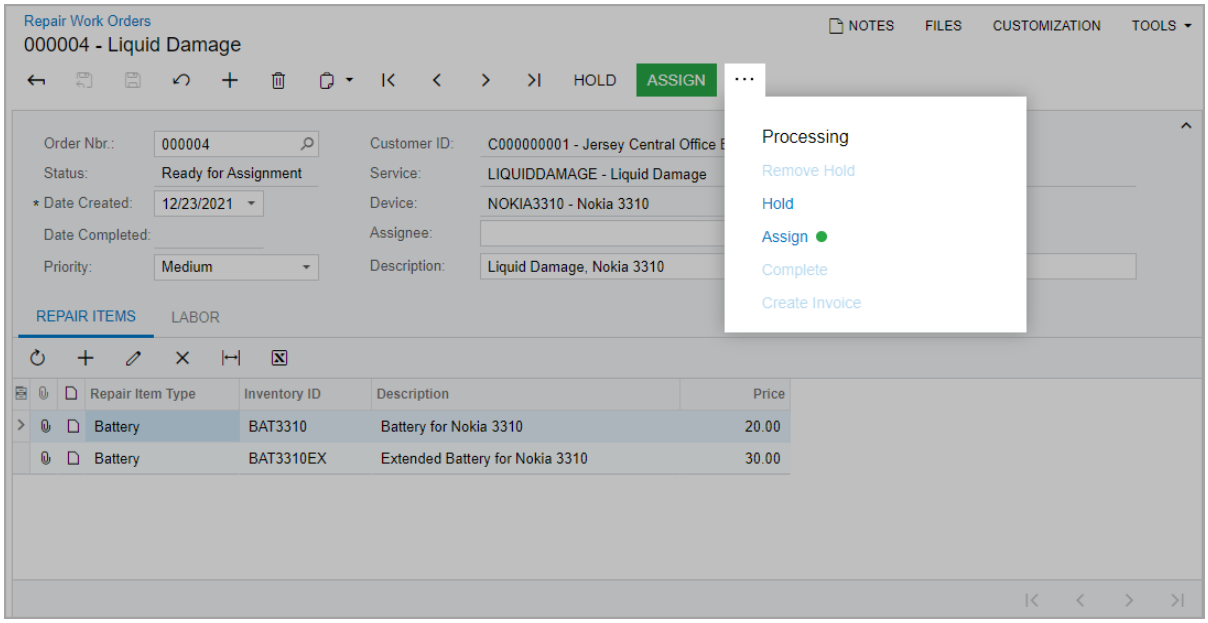


Figure: The resulting form

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. Accordingly, you need to customize the predefined workflow of that form. The command corresponding to the action should be named **View Repair Work Order** and should be displayed on the More menu under the **Repair Work Orders** category, as shown in the following screenshot.

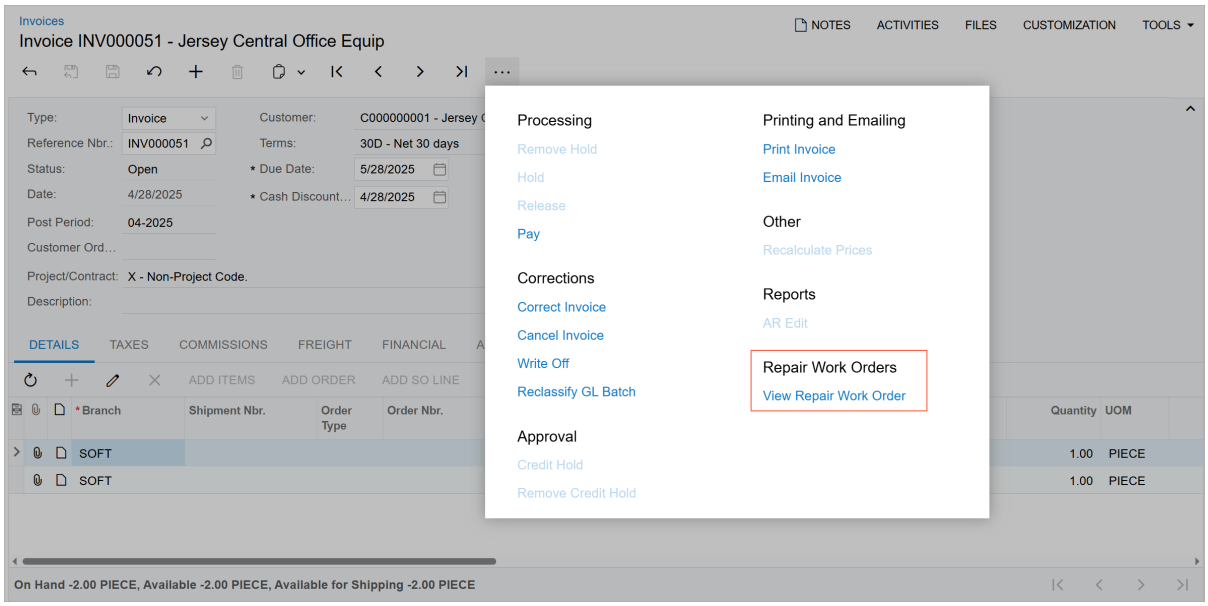


Figure: The View Repair Work Order action

To implement this task, you will extend the graph to define the action and customize the predefined workflow of the *Invoices* (SO303000) 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

Preparing an Instance for Workflow Customization

In this chapter, you will learn how to configure an instance of Acumatica ERP for testing the activities of this guide.

Test Instance for Workflow Customization: General Information

In this chapter, you will learn how to deploy an Acumatica ERP test instance that contains custom and customized forms. You can use this instance to complete a training course or test a scenario described in this guide. You can use this instance to complete the training course. *Custom forms* are forms that are created entirely from scratch by a customizer. *Customized forms* are forms that are based on predefined forms in Acumatica ERP, but are modified by a customizer to fit a specific business process.

Learning Objectives

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

- Prepare the environment
- Deploy an Acumatica ERP test instance

Applicable Scenarios

You deploy an Acumatica ERP test instance by using the instructions in this chapter in the following cases:

- You want to test the activities described in this guide.
- You need to complete a training course.

You deploy an Acumatica ERP test instance by using the instructions in this chapter if you need to complete the training course.

Deploying a Test Instance

You can use the Acumatica ERP Configuration wizard to deploy instances based on predefined datasets. It also makes it possible to deploy test instances that contain custom and customized forms.

In the Acumatica ERP Configuration wizard, you click **Deploy a New Acumatica ERP Instance for T-Series Developer Courses** and select the applicable training course in the list. The instance that you are preparing contains the data that is required to complete the activities of this guide.The instance that you are preparing contains the data that is required to complete a training course. The instance can also include the published customization project.

Test Instance for Workflow Customization: To Deploy a Test Instance

The following activity will walk you through the process of preparing and deploying an Acumatica ERP instance that you can use to test workflow customization.

Story

Suppose that you need to perform customization tasks for the Smart Fix company, as described in *Company Story and Customization Description*. You need to deploy an instance of Acumatica ERP with the *PhoneRepairShop* customization project published.

Process Overview

In this activity, you will install tools that will help you to perform customization tasks and then deploy the instance of Acumatica ERP with the dataset from the *T220 Data Entry and Setup Forms* course.

Step 1: Preparing the Environment

To prepare the environment, do the following:

1. Make sure that the environment that you are going to use conforms to the *System Requirements for the Acumatica ERP Installation*.
2. Make sure that the Web Server (IIS) features that are listed in *Configuration of IIS Web Server 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 Acumatica ERP Setup wizard, 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. You can also install the Acumatica ERP Tools separately. For details, see *Acumatica ERP Installation On-Premises: To Install the Acumatica ERP Tools (Optional)*.

Step 2: Deploying the Instance

To perform customization for the Smart Fix company as described in *Company Story and Customization Description*, you need deploy an instance of Acumatica ERP and publish the customization project prepared for the *T270 Workflow API* training course on the instance.

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 a New Acumatica ERP Instance for T-Series Developer Courses**.
   b. On the **Instance Configuration** page, do the following:
      a. In the **Training Course** box, select *T270 Workflow API*.
      b. In the **Local Path to the Instance** box, select a folder that is outside of the `C:\Program Files (x86)`, `C:\Program Files`, and `C:\Users` folders. (We recommend that you store the website folder outside of these folders to avoid an issue with permission to work in these folders when you customize the website.)
   c. On the **Database Configuration** page, make sure the name of the database is `SmartFix_T270`.

   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 activities of this guide.
2. Make sure that 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 the activities of this guide.

   This is the solution of the extension library that you will modify in the activities of this training course.
3. Sign in to the new tenant by using the following credentials:

- **Username**: `admin`
- **Password**: `setup`

Change the password when the system prompts you to do so.

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

   In subsequent sign-ins to this account, you will be signed in to this branch.

5. Optional: Add the *Customization Projects* (SM204505) and *Generic Inquiry* (SM208000) forms to your favorites. For details about how to add a form to your favorites, see *Favorites: General Information*.

Test Instance for Workflow Customization: To Turn On Workflow Validation

The following activity will walk you through the process of preparing an instance of Acumatica ERP for the validation of a workflow that you create by using Workflow API.

The code written by using Workflow API is declarative. As a result, the usual approaches to debugging do not work with the workflow code. To catch errors that occur in a workflow, you need to turn on workflow validation. This way, when an error occurs on a form that uses the workflow, you can see detailed information about the error on the **System Events** tab of the *System Monitor* (SM201530) form of Acumatica ERP.

Story

Suppose that you need to develop a workflow by using Workflow API. In this case, you need to learn how to debug the code and catch errors.

Process Overview

You will enable workflow validation in the `Web.config` file of your instance.

Step: Enabling Workflow Validation

To enable workflow validation, do the following:

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" />
```

3. Save your changes.

Test Instance for Workflow Customization: To Configure the Instance

Based on the *Customization Description*, you need to configure the instance to be able to test the logic you implement while completing the activities in this guide.

Story

In the Smart Fix company, there are no shipments or sales orders associated with repair work orders. Therefore, you need to configure Acumatica ERP 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.

After you perform the customization tasks, you will be testing your changes, including the **Create Invoice** button or command (which corresponds to the `CreateInvoice` action that you will create) of the Repair Work Orders (RS301000) form. If you click the button multiple times during testing, you may run into an issue with insufficient 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. You need to prevent this issue from occurring.

Process Overview

You will enable the *Advanced SO Invoices* feature on the *Enable/Disable Features* (CS100000) form to make it possible to process repair work orders.

On the *Item Classes* (IN201000) form, you will also change the *STOCKITEM* item class to allow negative quantities for the stock items. This will prevent possible issues during the testing of the **Create Invoice** action.

You will also specify accounts receivable preferences.

Step 1: Turning on the Feature

To turn on the feature, on the *Enable/Disable Features* (CS100000) form, do the following:

1. On the form toolbar, click **Modify**.
2. Select the **Advanced SO Invoices** check box.
3. Click **Enable** on the form toolbar.

Step 2: Allowing 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 the activities of this guide belong to this class.All the stock items used in the activities of this training course belong to this class.
2. On the **General** tab (**General Settings** section), select the **Allow Negative Quantity** check box, as shown in the following screenshot.

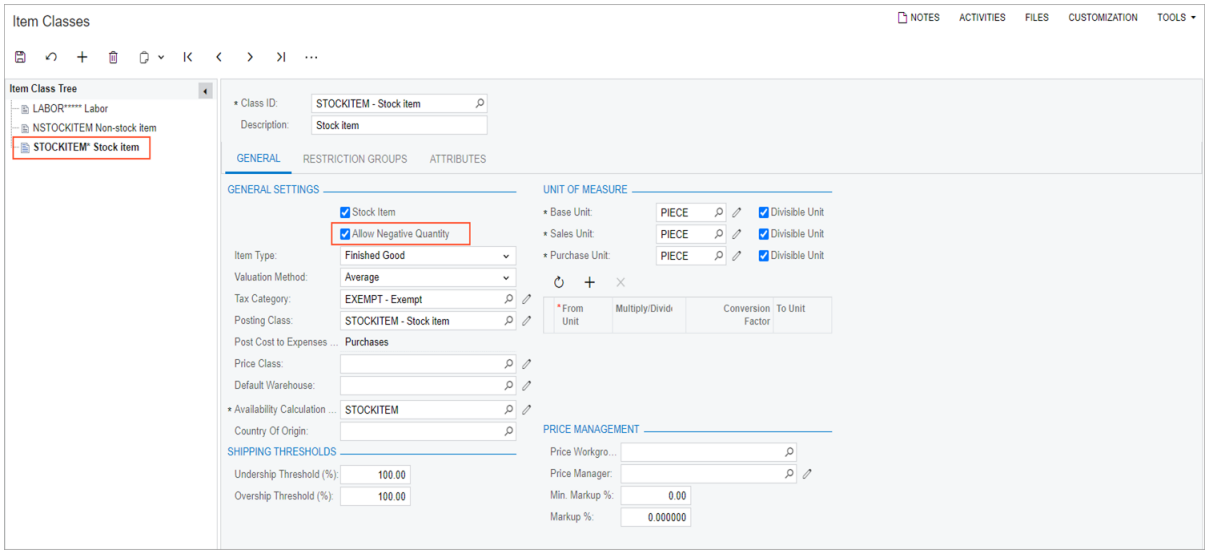


Figure: Item Classes form

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

Step 3: Specifying Accounts Receivable Preferences

Specify accounts receivable preferences as follows:

1. Open the *Accounts Receivable Preferences* (AR101000) form.
2. On the **Data Entry Settings** section of the **General** tab, clear the **Validate Document Totals on Entry** and **Require Payment Reference on Entry** check boxes. This will make it possible for users to skip entering values during the release of an invoice.
3. On the form toolbar, click **Save**.

Preparing a Screen Configuration

A screen configuration corresponds to each Acumatica ERP form and consists of the configuration of the elements of the screen (form), including the definition of the form's workflow or workflows. In this chapter, you will learn what the components of a screen configuration are and how to prepare a graph extension and a screen configuration for developing a workflow.

Screen Configuration: General Information

Each form of Acumatica ERP (including existing and custom forms) includes a screen configuration—that is, the configuration of the screen that corresponds to the form. The screen configuration contains the configuration of the components of the particular screen, such as actions and fields, and the definitions of workflows.

When a form is created, a screen configuration of this form is empty. To define a workflow or to add configuration of components of a screen, you need to edit the screen configuration of the corresponding form. Some of the existing forms of Acumatica ERP have predefined workflows, so their screen configuration is not empty.

In this chapter, you will edit a screen configuration and start defining a workflow.

Learning Objectives

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

- Create a graph extension where a workflow can be implemented
- Determine the workflow's set of states
- Define the set of states for the workflow
- Override the `Configure` method that contains the screen configuration
- Locate the class where the predefined workflow is implemented

Applicable Scenarios

You will work with a screen configuration in the following cases:

- You want to create a new workflow. You create a workflow when the form you want to customize does not have one, or when the existing workflow does not fit your business processes.
- Update (that is, create an extension of) a predefined workflow. Customizing a predefined workflow can save you time over creating a custom workflow from scratch, especially if you want to make only minor changes to the functionality of the predefined workflow.
- You want to add or update components of a screen configuration, such as actions or fields. You can configure actions and states of fields in a screen configuration even if they are not used in a workflow.

Workflow Implementation

A workflow class is a class where the workflow is implemented. A workflow class is 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 a workflow class as an extension of the `OpportunityMaint` graph. If you need customize an existing workflow, you create a second-level graph extension of the existing workflow class and the graph of the form.

The first step in creating a workflow is determining the set of states that a record created on the form can have in the workflow. The workflow state is determined by one of the fields of the record, which is usually the `Status` field. Each workflow state is defined by a combination of a constant string value and a class derived from the

`BqlString.Constant` class. The name of the class starts with a lowercase letter. For an example, see *Step 2: Defining the Set of States of the Workflow* in the *Screen Configuration: To Prepare a Screen Configuration for a Form Without a Predefined Workflow* activity.

To implement a workflow, you override the `PXGraphExtension.Configure` method, which accepts the `PXScreenConfiguration` instance. Inside the `Configure(PXScreenConfiguration config)` method, you call the static `Configure` method. In the static `Configure` method, you get the configuration context object, which you use to add the new screen configuration.

> ⚠ Declaring the static `Configure` method and calling in it the overridden `Configure(PXScreenConfiguration config)` method is important to ensure that no static methods are called inside the `Configure(PXScreenConfiguration config)` method and the `PXOverride` attribute cannot be applied to the method. For more details, see *Screen Configuration: Restrictions of the Configure Method*.

Inside the static `Configure` method, you define a workflow by using Workflow API (which is also called the *screen configuration API*). Workflow 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.

Components of a Screen Configuration

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

- *Conditions*: To define conditions in a screen configuration, you use the `WithConditions` method. For details, see *Defining Conditions*.
- *Dialog boxes*: To define dialog boxes in a screen configuration, you use the `WithForms` method. For details, see *Implementing Workflow Dialog Boxes*.
- *Fields*: To define the states of fields in a screen configuration, you use the `WithFieldStates` method. For details, see *Configuring Field States*.
- *Actions and their categories*: To define actions and action categories, you use the `WithActions` and `WithCategories` methods. For details, see *Implementing Workflow Actions*.
- *Workflow event handlers*: To define workflow event handlers, you use the `WithHandlers` method. For details, see *Implementing Workflow Events*.
- *Any number of workflows*. For each workflow, you define the following:
  - The state-identifying field, which is the field that holds the state value for the workflow definition. You specify this field by calling the `StateIdentifierIs` method and providing the field name as a generic parameter. For an example, see *Step 3: Overriding the Configure Method* in the *Screen Configuration: To Prepare a Screen Configuration for a Form Without a Predefined Workflow* activity.

    > ⚠ If you have specified the state identifier field, you need to add a workflow by calling either the `AddDefaultFlow` method or the `AddWorkflow` method.

  - Workflow states, which can include action definitions and field states

    To define workflow states, you use the `WithStates` method. For details, see *Defining Workflow States*.
  - Workflow transitions, which can be triggered based on conditions

    To define transitions, you use the `WithTransitions` method. For details, see *Implementing Transitions*.

  For workflow other than the default workflow, you also define the workflow-identifying field. This field holds the value that determines the workflow to be used.

The following diagram shows all components of the screen configuration.

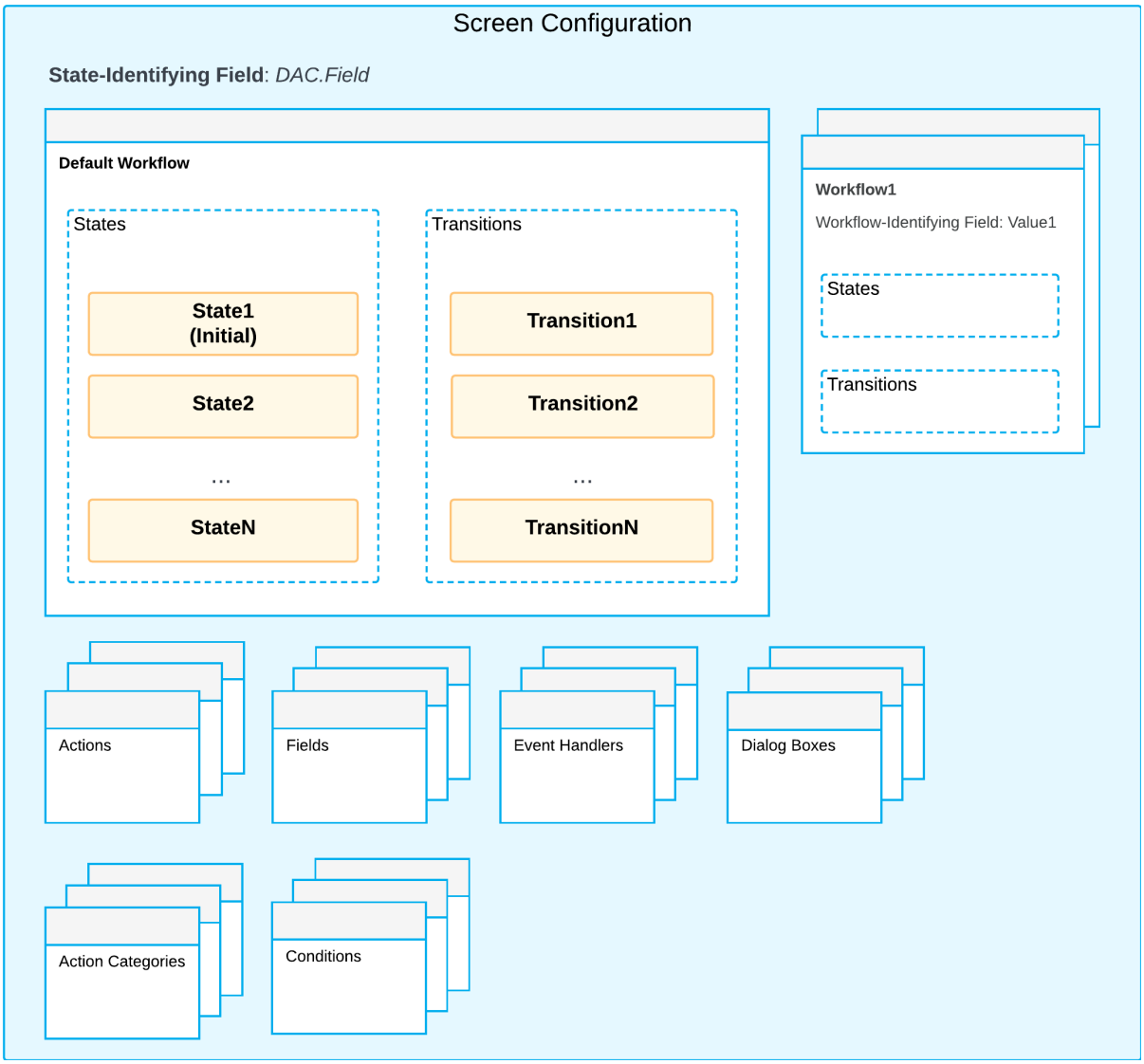


Figure: Components of the screen configuration

Workflow Inheritance

In a customization project, sometimes a derived (or inherited) graph needs to inherit the behavior of the workflow that is defined in its parent graph because similar workflow behavior is needed in the derived graph. To cause the descendant graph to inherit the workflow behavior, you should specify the *`PXWorkflowInheritance`* attribute on the descendant graph. As the result, the descendant graph will utilize all workflow extensions of its parent. When a descendant graph is marked with the `PXWorkflowInheritance` attribute, you can define workflow extension for this exact descendant, and this extension will be applied strictly after all the extensions of the parent class.

Screen Configuration: To Prepare a Screen Configuration for a Form Without a Predefined Workflow

This activity will walk you through the process of performing the initial steps for implementing a workflow for a form that has no predefined workflows.

Story

Suppose that you need to define a workflow for the Repair Work Orders (RS301000) form, which has no predefined workflows.

Process Overview

In this activity, you will prepare the basic components of a workflow by performing the following steps in the extension library:

1. Defining the graph extension where the workflow will be implemented
2. Defining a list of the states of the workflow
3. Overriding the screen configuration method
4. Specifying the state-identifier field

System Preparation

Configure your instance by performing the following prerequisite activities:

1. *Test Instance for Workflow Customization: To Deploy a Test Instance*
2. *Test Instance for Workflow Customization: To Turn On Workflow Validation*

Make sure that you have the Repair Work Orders (RS301000) form available in the system and located in the Phone Repair Shop workspace.

Step 1: Defining the Workflow Class

In this step, you will define the `RSSVWorkOrderEntry_Workflow` class that will contain a workflow for the Repair Work Orders (RS301000) form.

The business logic for the Repair Work Orders (RS301000) form is defined in the `RSSVWorkOrderEntry` graph, so you need to create the `RSSVWorkOrderEntry_Workflow` class as an extension of the `RSSVWorkOrderEntry` graph.

To define the `RSSVWorkOrderEntry_Workflow` workflow class, do the following:

1. In the `PhoneRepairShop_Code` project, create the `Workflows` folder.
2. In the `Workflows` folder, add a new item named `RSSVWorkOrderEntry_Workflow.cs` based on the C# class template.
3. In the `RSSVWorkOrderEntry_Workflow.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
{
    public class RSSVWorkOrderEntry_Workflow : PXGraphExtension<RSSVWorkOrderEntry>
    {

    }
```

```
}
```

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

Step 2: Defining the Set of States of the Workflow

In this step, you will define the set of states for the workflow of a repair work order. The workflow state is determined by one of the fields of the record, 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 *Business Process Overview*, 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)

To create the set of workflow states, do the following:

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

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 3: Overriding the Configure Method

To implement a workflow for the Repair Work Orders (RS301000) form, in the workflow class, you will override the `PXGraphExtension.Configure` method, which accepts the `PXScreenConfiguration` instance. Inside the `Configure(PXScreenConfiguration config)` method, you will call the static `Configure` method. In the static `Configure` method, you will get the configuration context object for the Repair Work Orders (RS301000) form, which you will later use to add the new screen configuration.

Declaring the static `Configure` method and calling in it the overridden `Configure(PXScreenConfiguration config)` method is important to ensure that no static methods are called inside the `Configure(PXScreenConfiguration config)` method and the `PXOverride` attribute cannot be applied to the method. For more details, see *Screen Configuration: Restrictions of the Configure Method*.

Do the following:

1. In the `RSSVWorkOrderEntry_Workflow` class, declare the static `Configure` method as the following code shows.

```
protected static void Configure(WorkflowContext<RSSVWorkOrderEntry,
                                                 RSSVWorkOrder> context)
{
}
```

   In the code above, the `Configure` method has one parameter of the `WorkflowContext` type, which has two type parameters: the graph of the form for which workflow is defined, and the primary DAC of the form.

2. Override the `Configure(PXScreenConfiguration config)` method and call the static `Configure` method inside it, as the following code shows.

```
public sealed override void Configure (PXScreenConfiguration config)
{
    Configure(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. In the static `Configure` method, 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 (RS301000) form, specified the field that is the state identifier and started adding the default workflow by calling the `AddDefaultFlow` method.

4. Save your changes.

Now you are ready to add the definition of the workflow to the screen configuration.

Screen Configuration: To Prepare a Screen Configuration with a Predefined Workflow

Most data entry forms of Acumatica ERP have at least one *predefined* workflow—that is, a workflow that is implemented in the original code of Acumatica ERP.

The following activity will walk you through the process of performing the initial steps for customizing a predefined workflow.

Learning Objectives

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

- Locate a class where the predefined workflow is implemented
- Create a graph extension where a customized workflow can be implemented

- Override the `Configure` method

Story

Suppose that you need to define a workflow for the *Invoices* (SO303000) form, which has a predefined workflow—that is, a workflow that has been implemented by Acumatica developers.

Process Overview

To extend the predefined workflow of the *Invoices* (SO303000) form, in the source code of Acumatica ERP, you will first find the class where the workflow is defined. You will then create an extension of this class and prepare the method where you can later customize the workflow.

System Preparation

As a prerequisite activity, perform the activities of the *Preparing an Instance for Workflow Customization* chapter to configure your instance.

Make sure you have enabled the workflow validation as described in *Test Instance for Workflow Customization: To Turn On Workflow Validation*.

Step 1: Locating the Class that Defines the Workflow

To locate the class where the workflow is defined, you should look into the `PX.Objects/<area>` folder in the source code of Acumatica ERP, where `<area>` is the code of the functional area of the form for which the workflow is defined. In most cases, workflows of this functional area are located in the `Workflows` folder of this location. Sometimes all workflow classes are located at the same level as the graph classes—that is, in the `PX.Objects/<area>` folder.

To locate the class where the workflow for the *Invoices* (SO303000) form is defined, do the following:

1. In the source code of Acumatica ERP, locate the `PX.Objects/SO/Workflow` folder.
2. In this folder, find the `SOInvoiceEntry_Workflow` class, 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 if you want to use the states of the workflow in your code.

Step 2: Extending the Predefined Workflow

To extend a predefined workflow, you need to create an extension of the class where the workflow is defined. In this extension, you need to override the `Configure(PXScreenConfiguration)` method, which initializes the screen configuration.

To extend a predefined workflow, do the following:

1. In the `Workflows` folder of the `PhoneRepairShop_Code` project, create the `SOInvoiceRepairOrder_Workflow.cs` file.
2. In the `SOInvoiceRepairOrder_Workflow.cs` file, define the `SOInvoiceRepairOrder_Workflow` class, as the following code shows.

   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 have declared the `Configure(WorkflowContext)` method, where you can later update the workflow.

3. Make sure that you have added the needed `using` directives.
4. Save your changes.

Screen Configuration: Restrictions of the Configure Method

You should be aware of the limitations of the `Configure` method when you are working with it. These limitations are described in the following sections.

Creation of Graph Instances and Base Graph Access

Inside the `Configure` method, you should not create a graph instance or access the `Base` graph because either of these actions can lead to deadlocks. If you need to read data from the database, you can use database slots by doing the following:

1. In the graph extension where you need to define the `Configure` method, you also define a database slot that is based on the DAC from which you need to read data. In the slot, you cache the data that you need to access.
2. You attach the `PXWorkflowDependsOnType` attribute to the `Configure` method and specify the DAC from which you want to read data.
3. In the `Configure` method, you access the data from the slot.

For example, suppose that you need to enable the extension in the `Configure` method only if the `InspectionEnabled` property of `SOSetup` DAC is *true*. The following code shows how you can access the `InspectionEnabled` property by using a database slot.

