Sitecore E-Commerce Services 2.2 for CMS 7.0 or Later

Order Manager Developer's Cookbook

A developer's guide to configuring the Order Manager.

Table of Contents

Chapter 1

Introduction

The Order Manager is an application that is based on SPEAK, easy to use, and easy to configure to suit your own requirements.

This guide describes how to customize and extend the Order Manager (OM) application in the backend. It is useful for developers who are looking for information about the Order Manager application.

This manual contains the following chapters:

- **Chapter 1 — Introduction**
  This chapter is introduction to the guide.
- **Chapter 2 — Setting up the Application**
  This chapter describes the different layouts of SPEAK and how to set up the OM application in a certain layout.
- **Chapter 3 —Configuring the Order Manager Application in SPEAK**
  This chapter describes how to configure all the pages in the SPEAK layout.
- **Chapter 4 — Configuring the Order Report in Stimulsoft**
  This chapter describes how to configure the order report in Stimulsoft.
- **Chapter 5 — Using the Order Manager API**
  This chapter is an API reference guide for OM.

Chapter 2

Setting up the Application

This chapter provides an overview of the default pages and controls that are available in the Order Manager. All the controls in the pages are customizable. It is also easy to add more pages if the default configuration is not suitable for your business needs.

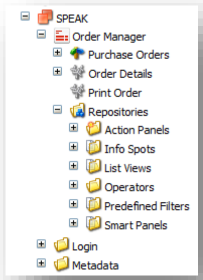
This chapter describes:

- The structure of the order manager application in SPEAK.
- The navigation diagram of SPEAK.

2.1 The Structure of the Order Manager Application in SPEAK

To work with the OM application, you must install the Order Manager package. For information about installing the OM package, see the *SES installation guide*.

In the **Content Editor**, you can navigate to the **Order Manager** root item:
`/sitecore/system/Modules/SPEAK/Order Manager`



The following table describes the folders that you can configure in the Order Manager:

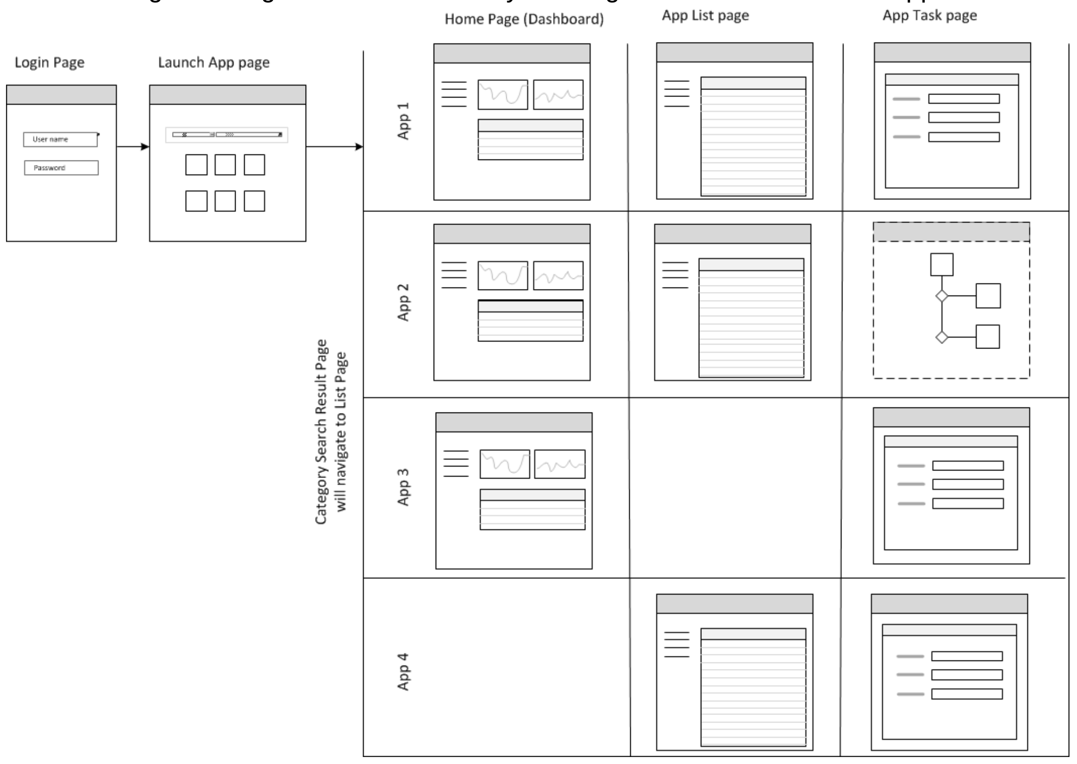
Folder	Description
Purchase Orders	This folder contains links to the navigation filters that appear on the left side navigation panel. Template: `/sitecore/templates/SPEAK/Base templates/Navigation`
Order Details	This folder contains definitions of the sections and fields on the order details task page. For more information, see *Configuring the Order Details Task Page*. Template: `/sitecore/templates/SPEAK/Pagetypes/Task page`
Print Order	This folder contains the definition of the task page that renders the order details report. For more information, see *Configuring the Order Report in* Stimulsoft. Template: `/sitecore/templates/SPEAK/Pagetypes/Task page`
Repositories	This folder contains miscellaneous SPEAK controls configurations. Template: `/sitecore/templates/SPEAK/Folders/Repositories`
Repositories/Action Panels	This folder contains actions that appear in different action panels. Template: `/sitecore/templates/SPEAK/Folders/Actions`
Repositories/Info Spots	This folder contains the controls that appear in the right hand side of the order details task page. Template: `/sitecore/templates/SPEAK/Folders/Info spots`