```
private class SOSetupInspection : IPrefetchable
{
    public static bool InspectionEnabled =>
        PXDatabase.GetSlot<SOSetupInspection>("SOSetupInspection",
 typeof(SOSetup))._inspectionEnabled;
    private bool _inspectionEnabled;
    void IPrefetchable.Prefetch()
    {
        using (PXDataRecord soSetup =
            PXDatabase.SelectSingle<SOSetup>(new PXDataField<SOSetup.inspectionEnabled>()))
            if (soSetup != null) _inspectionEnabled = (bool)soSetup.GetBoolean(0);
    }
}

[PXWorkflowDependsOnType(typeof(SOSetup))]
public sealed override void Configure(PXScreenConfiguration config){
     //if (!Base.sosetup.Current.InspectionEnabled) return;
     if (!SOSetupInspection.InspectionEnabled) return;
     ...
}
```

The code above has done the following:

1. In the extension, defined the `SOSetupInspection` database slot, which depends on the `SOSetup` table. In the slot, the `SOSetup.InspectionEnabled` value has been cached.

2. Attached the `PXWorkflowDependsOnType` attribute to the `Configure` method and specified that the workflows of the form depend on data from the `SOSetup` DAC.
3. In the `Configure` method, instead of accessing the `Base` graph, checked the `SOSetupInspection.InspecionEnabled` property, which holds the cached value of the `SOSetup.InspectionEnabled` property.

For more information on database slots, see *Use of Slots to Cache Data Objects*.

Customization with the `PXOverride` Mechanism

The `Configure` method cannot be customized with the `PXOverride` mechanism, despite being a `public virtual` extension method. The place in the platform where the `Configure` method is called does not support `PXOverride` extensions of the method. Thus, we recommend that you explicitly use the `sealed` modifier in C# when you are overriding the `Configure` method. The following code shows the recommended way of overriding the `Configure` method.

```
//Use sealed modifier explicitly to specify that this method cannot be overridden

public sealed override void Configure(PXScreenConfiguration config) =>
Configure(config.GetScreenConfigurationContext<TGraph, TPrimaryEntity>());

//Replace the TGraph and TPrimaryEntity above with your actual graph and DAC classes,
 respectively
```

Access to an Instance State

The `Configure` method is executed on uninitialized graphs and graph extensions. You cannot access an instance state of a graph or a graph extension within the `Configure` method or within any methods called from it. This also means that you cannot access any instance fields or properties, including views and actions, of a graph or a graph extension, within the `Configure` method. You should thus configure a workflow in a static context, which means configuring it in a static method. This technique is illustrated in the following sample code, which expands on the code example in the preceding section of this topic.

```
public sealed override void Configure(PXScreenConfiguration config) =>
 Configure(config.GetScreenConfigurationContext<TGraph, TPrimaryEntity>());

 //Apply workflow changes in a static method

 //Replace the TGraph and TPrimaryEntity below with your actual graph and DAC classes,
 respectively

 protected static void Configure(WorkflowContext<TGraph, TPrimaryEntity> context)
     {
     ...
     context.AddScreenConfigurationFor(screen => screen
                                          ...
                                       );
     }
```

In addition to the restrictions regarding access to an instance state, current best practices forbid calling any instance methods from the `Configure` method. However, it should be noted that calling instance methods from the `Configure` method will not cause an error by itself, whereas accessing the instance state will lead to a runtime error.

Graph Class Hierarchy

The `Configure` method is present both in graphs and graph extensions. The methods have the same signature. However, there are notable differences in how they behave at runtime. The `Configure` method in a graph is treated more like a static method than an instance virtual method by the Acumatica Framework. You can notice this phenomenon when working with a graph class hierarchy. Suppose that you have a `Base` graph and a `Derived` graph that is derived from the `Base` graph. The `Base` graph declares a workflow as shown in the following code.

```
public class Base : PXGraph<Base>
{
    public override void Configure(PXScreenConfiguration screenConfig)
    {
         //Some workflow logic here
    }
}
```

The code above has defined the screen configuration for the base graph. However, this `Configure` method will not be called for the `Derived` graph, and its screen configuration will remain empty because the system ignores all derived graphs while processing screens. Notice that the `Configure` method is not sealed in the `Base` graph. To make sure that the `Configure` method in the `Base` graph gets called for the `Derived` graph, you might attempt to declare a trivial override of the `Configure` method in the `Derived` graph. An example is shown in the following code.

```
 public class Derived : Base
{
    public override void Configure(PXScreenConfiguration screenConfig)
           => base.Configure(screenConfig);
}
```

It is important to note that overriding a graph's `Configure` method in a derived graph, as shown in the preceding code example, is not allowed. We recommend that you explicitly seal the `Base` graph's `Configure` method to prevent such override attempts. When the `Configure` method is used in graph extensions, the Acumatica Framework treats it like a normal virtual method. The caveats discussed above with regard to the `Configure` method in a graph do not apply in this instance, but you still should avoid relying on the `Configure` method's virtuality in a graph extension. You should instead explicitly seal the `Configure` method in graph extensions as well.

Order of Execution

The `Configure` method override in a graph is always executed first. The order of execution for multiple `Configure` methods declared in several graph extensions is defined by the way the graph extensions are chained and their explicit sort order. For more details, see *To Sort Multiple Generic Graph Extensions*

Defining Workflow States

In this chapter, you will learn how to define states of a worfklow. The states in a workflow for an Acumatica ERP form represent the stage of processing of a record that is created on this form.

Workflow States: General Information

A *workflow state* is a stage within a workflow of processing of a record that is created on an Acumatica ERP form. A workflow state represents the current status of a record.

Learning Objectives

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

- Define the set of states of the workflow
- Add states to the screen configuration
- Configure states in a workflow

Applicable Scenarios

You define and configure states of a workflow when you are defining a new workflow or customizing an existing workflow.

State-Identifying Field

Each workflow consists of multiple workflow states. Each workflow state corresponds to a value of the state-identifying field.

Currently, the workflow engine supports only state-identifying fields of the `String` type that have a predefined set of values. The values of the state-identifying field are usually defined in the *`PXStringList`* attribute. To define the state-identifying field, you need to call the `StateIdentifierIs` method for the new screen configuration and provide the field as the parameter. For details, see *Step 3: Overriding the Configure Method* of the *Screen Configuration: To Prepare a Screen Configuration for a Form Without a Predefined Workflow* activity.

Adding of a Workflow State

To define the set of workflow states, you call the `WithFlowStates` method in the lambda expression specified for the `AddDefaultFlow` method and provide the set of workflow states in the parameter, as the following code shows. You define each workflow 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 workflow state.

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

> ⚠ To add a workflow state to an existing workflow, you call the `Add` method in the `UpdateDefaultFlow` method inside the `UpdateScreenConfigurationFor` method.

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

Configuration of Workflow States

You can add, update, or delete workflow states in a screen configuration by calling the `Add`, `Update`, or `Delete` method, respectively.

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

- Specify that a workflow state should be the initial state of the workflow by calling the `IsInitial` method.

  > ⓘ Only one state can be the initial state of a workflow.

- Specify the properties that the specified DAC fields should have in this workflow state by calling the `WithFieldStates` method.

  You can specify such properties as requirement, visibility, and availability. Settings specified in the `WithFieldStates` method override the values of parameters of the DAC field attributes. For details about specifying field properties for a workflow state, see *Workflow States: Configuration of Field States*.
- Specify which actions are available in this workflow state and configure the appearance of these actions in this state, such as whether the action should be duplicated on the form toolbar. You do this by calling the `WithActions` method. For details on configuration of actions for a workflow state, see *Workflow States: Configuration of Actions*.
- Specify which workflow event handlers are available in this workflow state by calling the `WithEventHandlers` method. For details about workflow event handlers, see *Workflow Events: General Information*.
- Specify a list of field assignments that will be performed when a record enters the workflow state and when a record leaves the workflow state.

  To provide the list of fields whose values should be assigned when a record enters the workflow state, you use the `WithOnEnterAssignments` method when defining the workflow state. The field assignments listed in the method are applied to the workflow state for which the method was called.

  To provide the list of fields whose values should be assigned to a state when a record leaves the workflow state, you use the `WithOnLeaveAssignments` method when defining the workflow state. The field assignments listed in the method are applied to the workflow state for which the method was called.

  An example of using these methods is shown in the following code.

```
flowStates.Add<State.hold>(flowState =>
{
  return flowState
    .WithActions(actions => …);
    .WithOnEnterAssignments(fields => fields.Add<inclCustOpenOrders>(false));
    .WithOnLeaveAssignments(fields => fields.Add<inclCustOpenOrders>(true));
    ...
}
```

  In the code above, the `inclCustOpenOrders` field is assigned *false* when the record enters the `hold` workflow state (after the transition to this state is completed), and the `inclCustOpenOrders` field is assigned *true* before the `hold` workflow state is changed to some other state (before the transition is started).

Workflow States: Configuration of Field States

You can configure a state of a field in a screen configuration and in a workflow state.

The description of field states in a workflow state is applied after the system applies the description of field states in the screen configuration and in the DAC so the configurations described in the workflow state can override other field state configuration. For example, if a field was visible under a condition in a screen configuration, it can be made invisible in a workflow state.

For details on configuring the state of a field in the screen configuration, see *Configuring Field States*.

Adding or Updating a State of a Field in a Workflow State

To add or update a state of a field in a workflow state, you call the `WithFieldStates` method in a lambda expression for the `Add` or `Update` method that defines a workflow state. Inside the `WithFieldStates` method, you can do the following:

- Add a single field or all fields from a specified DAC by using the `AddField`, `AddAllFields`, or `AddTable` method. When adding a field, you can specify how the workflow will affect this field in this state.

  For example, the following code adds fields and tables to a workflow state and specifies how these fields and tables should be affected (in this case, hidden).

```
.WithFieldStates(fields => {
  fields.AddField<printed>(c => c.IsHidden());
  fields.AddField<emailed>(c => c.IsHidden());
  fields.AddField<dontEmail>(c => c.IsHidden());
  fields.AddField<dontPrint>(c => c.IsHidden());

  fields.AddTable<SOOrder>(c => c.IsHidden());
  fields.AddTable<SOLine>(c => c.IsHidden());
  fields.AddTable<SOLineSplit>(c => c.IsHidden());
  fields.AddTable<SOAddress>(c => c.IsHidden());
}
```

  If you need to add multiple fields or multiple tables, you can use the `AddFields`, `AddAllFieldsFromTables`, and `AddTables` methods. For example, the following code adds the fields and tables from the previous code example by using only two method calls. If the fields and properties have the same configuration, such as being hidden, it can be assigned to all of them in a single method.

```
.WithFieldStates(fields => {
  fields.AddFields<BqlFields.FilledWith<printed, emailed, dontEmail, dontPrint>>(c =>
    c.IsHidden());
  fields.AddTables<BqlTables.FilledWith<SOOrder, SOLine, SOLineSplit, SOAddress>>(c
 =>
    c.IsHidden());
}
```

  > If you need to configure the state of a DAC, use the `AddTable` or `AddTables` method. If you need to configure the states of all fields inside the DAC, use the `AddAllFields` or `AddAllFieldsFromTables` methods. If you use the `AddTable` or `AddTables` method, you can also restrict adding or removing records from the specified tables.

- Update a field state by calling the `Update` method.
- Replace the entire configuration of the field state by calling the `Replace` method.

- Delete the entire configuration of the field state by calling the `Remove` method.

Configuring a State of a Field in a Workflow State

You can configure a field state in the following ways:

- Configure the appearance and requirement of the field (or all fields in a DAC) by calling the following methods:
  - `IsHidden`
  - `IsDisabled`
  - `IsRequired`
- For a combo box field, specify which values of the combo box field should be available by calling the `ComboBoxValues` method.

  Only the value defined in the state of a field in a screen configuration can be restricted. For details on defining the list of combo box values in a screen configuration, see *Field States: General Information*.
- Specify a default value for the field by calling the `DefaultValue` method.

  Settings a default value is available only for the initial state of a workflow.

An example of configuring field states in a workflow state is shown in the following code.

```
fss.Add<State.onHold>(flowState =>
{
  return flowState
    .WithFieldStates(states => {
      states.AddField<dontEmail>(state =>
        state.DefaultValue(true));
      states.AddTable<ARInvoice>(state =>
        state.IsDisabled());
    });
});
```

In the example above, field states have been configured in the following ways:

- The `dontEmail` field has been assigned a default value of *true*.
- The `ARInvoice` DAC has been disabled.

Workflow States: Configuration of Actions

You can configure an action in the following locations:

- A definition of the action in a graph or in one of its extensions
- A screen configuration
- A workflow state

This topic describes the configuration of actions in a workflow state. For details on configuring actions in a screen configuration, see *Workflow Actions: Configuration of Actions*.

Adding or Updating an Action in a Workflow State

To add or update an action in a workflow state, you call the `WithActions` method in a lambda expression for the `Add` or `Update` method that defines a workflow state. Inside the `WithActions` method, you can do the following:

- Add a new action to the workflow state by calling the `Add` method.

  > ⚠ When you add an action in a workflow state, the action will be available in this state unless it is restricted by the settings defined in the screen configuration,—for example, by the condition in the `IsHiddenWhen` method.

- Update an action if it exists in the state of the base workflow by calling the `Update` method.
- Replace the entire configuration of the action by calling the `Replace` method.
- Delete the entire configuration of the action by calling the `Remove` method.

Configuring an Action in a Workflow State

You can configure an action in the following ways:

- Display the button that corresponds to the action on the form toolbar.

  By default, all commands, which correspond to actions, are displayed on the More menu. In addition to the command on the More menu, you can display on the form toolbar the button that corresponds to the action by calling the `IsDuplicateInToolbar` method while adding the action to a workflow state.
- Specify a connotation for the button and command by calling the `WithConnotation` method while adding the action to a workflow state. The connotation will be displayed both for the command on the More menu and for the button on the form toolbar.

  You can also cancel the connotation by calling the `WithNoConnotation` method.
- Specify that the action should be executed automatically when a provided condition is true by calling the `IsAutoAction` method.

An example of configuring an action in a workflow state is shown in the following code.

```
sss.Add<State.hold>(flowState =>
{
  return flowState
    .WithActions(actions =>
    {
      actions.Add(graph => graph.releaseFromHold,
        action => action.IsDuplicatedInToolbar()
              .WithConnotation(ActionConnotation.Success));
    });
});
```

In the example above, the appearance of the button and command that correspond to the `releaseFromHold` action has been adjusted as follows:

- The button that corresponds to the action is displayed on the form toolbar.
- The button on the form toolbar and the command on the More menu are displayed with the `Success` connotation.

Workflow States: To Define a Workflow State

The following activity will walk you through the process of defining a workflow state.

Story

Suppose that on the Repair Work Orders (RS301000) form, you need to add the `OnHold` and `ReadyForAssignment` states to the workflow. The `OnHold` workflow state should be the initial state of the workflow.

Process Overview

To define the workflow states, in the `Configure` method of the `RSSVWorkOrderEntry_Workflow` class, you will call the `WithFlowStates` method in the lambda expression specified for the `AddDefaultFlow` method and provide the set of states in the parameter. You will define each state by calling the `Add` method and specifying the state-identifying string as a generic parameter. As a parameter of the `Add` method, you will specify a lambda expression where you will configure the workflow state.

System Preparation

Make sure that you have done the following:

1. Prepared an instance with the *PhoneRepairShop* customization project and enabled the workflow validation by performing the following prerequisite activities:
   a. *Test Instance for Workflow Customization: To Deploy a Test Instance*
   b. *Test Instance for Workflow Customization: To Turn On Workflow Validation*
2. Prepared the screen configuration and defined the set of states by performing the *Screen Configuration: To Prepare a Screen Configuration for a Form Without a Predefined Workflow* prerequisite activity.

Step 1: Defining the Initial Workflow State

To define the initial workflow state, which is `OnHold` in this activity, do the following:

1. In the `RSSVWorkOrderEntry_Workflow` class, in the static `Configure` method, locate the `AddDefaultFlow` method. (You have added this method in *Step 3: Overriding the Configure Method* of the *Screen Configuration: To Prepare a Screen Configuration for a Form Without a Predefined Workflow* activity.)
2. Modify the `WithFlowStates` method, as the following code shows.

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

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

```
                        flowStates.Add<States.onHold>(flowState =>
                            GetOnHoldBehavior(flowState));
```

   In the code above, you have added the `OnHold` state by specifying the `States.onHold` value as the generic parameter of the `Add` method.

4. In the `RSSVWorkOrderEntry_Workflow` class, add the `Workflow States` region and define the following private static method, which defines the `OnHold` workflow state.

```
#region Workflow States
private static BaseFlowStep.IConfigured GetOnHoldBehavior(
    FlowState.INeedAnyFlowStateConfig flowState)
{
    return flowState
    .IsInitial()
    .WithActions(actions =>
    {
        actions.Add(graph => graph.ReleaseFromHold,
            action => action.IsDuplicatedInToolbar()
                .WithConnotation(
                    ActionConnotation.Success));
    });
}
#endregion
```

In the code above, you have done the following:

- Specified that the `OnHold` workflow state is the initial state of the workflow by calling the `IsInitial` method.
- Specified the actions that are available in the `OnHold` workflow 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` workflow state.
- By calling the `IsDuplicatedInToolbar` method, specified that the action should be shown as a button on the toolbar for this workflow 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 *Action Customization: Connotation for an Action*.

5. Save your changes.

Step 2: Defining the ReadyForAssignment Workflow State

To define the `ReadyForAssignment` workflow state, do the following:

1. Inside the lambda expression of the `WithFlowStates` method, after the `OnHold` workflow state, define the `ReadyForAssignment` workflow state, as the following code shows.

```
flowStates.Add<States.readyForAssignment>(flowState =>
    GetReadyForAssignmentBehavior(flowState));
```

In the code above, you have added the `ReadyForAssignment` workflow state by specifying the `States.readyForAssignment` value as the generic parameter of the `Add` method.

2. In the `RSSVWorkOrderEntry_Workflow` class, in the `Workflow States` region, define the following private static method, which defines the `ReadyForAssignment` workflow state.

```
private static BaseFlowStep.IConfigured GetReadyForAssignmentBehavior(
    FlowState.INeedAnyFlowStateConfig 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());
            });
        }
```

By calling the `WithFieldStates` method, you have specified the states of DAC fields in this workflow state. In this case, you have specified that the `customerID`, `serviceID`, and `deviceID` fields of the `RSSVWorkOrder` DAC should be unavailable in this workflow state.

3. Save your changes.

Workflow States: Tracking of Changes in States

The Acumatica Framework automatically tracks the state changes whenever a transition moves a record from one state to another in a workflow. This provides you with the ability to audit the state changes. To enable this functionality, you must declare the following fields in the underlying DAC of the workflow:

- A field of the `Guid` type that is declared with the *`PXDBStateChangedByID`* attribute.

  The field stores the ID of the user who made the last workflow transition for the data record.
- A field of the `string` type that is declared with the *`PXDBStateChangedByScreenID`* attribute.

  This field stores the screen ID of the application screen on which the last workflow transition occurred for the data record.
- A field of the `DateTime` type that is declared with the *`PXDBStateChangedDateTime`* attribute.

  The field stores the date and time in UTC of the data record's last workflow transition.

These fields are updated whenever a workflow transition happens and the workflow state is changed. The following code shows how to declare these fields in a DAC.

```
[PXDBStateChangedByID()]
public virtual Guid? StateChangedByID { get; set; }
public abstract class stateChangedByID :
    PX.Data.BQL.BqlGuid.Field<stateChangedByID> { }

[PXDBStateChangedByScreenID()]
public virtual string StateChangedByScreenID { get; set; }
public abstract class stateChangedByScreenID :
    PX.Data.BQL.BqlString.Field<stateChangedByScreenID> { }

[PXDBStateChangedDateTime]
public virtual DateTime? StateChangedDateTime { get; set; }
public abstract class stateChangedDateTime :
    PX.Data.BQL.BqlDateTime.Field<stateChangedDateTime> { }
```

For each field, you also need to create the corresponding database column. For details, see *Audit Fields*.

Implementing Workflow Actions

In this chapter, you will learn how to define and configure workflow actions. You can implement actions, specify the categories of the associated commands on the More menu, configure conditional states, and perform field assignments.

Workflow Actions: General Information

This chapter will give you experience with and an understanding of the ways you can define and configure workflow actions.

Learning Objectives

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

- Define and configure a workflow action
- Define a category for an action
- Configure the conditional appearance of an action

Applicable Scenarios

You define a workflow action when you need to add an action to a form that uses a workflow. You can display an action and make it available only in particular states, and you can use an action to trigger a transition from one state to another.

Workflow Actions

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

Workflow actions are actions that are added to the screen configuration and to a workflow. The declaration of a workflow action consists of the following:

- The declaration of an action in a graph.

  You can skip the declaration of an action in a graph if the action does not have complex logic—that is, if the only thing that action does is assign values to fields. In that case, you declare an action in the `Configure` method and define the action's logic by using capabilities of Workflow API.
- The declaration of an action in the screen configuration.

Action Declaration in a Graph

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

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

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

  In the `PXAction<>` type parameter, you should specify the main DAC of the primary data view. Otherwise, the button corresponding to the action cannot be displayed on the form toolbar (and the corresponding command cannot be displayed on the More menu).
- A method that implements the action; this method has the `PXButton` and `PXUIField` attributes. This method has the following forms of declaration:

- Without parameters and returning `void`: This standard form of declaration is shown in the following code example.

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

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

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

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

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

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

For more information about actions, see *Action Definition: General Information* and the *T230 Actions* training course.

Action Declaration 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.
   - You can specify in which category of the More menu the command corresponding to the action should be displayed by calling the `WithCategory` method in a lambda expression. If you do not specify a category for an action, its command will be listed under the **Other** category on the More menu and placed on the form toolbar as a button.

The following code shows an example of an action definition.

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

Workflow Actions: Configuration of Actions

You can configure workflow actions in the following ways:

- Add, update, or delete an action in the screen configuration by calling the `Add`, `Update`, or `Delete` method, respectively, in the `WithActions` method at the screen configuration level.

  For example, see *Step 3: Adding the Action to the Workflow* of the *Workflow Actions: To Implement a Simple Action* activity.
- Add, update, or delete an action in a specific state of a workflow by calling `Add`, `Update`, or `Delete` method, respectively, in the `WithActions` method at the state level. By adding an action to a state, you make it available in this state unless it is restricted by the settings specified in the screen configuration.

  For details on configuring an action in a workflow state, see *Workflow States: Configuration of Actions*.

  For example, see *Workflow States: To Define a Workflow State*.
- Assign field values when the action is performed by calling the `WithFieldAssignments` method while adding or updating the action in the screen configuration. For a detailed example, see *Workflow Actions: To Implement an Action with Field Assignments*.
- Provide the parameters of an action by calling the `WithParameterAssignments` method.
- Display a dialog box when an action is clicked by calling the `WithForm` method. For details on defining dialog boxes, see *Workflow Dialog Boxes: General Information*.
- Configure the appearance of an action on the More menu and form toolbar. For more information, see *Workflow Actions: Appearance of Buttons and Commands*.
- Configure the persisting options. That is, you can specify when the system should save the current changes to the database. For details about persisting options, see *Workflow Actions: Persisting Options*.
- Configure mass processing, as described in *Workflow Actions: Mass Processing Actions*.
- Configure the behavior of an action. For details, see *Workflow Actions: Action's Behavior*.
- Map an action to the Acumatica mobile app.
- Display an action on a side panel by calling the `IsSidePanelScreen`method, as described in *Workflow Actions: Opening of a Side Panel*

Levels of Action Configuration

You can configure an action on the following levels, which are listed in order:

1. In the definition of the action in a graph or in one of its extensions
2. In a screen configuration
3. In a workflow state

Configurations defined on each level are applied to the subsequent level. For example, if you specified a category for an action in the `PXButton` attribute in a graph, this category is the default category for this action for the whole screen configuration, and you can change it in the screen configuration. But for some properties (see the table below), the configuration specified on the next level can only narrow what has been defined in the previous level.

The following table lists all possible configuration types and the levels where they can be changed.

Configuration Type	Graph	Screen Configuration	Workflow State	Comment
Visibility	+	+		The condition on the next level can only narrow what has been defined in the previous level.
Availability	+	+		
Category	+	+		The location inside the category can be specified only in the screen configuration.
Connotation	+		+	
Display on the form toolbar	+		+	
Assign field values	+	+		
Provide parameters		+		
Display a workflow dialog box		+		
Configure the persisting option	+	+		
Run report, open a generic inquiry, create a new record	+	+		
Expose the action to the mobile app		+		

Mapping a Workflow Action to the Mobile App

You can map a workflow action to the Acumatica mobile app without using MSDL. To map a workflow action, call the `IsExposedToMobile` method while adding an action to a screen configuration.

For this method to work, you must use MSDL to map the screen where the action should be displayed. For more information about mapping forms to the Acumatica mobile app, see *Mobile Framework Guide*.

Workflow Actions: Appearance of Buttons and Commands

You can configure the appearance of the button or command that corresponds to an action in a specific state of a workflow and in a screen configuration. The appearance settings specified in a screen configuration are applied to the action in all states of all workflows defined in the screen configuration.

Adjusting Appearance in the Screen Configuration

To adjust the appearance of a button or command for all states of all workflows of the form, you work with the screen configuration of this form. You call the methods mentioned in the list below in a lambda expression for the `Add` or `Update` method when you register an action in the screen configuration. You can specify the appearance settings for an action in a screen configuration in the following ways:

- Specify the category in which the command that corresponds to the action will be displayed on the More menu by calling the `WithCategory` method.

  By default, a command is placed in the **Other** category.
- Specify the location of the command inside the category.

  By default, commands are displayed in the order in which the respective actions are added in the screen configuration. You can specify the command's location in one of the following ways while adding the action to the screen configuration:
  - By specifying the `Placement` parameter in the `WithCategory` method.

    You can use this method when adding a new action to a category.
  - By calling the `Place` method.

    You can use this method when you need to customize the location of a predefined command in a category.
  - By calling one of the following methods:
    - `PlaceInCategory`
    - `PlaceFirst`
    - `PlaceLast`
    - `PlaceBefore`
    - `PlaceAfter`
- Make the command or button that corresponds to the action unavailable depending on a condition by calling the `IsDisabledWhen` method.

  You can also make the command or button unavailable unconditionally by calling the `IsDisabledAlways` method. You may need this method if you want to make unavailable a predefined button or command.
- Hide the command or button that corresponds to the action if a condition is met by calling the `IsHiddenWhen` method.

  You can also hide a command or button unconditionally by calling the `IsHiddenAlways` method. You may need this method if you want to remove a predefined command or button from the form.

The `IsDisabledWhen`, `IsDisabledAlways`, `IsHiddenWhen` and `IsHiddenAlways` methods can only restrict the behavior that is implemented in the code of a graph or a graph extension.

Adjusting Appearance in a Workflow State

To adjust the appearance of a button or command in a specific state of a workflow, you configure the workflow state as described in *Workflow States: Configuration of Actions*.

Workflow Actions: Persisting Options

In the code of a graph, you can save the changes to the database by calling the *`PXGraph.Persist`* method. When adding an action to a screen configuration, you can configure whether and when the current changes in the cache should be saved to the database—that is, when the system should invoke the `Persist` method.

This configuration can be defined in the `WithPersistOptions` method by using the values of the *`ActionPersistOptions`* enumerator.

> ⚠ If the action being executed triggers a transition, then the transition, by default, will cause the entity to be saved to the database. In this case, specifying the `WithPersistOptions(ActionPersistOptions.NoPersist)` method in that action would have no effect. To prevent the transition from saving the entity to the database, call the `DoesNotPersist` method in the transition.

The following example shows an action from the workflow on the *Opportunities* (CR304000) form.

```
.WithActions(actions =>
{
  actions.Add(g => g.Open, c => c
    .WithForm(formOpen)
    .WithFieldAssignments(fields =>
    {
      fields.Add<CROpportunity.resolution>(f => f.SetFromFormField(formOpen,
 ReasonFormField));
      fields.Add<CROpportunity.stageID>(f => f.SetFromFormField(formOpen, StageFormField));
      fields.Add<CROpportunity.isActive>(f => f.SetFromValue(true));
      fields.Add<CROpportunity.closingDate>(f => f.SetFromValue(null));
    })
    .IsHiddenWhen(conditions.IsInNewState)
    .WithPersistOptions(ActionPersistOptions.PersistBeforeAction)
    .WithCategory(categoryProcessing)
    .IsExposedToMobile(true)
    .MassProcessingScreen<UpdateOpportunityMassProcess>());
```

If you need to save the changes of one entity on a form where this entity is not used, you need to override the `Persist` method of this form and manually save these changes of this custom entity. This might be useful, for example, if you need to save the changes of a custom entity on a predefined form. For more information, see *Step 6: Overriding the PerformPersist Method*.

Workflow Actions: Mass Processing Actions

You can use a workflow action on a mass processing form, which gives a user the ability to process any number of records simultaneously. To specify that the action should be used for mass processing, you should call the `MassProcessingScreen` method when adding the action to the screen configuration and specify the graph of the processing form as a type parameter of the method.

You can also call the `InBatchMode()` method so that the action processes the list of records at once. Otherwise, the action is invoked for each record separately.

An example of the configuration of mass processing is shown in the following code.

```
actions.Add(g => g.Assign,
  c => c.WithCategory(processingCategory, g => g.PutOnHold)
  .MassProcessingScreen<RSSVAssignProcess>()
  .InBatchMode());
```

For a detailed example, see *Processing Forms: Activity 1.1.1: To Create a Simple Processing Form*.

Workflow Actions: Action's Behavior

You can specify the following behavior for a workflow action without implementing an action method in a graph:

- Open a report by calling the `IsRunReportScreen` method and specifying the configuration of the report in the parameter. The following code shows an example from the `INSiteMaint` graph.