Folder	Description
Repositories/List Views	This folder contains all the list controls that appear on the dashboard page and the list page. It defines the filter options and list columns. For more information see, *Configuring the Order Manager Application in* SPEAK This chapter describes the configuration options in all the SPEAK pages. These configuration options are described in the section *The Navigation Diagram of* SPEAK. The following sections describes how to: • Set up the controls on the Dashboard. • Configure the navigation filters. • Configure the list page. • Configure the order details task page. • Configure the smart panel. Setting up the Controls on the Dashboard and *Configuring the List Page*. Template: `/sitecore/templates/SPEAK/Folders/List Views`
Repositories/Operators	This folder contains the Order Manager specific operators for filtering orders. By default, it contains the operators: • Is equal to • Search • between **Note** You should not edit any of these operators because they are part of the implementation details of the Order Manager. However, you can add more operators. Template: `/sitecore/templates/Common/Folder`
Repositories/Predefined Filters	This folder contains the list views and filter controls. These are the predefined filter options through which you can filter the order. You can add your own custom options here. For more information, see *Configuring the Predefined Filters*. Template: `/sitecore/templates/SPEAK/Folders/List Views filters`
Repositories/Smart Panels	This folder contains the smart panels content also known as quick views. For more information, see the section *Configuring the Smart* Panel. Template: `/sitecore/templates/SPEAK/Folders/SmartPanels`

Note
If you upgrade your installation, any configuration changes that you make are overwritten. Therefore, you must always create a backup of your configuration settings in the Order Manager.

2.2 The Navigation Diagram of SPEAK

The following flow diagram illustrates how you navigate in a SPEAK based application.



As illustrated in the previous image, you can configure the layouts of the SPEAK pages in different ways but we have chosen the *App 1* theme.

The following table describes every page in the navigation architecture:

SPEAK Page	**Description**
Login Page	To launch the SPEAK login page, enter its URL in your web browser.
Launch Application Page	Contains a list of the applications that are available in SPEAK.
Home Page (Dashboard)	Provides users with an overview of their tasks and what they are currently working on. In the Order Manager, users can have their own individual home page depending on the security roles they have been assigned in Sitecore.
List Page	Displays the results of saved navigation filters in the left hand navigation panel. The default Order Manager application comes with the following pre-defined navigation filters: • Orders • Open orders • Orders in process • Closed orders • Cancelled orders
Task Page	This page is also known as the order details page, displays the full details of an order and enables you to complete specific order management tasks

This guide describes how to configure the dashboard, list pages, and task pages.

Chapter 3

Configuring the Order Manager Application in SPEAK

This chapter describes the configuration options in all the SPEAK pages. These configuration options are described in the section *The Navigation Diagram of* SPEAK.

The following sections describes how to:

- Set up the controls on the Dashboard.
- Configure the navigation filters.
- Configure the list page.
- Configure the order details task page.
- Configure the smart panel.

3.1 Setting up the Controls on the Dashboard

The dashboard is the first page that you see when you choose the Order Manager application in SPEAK. By default, the dashboard contains the navigation filters on the left hand panel, and the last new orders created and the latest orders that are ready to be captured.

This section describes how to configure the dashboard on the Order Manager page.

3.1.1 Configuring Data Sources

You can configure the data source of an in item in the Order Manager. There are two types of data source in SPEAK: fast query and object detail list. In the Order Manager, you use the object detail list to access the order data that is stored in a separate order database.

To configure the data source of the purchase orders in the **Content Editor**:

1. Navigate to the **Purchase Orders** item: `/sitecore/system/Modules/SPEAK/Order Manager/Repositories/List Views/Purchase Orders`
2. Right click **Purchase Orders**, click **Insert**, select **ObjectDetailList** and then call it **Sample Orders.**
3. In the **Content** section, navigate to the **ObjectDataSourceSettings** section.

The following table describes the fields in the Object data source:

Field	Description
EnablePaging	Indicates whether or not the data source control supports paging through the data that it retrieves.
TypeName	The name of the class on which the `ObjectDataSource` object is based — for example, the type that is responsible for handling the Select, Update, Delete, Insert operations.
DataObjectTypeName	The name of a class that the `ObjectDataSource` object uses as the return value in an update, insert, or delete data operation.
SelectMethod	The name of the method that the `ObjectDataSource` control invokes in the object that is specified in the `TypeName` property, to retrieve data.
SelectParameterName	The name of the parameter that is used in the method specified by the `SelectMethod` property.
SelectParameterValue	The value of the parameter that is used in the method specified by the `SelectMethod` property.
UpdateMethod	The name of the method that the `ObjectDataSource` control invokes to update the data.
OldValuesParameterFormatString	The format string that should be applied to the names of the parameters for original values that are passed to the Delete or Update methods.
DeleteMethod	The name of the method that the `ObjectDataSource` control invokes on the object that is specified in `TypeName` to delete data.

Field	Description
DeleteParameterName	The name of the parameter that is used by the `DeleteMethod` method.
DeleteParameterValue	The parameters collection that is used by the `DeleteMethod` method.
InsertMethod	The name of the method or function that the `ObjectDataSource` control invokes on the object that is specified in `TypeName` to insert data.
SelectCountMethod	The name of the method or function that the `ObjectDataSource` control invokes to retrieve a row count.

You can also configure the data sources in:

- Lists in the dashboard.
- List pages.
- Order details that have one overall data source and another for each list control.

3.1.2 Configuring a Shop Context

In the **Content Editor**, a shop context represents a webshop that appears in the Web Store Selector in the client interface. You must configure a shop context for each individual webshop. You can configure as many shop contexts as you need.

To configure the **Shop Contexts** item in the **Content Editor**:

1. In the **Content Editor**, navigate to the *Shop Contexts* item:
   `/sitecore/system/Modules/SPEAK/Order Manager/Repositories/Shop Contexts`

1. Right click **Shop Contexts**, click **Insert**, and then click **Shop Context**.
2. Call it **Third Web Store**.
3. On the **Content** tab, assign values to the following fields.

Field	Description
Name	The logical name of the web shop. In this example, call it *thirdwebstore*.
Title	The webshop name that appears in the client interface in the web store selector. In the previous step, you called it *Third Web Store*.
Tooltip	The hint that describes the shop in the client interface. You can enter *third web shop for testing purposes*.
Icon	The icon that appears next to this shop context item in the **Content Editor**. You can leave it as *business/32x32/shoppingcart.png*.

4. In Visual Studio, open the `Sitecore.Ecommerce.Examples.config` file of the solution and register the *Third Web Store*.

```
< sites>
  <site name="thirdwebstore" .../>
</sites>
```

Once you have created a shop context, you must specify the users who have access to the webshop:

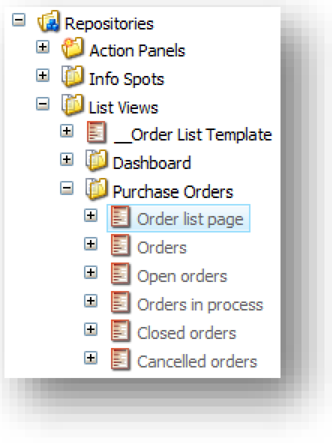
1. In Sitecore **Desktop**, click **Sitecore Security Tools**, and then select **Role Manager**.
2. In the **Role Manager** dialog box, click the **New** tab.
3. In the role **Name** field, enter *Order Manager Third Web Store Processing* and in the domain field, enter *Sitecore.*
4. Click the **Members** tab, click **Add**, click **Users**, and then select the user who you want to make a member of the role and then click **OK.**

3.2 Configuring the Navigation Filters

You can configure as many navigation filters as you want in your application so that you can search for orders according to your own criteria.