```
actions.AddNew(locationLabels.ActionName, a => a
  .WithCategory(PredefinedCategory.Reports)
  .DisplayName(Messages.INLocationLabels)
  .IsRunReportScreen(report =>
  {
    return report
      .ReportID(locationLabels.ReportID)
      .WithWindowMode(PXBaseRedirectException.WindowMode.New)
      .WithAssignments(rass =>
      {
        rass.Add(locationLabels.Parameters.Site, z =>
 z.SetFromField<INSite.siteCD>());
      });
}));
```

  In the code above, the action has opened a report in a new window and assigns a value to a report parameter.
- Open a generic inquiry form—for example, a substitute form—by calling the `IsSearchRecordScreen` method.
- Open a form where a user can create a new record by calling the `IsCreateRecordScreen` method.

You should define the behavior of an action either in the code of a graph or in the screen configuration. If you define the behavior of an action in a workflow, the action method in a graph should be empty.

Configuring an Action to Run Automatically

You can configure an action to run automatically in a workflow without the user having to click a button to trigger the action. This can be done in one of the following ways:

- By defining the action in the graph to be of type `PXAutoAction` and calling the `IsAutoAlways` method or the `IsAutoWhen` method on it in the screen configuration.
- By defining the action in the screen configuration itself (instead of in a graph) and calling the `IsAutoAlways` method or the `IsAutoWhen` method on it.

The `IsAutoAlways` method will cause the action to always be triggered automatically without a condition being checked. Alternatively, you can call the `IsAutoWhen` method and pass in a `condition` as a parameter to this method. The action will be triggered automatically when the condition is `True`. An example is shown in the following code.

```
public override void Configure(PXScreenConfiguration config)
{
          var context = config.GetScreenConfigurationContext<SOOrderEntry,
                                SOOrder>();
    var IsBackOrderedbyPartialShipment = context.Conditions.
    FromBql<SOOrderCustomUnboundFields
          .usrbackOrderedbyShipmentConfirmation.IsEqual<True>>()
```

```
    .WithSharedName("IsBackOrderedbyPartialShipment");
    var sendEmailAction = context.ActionDefinitions.CreateNew("sendEmail",
                        a => a.DisplayName("Send Email"));
    context.UpdateScreenConfigurationFor(screen =>
   screen
   .WithActions(actions => {
    actions.Add(sendEmailAction);
   })
   .WithFlows(flows => flows.Update<SOBehavior.sO>(f =>
    f.WithFlowStates(fss =>
    {
      fss.Update(State.Initial, state =>
             state.WithActions(actions =>
        {
          actions.Add(sendEmailAction, a =>
                      a.IsAutoWhen(IsBackOrderedbyPartialShipment));
        }
     ));
       }))
   )
  );
}
```

In the code above, a workflow action, `sendEmailAction`, has been defined in the screen configuration itself through the use of the `Configure` method. The screen configuration has then been updated through the use of the `UpdateScreenConfigurationFor` method. Inside the lambda expression of this method, the `sendEmailAction` action has been added to the lambda expression of the `WithActions` method. Finally, in the lambda expression of the `WithActions` method of the state, the `sendEmailAction` action has been added. The action is set to trigger automatically through the use of the `IsAutoWhen` method when the `IsBackOrderedbyPartialShipment` condition is `True`.

An action of the `PXAction` type that is declared in a graph cannot be configured to run automatically through the use of the `IsAutoAlways` and `IsAutoWhen` methods.

Workflow Actions: Opening of a Side Panel

You can create an action in a workflow and configure it to open a side panel. A side panel is a navigation option that you can use to drill down to relevant details about any record listed on a form. The action that you configure in a workflow to open a side panel appears as a tab icon in the side panel on the UI when the relevant form is opened. For more details, see *Side Panels of Generic Inquiry Forms*.

To configure an action to open a side panel, you perform the following general steps:

1. You get the screen configuration and configure the side panel settings as follows:
   a. You get the screen configuration by using the `GetScreenConfigurationContext` method, and you add a set of settings (for the new action to be defined in the next step) to the screen configuration by using the `SidePanel` method. This method is provided by the `Navigation` helper object of the screen configuration. See the following code for an example.

```
        var context = config.GetScreenConfigurationContext<SOOrderEntry,
SOOrder>();
        var sidePanel = context.Navigation.SidePanel(screen => screen
                        ... );
```

b. In the lambda expression of the `SidePanel` method in the code above, you call the `NavigateToScreen` method and pass in a `ScreenID` as a parameter. This `ScreenID` should belong to the form that you want to open when the action is triggered on the UI. You can pass in the `ScreenID` as a `string` or as a graph type such as `NavigateToScreen<SOOrderEntry>()`.

   You will also call the `WithIcon` method and pass in an icon name as a parameter. This method specifies the icon for the action on the UI. Optionally, you can call the `WithAssignments` method to specify a list of navigation parameters. An example is shown in the following code.

```
                var sidePanel = context.Navigation.SidePanel(screen => screen
     .NavigateToScreen("SO3031SP")
                        .WithIcon("new_quote")
                        .WithAssignments(filler =>
               {
                   filler.Add(nameof(SOOrder.OrderNbr), c =>
c.SetFromField<orderNbr>());
               }));
```

2. Add a new action to the screen configuration and configure it to open a side panel as follows:

   a. You add a new action to the screen configuration by using the `CreateNew` method provided by the `ActionDefinitions` helper object. You provide a name for the action and a lambda expression as parameters of the method. The lambda expression is used to configure the action. You then call the `DisplayName` method to set the display name of this action on the user interface.

   b. You call the `IsSidePanelScreen` method and pass as a parameter the `sidePanel` variable that you created in Step 1a. The `sidePanel` variable defines the side panel configuration of the new action. An example is shown in the following code.

```
              var action =
                   context.ActionDefinitions.CreateNew("ShowInvoicesAndMemos", a =>
a
                   .DisplayName("Invoices and Memos")
                   .IsSidePanelScreen(sidePanel));
```

3. You add the `action` defined in the previous step to the lambda expression of the `WithActions` method of the `AddScreenConfigurationFor` or the `UpdateScreenConfigurationFor` method for the screen configuration. An example is shown in the following code.

```
          context.UpdateScreenConfigurationFor(screen =>
          {
              return screen
                  .WithActions(actions =>
                  {
                      actions.Add(action);
                  });
          });
```

Workflow Actions: Categories of the More Menu

On the More menu of an Acumatica ERP form, the commands corresponding to actions are listed under categories. You can specify the category of an action by using Workflow API or attributes. If a category is not specified for an action, the action is represented in the following ways:

- As a command listed under the **Other** category of the More menu
- As a button on the form toolbar

Conversely, if a category has no actions added to it, then it will not be displayed on the More menu until at least one action is assigned to it.

Specifying a Category for an Action on a Form with a Workflow

To specify a category for an action on a form that uses a workflow, you perform the following general steps:

1. You define and configure the category, as described in *Creating a Category*
2. In the action definition, you call the `WithCategory` method and specify the object that defines the category. An example is shown in the following code.

```
actions.Add(g => g.printSalesOrder, c =>
    c.WithCategory(ActionCategory.printAndEmailCategory));
```

Specifying the Order of Actions in a Category

You can specify the location of an action in a category, which determines the placement of its corresponding command on the More menu. An action can be placed first, last, or before or after another specified action. To specify the location, you use the `Place` method or the `Placement` parameter of the `WithCategory` method.

The following code shows an example of the usage of the `Placement` parameter.

```
actions.Add(g => g.putOnHold, c =>
     c.WithCategory(processingCategory, Placement.First));
```

In the code example above, the `putOnHold` action is placed first in the `processingCategory` by specifying `Placement.First`. For more details, see *Enum Placement*.

Creating a Category

You can create a category on the More menu by creating a new object that can later be used for multiple actions. To define a category by creating an object, you perform the following general steps:

1. You call the `Categories.CreateNew` method and provide the display name of the category. An example is shown in the following code.

```
var processSalesCategory = context.Categories.CreateNew
  (ActionCategories.ProcessSalesCategoryID,
    category => category.DisplayName(CategoryNames.ProcessSales)
      .Place(Placement.Before, labelsCategory));
);
```

   In the preceding code, you have created a new category with the `CategoryNames.ProcessSales` display name and placed it before the `labelsCategory`.

2. You add the category to the screen configuration by using the `WithCategories` method, as the following code shows.

```
.WithCategories(categories =>categories.Add(processSalesCategory))
```

Specifying the Order of Categories

You can specify the location of a category on the More menu by calling the `Place` method. A category can be placed first, last, or before or after a specified category by using the `Placement` parameter of this method.

In the following code example, the existing **Printing Email** category (which corresponds to `printingEmailingCategory` in code) is placed last on the More menu.

```
.WithCategories(
  actions.Update(printingEmailingCategory,
    category => category.Place(Placement.Last));
);
```

You can define the order of categories by using the `PlaceAfter` and `PlaceBefore` methods in the `WithCategories` method. An example is shown in the following code.

```
.WithCategories(categories =>
{
  categories.Add(processingCategory);
  categories.Add(intercompanyCategory);
  categories.Add(printingEmailingCategory);
  categories.Add(otherCategory);
  categories.Update(FolderType.ReportsFolder,
    category => category.PlaceAfter(otherCategory));
})
```

By default, the **Actions**, **Inquiries**, and **Reports** categories are the first categories on the menu if there is at least one action in any of these categories.

Specifying a Category for an Action on a Form Without a Workflow

You can place an action in a category by specifying the category in the `PXButton` attribute for an action. An example is shown in the following code.

```
[PXButton(Category = "Publish")]
protected virtual void ViewXml()
```

In the code above, the `ViewXml` action is placed in the **Publish** category. If the specified category does not exist, it will be created by the system.

If there are multiple actions in the same category, the order of actions in the category is the same as the order in which the actions are declared in code: first, the actions declared in the code of the graph, in their order of declaration in code; and then the actions in all graph extensions, in the order in which they are declared in code.

Workflow Actions: To Implement a Simple Action

The following activity will walk you through the process of implementing a workflow action that can later be used to trigger a transition.

Story

On the Repair Work Orders (RS301000) form, you need to create a workflow action that can later be used to trigger a transition. The command that corresponds to this action should be displayed in the **Processing** category of the More menu.

Process Overview

In your extension library, you will implement the `ReleaseFromHold` graph action with an empty method and add this action to the screen configuration of the Repair Work Orders (RS301000) form. You will also add the predefined **Processing** category to the workflow and specify this category for the `ReleaseFromHold` action.

System Preparation

Make sure that you have done the following:

1. Prepared an instance with the *PhoneRepairShop* customization project and enabled the workflow validation by performing the following prerequisite activities:
   a. *Test Instance for Workflow Customization: To Deploy a Test Instance*
   b. *Test Instance for Workflow Customization: To Turn On Workflow Validation*
2. Prepared the screen configuration and defined the set of states by performing the *Screen Configuration: To Prepare a Screen Configuration for a Form Without a Predefined Workflow* prerequisite activity.

Step 1: Implementing the Action in the Graph

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

To define the action in the graph, do the following:

1. Add the `using System.Collections;` directive to the `RSSVWorkOrderEntry.cs` file (if it has not been added yet). Then, add the `Actions` region.
2. In the `Actions` region, add the following code.

```
#region Actions
public PXAction<RSSVWorkOrder> ReleaseFromHold = null!;
[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.

3. Save your changes.

Step 2: Defining a Category for the Action

Acumatica ERP provides a list of predefined 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. If a command is associated with an action related to changing the status of a record created on the form, it is usually displayed in the **Processing** category. To add the **Processing** category to your workflow, do the following:

1. In the static `Configure` method of the `RSSVWorkOrderEntry_Workflow` class, define the **Processing** category before the `context.AddScreenConfigurationFor` method call, 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 static `Configure` method, locate the `AddDefaultFlow` method, which you have added in *Step 3: Overriding the Configure Method* in the *Screen Configuration: To Prepare a Screen Configuration for a Form Without a Predefined Workflow*.

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, you have added the **Processing** category to the screen configuration by calling the `Add` method.

Step 3: Adding the Action to the Workflow

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

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

1. In the `RSSVWorkOrderEntry_Workflow` class, in the static `Configure` method, locate the `AddDefaultFlow` method. (You have added this method in *Step 3: Overriding the Configure Method* in the *Screen Configuration: To Prepare a Screen Configuration for a Form Without a Predefined Workflow* activity.)

2. For the screen parameter, call the `WithActions` method after the `WithCategories` method, as the following code shows.

```
    .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(graph => graph.ReleaseFromHold,
                          action => action.WithCategory(processingCategory));
```

   In the code above, you have added the `ReleaseFromHold` action, which you defined in the `RSSVWorkOrderEntry` graph in *Step 1: Implementing the Action in the Graph* in this activity, to the screen configuration. 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.

You will use this action later to trigger a transition.

Workflow Actions: To Implement an Action with Field Assignments

This activity will walk you through the process of implementing a workflow action that assigns values to DAC fields without any graph logic.

Story

Suppose that a user should be able to mark a repair work order as completed when it has the *Assigned* status. To do that, the user should be able to click the **Complete** button. The underlying 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. The **Complete** action should always be displayed last in the **Processing** category.

Process Overview

In the customization code, you will define the `Complete` workflow action, the `Completed` state, and the transition from the `Assigned` state to the `Completed` state. You will learn how to define the assignment of the **Date Completed** box in a workflow action. You will also specify the location of the **Complete** command on the More menu.

System Preparation

Make sure that you have done the following:

1. Prepared an instance with the *PhoneRepairShop* customization project and enabled the workflow validation by performing the following prerequisite activities:
   a. *Test Instance for Workflow Customization: To Deploy a Test Instance*
   b. *Test Instance for Workflow Customization: To Turn On Workflow Validation*
2. Prepared the screen configuration and defined the set of states by performing the *Screen Configuration: To Prepare a Screen Configuration for a Form Without a Predefined Workflow* prerequisite activity.
3. Defined the `Assigned` workflow state by performing the *Workflow Dialog Boxes: To Implement a Transition with a Dialog Box* prerequisite activity.

Step 1: Adding and Configuring the Action

In this step, you will add the `Complete` action in the graph of the Repair Work Orders (RS301000) form and in the screen configuration of this form. You will specify the location of the associated command by using the `WithCategory` method and assign the DAC field values by using the `WithFieldAssignments` method.

Do the following:

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

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

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

```
actions.Add(graph => graph.Complete, action => action
  .WithCategory(processingCategory, Placement.Last)
  .WithFieldAssignments(fields => fields
    .Add<RSSVWorkOrder.dateCompleted>(field =>
      field.SetFromToday())));
```

   In the code above, you have defined the `Complete` action in the screen configuration. In the `WithCategory` method, you have specified that the associated command will be displayed last in the **Processing** category of the More menu by specifying the `Placement.Last` parameter. 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` workflow state, add the action to the configuration of the `Assigned` workflow state in the `GetAssignedBehavior` private static method, as the following code shows.

```
.WithActions(actions =>
{
    actions.Add(graph => graph.Complete, action => action
      .IsDuplicatedInToolbar()
      .WithConnotation(ActionConnotation.Success));
});
```

Step 2: Adding a Workflow State and a Transition

In this step, you will define the `Completed` workflow state and the transition from the `Assigned` workflow state to the `Completed` workflow state. Do the following:

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

```
flowStates.Add<States.completed>(flowState =>
    GetCompletedBehavior(flowState));
```

2. In the `RSSVWorkOrderEntry_Workflow` class, add the `GetCompletedBehavior` private static method, as the following code shows.

```
private static BaseFlowStep.IConfigured GetCompletedBehavior(
    FlowState.INeedAnyFlowStateConfig 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 method, you have made the `CustomerID`, `ServiceID`, and `DeviceID` fields unavailable for editing.

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

```
transitions.AddGroupFrom<States.assigned>(
    transitionGroup =>
{
    transitionGroup.Add(transition =>
        transition.To<States.completed>()
        .IsTriggeredOn(graph => graph.Complete));
});
```

Step 3: Testing the Field Assignment

In this step, you will make sure that the transition from the `Assigned` workflow state to the `Completed` workflow state assigns the value as planned. Do the following:

1. Rebuild the `PhoneRepairShop_Code` project.
2. In Acumatica ERP, open a repair work order with the *Assigned* status on the Repair Work Orders (RS301000) form.
3. On the form toolbar, click **Complete**. Notice that the document's status has been changed to *Completed* and the **Date Completed** box has the current business date.

Workflow Actions: To Configure the Conditional Appearance of the Action

This activity will walk you through the process of implementing an action that appears conditionally and adding it to a workflow.

Story

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. You need to implement the `CreateInvoice` action, as well as to configure the conditional availability of the associated button on the form toolbar and the equivalent command on the More menu.

Process Overview

In the customization code, first, you will define the `CreateInvoice` action in a graph. Then you will add this action to the screen configuration and to the states of the workflow where the action should be available. You will also define a condition depending on which the action should be available.

System Preparation

Make sure that you have done the following:

1. Prepared an instance with the *PhoneRepairShop* customization project and enabled the workflow validation by performing the following prerequisite activities:
   a. *Test Instance for Workflow Customization: To Deploy a Test Instance*
   b. *Test Instance for Workflow Customization: To Turn On Workflow Validation*
   c. *Test Instance for Workflow Customization: To Configure the Instance*
2. Prepared the screen configuration and defined the set of states by performing the *Screen Configuration: To Prepare a Screen Configuration for a Form Without a Predefined Workflow* prerequisite activity.
3. Defined the `PendingPayment` and `Completed` workflow states by performing the *Transitions: To Implement a Transition Triggered by an Action* and *Workflow Actions: To Implement an Action with Field Assignments* prerequisite activities.

Step 1: Defining the Graph Action

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

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

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

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

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

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

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

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

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

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

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

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

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

```
            workOrderEntry.Actions.PressSave();

            ts.Complete();
        }
    }
```

In the method above, you have done the following:

a. You have created an instance of the `SOInvoiceEntry` graph. This graph works with SO invoices.

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

You have initialized the summary of the invoice by using the `ARInvoice` class. You have assigned a value to the `CustomerID` field, which is required to create an invoice. After that you have updated the `ARInvoice` instance in cache.

b. You have created an instance of the `RSSVWorkOrderEntry` graph, which you need to get access to the current work order and to save the generated invoice number to the current work order.

c. You have selected the repair and labor items specified on the Repair Work Orders form by using the instance of the `RSSVWorkOrderEntry` graph. Then you have added lines to the invoice by adding instances of the `ARTran` class: the lines associated with repair items, followed by the lines associated with labor items.

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

e. You have saved the number of the generated invoice to the work order and updated its value in the cache. Then you have saved the changes to the database by invoking the `Actions.PressSave()` method.

f. At the end of the method, you have completed the transaction.

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

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

```
    public PXAction<RSSVWorkOrder> CreateInvoiceAction = null!;
    [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;
        }
```

You have used the `PXButton` attribute to set up a button on the form. You have specified the button name in the `PXUIField` attribute.

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

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

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

Finally, you have returned the list of work orders.

Step 2: Configuring the Conditional Appearance of the Button

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). Configure the availability as follows:

1. Configure a condition that checks whether the `InvoiceNbr` field is not null, and make the action unavailable 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 HasInvoice => GetOrCreate(condition =>
    condition.FromBql<Where<RSSVWorkOrder.invoiceNbr.IsNotNull>>());
```

   b. Make sure an instance of the `Conditions` class is created in the static `Configure` method.
   c. In the `WithActions` method of the screen configuration, register the `CreateInvoice` action, as the following code shows.

```
actions.Add(graph => graph.CreateInvoiceAction,
    action => action.WithCategory(processingCategory)
    .IsDisabledWhen(conditions.HasInvoice));
```

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

```
.WithActions(actions =>
{
    actions.Add(graph => graph.CreateInvoiceAction,
        action => action.IsDuplicatedInToolbar()
```

```
                    .WithConnotation(ActionConnotation.Success));
            });
```

Step 3: Testing the Action

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

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

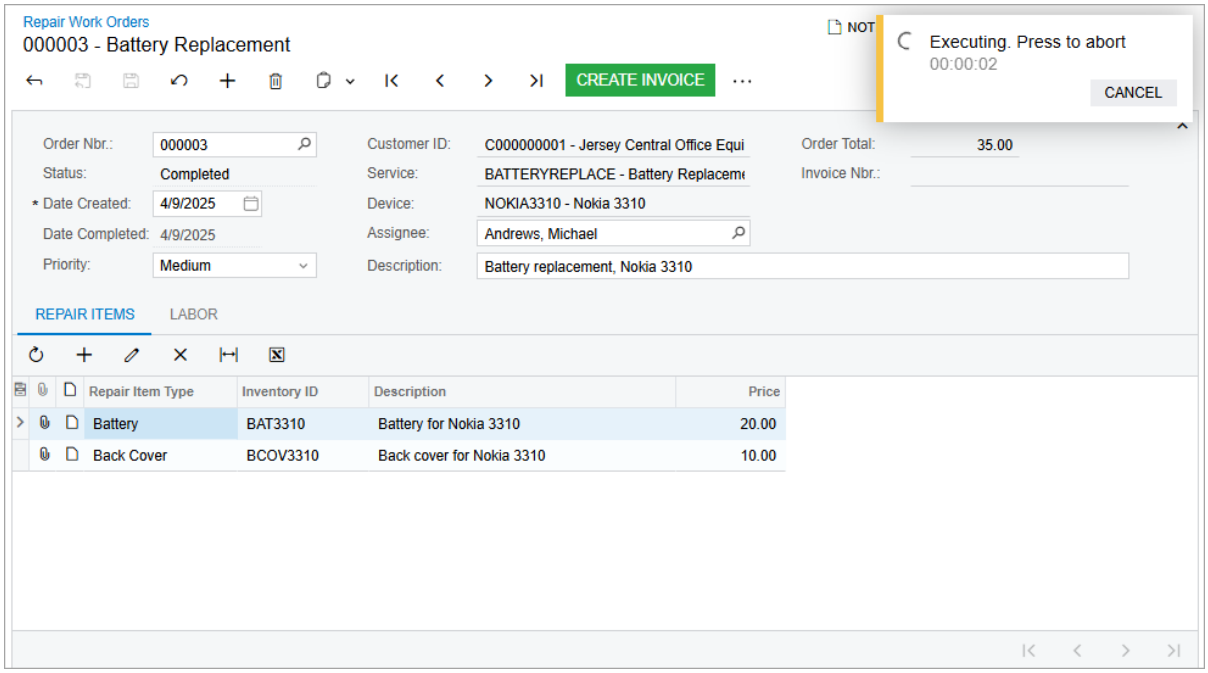


Figure: Creation of an invoice

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

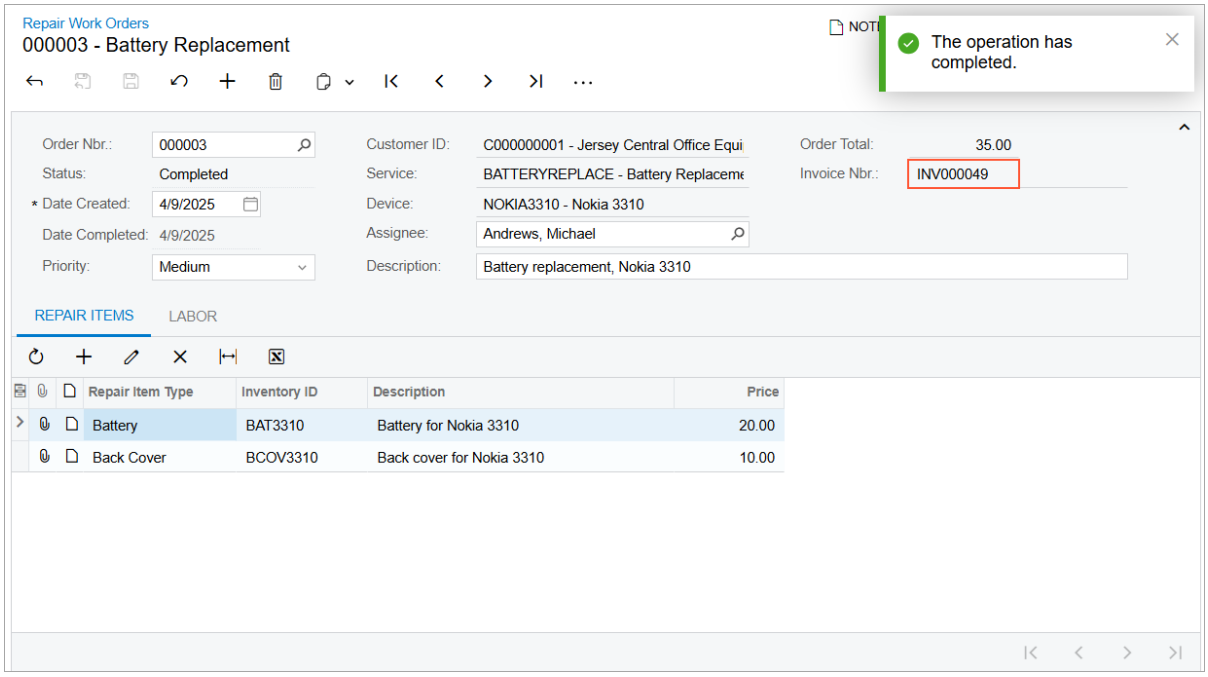


Figure: The invoice number of the repair work order

Workflow Actions: Order of Field Assignments

The order in which field assignments are declared in code is important because the system processes the field assignments in that order. You can define the order in which a field assignment is processed by specifying whether a field assignment should be placed before or after another field assignment. You do this by calling the `PlaceBefore` or `PlaceAfter` method.

To illustrate why this order is important, suppose that you plan to have two dependent fields on a form: One of the fields (`City`) represents a list of cities, and the other (`Country`) represents a list of countries. Both fields are declared to be of the `PXSelector` type. The values available in the `City` field depend on the value of the `Country` field. Suppose that the default values for the `City` and `Country` fields are *London* and *United Kingdom*, respectively.

Further suppose that you have a workflow action defined for the form and that you initially had the `City` field and the workflow action added to your form. You are using the `WithFieldAssignments` method to assign a new value to the `City` field, and this value is *New York*. The following code shows an example of the definition of this workflow action.

```
actions.Add(g => g.assignAddress, c => c
                            .WithCategory(addressCategory)
                            .WithFieldAssignments(fass =>
                           {
                                fass.Add<firstName>(firstName);
                                fass.Add<lastName>(lastName);
                                fass.Add<phone>(phoneNumber);
                                fass.Add<city>(cityName);
                            }
                            ));
```

Now suppose that you later realize that you need to add the `Country` field to your form in addition to the workflow action defined in the preceding code example. This means that you need to update your existing

workflow action and add the `Country` field by using the `WithFieldAssignments` method. Because the `City` field is dependent on the `Country` field, the order in which you declare these field assignments is important. If you declare the `City` field first, as shown in the code above, then the field assignment will most likely fail. This is because the value, *New York*, that you want to set for the `City` field would be available only if the value in the `Country` field is already set to *USA*.

However, in this case, the `Country` field has not being added to the form and the workflow action yet. Even if it had been added, it would have to be declared before the `City` field in the preceding code example because the `City` field depends on its value. To resolve this issue, you can either declare the `Country` field assignment first when you initially define the action in the code above or you can add this field assignment elsewhere in the code and use the `PlaceBefore` or `PlaceAfter` method to update the order of the field assignments.

The following code shows how to update the workflow action defined in the preceding code example. This code adds the field assignment for the `Country` field and changes the order of field assignments by calling the `PlaceBefore` method to place the `Country` field assignment before the `City` field assignment.

```
actions.Update(g => g.assignAddress, c => c
                            .WithFieldAssignments(fass =>
                           {
                                fass.Add<country>(f =>
                                f.SetFromValue(countryName).PlaceBefore<city>());
                            }
                             ));
```

The code above would result in the `Country` field being added and placed before the `City` field in the order of field assignments. Thus, the `Country` field will be assigned its value before the `City` field, which is the correct order because the `City` field depends on the value of the `Country` field in order for its value to be assigned correctly.

Implementing Transitions

In this chapter, you will learn how to implement transitions in a worfklow. Transitions are used to link the states of the workflow, and they define the changes that a record proceeds through.

Transitions: General Information

In a workflow, all workflow states are linked with transitions. Transitions define the changes during the lifecycle of a record.

Learning Objectives

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

- Define a simple transition
- Define a transition group
- Define conditions
- Define transitions that are triggered depending on a condition

Applicable Scenarios

You implement a transition when you need to implement a change of workflow state.

Definition of a Transition

To implement a transition, you define the following parts of the screen configuration:

- The source workflow state for the transition
- The target workflow state for the transition
- The action or workflow event that triggers the transition
- The conditions that are checked before the transition is applied (optional)
- The transition itself

After you have defined all the entities required for a transition (which are the source workflow state, the target workflow state, and the entity that triggers the transition), you can 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 source workflow state for the transition
- `To`: The target workflow state for the transition
- `IsTriggeredOn`: The trigger of the transition

Configuration of a Transition

While configuring a transition, you can do the following:

- Add, update, or delete a transition by calling the `Add`, `Update`, or `Delete` method.
- Specify the condition on which a transition is performed by calling the `When` method. For an example of this implementation, see *Transitions: To Implement a Group of Transitions*.

- Define the order in which the system checks a transition by specifying whether the transition should be placed before or after another transition. You do this by calling the `PlaceBefore` or `PlaceAfter` method. By default, transitions are executed based on their order of declaration in code.
- Specify the field assignments that are executed if the current transition matches all criteria and is applied to the current record. You do this by calling the `WithFieldAssignments` method.
- Specify that the system should not save the current record to the database when the transition is performed. You do this by calling the `DoesNotPersist` method. By default, the system always saves the current record to the database when the transition is performed.

Transitions: To Implement a Transition Triggered by an Action

This activity will walk you through the process of implementing a transition that is triggered by an action.

Story

Suppose that 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 the corresponding command on the More menu. As a result, the status of the repair work order should be changed to *Ready For Assignment*.

You need to implement a simple transition from the `OnHold` workflow state to the `ReadyForAssignment` workflow state. This transition is performed without the system considering the prepayment requirements.