To create a navigation filter:

1. In the **Content Editor**, navigate to the **Order List Template**
   `/sitecore/system/Modules/SPEAK/Order Manager/Repositories/List Views/__Order List Template`
2. Create a clone of the **Order List Template** in the **Purchase Orders** repository.
   `/sitecore/system/Modules/SPEAK/Order Manager/Repositories/List Views/Purchase Orders`



Note
You can also create the filter in the repository instead of cloning the **__Order List Template** item. However, we recommend that you use cloning for maintenance reasons. If you modify the template, all of the clones are modified as well. For example, you can add a field to all the filters by adding it to the **__Order List Template** item.

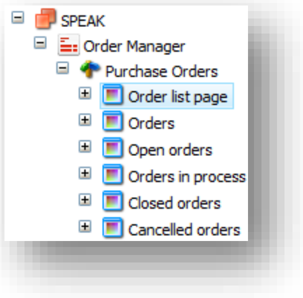
3. Name the new repository **Order list page**. In the **Content** section, you can also configure the following if you want:
   - **Enable Collapsing**
   - In the **LoadDataWith** field, enter **PageScroll** or **ElementScroll**.

   Note
   The recommended setting for **LoadDataWith** in SPEAK is **PageScroll**.

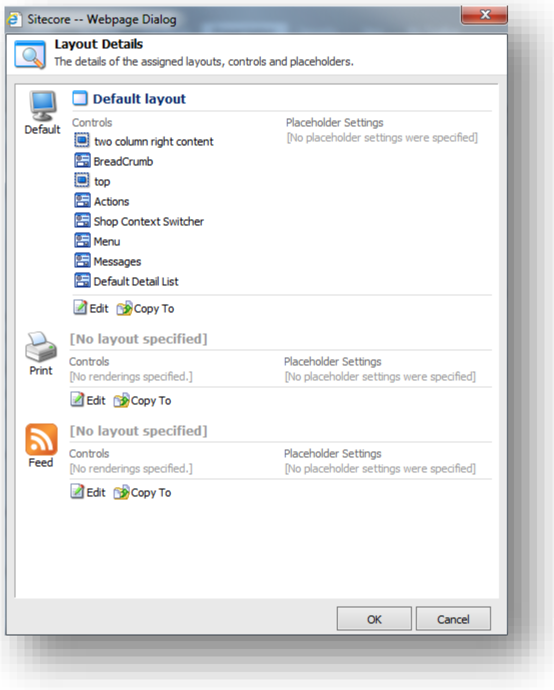
4. Select the **EnableFiltering** option.
5. In the **ObjectDataSourceSettings** group, configure the object data source.

   For more information, see the section *Configuring Data Sources*.

6. Navigate to the **Purchase Orders** item.
`/sitecore/system/Modules/SPEAK/Order Manager/Purchase Orders`



7. Right click **Purchase Orders**, click **Insert** and then select **List page** and then call it **Order list page**.
8. On the ribbon, click the **Presentation** tab, in the **Layout** group, click **Details**.



9. On the **Layout Details** dialog box, click **Default Details List**, and then enter the path to the filter created in the **Data Source** field.
`/sitecore/system/Modules/SPEAK/Order Manager/Repositories/List Views/Purchase Orders/Order list page`

Note
You can create multiple navigation filters that refer to the same list view.

3.2.1 Configuring Navigation Filters According to a User Role

You can configure the Order Manager application to show or hide items according to the user's role.

To use the standard Sitecore security feature to show or hide navigation filters for different users:

1. In the **Content Editor**, navigate to the **Order list page** filter that you have just created.
2. Click the **Security** tab and then click **Access Viewer**.
3. In the **Access Viewer**, you can then deny any user read access to, for example, the **Order list page** and the **Cancelled orders** page.

   For more information about how to use the **Access Viewer**, see the *Security Administrator's Cookbook*.

3.3 Configuring the List Page

Once you have created the navigation filter, you can configure the list page that the navigation filter generates. This section describes how to:

- Configure the columns on the list page.
- Configure the predefined filters.

3.3.1 Configuring a Column

The list page contains the orders that result from the navigation filter.

You can present any information that belongs to the order on the list page. To add a column to the table on the list page of a navigation filter:

4. In the **Content Editor**, locate the filter to which you want to add a new column, for example, `/sitecore/system/Modules/SPEAK/Order Manager/Repositories/List Views/Purchase Orders/Orders`
5. Right click **Orders**, click **Insert**, and then select **Column Field**.

   The default selection is the standard **Column Field**.
6. Name it **Currency**.
7. Enter a value for the **HeaderText**. You can use the same name as that of the **Column Field** — **Currency.**
8. In the **DataField** general property, enter the property name that you want to fill the column with — **Currency**.

Note
In the **DataField**, you can only enter a property that exists in the `Sitecore.Ecommerce.Apps.OrderManagement.Models` class.

For more information, see the section *Adding a Column to an Order Details List*.

3.3.2 Configuring the Predefined Filters

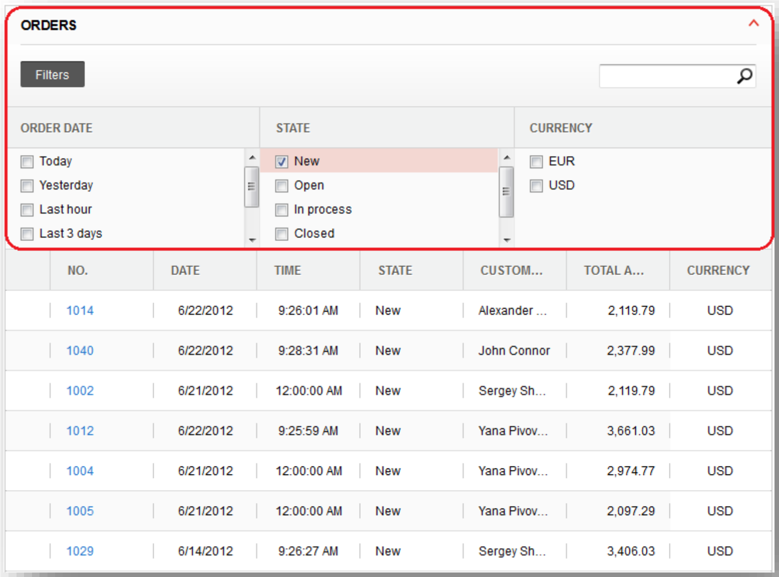
You can use a predefined filter to further refine the list of orders that result from the navigation filter.

This section describes how to configure a predefined filter.

To navigate to the predefined filter options in the **Order Manager**:

1. In the **Order Manager**, navigate to the home page.
2. Click a navigation filter to navigate to a list page, for example, the **Orders** page.

3. On the **Orders** page, click **Filters** and you should see the following filters:



ORDER DATE	STATE	CURRENCY
☐ Today	☑ New	☐ EUR
☐ Yesterday	☐ Open	☐ USD
☐ Last hour	☐ In process	
☐ Last 3 days	☐ Closed	

NO.	DATE	TIME	STATE	CUSTOM...	TOTAL A...	CURRENCY
1014	6/22/2012	9:26:01 AM	New	Alexander ...	2,119.79	USD
1040	6/22/2012	9:28:31 AM	New	John Connor	2,377.99	USD
1002	6/21/2012	12:00:00 AM	New	Sergey Sh...	2,119.79	USD
1012	6/22/2012	9:25:59 AM	New	Yana Pivov...	3,661.03	USD
1004	6/21/2012	12:00:00 AM	New	Yana Pivov...	2,974.77	USD
1005	6/21/2012	12:00:00 AM	New	Yana Pivov...	2,097.29	USD
1029	6/14/2012	9:26:27 AM	New	Sergey Sh...	3,406.03	USD

Configuring an Expression Group

Expression groups are predefined filters.

To create an expression group:

1. In the content tree, navigate to the **Purchase Orders** item `/sitecore/system/Modules/SPEAK/Order Manager/Repositories/Predefined Filters/Purchase Orders`
2. Right click **Purchase Orders**, click **Insert from Template**.
3. In the **Insert from Template** dialog box, navigate to the template `/sitecore/templates/SPEAK/Expression Group` and then call the new expression group **Total Amount**.
4. Select **Total Amount**, click the **Content** tab and then assign values for the following fields.