Process Overview

To implement a transition from the `OnHold` workflow state to the `ReadyForAssignment` workflow state, you will use or define the following components in the screen configuration:

- The `OnHold` workflow state, which is the initial workflow state for the transition.
- The `ReadyForAssignment` workflow state, which is the target workflow state for the transition.

  You have defined both workflow states in *Workflow States: To Define a Workflow State*.
- The `ReleaseFromHold` action, which is the element that triggers the transition.

  In this activity, you will use the implementation of the `ReleaseFromHold` action from *Workflow Actions: To Implement a Simple Action*.
- The transition.

System Preparation

Make sure that you have done the following:

1. Prepared an instance with the *PhoneRepairShop* customization project and enabled the workflow validation by performing the following prerequisite activities:
   a. *Test Instance for Workflow Customization: To Deploy a Test Instance*
   b. *Test Instance for Workflow Customization: To Turn On Workflow Validation*
2. Prepared the screen configuration and defined the set of states by performing the *Screen Configuration: To Prepare a Screen Configuration for a Form Without a Predefined Workflow* prerequisite activity.
3. Defined the `ReleaseFromHold` action by performing the *Workflow Actions: To Implement a Simple Action* activity.
4. Defined the `OnHold` and `ReadyForAssignment` states by performing the *Workflow States: To Define a Workflow State* activity.

Step 1: Defining a Transition

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

1. In the `RSSVWorkOrderEntry_Workflow` class, in the static `Configure` method, locate the `AddDefaultFlow` method.
2. Call the `WithTransitions` method in the lambda expression for the `AddDefaultFlow` method, as the following code shows.

```
.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(transition => transition
    .From<States.onHold>().To<States.readyForAssignment>()
    .IsTriggeredOn(graph => graph.ReleaseFromHold));
```

   In the code above, you have specified the following:

   - The source workflow state for the transition (which is `OnHold`) as a type parameter of the `From` method
   - The target workflow state for the transition (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

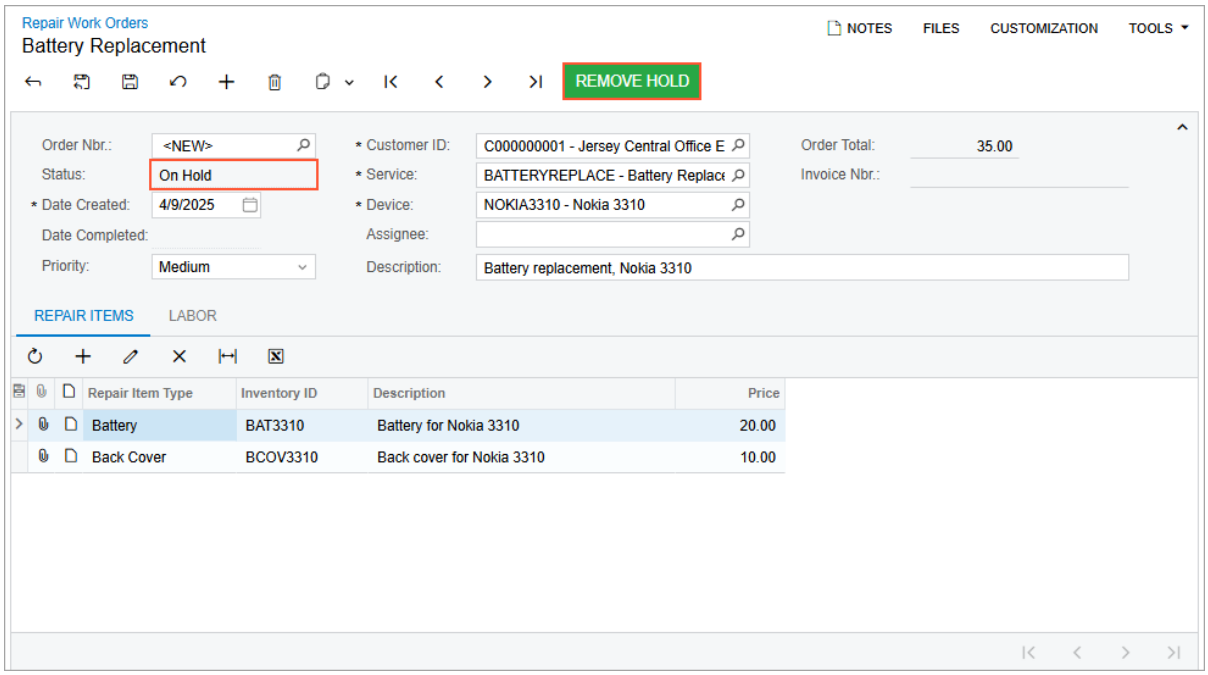
4. Save your changes.

Step 2: Testing the Transition

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

1. Rebuild the `PhoneRepairShop_Code` project.
2. In Acumatica ERP, on the Repair Work Orders (RS301000) form, create a record, and specify the following settings:
   - **Customer ID**: *C000000001*
   - **Service**: *Battery Replacement*
   - **Device**: *Nokia 3310*
   - **Description**: `Battery replacement, Nokia 3310`
3. Make sure that the record 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 record on the Repair Work Orders form



> The More menu is not displayed because the workflow includes only one action and this action fits on the form toolbar. If there was only one workflow action and it could not be displayed on the form toolbar, then the More menu would be displayed.

4. Save the repair work order.
5. On the form toolbar, click **Remove Hold**.
6. Notice that the status of the record 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.

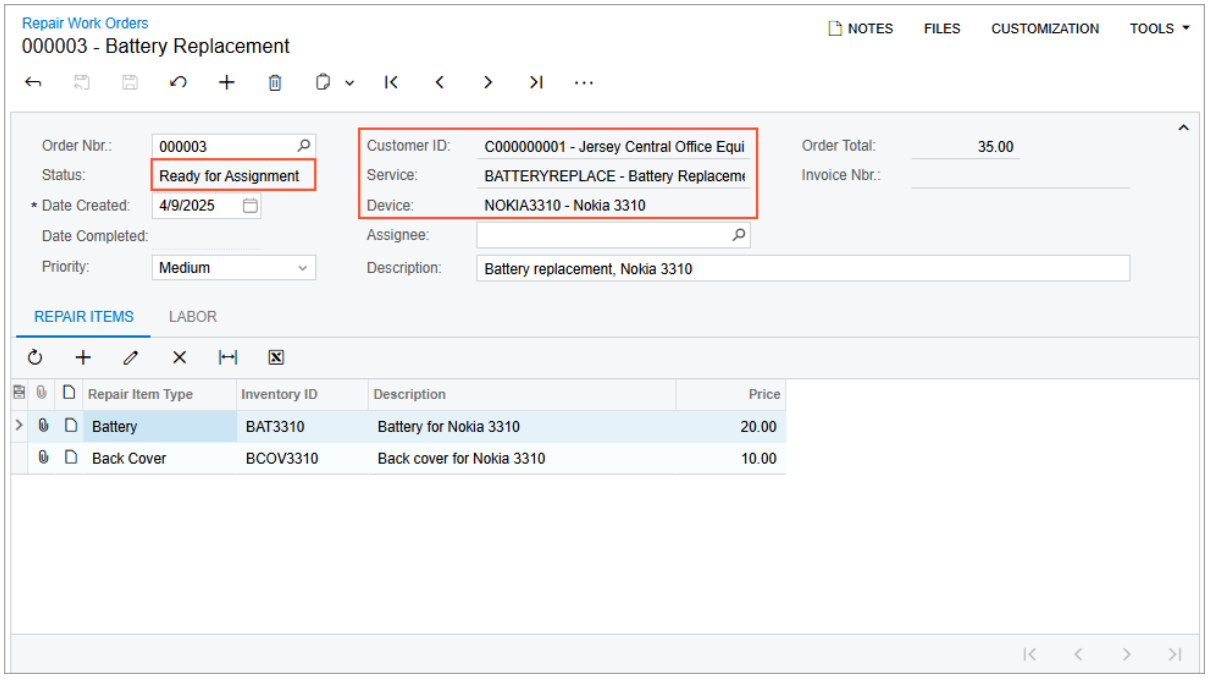


Figure: A record with the Ready for Assignment status

Transitions: Transition Groups

You can group transitions by their source workflow states. In this case, you can structure your code and do not have to specify the source workflow state for the transition in 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. An example is shown in the following code.

```
.WithTransitions(transitions =>
{
  transitions.AddGroupFrom<State.open>(transitionGroup => {
    transitionGroup.Add(transition => transition
      .To<State.hold>()
      .IsTriggeredOn(graph => graph.putOnHold)
      .WithFieldAssignments(fields => fields.Add<hold>(true)));
    transitionGroup.Add(transition => transition
      .To<State.confirmed>()
      .IsTriggeredOn(graph => graph.confirmShipmentAction));
  });
})
```

Transitions: To Implement a Group of Transitions

The following activity will walk you through the process of implementing a group of transitions—that is, a set of transitions that have the same source workflow state.

Story

Suppose that a user should be able to remove a repair work order from hold when the order has the *On Hold* status. The user should be able to do this by clicking the **Remove Hold** button on the form toolbar or the equivalent command on 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.

Process Overview

In *Transitions: To Implement a Transition Triggered by an Action*, you have implemented an unconditional transition from the `OnHold` workflow state to the `ReadyForAssignment` workflow state.

Here you will add a transition from the `OnHold` workflow state to the `PendingPayment` workflow state and modify the transition from the `OnHold` workflow state to the `ReadyForAssignment` workflow state. You will also define the conditions on which these transitions are performed.

System Preparation

Make sure that you have done the following:

1. Prepared an instance with the *PhoneRepairShop* customization project and enabled the workflow validation by performing the following prerequisite activities:
   a. *Test Instance for Workflow Customization: To Deploy a Test Instance*
   b. *Test Instance for Workflow Customization: To Turn On Workflow Validation*
2. Prepared the screen configuration and define the set of states by performing the *Screen Configuration: To Prepare a Screen Configuration for a Form Without a Predefined Workflow* prerequisite activity.
3. Defined the `OnHold` and `ReadyForAssignment` workflow states by performing the *Workflow States: To Define a Workflow State* prerequisite activity.
4. Implemented a transition from the `OnHold` workflow state to the `ReadyForAssignment` workflow state by performing the *Transitions: To Implement a Transition Triggered by an Action* activity.

Step 1: Adding a Condition

To implement a transition from the `OnHold` workflow state to the `ReadyForAssignment` or `PendingPayment` workflow state, based on the properties of the current repair work order, you need to define a condition that is true when a repair work order requires prepayment. You will use the logical NOT C# operator and this condition to define a condition that is true when a repair work order does not require prepayment.

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 condition, do the following:

1. In the `RSSVWorkOrderEntry_Workflow` class, add the `Conditions` region.
2. In the `Conditions` region, declare a condition class, 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(condition =>
            condition.FromBql<Where<RSSVRepairService.prepayment
                .FromSelectorOf<RSSVWorkOrder.serviceID>.IsEqual<True>>>());
```

   In the code above, you have created a condition by calling the `GetOrCreate` method and provided 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 of this record equals *true*.

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

```
        // Create an instance of the Conditions class
        var conditions = context.Conditions.GetPack<Conditions>();
```

5. Save your changes.

Step 2: Adding the PendingPayment Workflow State (Self-Guided Exercise)

To define a transition from the `OnHold` workflow state to the `PendingPayment` workflow state, you first need to define the `PendingPayment` workflow 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 workflow state in *Workflow States: To Define a Workflow State*.

While defining a state, you need to use the `States.pendingPayment` class, which you have added in *Step 2: Defining the Set of States of the Workflow* of the *Screen Configuration: To Prepare a Screen Configuration for a Form Without a Predefined Workflow* activity. In the `PendingPayment` workflow state, make the **Customer ID**, **Service ID**, and **Device ID** boxes of the Repair Work Orders (RS301000) form are unavailable for this workflow state.

Step 3: Grouping Transitions and Adding Conditions to Transitions

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

To specify the conditions you defined in *Step 1: Adding a Condition* of this activity, 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>(transitionGroup =>
            {
            });
        })
```

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

2. Move the definition of the transition that you added in *Transitions: To Implement a Transition Triggered by an Action* 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>(transitionGroup =>
                {
                    transitionGroup.Add(transition =>
                        transition.To<States.readyForAssignment>()
                        .IsTriggeredOn(graph => graph.ReleaseFromHold)
                    );
                });
```

3. After the `IsTriggeredOn` method call, add the `When` method call, and specify the `RequiresPrepayment` condition with the logical NOT operator, as the following code shows.

```
                        transition.To<States.readyForAssignment>()
                        .IsTriggeredOn(graph => graph.ReleaseFromHold)
                        .When(!conditions.RequiresPrepayment));
```

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

```
                    transitionGroup.Add(transition =>
                        transition.To<States.pendingPayment>()
                        .IsTriggeredOn(graph => graph.ReleaseFromHold)
                        .When(conditions.RequiresPrepayment));
```

5. Save your changes.

The full code of the section should look as follows.

```
            .WithTransitions(transitions =>
            {
                transitions.AddGroupFrom<States.onHold>(transitionGroup =>
                {
                    transitionGroup.Add(transition =>
                        transition.To<States.readyForAssignment>()
                        .IsTriggeredOn(graph => graph.ReleaseFromHold)
                        .When(!conditions.RequiresPrepayment));
                    transitionGroup.Add(transition =>
                        transition.To<States.pendingPayment>()
                        .IsTriggeredOn(graph => graph.ReleaseFromHold)
                        .When(conditions.RequiresPrepayment));
                });
            }))
```

Step 4 (Optional): Specifying the Order of Transitions

Transitions are checked based on their order of declaration in code. To specify that a transition should be checked before or after another transition, you should call the `PlaceBefore` or `PlaceAfter` method,

In *Step 3: Grouping Transitions and Adding Conditions to Transitions*, you have defined the following transitions from the `onHold` workflow state:

- A transition to the `readyForAssignment` workflow state
- A transition to the `pendingPayment` workflow state

Suppose that you want to make sure that the transition to the `pendingPayment`workflow state is always checked before the transition to the `readyForAssignment` workflow state.

To explicitly specify this order of transitions, call the `PlaceBefore` method in the definition of the transition from the `onHold` workflow state to the `pendingPayment` workflow state, as the following code shows.

```
                    .WithTransitions(transitions =>
                    {
                        transitions.AddGroupFrom<States.onHold>(transitionGroup =>
                        {
                            transitionGroup.Add(transition =>
                                transition.To<States.readyForAssignment>()
                                .IsTriggeredOn(graph => graph.ReleaseFromHold)
                                .When(!conditions.RequiresPrepayment));
                            transitionGroup.Add(transition =>
                                transition.To<States.pendingPayment>()
                                .IsTriggeredOn(graph => graph.ReleaseFromHold)
                                .When(conditions.RequiresPrepayment)
                                .PlaceBefore(transition =>
                                    transition.To<States.readyForAssignment>()));
                        });
                    }))
```

In this code,you have called the `PlaceBefore` method after calling the `When` method. Notice that you have passed in a lambda expression as a parameter to the `PlaceBefore` method, and within it, you have specified `transition.To<States.readyForAssignment>()` as the transition before which you want the transition being defined to be checked.

Step 5: Testing 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, create a new record on 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 *Transitions: To Implement a Transition Triggered by an Action*), because the service specified for the order requires prepayment. You can see that the service requires prepayment on the Repair Services (RS201000) form.

Defining Conditions

In this chapter, you will learn how to define the conditions that can be used in a screen configuration and in a worfklow. You can use conditions to specify when actions should be performed, when transitions should be triggered, and when certain fields and dialog boxes are available on the form.

Conditions: General Information

A *condition* is a Boolean statement that is checked in a workflow. When the statement is true, a workflow component can change to your specifications.

Learning Objectives

In this chapter, you will learn where and how a condition can be applied.

Applicable Scenarios

You can use a condition in the following ways:

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

Declaring of Conditions

You declare conditions in a class that inherits from the `Condition.Pack` class. In that 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 definition of conditions is shown in the following code.

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

You can combine conditions by using logical AND, OR, and NOT C# operators to dynamically create and register new conditions.

A condition that is declared using the `Condition.Pack` class is automatically registered in the screen configuration and does not need to be added to it manually.

Use of Conditions

You can use conditions in the following ways:

- To make an action unavailable depending on a condition.

  You should call the `IsDisabledWhen` method while adding an action to a screen configuration and provide a condition as a parameter. For an example, see *Workflow Actions: To Configure the Conditional Appearance of the Action*.
- To hide a field, make the field unavailable, or make the field required depending on a condition.

  You do this while configuring the state of the field in the screen configuration. For details, see *Field States: General Information*.
- To hide a dialog box field depending on a condition.

  You call the `IsHiddenWhen` method while adding this field to a dialog box and provide a condition as a parameter. For details, see *Workflow Dialog Boxes: Configuration of Fields*.
- To check whether a transition should be performed.

  To do this, you call the `When` method while adding the transition in a transition group and provide a condition as a parameter. This way, you can also create multiple transitions with a single initial workflow state if these transitions should be triggered by the same action or event. The transition is performed when the provided condition is true. An example is shown in the following code.

  ```
  .WithTransitions(transitions =>
  {
    transitions.AddGroupFrom(initialState, transitionGroup =>
    {
      transitionGroup.Add(transition => transition
        .To<State.balanced>()
        .IsTriggeredOn(graph => graph.initializeState)
        .When(conditions.IsBalanced)
    }
  }
  ```

  For the full example, see *Transitions: To Implement a Group of Transitions*.

Implementing Workflow Dialog Boxes

In this chapter, you will learn how to implement a dialog box that can be shown when a user clicks a button on the form toolbar or a command on the More menu to invoke a workflow action. In the workflow dialog box, the user provides particular field values, which can be used in a transition that is triggered by this action.

Workflow Dialog Boxes: General Information

A *workflow 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 or a menu command. A dialog box is referred to in code as a `form`.

This chapter describes how to define a dialog box that can be used in a workflow by using Workflow API.

Learning Objectives

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

- Define a dialog box in a workflow
- Specify a workflow dialog box for an action
- Assign values specified in the workflow dialog box to the DAC fields

Applicable Scenarios

You define a dialog box in a workflow when you need a user to provide particular field values when the user clicks a button or a menu command that corresponds to an action. These field values can later be used in a transition that is triggered by this action.

Definition of a Workflow Dialog Box

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

1. You define the dialog box in the static `Configure` method of the workflow class by using the `WorkflowContext.Forms.Create` method. In the method parameters, you provide the internal name of the dialog box and its configuration. 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.

   ```
   protected static void Configure(WorkflowContext<GraphType, DACType> context)
   {
     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 has been declared.

2. You configure fields and layout of the workflow dialog box as described in *Workflow Dialog Boxes: Configuration of Fields* and *Workflow Dialog Boxes: Configuration of the Layout*.

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)))
```

Every dialog box has the **OK** and **Cancel** buttons. You cannot modify the buttons of a dialog box.

Linking of a Workflow Dialog Box to an Action

To display a workflow 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 workflow dialog box field in 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 workflow dialog box and the field name as parameters.

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

```
actions.Add(graph => graph.close, action => action
  .WithForm(formClose)
  .WithFieldAssignments(fields =>
  {
    fields.Add<RequestForInformation.reason>(field =>
      field.SetFromFormField(formClose, "Reason"));
  }));
```

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

> The original code of the action is executed only after the workflow 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.

Workflow Dialog Boxes: Configuration of Fields

You configure fields of the workflow dialog box as described in the sections below.

Types of Fields in a Workflow Dialog Box

A workflow 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>();
```

> You can add two dependent fields, such as two fields of `PXSelector` type, to a workflow dialog box so that the value selected in one field updates the available values in the other field. The relationship between them should be defined in the DAC where they are declared.

- A rich text box field. It is defined with the `WithRichTextEditorField` method. An example is shown in the following code:

```
fields.Add("Reason", field => field
    .WithRichTextEditorField()
    .Prompt("Reason")
    .DefaultValue(defaultValue));
```

> The `WithRichTextEditorField` method must be the first method in the line of configuration methods. Also, you cannot use the `WithSchemaOf` method for the rich text box field.

Configuration of Fields in a Workflow Dialog Box

You can configure the parts of a workflow dialog box in the following ways:

- Specify the label text for each field by calling the `Prompt` method
- Specify that the value of any field is required unconditionally by calling the `IsRequired` method, or that the field is required depending on a condition by calling the `IsRequiredWhen` method
- Specify the default value of any field by calling one of the following methods:
  - `DefaultValue`
  - `DefaultExpression`
  - `DefaultValuefromSchemaField`
- Specify the location of a button on the workflow dialog box by calling one of the following methods:
  - `PlaceBefore`
  - `PlaceAfter`
- Hide any field by calling one of the following methods:
  - `IsHiddenWhen`

    This method hides a field if a condition is met. When using this method, you provide a condition as a parameter.

    > The condition specified in the `IsHiddenWhen` method is not checked during the mass processing of records.

  - `IsHiddenAlways`

    This method hides the field unconditionally. This method may be useful if you need to hide a field in a dialog box of a custom workflow if this field is included in this dialog box in a predefined workflow.

An example of using these methods is shown in the following code.

```
.WithForms(forms => {
  forms.Update("FormOpen", form =>
    form.WithFields(fields =>
    {
      fields.Add("Reason", field => field.WithSchemaOf<CROpportunity.resolution>()
        .Prompt("Reason")
```

```
        .IsHiddenWhen(hideReason));

      fields.Update("Stage", field => field.WithPrompt("Stage ID").IsHiddenAlways());
    }
  ));
})
```

In the code above, the `FormOpen` dialog box has been updated in the following ways:

- The `Reason` field of the `FormOpen` dialog box has been added; it will be hidden when the `hideReason` condition is *True*.
- The `Stage` field is always hidden.

Configuration of Combo Box Fields

For a combo box field, you can do the following:

- Define a set of combo box values by calling the `ComboBoxValues` method
- Specify that the set of combo box values of a dialog box field should be taken from a source or target state by calling the `ComboBoxValuesSource` method. In the method parameter, you should specify from where the values should be taken (which can be a source state, a target state, or explicit values specified in the workflow dialog box definition).

  For example, suppose that in a workflow state, a set of values for the `CRLead.resolution` field is specified. The workflow includes a transition where this workflow state is a target state, and a workflow dialog box that is shown during this transition. Further suppose that the workflow dialog box includes the `Reason` field, whose configuration is provided by the `CRLead.resolution` field (through the use of the `WithSchemaOf` method). To specify that the values of the `Reason` field should be taken from the target workflow state of a transition where the workflow dialog box is used, you should use the following code for the workflow dialog box.

  ```
  .WithForms(forms => {
    forms.Add("FormOpen", form =>
      form.WithFields(fields =>
      {
        fields.Add("Reason", field => field
          .WithSchemaOf<CRLead.resolution>()
          .IsRequired()
          .Prompt("Reason")
          .ComboBoxValuesSource(ComboBoxValuesSource.TargetState));
      }
    ));
  })
  ```

- Add a new value and its display name by calling the `ComboBoxValue` method.

  You should use this method when updating a field in a predefined workflow dialog box.
- Restrict the set of predefined combo box values by calling the `OnlyComboBoxValues` method.

Workflow Dialog Boxes: Configuration of the Layout

You can configure the layout of controls in a workflow dialog box in the following ways:

- Split controls into multiple columns by calling the `ColumnsCount` method in a workflow dialog box configuration.

  By default, all controls are displayed in a single column.

- Span multiple columns of a control by calling the `ColumnSpan` method in the configuration of a field in the workflow dialog box.
- Specify the size of a control by calling the `ControlSize` method in the configuration of a field in the workflow dialog box.
- Arrange the order of controls by calling the `PlaceBefore` and `PlaceAfter` methods in the configuration of a field in the workflow dialog box.

  These methods may be useful if you are updating a workflow dialog box in a predefined workflow. By default, all fields are displayed in the order in which they are added to the dialog box.

Workflow Dialog Boxes: To Implement a Transition with a Dialog Box

This activity will walk you through the process of creating a workflow dialog box and implementing a transition that uses the values the user provides in this dialog box.

Story

On the Repair Work Orders (RS301000) form, 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 the equivalent command on the More menu) and enter the name of the assignee in the dialog box. When the user clicks **OK**, the repair work order will get the *Assigned* status. You need to implement these changes in the customization project.

Process Overview

In the workflow code, you will define the **Assign** workflow dialog box. To do this, you will define the `Assign` action, register it in the workflow, and specify the dialog box for it. You will also define the `Assigned` workflow state and the transition from the `ReadyForAssignment` workflow state to the `Assigned` workflow state.

System Preparation

Make sure that you have done the following:

1. Prepared an instance with the *PhoneRepairShop* customization project and enabled the workflow validation by performing the following activities:
   a. *Test Instance for Workflow Customization: To Deploy a Test Instance*
   b. *Test Instance for Workflow Customization: To Turn On Workflow Validation*
2. Prepared the screen configuration and defined the set of states by performing the *Screen Configuration: To Prepare a Screen Configuration for a Form Without a Predefined Workflow* activity.
3. Implemented the `ReadyForAssignment`workflow state by performing the *Workflow States: To Define a Workflow State* activity.

Step 1: Defining a Workflow Dialog Box

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

Do the following:

1. In the static `Configure` method of the `RSSVWorkOrderEntry_Workflow` class, define the **Assign** workflow dialog box, as the following code shows.

```
            // Define the Assign dialog box
```

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

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

2. Inside the lambda expression for the `WithFields` method, add the `Assignee` field, which will be 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 workflow 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 in the lambda expression for the `AddScreenConfigurationFor` method, as the following code shows.

```
                .WithForms(forms => forms.Add(formAssign))
```

Step 2: Defining the Action That Opens the Workflow Dialog Box

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

2. In the `RSSVWorkOrderEntry_Workflow` class, in the lambda expression for the `WithActions` method at the screen configuration level, add the `Assign` action, as the following code shows.

```
                    actions.Add(graph => graph.Assign, action => action
                      .WithCategory(processingCategory)
                      .WithForm(formAssign)
                      .WithFieldAssignments(fields => {
                          fields.Add<RSSVWorkOrder.assignee>(field =>
                            field.SetFromFormField(formAssign, "Assignee"));
                      }));
```

In the code above, you have added the `Assign` action to the screen configuration. You have specified the workflow 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 3: Defining the Workflow State and the Transition (Self-Guided Exercise)

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

While defining a workflow state, you need to use the `States.assigned` class, which you added in *Workflow States: To Define a Workflow State*. In the `Assigned` workflow state, you should make the **Customer ID**, **Service ID**, and **Device ID** boxes unavailable for the state. Also, you need to add the `Assign` action to the `ReadyForAssignment` workflow state, display it on the form toolbar, and specify the *Success* connotation for it.

For details on defining transitions, see *Transitions: To Implement a Transition Triggered by an Action* and *Transitions: To Implement a Group of Transitions*. The transition should be defined inside a group and is performed without any conditions.

Step 4: Testing the Assign Action

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

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

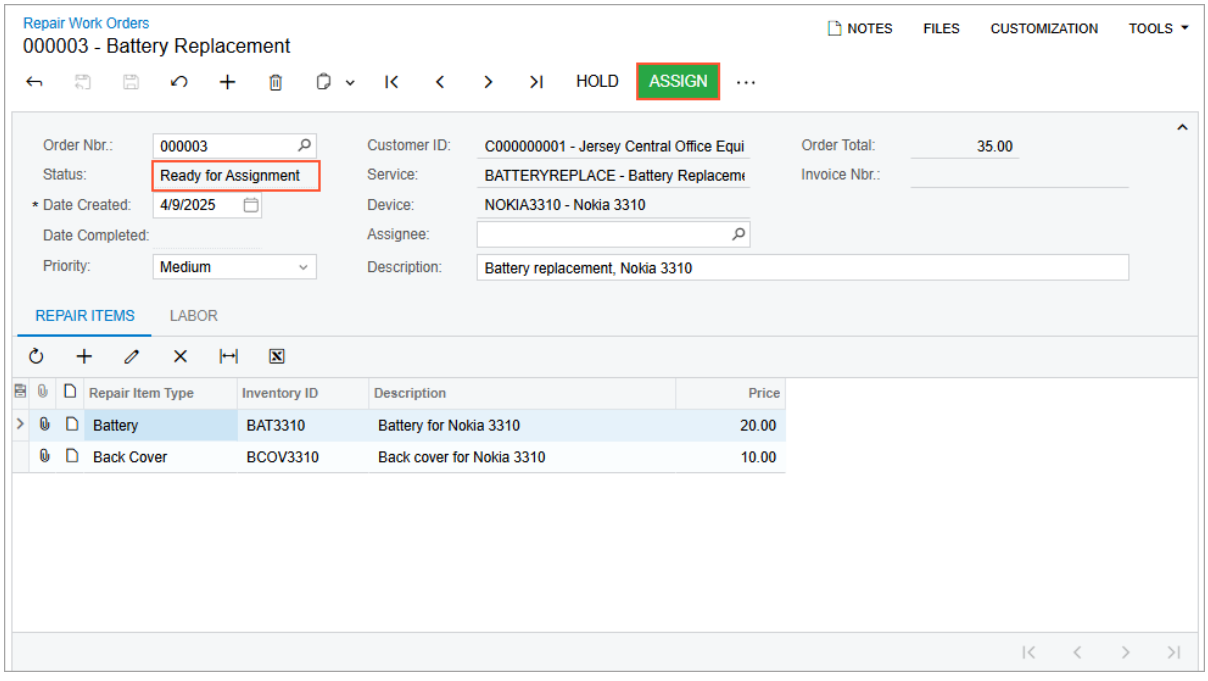


Figure: The Assign button

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

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

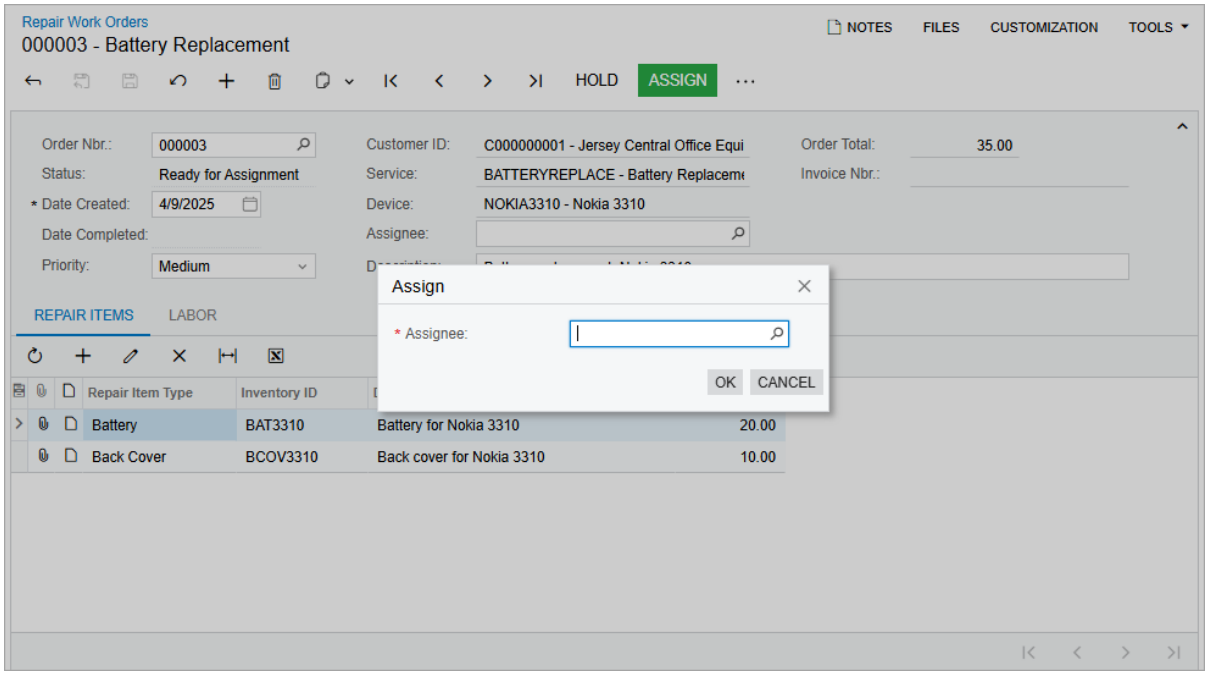


Figure: The Assign dialog box

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

Configuring Field States

This chapter describes how to configure field states in a screen configuration. For details on configuring field states in a workflow state, see *Workflow States: General Information*.