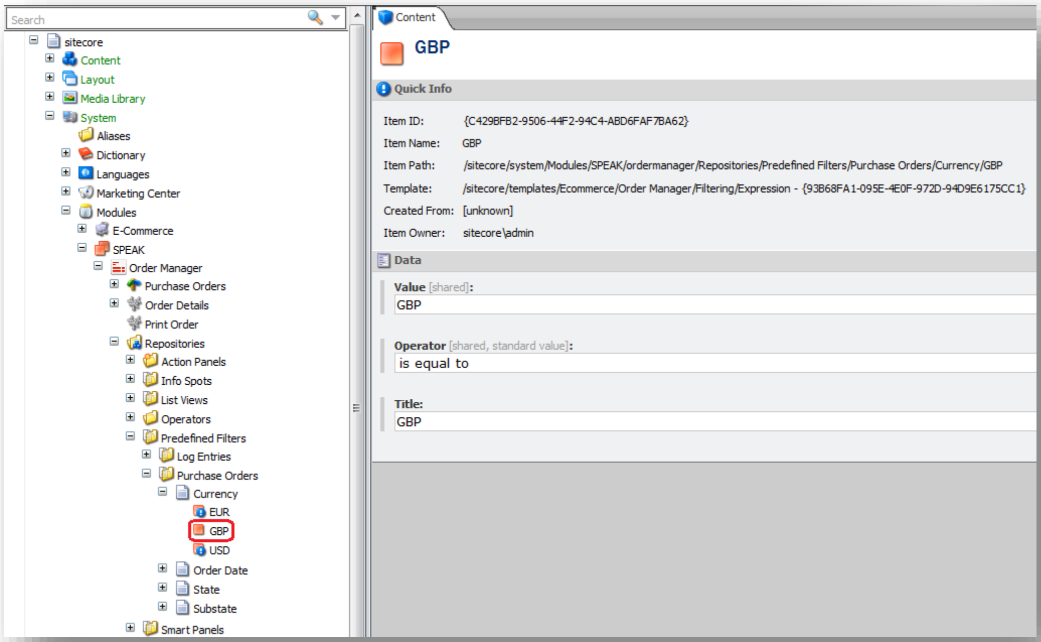
Field	Description
Title	The name of the filter in the UI. You called it **Total Amount** in the previous step.
Name	The logical name of the filter. Call it **TotalAmount**.
Type	The type of data that the expression group is filtering. You can set it to **Date**, **Enum** or **UTC Date**. In this example, you must select **Enum**.

5. Navigate to the **Orders** filter: `/sitecore/system/Modules/SPEAK/Order Manager/Repositories/List Views/Purchase Orders/Orders`
6. Click **Orders**, and on the **Content** tab, click **DetailList settings**, **filters**, and then add the **Total Amount** expression group to the **Orders** navigation filter.

Configuring a Value Based Expression

To add a value based expression to a predefined filter:

1. In the **Content Editor**, navigate to the predefined filter that you want to add the criteria to — for example, *Currency*
   `/sitecore/system/Modules/SPEAK/Order Manager/Repositories/Predefined Filters/Purchase Orders/Currency`
2. Right click **Currency**, and click **Insert from Template**.
3. In the **Insert from Template** dialog box, navigate to the template: `/sitecore/templates/Ecommerce/Order Manager/Filtering/Nonlocalizable Expression`, and name it **GBP**.



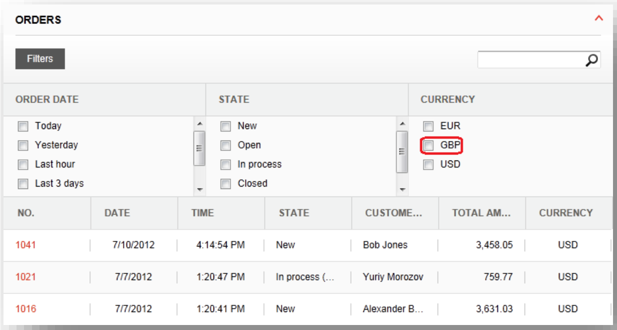
Item ID:	{C429BFB2-9506-44F2-94C4-ABD6FAF7BA62}
Item Name:	GBP
Item Path:	/sitecore/system/Modules/SPEAK/ordermanager/Repositories/Predefined Filters/Purchase Orders/Currency/GBP
Template:	/sitecore/templates/Ecommerce/Order Manager/Filtering/Expression - {93B68FA1-095E-4E0F-972D-94D9E6175CC1}
Created From:	[unknown]
Item Owner:	sitecore\admin

Value [shared]: GBP

Operator [shared, standard value]: is equal to

Title: GBP

4. On the webshop, navigate to the group orders and you can see that the new filter is added.



NO.	DATE	TIME	STATE	CUSTOME...	TOTAL AM...	CURRENCY
1041	7/10/2012	4:14:54 PM	New	Bob Jones	3,458.05	USD
1021	7/7/2012	1:20:47 PM	In process (...	Yuriy Morozov	759.77	USD
1016	7/7/2012	1:20:41 PM	New	Alexander B...	3,631.03	USD

Configuring a Range Based Expression

You can also configure the *Total Amount* predefined filter that you created in the section *Configuring an Expression Group*.

To configure a range based expression:

1. In the content tree, navigate to the *Total Amount* predefined expression group:
   `/sitecore/system/Modules/SPEAK/Order Manager/Repositories/Predefined Filters/Purchase Orders/Total Amount`
2. Right click the *Total Amount* item, click **Insert from Template**.
3. In the **Insert from Template** dialog box, navigate to the template:
   `/sitecore/templates/Ecommerce/Order Manager/Filtering/Nonlocalizable Expression` , and name it *1000-2000*
4. Click the*1000-2000* item and on the **Content** section, assign values to the following fields:

Field	Description
Value	*[1000,2000]*
Title	*[1000,2000]*
Operator	*between* — to select all the orders that have a total amount value within this range.

5. Repeat the previous two steps for the expressions: *2001-4000* and *4001-5000***.**

Creating an Operator

This section describes how to create an operator to search for orders with certain state and substates.

To create a new operator that should be used in the predefined filters:

1. In the content tree, navigate to the *Operators* item:
   `/sitecore/system/Modules/SPEAK/Order Manager/Repositories/Operators`
2. Right click **Operators**, click **Insert**, select **Operator** and name it *substate*.
3. Click **substate**, and in the **Content** section, in the Default field, assign the following condition code:

   ```
   o.field.Substates.Any(s => (s.Code == "values[0]") && s.Active)
   ```
4. Create an expression group and name it *Substates*.

   For more information about how to create an expression group, see the section *Configuring an Expression Group*.
5. Select the *Substates* item and in the **Content** section, assign values to the following fields:

Field	Value
Title	*Substates*
Name	*State*
Type	*Enum*

6. In *Susbstates*, create an expression and name it *Captured in Full*.

   For information about how to create an expression, see the section *Configuring a Value Based Expression*.

7. Click *Captured in Full* and in the **Content** section, assign values to the following fields:

Field	**Description**
Value	*Captured In Full*
Title	*Captured in full*
Operator	*Substate* — the new operator that you have created.

8. Repeat the previous two steps for the expressions: *Packed in Full* and *Shipped in Full***.**

Note
The values in the **Value** fields are case sensitive. Use the same names in the Order Manager database.

3.4 Configuring the Order Details Task Page

The Order Details Task Page is the page that you see after you have click on the order. It contains all the information that is available for the order.

Once you have configured the list page, you can configure the task page. This section describes how to:

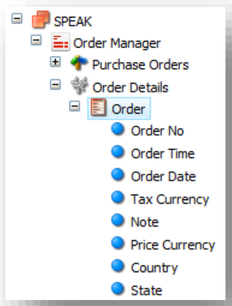
- Add a field editor.
- Add an order details list.
- Add a column to the order details list.
- Add a field to the order details list.
- Extend the Order Manager to see the order details that have one-to-many relationships with an order.

3.4.1 Adding a Field Editor

The field editor represents the details that have a one-to-one relationship with the order.

To add an **Order Field Editor** item:

1. In the **Content Editor**, navigate to the **Order Details** item `/sitecore/system/Modules/SPEAK/Order Manager/Order Details`.
2. Right click **Order Details** and then click **Insert from Template**.
3. In the **Insert from Template** dialog box, navigate to the template `/sitecore/templates/Ecommerce/Order Manager/Web Controls/Order Field Editor`.
4. Call it **Order**.



   You can also configure the following properties:

   - EnableCollapsing
   - 2 Columns
   - DataKeyNames

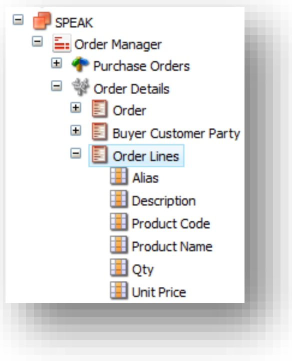
5. Configure the object data source in the **ObjectDataSourceSettings** group. For more information, see the section *Configuring Data Sources.*

3.4.2 Adding an Order Details List

The detail list represents the details that have a one-to-many relationship with the order.

To add an **Order Detail List** item:

1. In the **Content Editor**, right click **Order Details** and then click **Insert from Template**.
2. In the **Insert from Template** dialog box, navigate to the template /`sitecore/templates/Ecommerce/Order Manager/Web Controls/ Order Detail List`
3. Call it **Order Lines**.