Field States: General Information

You can configure the states of fields, including default values and appearance, in a screen configuration.

Learning Objectives

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

- Add a field state to a screen configuration
- Specify the default value of a field
- Specify the set of values for a combo box field

Applicable Scenarios

You define field states in a screen configuration when you need to configure the properties of this field for all workflows of a form.

Adding or Updating of a Field State

To add a state of a field to a screen configuration or update the state, you call the `WithFieldStates` method in the screen configuration (that is, in the `AddScreenConfigurationFor` or `UpdateScreenConfigurationFor` method). Inside the `WithFieldStates` method, you can do the following:

- Add a field state by calling the `Add` method
- Update a field state by calling the `Update` method
- Replace the entire configuration of the field state by calling the `Replace` method
- Delete the entire configuration of the field state by calling the `Remove` method

Configuration of a Field State

You can configure a field state in the following ways:

- Configure the appearance and requirement of the field depending on a condition by calling the following methods:
  - `IsHiddenWhen`
  - `IsDisabledWhen`
  - `IsRequiredWhen`
- Make a field hidden, disabled, or required by calling the respective method:
  - `IsHiddenAlways`
  - `IsDisabledAlways`
  - `IsRequiredAlways`

  These methods may be useful to edit the properties of a field from a predefined screen configuration.

- Add a combo box value by calling one of the following methods:
  - `SetComboValue`
  - `SetComboValues`
- Specify a default value for the field by calling one of the following methods:
  - `WithDefaultValue`

    This method allows the developer to specify the exact value.
  - `WithDefaultExpression`

    This method allows the developer to provide a BQL expression that determines the proper value.

  An example of the use of this method is shown in the following code.

```
context.AddScreenConfigurationFor(screen =>
{
  return screen
    .StateIdentifierIs<CRCase.status>()
    .AddDefaultFlow(DefaultCaseFlow)
    ...
    .WithFieldStates(fields =>
    {
      fields.Add<CRCase.resolution>(field => field
        .SetComboValues(
          (_reasonRejected, "Rejected"),
          (_reasonResolved, "Resolved"),
          ...
          (_reasonUnassigned, "Unassigned"),
          (_reasonAssigned, "Assigned"),
          (_reasonUpdated, "Updated"))
        .WithDefaultValue("Assigned"));
    })
});
```

  In the code above, the `CRCase.resolution` field is added to the screen configuration. The default value of the field is specified in the `WithDefaultValue` method.

  > ⚠ If a field has combo box values, the default value should be one of the values listed in the `SetComboValues` method.

Field States: Levels of Configuration

You can configure a field on the following levels, which are listed in order:

1. In the definition of a field in a DAC or one of its extensions
2. In a screen configuration
3. In a workflow state

Field configurations defined on each level are applied to the next level. For example, if you specified a default value for a field in the `PXDefault` attribute in a DAC, this value is the default value for this field for the whole screen configuration, and you can change it in the screen configuration. But for some properties (see the table below), the configuration specified on the next level can only narrow what has been defined in the previous level.

The following table lists all possible configuration types and the levels where they can be changed.

Configuration Type	DAC	Screen Configuration	Workflow State	Comment
Visibility	+	+	+	The condition on the next level can only narrow what has been defined in the previous level.
Requirement	+	+	+	
Availability	+	+	+	
Specify default value	+	+	+	
Specify combo box value	+	+	+	On the workflow state level, you can only restrict the set of values.

Implementing Workflow Events

In this chapter, you will learn how to use events to trigger a transition of a workflow. You will learn about using Workflow API to work with existing events and to create new ones.

Workflow Events: General Information

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.

In this chapter, you will use existing events and create your own events in a custom workflow by using Workflow API.

Learning Objectives

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

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

Applicable Scenarios

You use workflow events when you need to implement cross-graph interaction for forms that use workflows.

For example, 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 workflow events.

Defining of a Workflow Event

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

- Define the workflow event itself
- Define a workflow event handler
- Bind the workflow event handler to the workflow event
- Register the workflow event handler in the workflow state where the corresponding workflow event should be fired

After you have declared a workflow event, you need to fire it.

Workflow Event

A workflow 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 the DAC the workflow event relates to.

To define a new workflow event, you do the following:

1. You create a 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.

2. In the class that inherits from the `PXEntityEvent.Container` class, you create the workflow event as a field of the `PXEntityEvent` type.

   The `PXEntityEvent` class has one or two generic type parameters. The first parameter must be the same one that you used in the class that inherits from the `PXEntityEvent.Container` class. This parameter shows that this workflow event happens on records of the specified DAC. You can use the second parameter to provide additional information about the workflow event that occurred. The second parameter usually holds an additional DAC record that corresponds to the workflow event.

An example of a workflow event definition is shown in the following code. Among other events, the `InvoiceLinked` workflow event is declared. This workflow event occurs when the invoice, which is defined as the second type parameter, is linked with an `SOOrderShipment` record.

```
public partial class SOOrderShipment : PXBqlTable, 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;
  }
}
```

Workflow Event Handler

You can define a workflow 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 need to 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 need to specify the DAC where the workflow event for this handler is defined. The optional third parameter can be used to further restrict the handler to only those workflow events that provide additional information for a record. The second and third parameters must match the type parameters of the workflow event for which the handler is defined.

Binding of a Handler to a Workflow Event

To bind a workflow event handler to a workflow event, in the screen configuration (inside the `AddScreenConfigurationFor` or `UpdateScreenConfigurationFor` method), 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>()
      //PXEntityEvent<SOOrder>
      .OfEntityEvent<SOOrder.Events>(event => event.ShipmentCreationFailed)
      // PXWorkflowEventHandler<SOOrder, SOOrder>
      .Is(graph => graph.OnShipmentCreationFailed)
```

```
      .UsesTargetAsPrimaryEntity()
      .DisplayName("Shipment Creation Failed");
  });
})
```

In the `OfEntityEvent` method, you need to select the workflow event declared in the DAC. The workflow event must match the type specified in the `WithTargetOf` method and the type of the current screen configuration context.

In the `Is` method, you need to specify the workflow event handler that matches the workflow event signature.

By calling the `UsesTargetAsPrimaryEntity` method, you specify that the workflow should use the record on which the workflow event is fired. For a cross-graph execution, you need to call the `UsesPrimaryEntityGetter` method instead and select the DAC record that should be used as the source record for the workflow event. For an example, see *Step 4: Declaring the Workflow Event* in the *Workflow Events: To Create a Workflow Event* activity.

Registration of the Workflow Event Handler in a Workflow State

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

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

Firing of a Workflow Event

To fire a workflow event, you select the workflow event from the list of events declared in the class derived from the `PXEntityEvent.Container` class and call the `FireOn` method. In the `FireOn` method parameters, you provide the graph that should be used as a host for workflow event execution, as well as the instances of DACs that are used as type parameters in the workflow event definition. An example of firing the `InvoiceLinked` workflow event 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, the `InvoicedLinked` event is selected from the list of `SOOrderShipment` events and the `FireOn` method is called. As parameters, the following have been provided: the instance of the graph that will be used as a host for workflow event execution, and the instances of `SOOrderShipment` and `SOInvoice` DACs. The signature of the `InvoiceLinked` event restricts the `FireOn` method to an overload that has these exact parameters.

Triggering of a Transition by a Workflow Event

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

```
transitions.AddGroupFrom<States.completed>(transitionGroup =>
{
  transitionGroup.Add(transition =>
    transition.To<States.linked>().IsTriggeredOn(graph => graph.OnInvoiceLinked));
});
```

Workflow Events: To Use an Existing Event

The following activity will walk you through the process of creating a workflow event handler for an existing workflow event and implementing a transition that is triggered by this event.

Story

Suppose that on the Repair Work Orders (RS301000) form, 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 is assigned the *Closed* status. You need to implement the needed changes for this system behavior.

Process Overview

To change the status of the repair work order from *Completed* to *Paid*, you will use an existing workflow event that is fired when the invoice document is assigned the *Closed* state. First, you will explore the code of the *Invoices* (SO303000) form to find the workflow event that is fired. Then, in your extension library, you will create a workflow event handler for this event, bind it to the workflow on the Repair Work Orders (RS301000) form, and implement the transition. You will also implement the `Paid` workflow state and override the `Persist` method on the *Invoices* form.

System Preparation

Make sure that you have done the following:

1. Prepared an instance with the *PhoneRepairShop* customization project and enabled the workflow validation by performing the following activities:
   a. *Test Instance for Workflow Customization: To Deploy a Test Instance*
   b. *Test Instance for Workflow Customization: To Turn On Workflow Validation*
   c. *Test Instance for Workflow Customization: To Configure the Instance*
2. Prepared the screen configuration and defined the set of workflow states by performing the *Screen Configuration: To Prepare a Screen Configuration for a Form Without a Predefined Workflow* activity.
3. Implemented the `Completed` workflow state by performing the *Workflow States: To Define a Workflow State* activity.

4. Implemented the `CreateInvoice` action by performing the *Workflow Actions: To Configure the Conditional Appearance of the Action* activity.

Step 1: Exploring the Acumatica ERP Source Code

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

To change the status of a repair work order whose invoice has been paid, you need to use a workflow 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. In this step, you will locate the `Release` action.

To locate the `Release` action, do the following:

1. In Acumatica ERP, open the *Payments and Applications* (AR302000) form.
2. To 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.

Step 2: Preparing the Project for Debugging

To find the workflow event that you can use in a customization project, you can debug the Acumatica ERP source code with breakpoints and see which breakpoint is hit in which scenario. In this step, you will prepare the `PhoneRepairShop_Code` for debugging in Visual Studio.

To prepare the `PhoneRepairShop_Code` 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 this activity.

   The PDB files were 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 was selected in the Acumatica ERP Installation wizard. When you create a new instance or update an existing one, these PDB files are copied to the `Bin` folder of the instance. If you did not select 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 *Acumatica ERP Installation On-Premises: To Install the Acumatica ERP Tools (Optional)*.

   A PDB file holds debugging and project state information that is used for incremental linking of a debug configuration of your program. In general, a PDB file contains the link between compiler instructions and 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 website.

b. On 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.

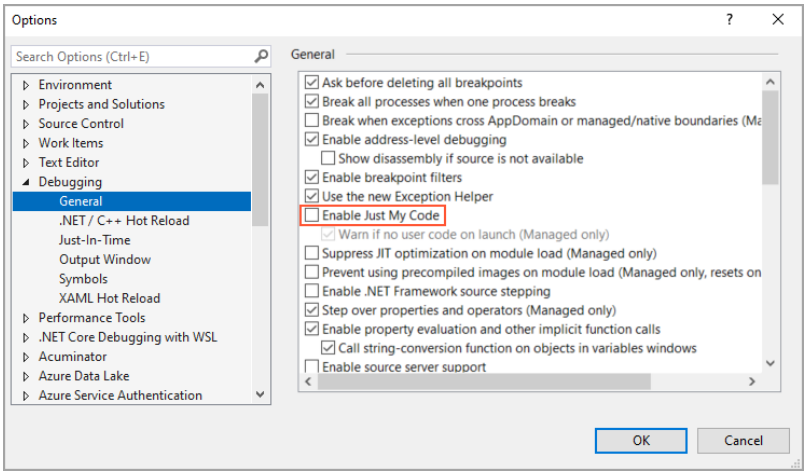


Figure: The cleared Enable Just My Code check box

d. In the **Debugging > Symbols** section, in the **Symbols file (.pdb) locations** list, add the path to the location of the PDB files in your Acumatica ERP instance.

e. Click **OK**.

4. 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 website folder in the Solution Explorer, select **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.

5. Add a breakpoint inside the `Release` method, as shown in the following screenshot.

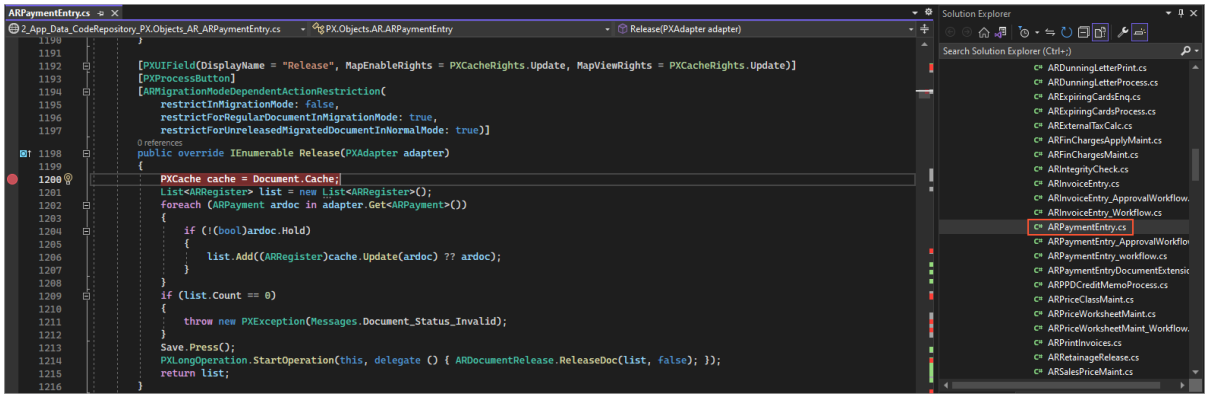


Figure: The source code of the Release action

6. Attach the Visual Studio debugger to the `w3wp.exe` running process.

   > For details about attaching to the process, see *To Debug the Customization Code*.

7. Start debugging by doing the following:
   a. In Acumatica ERP, open the *Payments and Applications* (AR302000) 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 would be created instead of the SO invoice. There is no difference for the release of a payment for AR and SO invoices, so you do not need to customize the closing of AR invoices as well.

Step 3: Exploring and Debugging 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 workflow event that you should use in the customization project. 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 which are displayed on the **Details** tab of the form.

   > 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 working 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.

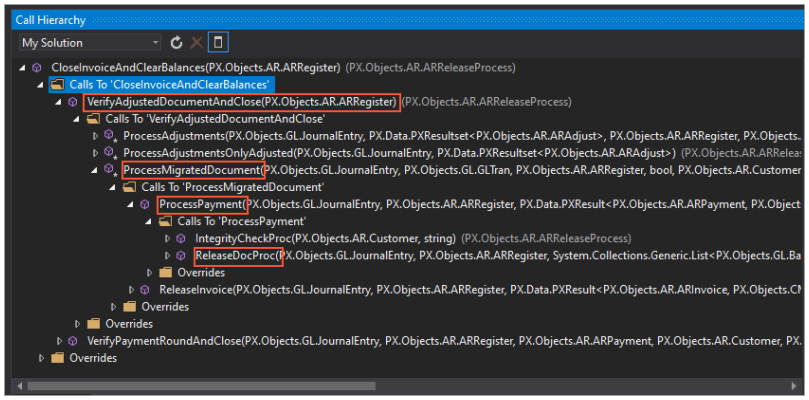


Figure: The call hierarchy for the CloseInvoiceAndClearBalances method

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(ARRegister doc, PX.Data.WorkflowAPI.SelectedEntityEvent<ARInvoice> invEvent)` method, you can find that the `CloseDocument` workflow 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 workflow 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` workflow 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` workflow event handler, as shown in the following screenshot.

```
transitions.AddGroupFrom<State.open>(ts =>
{
    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` workflow event in the `ARReleaseProcess.CloseInvoiceAndClearBalances` method indeed happens when an invoice is fully paid, do the following:
   a. Add a breakpoint to the line that fires the `OnCloseDocument` workflow event of the `ARReleaseProcess.CloseInvoiceAndClearBalances` method.
   b. Prepare a payment for debugging by doing the following:
      a. Open the invoice you created in *Workflow Actions: To Configure the Conditional Appearance of the Action* by searching for its number on the *Invoices* (SO303000) form.
      b. On the form toolbar of the *Invoices* (SO303000) 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* (AR302000) form, click **Remove Hold**, and then click **Release**.
   d. In Visual Studio, notice that a breakpoint in the `ARReleaseProcess.CloseInvoiceAndClearBalances` method was hit when an invoice was closed.
   e. Stop debugging.

Step 4: Defining the Paid Workflow State (Self-Guided Exercise)

To define a transition from the `Completed` workflow state to the `Paid` workflow state, you first need to define the `Paid` workflow 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 workflow state in *Workflow States: To Define a Workflow State*.

While defining the workflow state, you need to use the `States.paid` class, which you added in *Step 2: Defining the Set of States of the Workflow* of the *Screen Configuration: To Prepare a Screen Configuration for a Form Without a Predefined Workflow* activity. In the `Paid` workflow state, make the **Customer ID**, **Service ID**, and **Device ID** boxes unavailable.

Step 5: Defining the Workflow Event Handler

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

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

```
#region Workflow Event Handlers
public PXWorkflowEventHandler<RSSVWorkOrder, ARInvoice> OnCloseDocument
    = null!;
#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 `RSSVWorkOrderEntry_Workflow` class, bind the `OnCloseDocument` workflow event handler to the `CloseDocument` workflow event in the screen configuration: After the `AddDefaultFlow` method, call the `WithHandlers` method. In the method, add the `OnCloseDocument` event handler, as the following code shows.

```
.WithHandlers(handlers =>
{
    handlers.Add(handler => handler
        .WithTargetOf<ARInvoice>()
        .OfEntityEvent<ARInvoice.Events>(
            workflowEvent => workflowEvent.CloseDocument)
            .Is(graph => graph.OnCloseDocument)
            .UsesPrimaryEntityGetter<
                SelectFrom<RSSVWorkOrder>.
                Where<RSSVWorkOrder.invoiceNbr
                .IsEqual<ARRegister.refNbr.FromCurrent>>>());
})
```

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

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

```
using PX.Objects.AR;
using PX.Data.WorkflowAPI;
```

3. In the `GetCompletedBehavior` private static method, add the event handler by using the `WithEventHandlers` method, as the following code shows.

```
.WithEventHandlers(handlers =>
{
    handlers.Add(graph => graph.OnCloseDocument);
});
```

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

```
transitions.AddGroupFrom<States.completed>(
    transitionGroup =>
{
    transitionGroup.Add(transition =>
        transition.To<States.paid>()
        .IsTriggeredOn(graph => graph.OnCloseDocument));
});
```

5. Save your changes.

Step 6: Overriding the PerformPersist Method

The `Persist` method of any graph saves changes to the database. The `PerformPersist(IPersistPerformer persister)` method is a targeted override of the `Persist` method that offers you full control of the persist process without affecting any of the existing logic. We recommend that you use this method to override the persisting logic of a graph instead of overriding the `Persist` method directly.

In this activity, you are customizing an existing workflow to update another entity. For the changes to be saved to the database, you need to override the `PerformPersist(IPersistPerformer persister)` 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 `PerformPersist(IPersistPerformer persister)` method of the `ARReleaseProcess` graph so that the changes to a repair work order that are triggered by the workflow 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_Extension.cs` based on a C# template.
2. In the `ARReleaseProcess_Extension.cs` file, declare the `ARReleaseProcess_Extension` graph extension, as the following code shows.

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

Add the `using` directives shown in the following code.

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

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

```
public SelectFrom<RSSVWorkOrder>.View WorkOrdersForUpdate = null!;

[PXOverride]
public void PerformPersist(PXGraph.IPersistPerformer persister,
            Action<PXGraph.IPersistPerformer> base_PerformPersist)
```

```
        {
            base_PerformPersist(persister);
            persister.Update<RSSVWorkOrder>();
        }
```

> ⚠ When you override the `PerformPersist(IPersistPerformer persister)`, you must call the base version of the method. The only exception to this rule is when you need to suppress the base logic for some reason.

In the code above, you have declared a view for repair work orders and overridden the `PerformPersist` method. In the overridden method, you have called the base method and saved your changes to the database by calling the `PXGraph.IPersistPerformer.Update` method.

5. Save your changes.

Step 7: Testing the Transition

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

1. Rebuild the `PhoneRepairShop_Code` project.
2. In Acumatica ERP, on the Repair Work Orders (RS301000) form, open a repair work order that has the *Completed* status.
3. On the form toolbar, click **Create Invoice**.

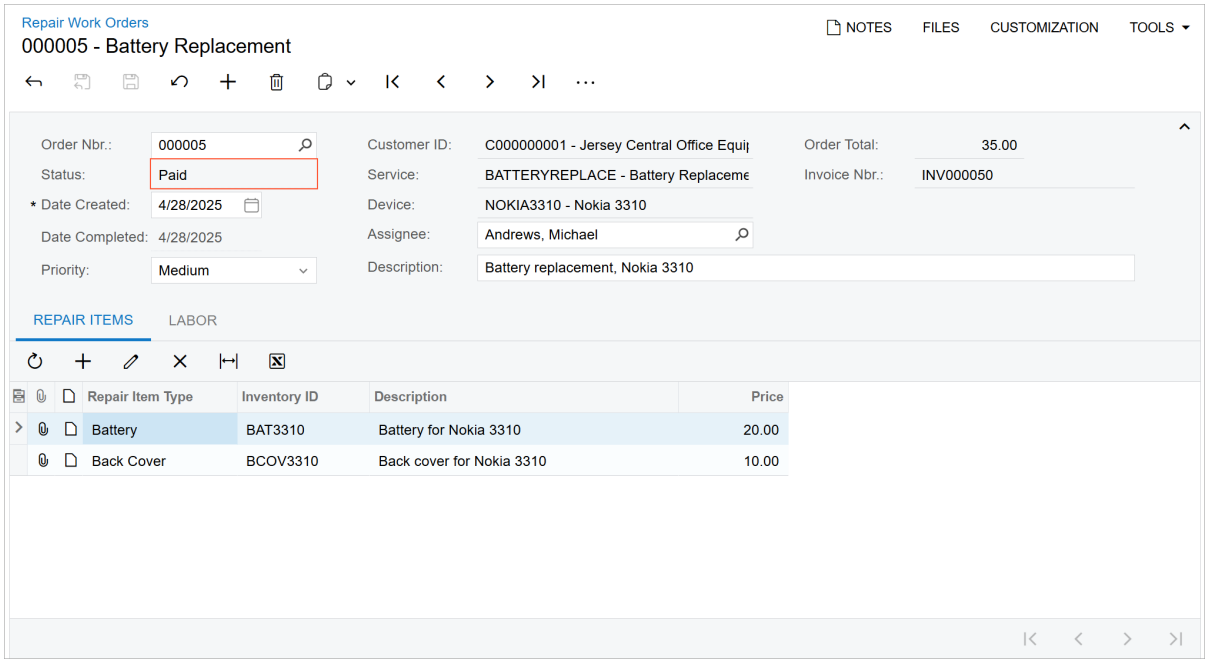
   Note the invoice number shown in the **Invoice Nbr** box.
4. On the *Invoices* (SO303000) form, open the invoice created in the previous instruction.
5. On the form toolbar, click **Remove Hold** and then **Release**.

   The invoice is assigned the *Open* status.
6. On the More menu, click **Pay**.

   The *Payments and Applications* (AR302000) form opens.
7. On the form toolbar, click **Remove Hold** and then **Release**.

   The invoice is now fully paid.
8. On the Repair Work Orders (RS301000) form, open the repair work order for which you have created the invoice in this step. Make sure that it has the *Paid* status, as shown in the following screenshot.

Figure: The repair work order with the Paid status



Workflow Events: To Create a Workflow Event

This activity will walk you through the process of creating a workflow event and implementing a transition by using this event.

Story

Suppose that 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 repair work order from *Pending Payment* to *Ready for Assignment* to indicate that the prepayment is applied. You need to implement the needed changes for this system behavior.

Process Overview

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

System Preparation

Make sure that you have done the following:

1. Prepared an instance with the *PhoneRepairShop* customization project and enabled the workflow validation by performing the following activities:
   a. *Test Instance for Workflow Customization: To Deploy a Test Instance*
   b. *Test Instance for Workflow Customization: To Turn On Workflow Validation*

c. *Test Instance for Workflow Customization: To Configure the Instance*

2. Prepared the screen configuration and defined the set of workflow states by performing the *Screen Configuration: To Prepare a Screen Configuration for a Form Without a Predefined Workflow* activity.
3. Implemented the `PendingPayment` and `ReadyForAssignment` workflow states by performing the *Step 2: Adding the PendingPayment Workflow State (Self-Guided Exercise)* and *Step 3: Defining the Workflow State and the Transition (Self-Guided Exercise)* prerequisite steps.
4. Implemented the `CreateInvoice` action by performing the *Workflow Actions: To Configure the Conditional Appearance of the Action* prerequisite activity.

Step 1: Creating a Custom Field

In Acumatica ERP, a user can create a payment for an invoice by clicking the **Pay** button or command 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* (AR302000) 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* (AR302000) form.

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

Adding a Custom Field to the Payments and Applications Form

To display and modify the prepayment percentage on the *Payments and Applications* (AR302000) form, you need to add the **Prepayment Percent** box to the form. Complete the following general tasks:

1. By using the Element Inspector, learn the name of the DAC and graph that define the Summary area of the *Payments and Applications* (AR302000) form. In this case, the DAC is `ARPayment` and the graph is `ARPaymentEntry`. You need the DAC name to know which database table and DAC to extend. You will need the graph name in the next step.
2. Add a column named `UsrPrepaymentPercent` to the `ARPayment` table with the same parameters as are specified for the `PrepaymentPercent` field of the `RSSVSetup` table. The data type of the column is `decimal(9, 6)`.
3. Create an extension of the `ARPayment` DAC, and add the `UsrPrepaymentPercent` field to the extension.

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

The code for the field is shown below.

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

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

4. Add a box for the `UsrPrepaymentPercent` field to the Summary area of the *Payments and Applications* (AR302000) form. On the *Customized Screens* page, the location of the element appears as shown in the following screenshot.

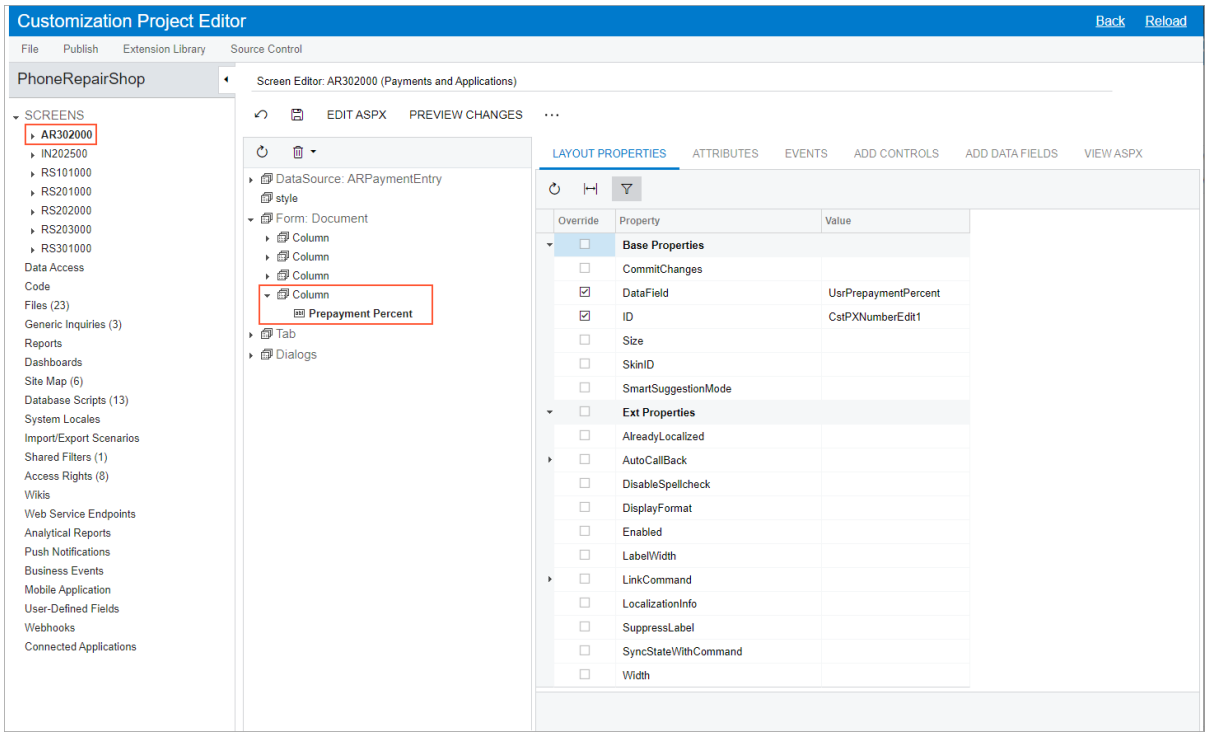


Figure: Prepayment Percent element

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

Step 2: Deriving 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 using one of the following:

- The `FieldDefaulting` event
- The `PXDefault` attribute

To populate the `UsrPrepaymentPercent` field of the `ARPayment` DAC extension when a payment is created, you will 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.

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

```
    }
```

2. Add the following `using` directives.

```
using PX.Data;
using PX.Data.BQL.Fluent;
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. 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)
        {
            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 `setupRecord` so that the `NullReferenceException` exception is not thrown if the data on the form has not been filled in yet.

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

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

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

This approach provides the following advantages:

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

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

Step 3: Exploring the Acumatica ERP Source Code

In this step, you will explore the source code of the *Invoices* (SO303000) and *Payments and Applications* (AR302000) forms of Acumatica ERP. You will explore the code of these 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` state to the `ReadyForAssignment` state.

Learning Field Names

To determine whether an invoice has been prepaid and whether the workflow event that triggers the transition from the `PendingPayment` workflow state to the `ReadyForAssignment` workflow 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. In Acumatica ERP, open the *Invoices* (SO303000) 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, open or refresh the *Invoices* (SO303000) form, and then perform this instruction.

Exploring the Source Code of the Release Method

To fire the workflow event that triggers the transition from the `PendingPayment` to `ReadyForAssignment` workflow 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.

If you explore the code of the **Release** button (as described in *Step 3: Exploring and Debugging the Code*), you can notice in the code 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 `UpdateBalances` is the method that you should override to calculate the prepaid percentage of the invoice and to fire the workflow event that triggers the transition from the `PendingPayment` workflow state to the `ReadyForAssignment` workflow state.

Step 4: Declaring the Workflow Event

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

Do the following:

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

```
        public class WorkflowEvents :
```

```
        PXEntityEvent<ARRegister>.Container<WorkflowEvents>
{
    public PXEntityEvent<ARRegister> InvoiceGotPrepaid = null!;
}
```

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

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

```
public PXWorkflowEventHandler<RSSVWorkOrder, ARRegister> OnInvoiceGotPrepaid
    = null!;
```

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

```
handlers.Add(handler => handler
    .WithTargetOf<ARRegister>()
    .OfEntityEvent<RSSVWorkOrder.WorkflowEvents>(
        workflowEvent => workflowEvent.InvoiceGotPrepaid)
        .Is(graph => graph.OnInvoiceGotPrepaid)
        .UsesPrimaryEntityGetter<
            SelectFrom<RSSVWorkOrder>.
            Where<RSSVWorkOrder.invoiceNbr
            .IsEqual<ARRegister.refNbr.FromCurrent>>>());
```

4. In the `GetPendingPaymentBehavior` private static method, add the workflow event handler by using the `WithEventHandlers` method, as the following code shows.

```
.WithEventHandlers(handlers =>
{
    handlers.Add(graph => graph.OnInvoiceGotPrepaid);
});
```

5. In the lambda expression for the `WithTransitions` method, declare the transition from the `PendingPayment` workflow state to the `ReadyForAssignment` workflow state in the `transitions.AddGroupFrom<States.pendingPayment>` group, as the following code shows.

```
transitionGroup.Add(transition =>
    transition.To<States.readyForAssignment>()
    .IsTriggeredOn(graph => graph.OnInvoiceGotPrepaid));
```

6. Save your changes.

Step 5: Firing the Event

As you learned in *Step 3: Exploring the Acumatica ERP Source Code* of this activity, you need to override the `UpdateBalances` method to fire the event that triggers the transition from the `PendingPayment` workflow state to the `ReadyForAssignment` workflow 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, which are available in an `ARRegister` record, have been calculated in the base `UpdateBalances` method.

> You can work with the fields of the `ARRegister` record instead of selecting the corresponding `ARInvoice` record because `ARRegister` is the base class for the `ARInvoice` data access class (DAC) and contains the same essential fields.

The `ARRegister` record has been modified in the base method and its field values have been updated in `PXCache`. But the values passed as the parameter to the overridden method have not been updated. Thus, in the overridden method, you should use the cached values. To get the cached values, you need to use the `Locate(Object)` method of *`PXCache<TNode>`*.

Firing the Workflow Event

To fire the workflow event that triggers the transition between workflow states, 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.WorkflowEvents
                          .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 values of the invoice record. You have selected the repair work order with the same invoice number as the invoice passed as a parameter. Then you have selected the prepayment and its extension (which contains the prepayment percent) and calculated the prepaid percentage for the invoice. If the prepaid percentage is greater than the required percentage, you have fired the `InvoiceGotPrepaid` event.

> ⚠ You should select both SO and AR payments and SO and AR invoices by using the `ARPayment` and `ARInvoice` DACs, respectively. To avoid selecting a 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 6: Testing the Transition

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

1. Rebuild the `PhoneRepairShop_Code` project.
2. In Acumatica ERP, on the Repair Work Orders (RS301000) form, open the repair work order with 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 is assigned the *Open* status.
6. On the More menu, click **Pay**.

   The *Payments and Applications* (AR302000) form opens. Notice that the **Prepayment Percent** box contains *10.00*, which has been copied from the Repair Work Order Preferences (RS101000) form.
7. On the *Payments and Applications* (AR302000) form, change the existing values to the following values, as shown in the following screenshot:
   - **Prepayment Percent**: `5.00`
   - **Amount Paid** (in the only row of the **Documents to Apply** tab): `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.

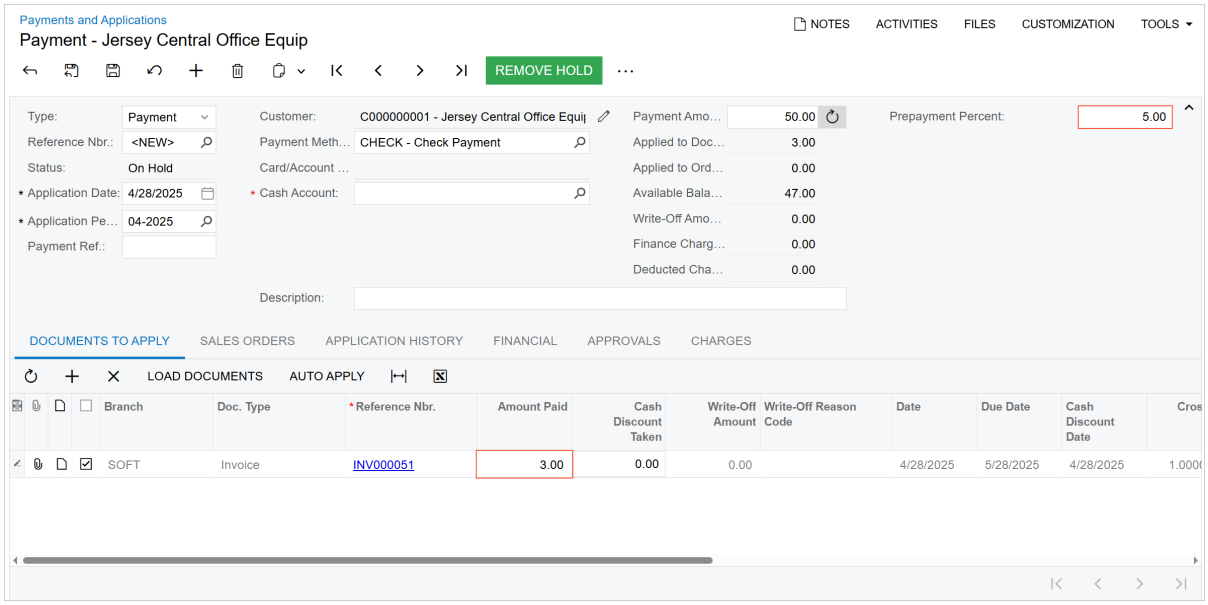


Figure: The prepayment for the created invoice

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

   The prepayment is applied.

9. On the Repair Work Orders form, open the repair work order for which you have created the invoice in this step.

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

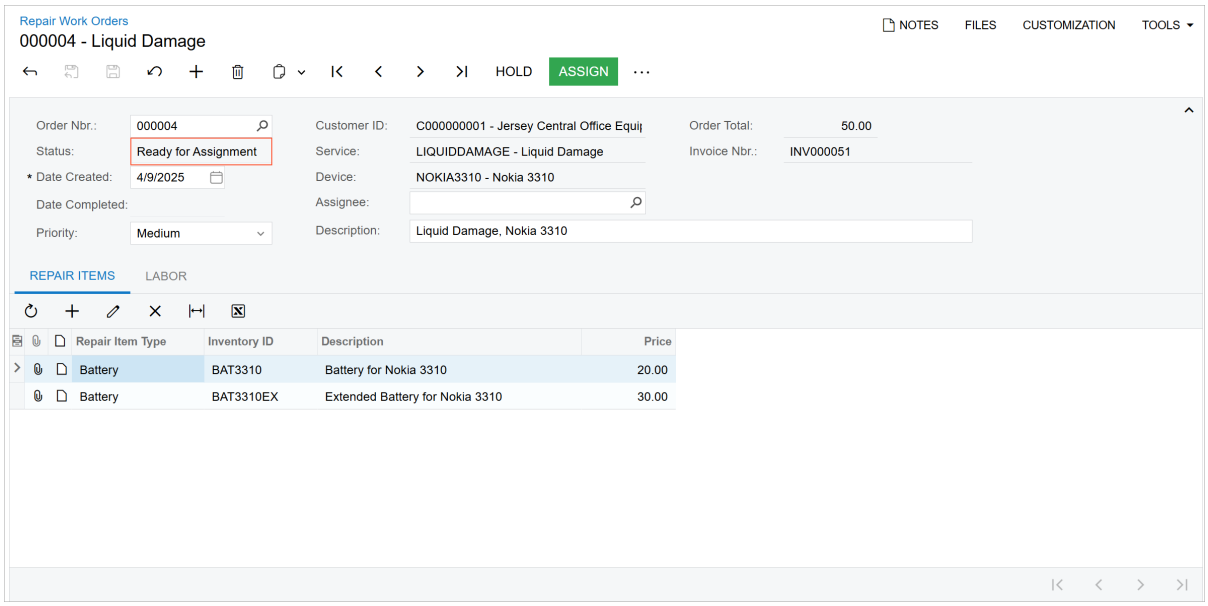


Figure: The repair work order with the Ready for Assignment status

Customizing a Predefined Workflow

In this chapter, you will learn how to customize a predefined workflow, which is a workflow that is provided as part of Acumatica ERP for a form. You customize a workflow by creating an extension of the workflow and modifying the screen configuration in this extension.

Workflow Customization: General Information

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

Learning Objectives

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

- Customize an existing workflow
- Add a new action to the workflow
- Define a new category in the workflow

Applicable Scenarios

You customize an existing workflow when you need to modify a workflow defined for an Acumatica ERP form.

Customization of Existing Workflows

You use the `Add` methods to add actions, states, transitions, and other elements of the screen configuration to the workflow. For each of these `Add` methods, Workflow API provides the `Update` method for updating the element and the `Delete` method for deleting the element from the workflow.

To define a new screen configuration and workflow, you use the `AddScreenConfigurationFor` and `AddDefaultFlow` methods to add a workflow to a form. Similarly, for the `AddScreenConfigurationFor` and `AddDefaultFlow` methods, Workflow API provides the `UpdateScreenConfigurationFor` and `UpdateDefaultFlow` methods, which 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`, and `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(
        graph => graph.putOnHold,
          action => action.WithFieldAssignments(fields =>
          {
            fields.Add<ARRegister.approved>(field => field.SetFromValue(false));
```

```
            fields.Add<ARRegister.rejected>(field => field.SetFromValue(false));
          }));
      actions.Update(
        graph => graph.releaseFromCreditHold,
        action => (BoundedTo<ARInvoiceEntry, ARInvoice>.
          ActionDefinition.ConfiguratorAction)action
          .InFolder(approvalCategory, reject));
    });
});
```

Workflow Customization: To Add an Action to an Existing Workflow

This activity will walk you through the process of adding an action to an existing workflow.

Story

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. You will customize the workflow on the *Invoices* (SO303000) form to add the **View Repair Work Order** command and its underlying action, which opens the repair work order. The **View Repair Work Order** command should be displayed under a new category called **Repair Work Orders**.

Process Overview

To define a new action, in your extension library, you will first define the new action in a graph extension, and then you will create an extension for the updated workflow. In this extension, you will add a new action category and the new action to the screen configuration.

System Preparation

Make sure that you have done the following:

1. Prepared an instance with the *PhoneRepairShop* customization project and enabled the workflow validation by performing the following activities:
   a. *Test Instance for Workflow Customization: To Deploy a Test Instance*
   b. *Test Instance for Workflow Customization: To Turn On Workflow Validation*
2. Prepared the screen configuration and defined the set of workflow states by performing the *Screen Configuration: To Prepare a Screen Configuration for a Form Without a Predefined Workflow* activity.
3. Performed the *Workflow Events: To Use an Existing Event* prerequisite activity to implement the `Paid` workflow state.

Step 1: Adding a Graph Action

In this step, you will create an extension of the 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* (SO303000) 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 PhoneRepairShop
{
    // Acuminator disable once PX1016 ExtensionDoesNotDeclareIsActiveMethod extension
 should be constantly active
    public class SOInvoiceEntry_Extension : PXGraphExtension<SOInvoiceEntry>
    {
    }
}
```

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

3. In the graph extension, implement the `viewOrder` action, as the following code shows.

```
        public PXAction<ARInvoice> ViewOrder = null!;
        [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. Make sure the following `using` directives are added.

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

5. Save your changes.

Step 2: Adding a 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* (SO303000) 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 step.

2. In the `Workflows` folder of the `PhoneRepairShop_Code` project, create the `SOInvoiceRepairOrder_Workflow.cs` file.
3. In the `SOInvoiceRepairOrder_Workflow.cs` file, in the `PhoneRepairShop` namespace, define the `SOInvoiceRepairOrder_Workflow` class as the following code shows.

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

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

In the code above, you have defined an extension of the `SOInvoiceRepairOrder_Workflow` class. As a second parameter of the extension, you have specified the graph of the *Invoices* (SO303000) form. In the extension, you have overridden the `Configure(PXScreenConfiguration)` method, which initializes the screen configuration, and declared the static `Configure(WorkflowContext)` method where you will update the workflow.

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

```
using PX.Data.WorkflowAPI;
using PX.Data;
using PX.Objects.SO;
using PX.Objects.AR;
```

5. Save your changes.

Step 3: Defining a 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 Work Orders Category";

        [PXLocalizable]
        public static class DisplayNames
        {
```

```
        public const string RepairOrders = "Repair Work Orders";
    }
}
```

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

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

2. In the static `Configure(WorkflowContext)` method, add the definition of the **Repair Work Orders** category and the `viewOrder` 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>(graph => graph.ViewOrder,
    action => action.WithCategory(repairCategory));
```

In the code above, you have created a workflow definition corresponding to the **Repair Work Orders** category of the More menu by using the `Categories.CreateNew` method. You have also created a workflow definition of the `viewOrder` action by using the `ActionDefinitions.CreateExisting` method. As the type parameter of the `CreateExisting` method, you have specified 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 `viewOrder` action to the `Open` state of the workflow.

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

4. Save your changes.

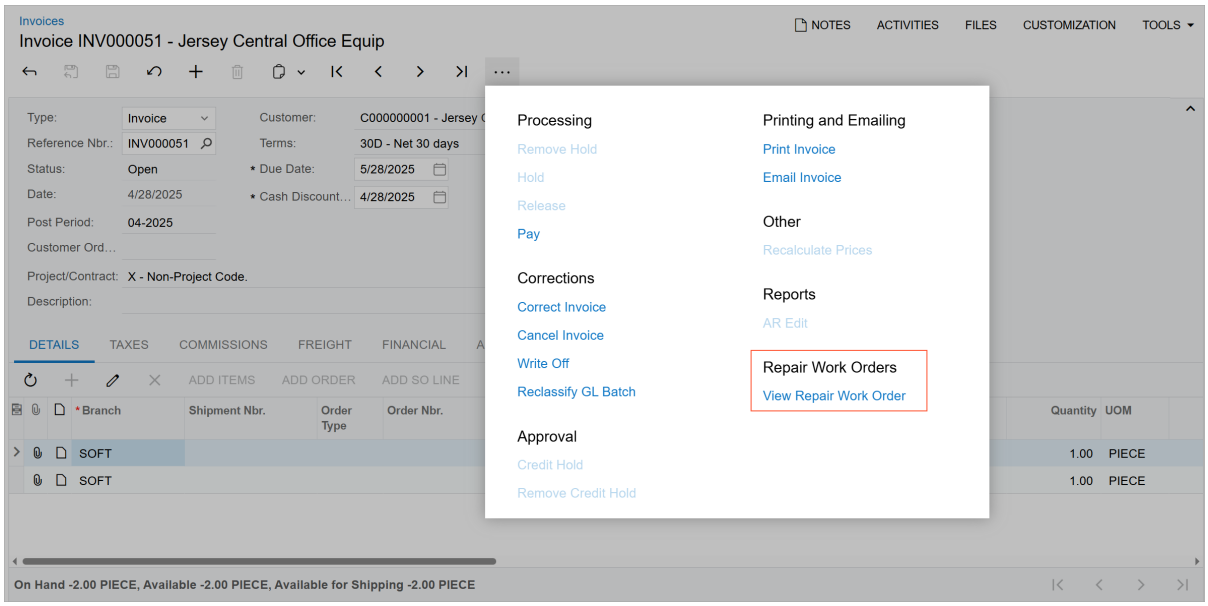
Step 4: Testing the New Action

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, open a repair work order with the *Completed* status. Create an invoice for this order by clicking **Create Invoice** on the form toolbar.
3. On the *Invoices* (SO303000) form, open the invoice that you have created for the repair work order.

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

   Figure: The More menu of the Invoices form



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

   In a new window, the Repair Work Orders (RS301000) form opens with the repair work order for which the opened invoice was created.
6. Return to the *Invoices* (SO303000) form.
7. Select an invoice with the *Closed* status. Notice that the **View Repair Work Order** command is unavailable on the More menu because this invoice has the *Closed* status.

Implementing Composite States

In this chapter, you will learn how to define a sequence of states and transitions using composite states. You will also learn how to customize an predefined composite state by adding a new state to it.

Composite Workflow States: General Information

A composite state of a workflow, which can also be referred to as a *parent workflow state*, is a workflow state that contains an ordered sequence of nested workflow states and the transitions to and from these states. With composite workflow states, you can specify common settings (actions, fields to be updated, and transitions) for a group of workflow states only once—in the composite state that includes these workflow states.

Learning Objectives

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

- Implement a composite state of a workflow
- Define a transition for a composite state of a workflow
- Update a composite state in a predefined workflow

Applicable Scenarios

You implement a composite state when you have a set of workflow states with common properties and you want to simplify the process of moving an entity though all these states. For example, you may need to implement a composite state in any of the following scenario examples:

- You need to implement a workflow with a sequence of workflow states and some of these states can be omitted for particular entities depending on conditions.
- You need to implement a workflow in which a transition to one of the workflow states is possible from multiple workflow states and this transition is triggered by the same action.
- You need to implement a workflow in which all transitions to one of the workflow states should specify the same value of a field (such as *True*), and all transitions from this workflow state should specify another same value of this field (such as *False*).

Use of Composite Workflow States

For each of the workflow states inside the composite state, you can do the following:

- Specify whether this workflow state should be skipped by using a skip condition
- Specify a transition to the next workflow state inside the composite state instead of a transition to a specific workflow state
- Specify a transition to the workflow state that is the next one after the composite state if this composite state is itself a nested workflow state in another composite state
- Override the settings inherited from the composite state, if needed

These capabilities make workflow customization much easier. You do not need to explicitly specify target workflow states for transitions. Thus, if you add or remove states in the workflow or change the order of states, you do not need to modify all the affected transitions.

Implementation of a Composite Workflow State

You add a composite workflow state inside the `AddFlowStates` method by using the `AddSequence` method. You specify the name of the composite workflow state as the type parameter of the method. In the lambda expression provided for the `AddSequence` method, you can add the workflow states that are a part of the composite state by using the `WithStates` method. The workflow states in a composite state are declared the same way as they are declared in the `AddFlowStates` method.

To specify that a workflow state should be skipped in the composite state, you call the `IsSkippedWhen` method in the workflow state definition. In the parameter of the `IsSkippedWhen` method, you specify the condition that determines whether the workflow state should be skipped.

The following code shows an example of the declaration of a composite workflow state.

```
.WithFlowStates(flowStates =>
{
  flowStates.Add(initialState, flowState => flowState
    .IsInitial(graph => graph.initializeState));
  flowStates.AddSequence<State.HoldToBalance>(seq =>
    seq.WithStates(seqStates =>
    {
      seqStates.Add<State.hold>(flowState =>
      {
        return flowState
          .IsSkippedWhen(conditions.IsNotOnHold)
          .WithActions(actions =>
          {
            ...
          });
      });
      seqStates.Add<State.creditHold>(flowState =>
      {
        return flowState
          .IsSkippedWhen(conditions.IsCreditHoldChecked)
          .WithActions(actions =>
          {
            ...
          });
      });
      seqStates.Add<State.balanced>(flowState =>
      {
        return flowState
          .WithActions(actions =>
      {
        ...
      });
    });
  }));
}
```

In the code above, the `HoldToBalance` composite workflow state is declared. The composite state includes the following wokflow states:

- `hold`, which is skipped when the `IsNotOnHold` condition is *true*
- `creditHold`, which is skipped when the `IsCreditHoldChecked` condition is *true*
- `balanced`, which is never skipped

Declaring of a Transition for a Composite Workflow State

You can define the following transitions involving a composite workflow state:

- A transition from any state of a workflow to a composite workflow state.
- A transition from a composite workflow state to any state of a workflow.

  This transition is inherited by the workflow states included in the composite state.
- A transition from a specific workflow state of the composite state to any state of the workflow.
- A transition from a composite workflow state to itself so that the workflow engine can search for the proper workflow state and check conditions again.

  To define this transition, in the `Add` method for the transition, you specify the target workflow state, which is the same as the source workflow state.
- A transition from one workflow state included in a composite state to the workflow state defined after it.

  To define this transition, in the `Add` method for the transition, you call the `ToNext` method.
- If a composite workflow state is defined inside a parent composite state, a transition from a workflow state in the child composite state to the next state in the parent composite state.

  To define this state, in the `Add` method for a transition, you call the `ToParentNext` method.

A composite workflow state cannot be the initial state of a workflow. You must specify a transition from an initial state to the composite state.

The following code shows an example of transitions that include a composite workflow state.

```
.WithTransitions(transitions =>
{
  transitions.AddGroupFrom(initialState, transitionGroup =>
  {
    transitionGroup.Add(transition => transition.To<State.HoldToBalance>()
      .IsTriggeredOn(graph => graph.initializeState)); // To composite state
  });
  transitions.AddGroupFrom<State.HoldToBalance>(transitionGroup =>
  {
    transitionGroup.Add(transition => transition
      .To<State.HoldToBalance>()
      .IsTriggeredOn(graph => graph.OnUpdateStatus)
      .When(conditions.IsARInvoice));
    transitionGroup.Add(transition => transition
      .To<State.open>()
      .IsTriggeredOn(graph => graph.OnReleaseDocument)
      .When(conditions.IsOpen));
  }
}
```

In the code above, the following transitions are declared:

- A transition from the initial workflow state to the `HoldToBalance` composite workflow state
- A transition from the `HoldToBalance` composite workflow state to itself
- A transition from the `HoldToBalance` composite workflow state to the `Open` workflow state

Customization of an Existing Composite State

You can customize a composite workflow state that has been defined in the source code of Acumatica ERP. To customize a composite state, you use the `UpdateSequence` method in the `WithFlowStates` method. In

the `WithStates` method of the `UpdateSequence` method, you can add, update, or delete a workflow state by using the `Add`, `Update`, or `Delete` methods, respectively. To specify the location of a workflow state in the updated composite state, you use the `PlaceAfter` method.

The following code shows an example of an updated composite state.

```
.WithFlowStates(states =>
{
  states.UpdateSequence<State.HoldToBalance>(seq =>
    seq.WithStates(seqStates =>
    {
      seqStates.Add<State.pendingApproval>(flowState =>
      {
        return flowState
          .IsSkippedWhen(conditions.IsApproved)
          .WithActions(actions =>
          {
            ...
          })
          .PlaceAfter<State.hold>();
      });
    }));
})
.WithTransitions(transitions =>
{
  transitions.AddGroupFrom<State.pendingApproval>(transitionGroup =>
  {
    transitionGroup.Add(transition => transition
      .To<State.HoldToBalance>()
      .IsTriggeredOn(graph => graph.OnUpdateStatus));
    transitionGroup.Add(transition => transition
      .ToNext()
      .IsTriggeredOn(aproveAction)
      .When(conditions.IsApproved));
    transitionGroup.Add(transition => transition
      .To<State.rejected>()
      .IsTriggeredOn(rejectAction)
      .When(conditions.IsRejected));
  });
});
```

In the code above, the `pendingApproval` workflow state has been added to the `HoldToBalance` composite workflow state and placed after the `hold` workflow state. Three transitions have been added from the `pendingApproval` workflow state.

Composite Workflow States: To Update a Composite State

This activity will walk you through the process of adding a nested workflow state to a composite state of a predefined workflow.

Story

Suppose that you need to implement the following behavior on the *Invoices* (SO303000) form:

- If a user specifies a discount for an invoice, you want this invoice to be reviewed and the discount to be approved before the user can proceed with processing the invoice. The invoice with the discount on review

should have the *Postponed* status. After the discount is approved, the system specifies the business date in the **Cash Discount Date** box.

- If the user does not specify any discount, the invoice does not obtain the *Postponed* status.

On the *Invoices* (SO303000) form, the default workflow includes a composite workflow state with multiple nested workflow states. To implement the described behavior, you need to add a new workflow state, `Postponed`, to the composite state after the workflow state that corresponds to the *Credit Hold* status of an invoice.

You also need to specify a skip condition for the new workflow state and add a transition from this workflow state to the next state in the workflow. The skip condition will check for no cash discount being applied to the invoice.

Process Overview

In the extension of the predefined workflow for the *Invoices* (SO303000) form, you will implement an action (and the corresponding button on the form toolbar and command on the More menu) that will trigger a transition from the new state. You will also add a nested workflow state, specify a condition that the system will use to skip this state, and then add a transition from this state to the next nested workflow state.

System Preparation

Make sure that you have done the following:

1. Prepared an instance with the *PhoneRepairShop* customization project and enabled the workflow validation by performing the following activities:
   a. *Test Instance for Workflow Customization: To Deploy a Test Instance*
   b. *Test Instance for Workflow Customization: To Turn On Workflow Validation*
2. Prepared the screen configuration and defined the set of workflow states by performing the *Screen Configuration: To Prepare a Screen Configuration for a Form Without a Predefined Workflow* activity.
3. Created an extension of the predefined workflow for the *Invoices* (SO303000) form, as described in the *Workflow Customization: To Add an Action to an Existing Workflow* prerequisite activity.

Step 1: Investigating the Source Code

To begin customizing the predefined workflow, you need to investigate how the composite workflow state is defined in it. In *Step 2: Adding a Workflow Extension* of the *Workflow Customization: To Add an Action to an Existing Workflow* activity, you learned that the workflow for the *Invoices* (SO303000) form is defined in the `SOInvoiceEntry_Workflow` class and that the states of the workflow are defined in the `ARDocStatus` class.

You will investigate the source code of the `SOInvoiceEntry_Workflow` class by doing the following:

1. In Visual Studio, in the *SmartFix_T270* website project, go to `App_Data/CodeRepository/PX.Objects/SO/Workflow`, and open the `SOInvoiceEntry_Workflow.cs` file.
2. In the `SOInvoiceEntry_Workflow` class, locate the `AddSequence<State.HoldToBalance>` method call. This call defines the `HoldToBalance` composite workflow state. Notice the order in which the nested workflow states are defined, and locate the `creditHold` workflow state. After the `creditHold` workflow state, you can see the `pendingPrint` workflow state, which will be the next state for the new `Postponed` state of the workflow.
3. Go to the definition of the `ARDocStatus` class, and note how the workflow states are defined.
4. For the purpose of adding a new status value, learn the name of the field that holds the status value on the *Invoices* (SO303000) form by using the *Element Inspector*. This field is `ARInvoice.Status`.

Step 2: Extending the ARDocStatus class

Before you add the `Postponed` state to the workflow, you need to add a string constant and class to the extension of the `ARDocStatus` class. You also need to add a corresponding value to the field that holds the record's status on the *Invoices* (SO303000) form. Do the following:

1. In the `SOInvoiceRepairOrder_Workflow.cs` file, extend the `ARDocStatus` class by adding the following code.

```
public class ARDocStatus_Postponed : ARDocStatus
{
    public const string Postponed = "O";
    public class postponed : BqlType<IBqlString, string>.Constant<postponed>
    {
        public postponed()
            : base("O")
        {
        }
    }
}
```

2. Add a combo box value to the **Status** box on the *Invoices* (SO303000) form by calling the `WithFieldStates` method in the lambda expression for the `UpdateScreenConfigurationFor` method, as the following code shows.

```
                .WithFieldStates(fs =>
                {
                    fs.Add<ARInvoice.status>(state =>
                        state.SetComboValue(ARDocStatus_Postponed
                          .Postponed, "Postponed"));
                })
```

3. Save your changes.

Step 3: Adding the ApproveDiscount Action

By using Workflow API, you can define actions without implementing them in a graph. You do this by calling the `ActionDefinitions.CreateNew` method and specifying all settings of the action, including the display name and category. This approach can be used only if the action does not have complex logic. In this step, you will define the `approveDiscount` action without implementing it in a graph.

To add the `approveDiscount` action, do the following:

1. In the `SOInvoiceRepairOrder_Workflow` class, define a string constant whose value will be used as the display name for the `approveDiscount` action, as the following code shows.

```
        public const string ApproveDiscount = "Approve Discount";
```

2. In the static `Configure` method of the `SOInvoiceRepairOrder_Workflow` class, add the following code.

```
            var approveDiscount = context.ActionDefinitions
                .CreateNew(ApproveDiscount, action => action
```

```
                .DisplayName("Approve Discount"));
```

In the code above, you have defined a new action with the **Approve Discount** display name.

3. In the `WithActions` method of the `UpdateScreenConfigurationFor` method call, add the `approveDiscount` action, as the following code shows.

```
                    actions.Add(approveDiscount);
```

4. Save your changes.

Step 4: Adding the Skip Condition

The `Postponed` workflow state should be skipped if no discount is being applied to the invoice—that is, if the **Document Discounts** box is *0.0* on the *Invoices* (SO303000) form. In this step, you will define this condition by doing the following:

1. On the *Invoices* (SO303000) form, learn the field name of the **Document Discounts** box by using the *Element Inspector* tool. The field name is `CuryDiscTot`. The field is defined in the `ARInvoice` DAC.
2. In the `SOInvoiceRepairOrder_Workflow` class, define the condition, as the following code shows.

```
        #region Conditions
        public class Conditions : Condition.Pack
        {
            public Condition DiscountEmpty => GetOrCreate(condition =>
              condition.FromBql<ARInvoice.curyDiscTot.IsEqual<decimal0>>());
        }
        #endregion
```

3. Add the necessary `using` directive, as the following code shows.

```
using static PX.Data.WorkflowAPI.BoundedTo<PX.Objects.SO.SOInvoiceEntry,
  PX.Objects.AR.ARInvoice>;
using PX.Objects.CS;
```

4. Create an instance of the `Conditions` class in the static `Configure` method, as the following code shows.

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

5. Save your changes.

Step 5: Adding the Postponed Workflow State to the Composite State

To add the `Postponed` workflow state to the composite state, do the following:

1. In the `SOInvoiceRepairOrder_Workflow` class, add the `Postponed` workflow state in the `WithFlowStates` method, as shown in the following code.

```
                                flowStates.UpdateSequence<ARDocStatus.HoldToBalance>(
                                    seq =>
                                {
                                    return seq.WithStates(states =>
                                    {
```

```
                                    states.Add<ARDocStatus_Postponed.postponed>(
                                        flowState =>
                                    {
                                        return flowState
                                    .PlaceAfter<ARDocStatus.creditHold>()
                                    .IsSkippedWhen(conditions.DiscountEmpty)
                                    .WithActions(actions =>
                                    {
                                        actions.Add(approveDiscount,
                                            action => action
                                            .IsDuplicatedInToolbar()
                                            .WithConnotation(
                                                ActionConnotation.Success));
                                    });
                                    });
                                });
                            });
```

In the code above, you have done the following:

- Added the `Postponed` workflow state after the `CreditHold` workflow state
- Specified that the `Postponed` workflow state should be skipped if the `DiscountEmpty` condition is *true*
- Added the `approveDiscount` action to the `Postponed` workflow state

2. Save your changes.

Step 6: Adding the Transition

Without a composite workflow state, you would need to remove or modify an existing transition and add at least two transitions instead: one to the new workflow state, and another from the new workflow state. But with a composite workflow state, you need to add only one transition: from the `Postponed` nested workflow state to the next nested workflow state of the composite state.

To add a transition from the `Postponed` workflow state to the next workflow state in the composite state, do the following:

1. In the `SOInvoiceRepairOrder_Workflow` class, in the lambda expression for the `UpdateDefaultFlow` method, add the following code.

```
                        .WithTransitions(transitions =>
                        {
                            transitions.AddGroupFrom<ARDocStatus_Postponed.postponed>(
                                transitionGroup =>
                            {
                                transitionGroup.Add(transition => transition
                                    .ToNext()
                                    .IsTriggeredOn(approveDiscount)
                                    .WithFieldAssignments(fields =>
                                        fields.Add<ARInvoice.discDate>(
                                            field => field.SetFromToday())));
                            });
                        });
```

In the code above, you have added a transition group where the initial workflow state is `Postponed`. In the transition group, you have done the following:

- Added the transition to the next workflow state in the composite state by calling the `ToNext` method
- Specified an action that triggered the transition in the `IsTriggeredOn` method
- Used the `WithFieldAssigments` method to specify that the **Cash Discount Date** box should be assigned the business date when the transition is performed

2. Save your changes.

Step 7: Testing the Transition

To test the transition, do the following:

1. Rebuild the `PhoneRepairShop_Code` project.
2. On the Repair Work Orders (RS301000) form, create an invoice for any repair work order.
3. On the *Invoices* (SO303000) form, open the created invoice.
4. In the **Document Discounts** box of the Summary area of the form, enter `5`.
5. On the form toolbar, click **Remove Hold**.

   The status of the invoice changes to *Postponed*, as shown in the following screenshot. On the More menu, notice that only one command is available (**Approve Discount**). The equivalent button appears on the form toolbar.

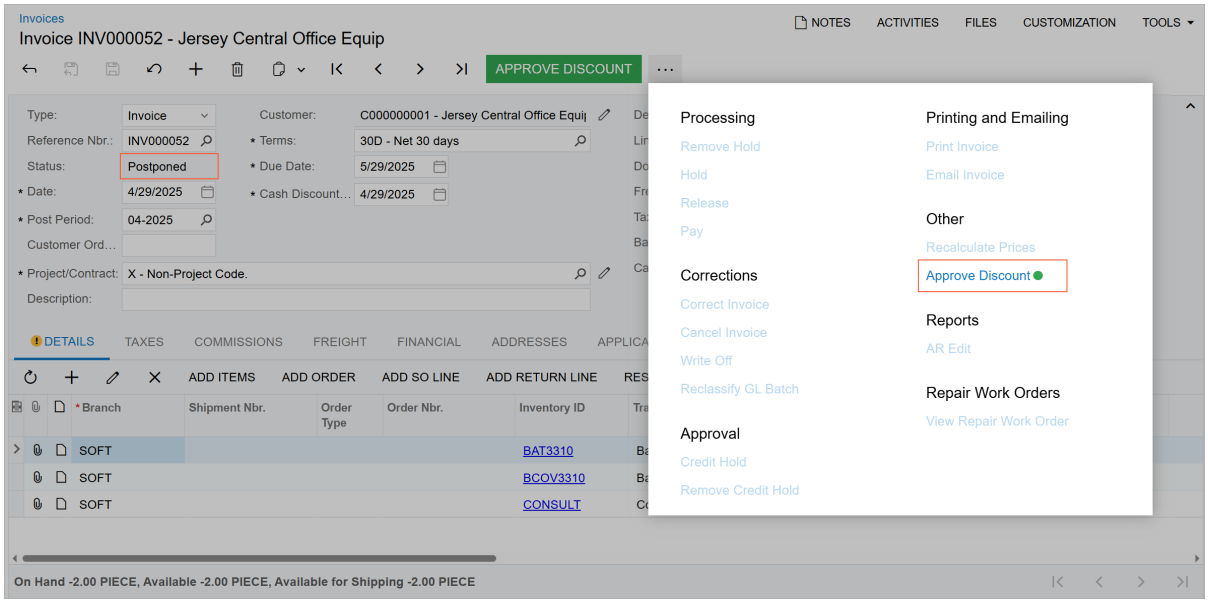


Figure: The Postponed Status of an Invoice

6. On the form toolbar, click **Approve Discount**.

   The status of the invoice changes to *Balanced*, and the value in the **Cash Discount Date** box is the business date.

7. On the Repair Work Orders (RS301000) form, create another invoice for a different repair work order.
8. On the *Invoices* (SO303000) form, open the created invoice.
9. On the form toolbar, click **Remove Hold**.

   The status of the invoice changes to *Balanced*. Because you have not specified any value in the **Document Discounts** box, the system skipped the `Postponed` workflow state and moved the invoice to the next workflow state in the composite state (`Balanced` in this case).

Defining Action Sequences

In this chapter, you will learn how to define sequences of actions that the system can perform automatically.

Action Sequences: General Info

You can make the system automatically execute actions one after another if certain conditions are met. Thus, you can optimize system performance and speed up the processing of records. When a sequence of actions is performed, the system shows a dialog box with the progress and results of performing the sequence.

Learning Objectives

In this chapter, you will learn what action sequences are. You will also learn how to configure these sequences in a workflow.

Applicable Scenarios

You implement action sequences in the following cases:

- You want the system to execute multiple actions one after another if certain conditions are met.
- You want the system to stop the processing of records if certain actions fail.
- You need to see which actions have been performed successfully and which ones have failed.

Execution of Sequential Actions

Each action can have multiple actions—referred to as *subscriber actions* or *subscribers*—that the system invokes after it executes this action. Each subscriber can also have its own subscribers. If a condition is specified for an action, the system executes a subscriber action only if the condition is met and the action is available in the current workflow state. The system executes the action even if it is hidden.

If a subscriber action is unavailable in the current state of the workflow, the system displays an error. The system then either continues the execution of other subscriber actions or stops the execution, depending on the settings specified for this action.

Only actions of the same form (screen) can be configured to run in a sequence.

If a user clicks a button (or command) that has an underlying action with subscribers, the system proceeds as follows:

1. If a dialog box is specified for this action, the system displays it and updates the fields as specified in the action's configuration.
2. The system performs all other steps in the workflow as specified in the configuration of the transitions and states.
3. If the action has been executed successfully, the system then executes its subscribers.

   If a dialog box is specified for a subscriber action, the system does not display this dialog box to the user; instead, it uses the dialog box values specified in the screen configuration.
4. For each of the subscribers, the system performs all other steps in the workflow.
5. If a subscriber action has its own subscribers, the system executes them one by one if they exist in the current workflow state and their respective conditions are met.

During the execution of each action in a sequence, the system displays a dialog box with the status of the action execution, as shown in the following screenshot. (It is the same dialog box that the system displays when the user clicks the **Quick Process** button on a form that supports quick processing.) For report and navigation actions, the system opens the corresponding report or data entry form in a new browser tab and continues the processing of other actions in the current tab.

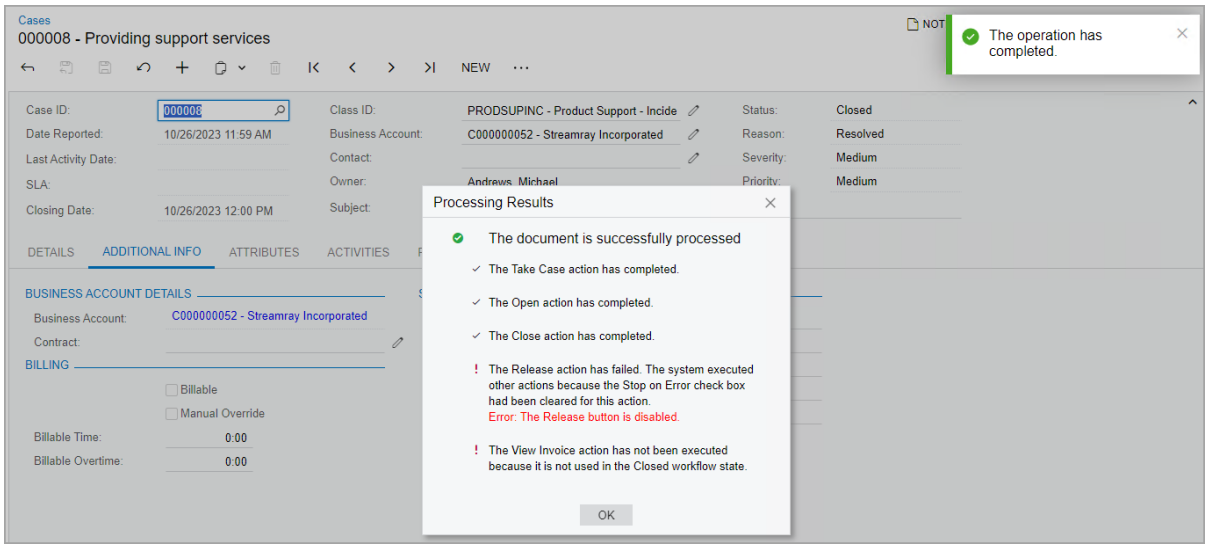


Figure: The dialog box during action execution

For details on implementing action sequences in Workflow UI, see *Action Configuration: Action Sequences* in the *Workflow UI Guide*.

Defining of an Action Sequence

To define an action sequence, you perform the following general steps:

1. At the screen configuration level, you call the `WithActionSequences` method.
2. In the lambda expression for the `WithActionSequences` method, you call the `Add` method to add a new sequence.

   > ⚠ You can instead call the `Replace` method to replace an existing sequence or the `Remove` method to remove an existing sequence. To avoid errors, Workflow API offers no option to update an existing sequence.

3. In the lambda expression for the `Add` method, you configure the sequence by using the following methods:
   - `AfterAction` (required): Specifies the triggering action (the action after which the sequence should be executed). A developer can specify either an action from the graph or an action defined by using Workflow API.
   - `RunAction` (required): Specifies the action that should be executed in the sequence.

     Only one action can be added to a sequence. To add a sequence of multiple actions, you need to add multiple sequences. For example, to add the A > B > C sequence of actions, you need to add the A > B sequence and the B > C sequence.
   - `StopOnError` (optional): Specifies whether the system should stop the execution of subscribers if the triggering action fails. By default, the sequence is stopped if an error occurs.
   - `AppliesWhen` (optional): Specifies the condition depending on which the sequence should be executed.

     A developer can instead call the `AppliesAlways` method to always execute the sequence.

- `WithFieldAssignments` (optional): Assigns the fields of a dialog box that is configured for the second action of the sequence (specified in the `RunAction` method). The assignment is performed after the first action (specified in the `AfterAction` method) is executed.

  > ⚠ If a dialog box is configured for an action specified in the `RunAction` method (the second action in the sequence), this dialog box is not displayed during the execution of the sequence. So the only way to provide values that are different from default ones in the dialog box is to specify them in the `WithFieldAssignments` method.

Order of Execution

The system executes action sequences in the order they are defined in the code. For example, if the code contains the action sequences A > B, B > C, and A > D with no conditions, the system will first execute the sequence A > B, then the sequence B > C, and then the sequence A > D, so the order of actions will be A > B > C > D. If the sequence A > B has a condition and it is not met, the system will proceed to the sequence A > D.

Action Sequences: To Define an Action Sequence

The following activity will walk you through the process of defining an action sequence.

Story

Suppose that when a user clicks the **Close** command on the *Cases* (CR306000) form, you need to execute the actions that correspond to the following commands on the form:

- **Release** to release the case
- **View Invoice** if the case is billable (that is, if the **Billable** check box is selected on the **CRM Info** tab of the *Cases* form)

Process Overview

First, you will learn the internal names of the entities used in the workflow: the actions and fields, workflow class, graph, and DAC.

Then you will extend the default workflow for the *Cases* (CR306000) form and define the action sequences in the extension. In order to define an action sequence for more than two actions, you need to define an action sequence for each pair. Thus, you will first define an action sequence for the `Close` and `Release` actions, and then an action sequence for the `Release` and `View Invoice` actions.

System Preparation

Before you begin, you should make sure that you have configured your instance of Acumatica ERP, as described in the *Preparing an Instance for Workflow Customization* chapter.

You also need to create an extension library for the instance. For details, see *To Create an Extension Library*.

Step 1: Investigate the Internal Names of Elements on the Cases Form

To define a sequence of existing actions, you need to learn the internal names of these actions in the graph of the *Cases* (CR306000) form, as well as other relevant internal names. Do the following:

1. Open the *Cases* form.

2. By using the Element Inspector, learn the name of the graph and the DAC for the form: `CRCaseMaint` and `CRCase`
3. By using the Element Inspector, learn the action names of the following commands on the More menu:
   - **Release**: `release`
   - **Close**: `Close`
   - **View Invoice**: `viewInvoice`
4. By using the Element Inspector, learn the internal name for the **Billable** check box on the **CRM Info** tab: `CRCase.IsBillable`.
5. In the website project of the Visual Studio solution, find the source code of the `PX.Objects.CR` namespace, and find the class that defines the code of the workflow for the *Cases* form: The `CaseWorkflow` class is located in `App_Data/CodeRepository/PX.Objects/CR/Workflows/CaseWorkflow.cs`.

Step 2: Implement the First Action Sequence

To implement the first action sequence to be executed, do the following:

1. In the extension library, create a C# file named `CaseWorkflow_Extension.cs`.
2. In the file, extend the class that implements the workflow for the *Cases* (CR306000) form, as shown in the following code.

```
public class CaseWorkflow_Extension : PXGraphExtension<CaseWorkflow, CRCaseMaint>
{
    public sealed override void Configure(PXScreenConfiguration config) =>
        Configure(config.GetScreenConfigurationContext<CRCaseMaint, CRCase>());

    protected static void Configure(WorkflowContext<CRCaseMaint, CRCase> context)
    {
    }
}
```

3. In the `Configure(WorkflowContext<CRCaseMaint, CRCase> context)` method, implement the action sequence as shown in the following code.

```
context.UpdateScreenConfigurationFor(screen =>
{
  return screen
    .WithActionSequences(sequences =>
    {
      sequences.Add(s => s
        .AfterAction("Close")
        .RunAction("release")
        .StopOnError(true));
    });
});
```

   In the code above, you have defined a sequence by calling the `Add` method in the `WithActionSequences` method. In the lambda expression for the `Add` method, you have specified the first action of the sequence (the triggering action) in the `AfterAction` method and the second action of the sequence in the `RunAction` method. In the `StopOnError` method, you have also specified that the second action should not be performed if an error occurs during the execution of the first action of the sequence.
4. Add the necessary `using` directives.

```
using PX.Data;
using PX.Data.WorkflowAPI;
using PX.Objects.CR;
using PX.Objects.CR.Workflows;
```

5. Build the project.

Step 3: Implement the Second Action Sequence

To implement an action sequence containing the `release` and `viewInvoice` actions, you need to add a separate sequence after the first one. Because the `viewInvoice` action should be executed only if the case is billable, you need to define a condition and specify it in the action sequence. Do the following:

1. Define a condition that is true when the case is billable.
   a. In the `CaseWorkflow_Extension` class, define the `Conditions` class with the condition.

```
public class Conditions : Condition.Pack
{
    public Condition IsBillable =>
      GetOrCreate(b => b.FromBql<CRCase.isBillable.IsEqual<True>>());
}
```

   b. Add the necessary `using` directive.

```
using static PX.Data.WorkflowAPI.BoundedTo<PX.Objects.CR.CRCaseMaint,
 PX.Objects.CR.CRCase>;
```

   c. In the `Configure(WorkflowContext<CRCaseMaint, CRCase> context)` method, create an instance of the `Conditions` class.

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

2. In the `WithActionSequences` method, add the second sequence after the sequence defined in the previous step. Specify the `IsBillable` condition in the `AppliesWhen` method.

```
sequences.Add(s => s
  .AfterAction("release")
  .RunAction("viewInvoice")
  .AppliesWhen(conditions.IsBillable)
  .StopOnError(true));
```

3. Build the project.

Step 3: Test the Modified Workflow

To test the implemented sequence, do the following:

1. Open the *Cases* (CR306000) form.

> ⚠ To make sure that the workflow has been modified correctly, note that the More button is displayed and the toolbar buttons have connotations. If you suspect an error, you can check the workflow output on the **System Events** tab of the *System Monitor* form. For details, see *Test Instance for Workflow Customization: To Turn On Workflow Validation*.

2. Create a new case with the following settings:

- **Case Class**: *PRODSUPINC*

  You need to specify this class because it is billable; this causes the **Billable** check box to be selected on the **CRM Info** tab of the form.
- **Business Account**: *C000000001*
- **Subject**: `Test Sequence`

3. On the form toolbar, click **Open**.
4. In the **Open** dialog box which opens, click **OK**.
5. On the form toolbar, click **Close**.
6. In the **Close** dialog box which opens, click **OK**.

   Note that the form shows the **Processing Results** dialog box and the timer indicating a long-running operation. The **Processing Results** dialog box indicates that the action sequence is being performed. The timer indicates that the `release` action which contains the long-running operation is being performed.

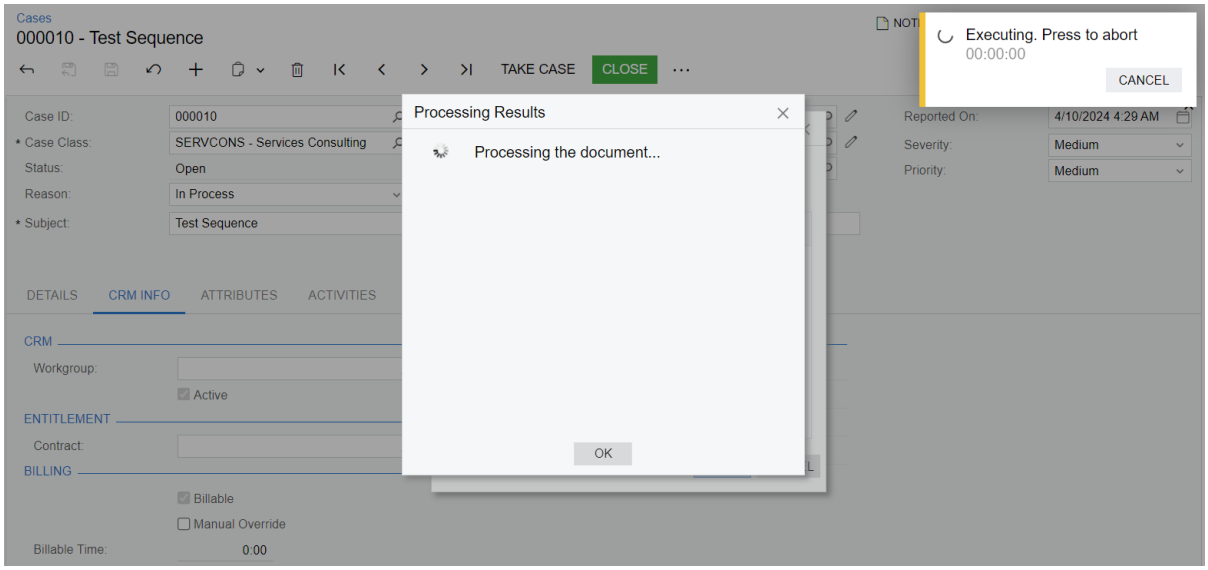


Figure: The system performing a long-running operation

After the sequence is completed, in the **Processing Results** dialog box, note the list of actions that have been performed: **Close**, **Release**, and **View Invoice**, as shown in the following screenshot. The *Invoices and Memos* (AR301000) form is opened in a new tab with the invoice created for the case.

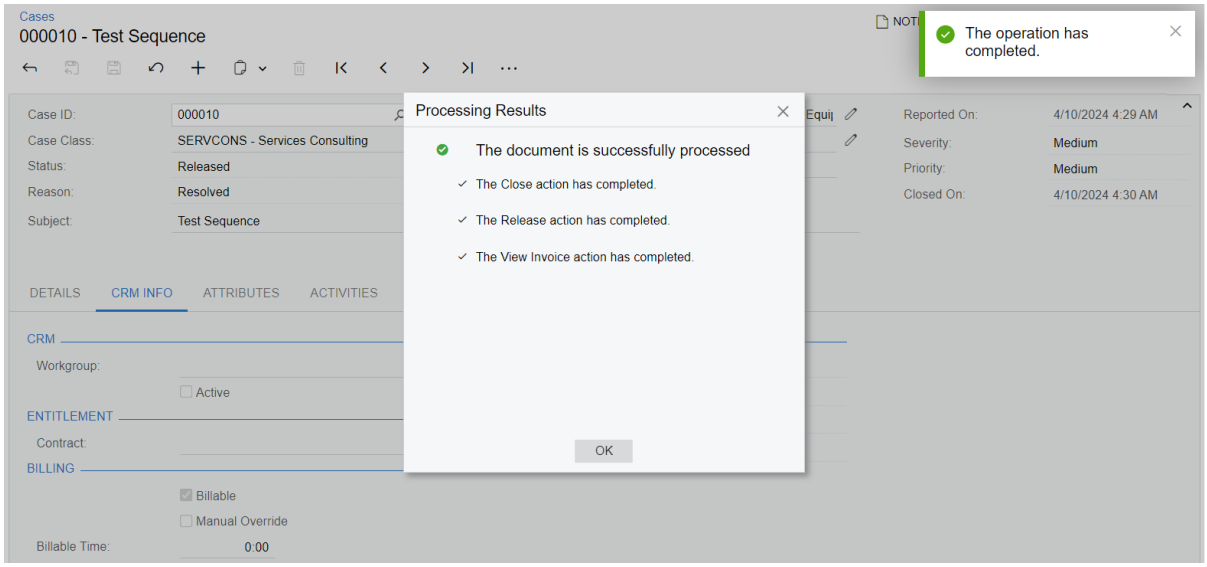


Figure: The list of performed actions

7. In the **Processing Results** dialog box, click **OK**.

Defining Workflows with a Workflow Identifying Field

In this chapter, you will learn how to customize workflows with a workflow-identifying field. The value of a workflow-identifying field determines which workflow will be used for the form. Thus, a record created on the form is processed differently based on the value of this field.

Workflow-Identifying Fields: General Information

You can add multiple workflows for a particular form so that each workflow is applied to all records that have a specific value in a particular field. This field is described as a *workflow-identifying field* because its value determines the workflow to be used.

Learning Objectives

In this chapter, you will gain experience creating a workflow that is based on the specific value of a selected field.

Applicable Scenarios

You customize a workflow with a workflow-identifying field if you need to make changes to the workflow so that it is better suited for your business processes, and you do not want to create such a workflow from scratch.

Workflow Types

A form can have the following types of workflows:

- The default workflow

  The form can have only one default workflow, which is defined with the `AddDefaultFlow` method. For details, see *Screen Configuration: To Prepare a Screen Configuration for a Form Without a Predefined Workflow*.
- A set of workflows, which are applied depending on the value of the workflow-identifying field

  If you have not defined a workflow for a value of the workflow-identifying field, the system applies the default workflow for a record if this value is selected for the record.

Definition of a Workflow-Identifying Field

You define a workflow-identifying field by specifying it in the screen configuration.

You can define workflow-identifying field of the first and second level. For details on defining a workflow-identifying field of the second level, see *Workflow-Identifying Fields of the Second Level: General Information*.

To specify a workflow-identifying field, you call the `FlowTypeIdentifierIs` method in the lambda expression for the `AddScreenConfigurationFor` or `UpdateScreenConfigurationFor` method and specify a string field from the primary DAC of the form as the type parameter. An example is shown in the following code.

```
context.AddScreenConfigurationFor(screen =>
{
  return screen
    .StateIdentifierIs<status>()
```

```
    .FlowTypeIdentifierIs<SOOrder.behavior>();
});
```

Definition of a Workflow for the Specific Value of the Workflow-Identifying Field

To define a workflow for the specific value of the workflow-identifying field, you do the following:

1. Inside the `AddScreenConfigurationFor` or `UpdateScreenConfigurationFor` method, you call the `WithFlows` method.
2. In the `WithFlows` method, you call the `Add` method.

   As the type parameter of the `Add` method, you specify the value of the workflow-identifying field.
3. Inside the `Add` method, you define workflow states and transitions, as described in *Defining Workflow States* and *Implementing Transitions*.

An example of the workflow for the specific value of the workflow-identifying field is shown in the following code.

```
context.UpdateScreenConfigurationFor(screen => screen
  .WithFlows(flows =>
  {
    flows.Add<SOBehavior.bL>(flow => flow
      .WithFlowStates(flowStates =>
      {
        ...
      }
      .WithTransitions(transitions =>
      {
        ...
      }));
  }));
```

Workflow-Identifying Fields: To Add a Workflow for a Value of the Workflow-Identifying Field

The following activity will walk you through the process of defining a workflow for a specific value of the workflow-identifying field.

Story

Suppose that you have a repair work order that can be processed in the following ways:

- Quickly while the customer awaits in the phone repair shop. In this case, the workflow can skip most of the traditional states and be paid right after it is created. So the workflow can have only three states (`OnHold`, `Completed`, `Paid`) and direct transitions from one to another.
- In a few days while the order awaits the assigned employee. In this case, the standard default workflow can be applied.
- In a few weeks because some of the required items are out of stock. In this case, the workflow might have an additional state, `AwaitingDelivery`, in addition to standard states of the workflow described in *Customization Description*.

These ways of processing a repair work order are defined by the **Order Type** box which can have one of the following values:

- *Simple*

- *Standard*
- *Awaiting Delivery*

In this activity, you will define a workflow for the *Simple* value of the **Order Type** box. The workflow is shown in the following diagram.

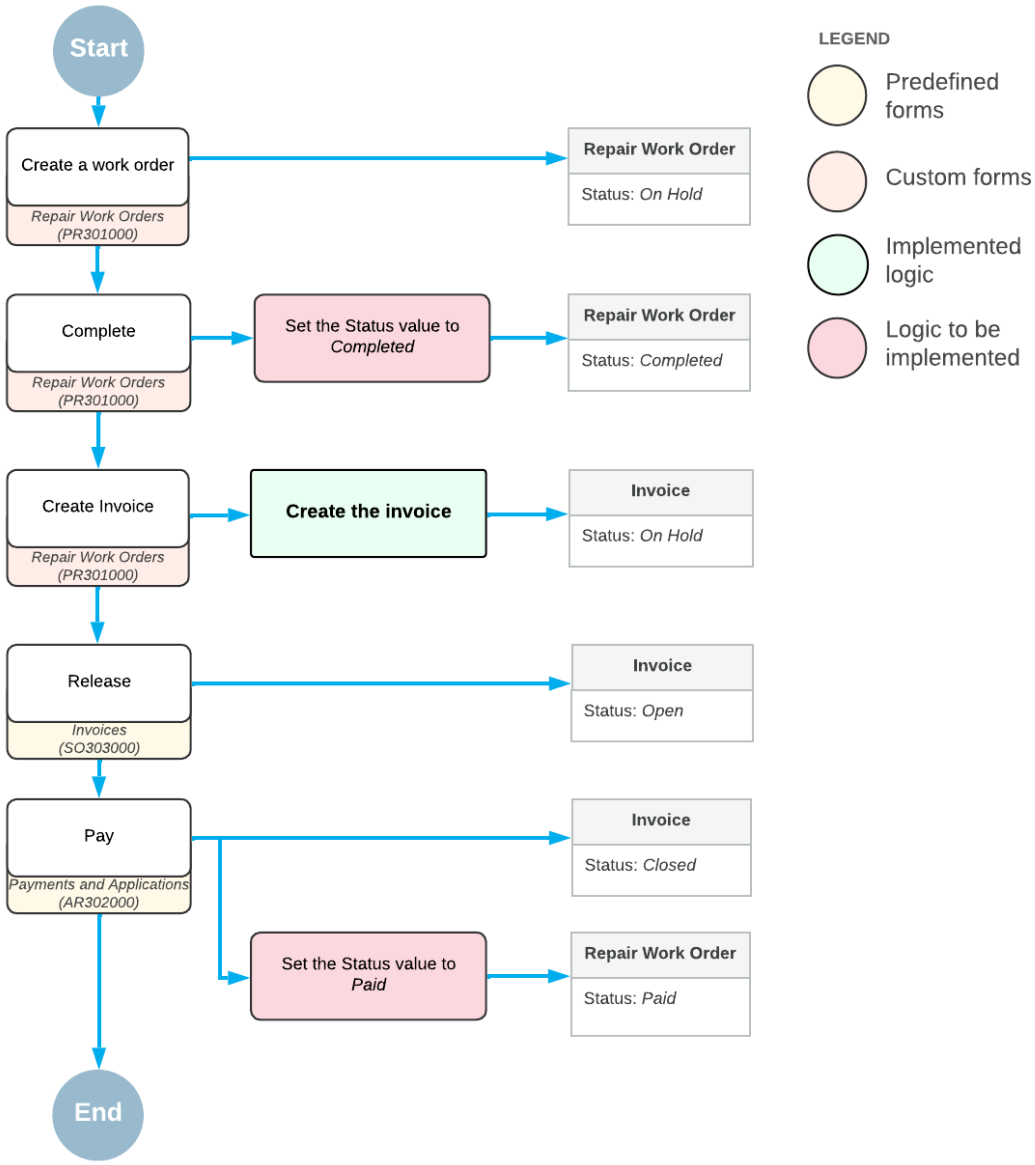


Figure: Workflow for the Simple order type

When the repair work order is created, it gets the *On Hold* status. Then a user can click the **Complete** action on the form toolbar, which changes the order to *Completed*. After that, a user can click **Create Invoice** on the form toolbar, which initiates creation of an invoice for this repair work order. When the invoice is created, a user can release and pay it on the *Invoices* (SO303000) and *Payments and Applications* (AR302000) forms respectively. As soon as the invoice is fully paid, the repair work order status should be changed to *Paid*.

Process Overview

You will first add the `UsrOrderType` field to the `RSSVWorkOrder` DAC and respective database table. The field will hold the type of a repair work order and will be used on the Repair Work Orders (RS301000) form. Then,

in the screen configuration for the Repair Work Orders (RS301000) form, you will specify this field as a workflow-identifying field. Then, you will add a workflow for the *Simple* value of the `UsrOrderType` field.

> Creation of workflows for other values of the `UsrOrderType` field is not covered in this activity.

System Preparation

Make sure that you have done the following:

1. Prepared an instance with the *PhoneRepairShop* customization project and enabled the workflow validation by performing the following activities:
   a. *Test Instance for Workflow Customization: To Deploy a Test Instance*
   b. *Test Instance for Workflow Customization: To Turn On Workflow Validation*
2. Prepared the screen configuration and defined the set of workflow states by performing the *Screen Configuration: To Prepare a Screen Configuration for a Form Without a Predefined Workflow* activity.
3. Performed the following prerequisite activities:
   a. *Workflow Actions: To Implement an Action with Field Assignments*
   b. *Workflow Actions: To Configure the Conditional Appearance of the Action*
   c. *Workflow Events: To Use an Existing Event*

Step 1: Adding the OrderType Field and Corresponding Box to the UI

To add multiple workflows to a form, you need to define a workflow-identifying field, which will hold a value that defines a particular workflow to be used for a record. In this step, you will add the workflow-identifying field to the Repair Work Orders (RS301000) form.

Do the following:

1. Add the `UsrOrderType` field to the `RSSVWorkOrder` database table by doing the following:
   a. In the Customization Project Editor, open the *Database Scripts* page.
   b. On the More menu, click **Add Custom Column to Table**.
   c. In the **Add Custom Column to Table** dialog box which opens, specify the following values:
      - **Table**: *RSSVWorkOrder*
      - **Field Name**: `UsrOrderType`

        > The field must have the `Usr` prefix because it is a custom field.

      - **Data Type**: *string*
      - **Length**: `2`
   d. Click **OK**.
   e. Publish the customization project to apply the database script.
2. In the `Constants.cs` file, add the list of possible values for the `UsrOrderType` field as shown in the following code.

```
//Constants for the repair work order types
public static class WorkOrderTypeConstants
{
```

```
        public const string Simple = "SP";
        public const string Standard = "ST";
        public const string Awaiting = "AW";
    }
```

3. In the `Messages.cs` file, add the list of UI string values corresponding to the added constant strings as shown in the following code.

```
        //Work order types
        public const string Simple = "Simple";
        public const string Standard = "Standard";
        public const string Awaiting = "Awaiting Delivery";
```

4. Create an extension of the `RSSVWorkOrder` DAC and add the `UsrOrderType` field to the `RSSVWorkOrder` DAC by using the following code.

```
        public sealed class RSSVWorkOrder_Extension :
            PXCacheExtension<RSSVWorkOrder>
        {
            #region Status
            [PXDBString(2, IsFixed = true)]
            [PXDefault(WorkOrderTypeConstants.Standard,
              PersistingCheck = PXPersistingCheck.Nothing)]
            [PXUIField(DisplayName = "Order Type")]
            [PXStringList(
              new string[]
              {
                  WorkOrderTypeConstants.Simple,
                  WorkOrderTypeConstants.Standard,
                  WorkOrderTypeConstants.Awaiting
              },
              new string[]
              {
                  Messages.Simple,
                  Messages.Standard,
                  Messages.Awaiting
              })]
            public string? UsrOrderType { get; set; }
            public abstract class usrOrderType :
                PX.Data.BQL.BqlString.Field<usrOrderType> { }
            #endregion
        }
```

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

5. Rebuild the `PhoneRepairShop_Code` project.
6. Add the **Order Type** box to the Repair Work Orders (RS301000) form by doing the following:
   a. In the Customization Project Editor, open **Screens > RS301000** page.
   b. In the control tree, select the first column in the **Form: WorkOrders** node.
   c. On the **Add Data Fields** tab, select the `UsrOrderType` field and click **Create Controls**.

      The **Order Type** field appears as the first element of the selected column in the control tree.
   d. On the **Layout Properties** tab, for the **CommitChanges** property, select *True*.

Once a new value is selected in the box, the system will apply the workflow that corresponds to the selected value.

e. Save your changes and publish the customization project.

f. Open the Repair Work Orders (RS301000) form and make sure the **Order Type** box is displayed in the Summary area of the form as a drop down control.

Step 2: Specifying the Workflow-Identifying Field in the Workflow

In this step, you will extend the screen configuration of the Repair Work Orders (RS301000) form and specify the workflow-identifying field in it by calling the `FlowTypeIdentifierIs` method.

Do the following:

1. In the extension library, create an extension of the `RSSVWorkOrderEntry_Workflow` class as shown in the following code.

```
public class RSSVWorkOrderEntry_Workflow_Extension :
    PXGraphExtension<RSSVWorkOrderEntry_Workflow, RSSVWorkOrderEntry>
{
}
```

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

2. In the `RSSVWorkOrderEntry_Workflow_Extension` class, define the set of constant string values and corresponding classes for the value of the `UsrOrderType` field which as the following code shows.

```
    #region Constants
    public static class OrderTypes
    {
        public const string Simple = WorkOrderTypeConstants.Simple;
        public const string Standard = WorkOrderTypeConstants.Standard;
        public const string Awaiting = WorkOrderTypeConstants.Awaiting;

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

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

        public class awaiting : PX.Data.BQL.BqlString.Constant<awaiting>
        {
            public awaiting() : base(Awaiting) { }
        }
    }
    #endregion
```

3. In the `RSSVWorkOrderEntry_Workflow_Extension` class, override the `Configure` method as the following code shows.

```
public sealed override void Configure(PXScreenConfiguration config)
{
    Configure(config.GetScreenConfigurationContext<RSSVWorkOrderEntry,
                                                   RSSVWorkOrder>());
}

protected static void Configure(WorkflowContext<RSSVWorkOrderEntry,
                                                RSSVWorkOrder> context)
{
}
```

4. In the static `Configure` method, update the screen configuration and specify the workflow-identifying field as the following code shows.

```
    context.UpdateScreenConfigurationFor(screen => screen
        .FlowTypeIdentifierIs<RSSVWorkOrder_Extension.usrOrderType>());
```

5. Save your changes.

Step 3: Adding a Workflow for the Specific Value of the Workflow-Identifying Field

In this step, you will add a workflow for the *Simple* value of the **Order Type** box. This workflow has three workflow states (which are `OnHold`, `Completed`, and `Paid`) and the following transitions:

- From the `OnHold` to `Completed` workflow state, which is triggered by the `Complete` action
- From the `Completed` to `Paid` workflow state, which is triggered by the `OnCloseDocument` event

You will add these workflow states and transitions to the screen configuration in the extension of the `RSSVWorkOrderEntry_Workflow` class.

You do not need to add actions and event handlers used in this workflow because they are already added in the original screen configuration defined in the `RSSVWorkOrderEntry_Workflow` class.

To add the new workflow, do the following in the static `Configure` method of the `RSSVWorkOrderEntry_Workflow_Extension` class:

1. In the lambda expression for the `UpdateScreenConfigurationFor` method, after calling the `FlowTypeIdentifierIs` method, call the `WithFlows` method and add a workflow for the *Simple* value of the `UsrOrderType` field as shown in the following code.

```
    .WithFlows(flows => flows
        .Add<OrderTypes.simple>(flow =>
            getSimpleBehavior(flow)))
```

   You have specified the value for the workflow-identifying field as the type parameter of the `Add` method.

2. Add the private static `getSimpleBehavior` method, and specify the list of states of the new workflow as shown in the following code.

```
private static Workflow.IConfigured getSimpleBehavior(
    Workflow.INeedStatesFlow flowState)
{
    return flowState
    .WithFlowStates(states =>
    {
        states.Add<RSSVWorkOrderEntry_Workflow.States.onHold>(flowState =>
```

```
                {
                    return flowState
                      .IsInitial()
                      .WithActions(actions =>
                      {
                          actions.Add(g => g.Complete, a => a
                          .IsDuplicatedInToolbar()
                          .WithConnotation(ActionConnotation.Success));
                      });
                });
                states.Add<RSSVWorkOrderEntry_Workflow.States.completed>(flowState =>
                {
                    return flowState
                        .WithFieldStates(fieldstates =>
                        {
                            fieldstates.AddField<RSSVWorkOrder.customerID>(state =>
                                state.IsDisabled());
                            fieldstates.AddField<RSSVWorkOrder.serviceID>(state =>
                                state.IsDisabled());
                            fieldstates.AddField<RSSVWorkOrder.deviceID>(state =>
                                state.IsDisabled());
                        })
                        .WithActions(actions =>
                        {
                            actions.Add(g => g.CreateInvoiceAction, a => a
                            .IsDuplicatedInToolbar()
                            .WithConnotation(ActionConnotation.Success));
                        })
                        .WithEventHandlers(handlers =>
                        {
                            handlers.Add(g => g.OnCloseDocument);
                        });
                });
                states.Add<RSSVWorkOrderEntry_Workflow.States.paid>(flowState =>
                {
                    return flowState
                        .WithFieldStates(fieldstates =>
                        {
                            fieldstates.AddField<RSSVWorkOrder.customerID>(state =>
                                state.IsDisabled());
                            fieldstates.AddField<RSSVWorkOrder.serviceID>(state =>
                                state.IsDisabled());
                            fieldstates.AddField<RSSVWorkOrder.deviceID>(state =>
                                state.IsDisabled());
                        });
                });
            });
        }
```

In the code above, you have added the following workflow states:

- `OnHold`: You have specified that the **Complete** action should be available in this workflow state.
- `Completed`: You have specified which fields should be unavailable in this worklfow state. You have also specified that the `CreateInvoice` action and the `OnCloseDocument` workflow event handler should be available in this workflow state.
- `Paid`: You have specified which fields should be unavailable.

3. In the private static `getSimpleBehavior` method, after the call of the `WithFlowStates` method, define the transitions by using the `WithTransitions` method as the following code shows.

```
            .WithTransitions(transitions =>
            {
                transitions.AddGroupFrom<RSSVWorkOrderEntry_Workflow.States.onHold>(
                    transitionGroup =>
                {
                    transitionGroup.Add(transition => transition
                        .To<RSSVWorkOrderEntry_Workflow.States.completed>()
                        .IsTriggeredOn(graph => graph.Complete));
                });
                transitions.AddGroupFrom<
                    RSSVWorkOrderEntry_Workflow.States.completed>(
                    transitionGroup =>
                {
                    transitionGroup.Add(transition => transition
                        .To<RSSVWorkOrderEntry_Workflow.States.paid>()
                        .IsTriggeredOn(graph => graph.OnCloseDocument));
                });
            });
```

4. Build your project.

Step 4: Testing the Workflow

To test the implemented workflow, do the following:

1. In Acumatica ERP, open the Repair Work Orders (RS301000) form and create a repair work order.
2. In the **Order Type** box, select *Simple*
3. Specify the following values:
   - **Customer ID**: *C000000001*
   - **Service**: *Battery Replacement*
   - **Device**: *Nokia 3310*
   - **Description**: `Battery replacement, Nokia 3310`

   Note that the button on the form toolbar changes to **Complete**.
4. On the form toolbar, click **Complete**.

   The record is saved. The status of the record changes to *Completed*.
5. On the form toolbar, click **Create Invoice**.

   When the invoice is created, note the number of the created invoice.
6. On the *Invoices* (SO303000) form, open the created invoice.
7. On the form toolbar, click **Remove Hold**, and then **Release**.
8. In the More menu, click **Pay**.

   The *Payments and Applications* (AR302000) form opens.
9. On the form toolbar, click **Remove Hold**, and then **Release**.
10. Open the repair work order that you have created in this step.

    Make sure that its status has changed to *Paid*.

Using Workflow-Identifying Fields of the Second Level

Workflow-Identifying Fields of the Second Level: General Information

This topic provides information about workflow identifiers (that is, workflow-identifying fields, which are fields whose values determine the workflow to be used) of the second level. You will also learn how to create different workflows that use these identifiers.

Learning Objectives

In this chapter, you will learn the following:

- What is a workflow-identifying field of the second level
- How to define a subflow (a workflow for a pair of the workflow-identifying field of the first and second level)

Applicable Scenarios

You use workflow identifiers of the second level when you have multiple types of entities on a form, and you need to create separate workflows for each of these entity types.

Workflow-Identifying Fields of the Second Level

For a screen configuration which has a workflow-identifying field, you can specify a workflow-identifying field of the second level. This way you can create a workflow for each pair of values of these two fields.

Workflow-identifying field of the first level and the second level can also be called *type identifier* and the *subtype identifier* especially in the method names.

The system will apply the workflow based on the value of a workflow-identifying field of the first level (defined using the `FlowTypeIdentifierIs` method) and then further based on the value of the workflow-identifying field of the second level.

A workflow defined for the second level inherits its configuration from the workflow defined on the first level. The following diagram shows the types and subtypes of workflows based on their workflow-identifier field values.

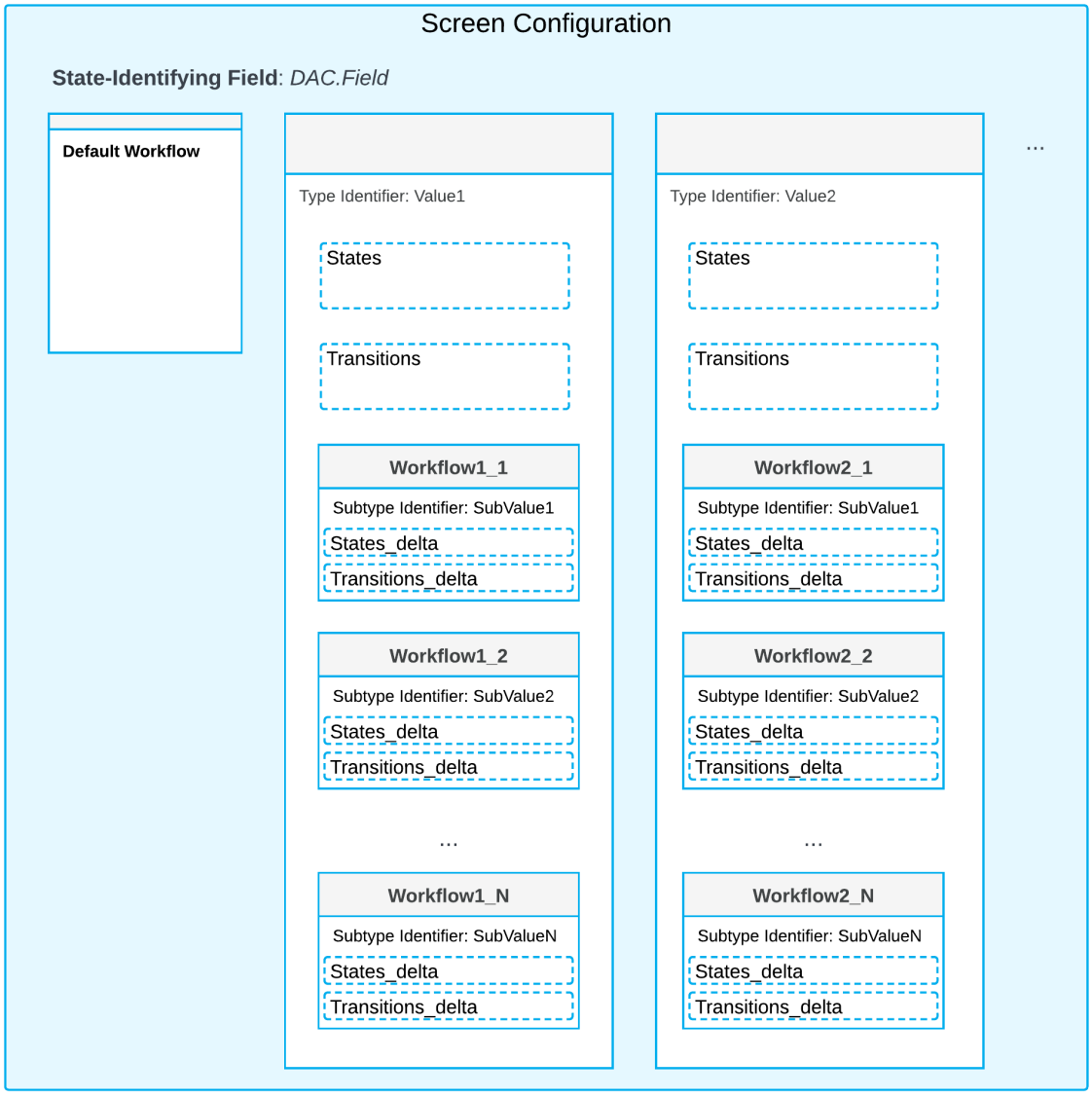


Figure: Types and subtypes of workflows

Definition of a Workflow of the Second Level

To create a workflow of the second level, do the following in the screen configuration:

1. Define the workflow-identifying field of the first level by calling the `FlowTypeIdentifierIs<>()` method
2. Define the workflow-identifying field of the second level by calling `FlowSubTypeIdentifierIs<>()` method
3. For each value of the workflow-identifying field of the first level, define the set of workflows in the `WithFlows()` method
4. For each value of the workflow-identifying field of the second level, define the set of workflows in the `WithSubFlows()` method

Workflow-Identifying Fields of the Second Level: To Define a Workflow

The following activity will walk you through the process of defining workflows for the workflow-identifying field of the second level.

Story

Suppose that on the Repair Work Orders (RS30100) form, a workflow for the repair work order may depend on values of two fields: `OrderType` and `Behavior`. The `OrderType` field is already used as the workflow-identifying field. Therefore, the `Behavior` field should be used as the workflow-identifying field of the second level.

The `Behavior` field can have the following values:

- *B1*

  For this value, the user should not be able to put a repair work order on hold. That means, the transitions from the *Ready For Assignment* to *On Hold* state and from *Pending Payment* to *On Hold* state should not be available (Items 1.1 and 1.2 on the diagram below).

- *B2*

  For this value, a new status of the repair work order should be available: *In Progress*. When a repair work order is in the *Assigned* status, a user should be able to click **Start** button and change the status of the order into *In Progress*. Then, a user should be able to click the **Complete** button which changes the status of the repair work order into *Completed*. That means, a new state and two new transitions should be available instead of an old transitions from the *Assigned* to *Completed* state (Item 2.1 on the diagram below).

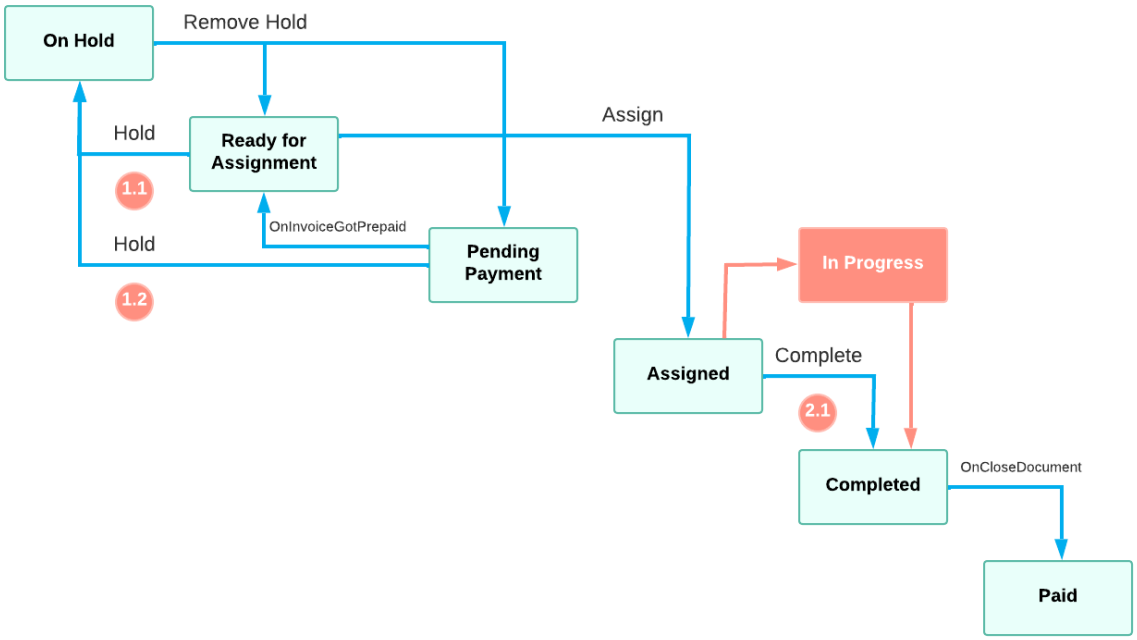


Figure: Changes of the workflow for the Repair Work Orders form

Process Overview

In this activity, you will first add the `UsrBehavior` custom field with corresponding string values to the Repair Work Orders (RS301000) form. Then, in the screen configuration for the Repair Work Orders (RS30100) form, you

will specify the `UsrBehavior` field as the workflow-identifying field of the second level. For each value of the `UsrBehavior` field, you will add a workflow of the second level.

For the B1 value of the UsrBehavior field, you will modify the default workflow by removing the following transitions:

- From the Ready For Assignment to On Hold state
- From the Pending Payment to On Hold state

For the B2 value of the UsrBehavior field, you will modify the default workflow by doing the following:

- Adding the In Progress state
- Removing the transition to the Completed state
- Adding a transition from the Assigned to In Progress state which is triggered by the new Start action

  You will also add the Start action to the screen configuration.
- Adding a transition form the In Progress to Completed state which is triggered by the existing Complete action

System Preparation

In an instance with the *T100* dataset, make sure that you have done the following:

- Prepared an instance with the *PhoneRepairShop* customization project and enabled the workflow validation by performing the prerequisite activities in the *Preparing an Instance for Workflow Customization* chapter
- Defined the default workflow for the Repair Work Orders (RS301000) form according to the customization description in *Company Story and Customization Description*
- Defined a workflow identifying field for the Repair Work Orders (RS301000) form as described in *Workflow-Identifying Fields: To Add a Workflow for a Value of the Workflow-Identifying Field*

Step 1: Add the Behavior Field (Self-Guided Exercise)

In this step, you will add the `UsrBehavior` field to the Repair Work Order (RS30100) form.

> ⚠ The field must have the `Usr` prefix because it is a custom field.

To add the field, use the same instructions as described in *Step 1: Adding the OrderType Field and Corresponding Box to the UI*.

Step 2: Specify the Workflow-Identifying Field of the Second Level

In this step, you will specify the `UsrBehavior` field as the workflow-identifying field of the second level in the extension of `RSSVWorkOrderEntry_Workflow` class. You will do it using the `FlowSubTypeIdentifierIs` method. Do the following:

1. In the `RSSVWorkOrderWorkflow_Extension` class which you defined in *Workflow-Identifying Fields: To Add a Workflow for a Value of the Workflow-Identifying Field*, define the set of constant string values and corresponding classes for the value of the `UsrBehavior` field which will later be used as values of the workflow-identifying field of the second level.

   > ⚠ You can do it the same way as you did for the `UsrOrderType` field.

2. In the `Configure` method, after the call to the `FlowTypeIdentifierIs` method, call the `FlowSubTypeIdentifierIs` method and specify the UsrBehavior field as the type parameter as shown in the following code.

```
context.UpdateScreenConfigurationFor(screen => screen
  .FlowTypeIdentifierIs<RSSVWorkOrder_Extension.usrOrderType>()
  .FlowSubTypeIdentifierIs<RSSVWorkOrder_Extension.usrBehavior>());
```

Step 3: Define Workflows for Each Value of the Workflow-Identifying Field of the Second Level

In this step, you will define a workflow for each value of the `UsrBehavior` field using the `WithSubFlows` method. This workflows will be applied to the *Simple* value of the `UsrOrderType` field.

You do not need to define a complete workflow for each value of the `UsrBehavior` field because the workflows of the second level inherit their configuration from the workflow of the first level. You can define only the differences such as add states or modify transitions.

Do the following:

1. In the `WithFlows` method call, in the `Add<OrderTypes.simple>` method, call the `WithSubFlows` method as shown in the following code.

```
.Add<OrderTypes.simple>(flow => flow
  // states and transitions here
  .WithSubFlows(subFlows =>
  {
    ...
  }));
```

2. Add the workflow for the *B1* value of the `UsrBehavior` field: In the `WithSubFlows` method, call the `Add` method and specify the *B1* value as the type parameter.

```
.Add<OrderTypes.simple>(flow => flow
  // states and transitions here
  .WithSubFlows(subFlows =>
  {
    subFlows.Add<Behaviors.b1>("S1", subFlow =>
      // define states and transitions here
    );
    ...
   }));
```

3. Add the workflow for the *B2* value of the `UsrBehavior` field: In the `WithSubFlows` method, call the `Add` method and specify the *B2* value as the type parameter.

```
.Add<OrderTypes.simple>(flow => flow
  // states and transitions here
  .WithSubFlows(subFlows =>
  {
    subFlows.Add<Behaviors.b1>("S1", subFlow =>
      // define states and transitions here
    );
    subFlows.Add<Behaviors.b2>("S2", subFlow =>
      // define states and transitions here
    );
  }));
```