You can also configure the following properties:

- EnableCollapsing
- LoadDataWith
- SmartPanel
- EnableFiltering

4. Configure the object data source in the **ObjectDataSourceSettings** group. For more information, see the section *Configuring Data Sources.*

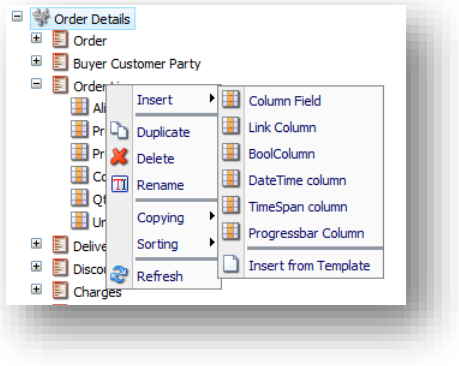
3.4.3 Adding a Column to an Order Details List

You can add a column to an order details item. You can take **OrderLines** as an example of the Order Details Items.

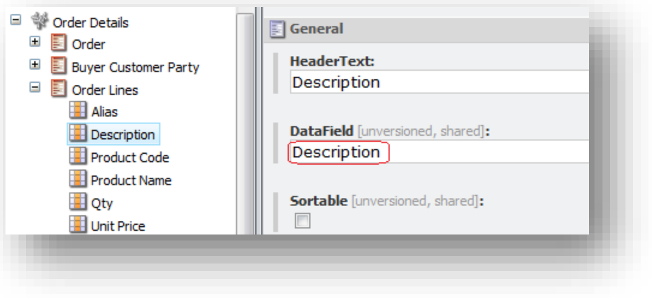
To add a column to **OrderLines**:

1. In the **Content Editor**, navigate to the **Order Lines** item: `/sitecore/system/Modules/SPEAK/Order Manager/order/Order Lines`

2. Right click **Order Lines**, click **Insert**, **Column Field** and then name it **Description.**



3. Click **Description**, in the **Content** tab, **General**, **DataField**, enter **Description** — the name of the property that represents this column in the implementation.



HeaderText: Description

DataField [unversioned, shared]: Description

Sortable [unversioned, shared]:

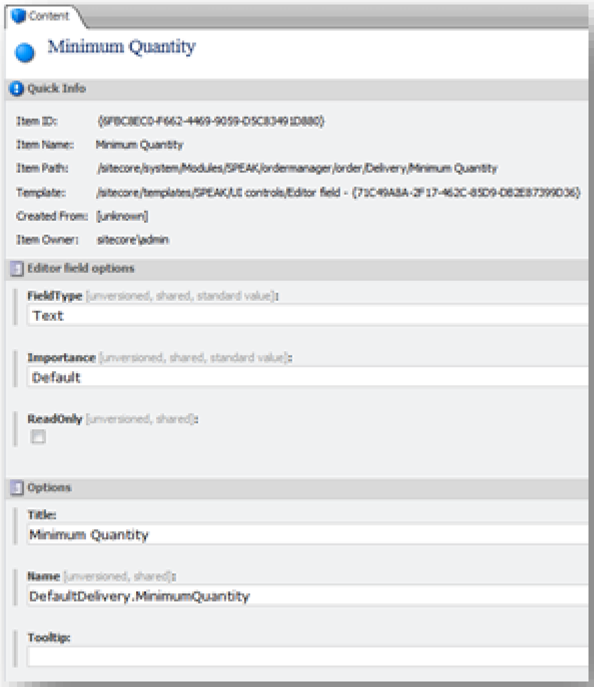
3.4.4 Adding a Field to the Details List

You can add a column to an order details item. You can take **Delivery** as an example of the details list.

To add a field to **Delivery**:

1. In the **Content Editor**, navigate to the **Delivery** details list `/sitecore/system/Modules/SPEAK/Order Manager/order/Delivery`
2. Right click **Delivery**, click **Insert**, select **Editor field** and then name it **Minimum Quantity**.
3. Change the value of the **Name** field to the data property name whose value is displayed in this field — *DefaultDelivery.MinimumQuantity*.

The following image shows the properties of a details item:



Minimum Quantity

Quick Info

Item ID:	{6FBC8EC0-F662-4469-9059-D5C83491D880}
Item Name:	Minimum Quantity
Item Path:	/sitecore/system/Modules/SPEAK/ordermanager/order/Delivery/Minimum Quantity
Template:	/sitecore/templates/SPEAK/UI controls/Editor field - {71C49A8A-2F17-462C-85D9-0B2E87399D36}
Created From:	[unknown]
Item Owner:	sitecore\admin

Editor field options

FieldType [unversioned, shared, standard value]: Text

Importance [unversioned, shared, standard value]: Default

ReadOnly [unversioned, shared]: ☐

Options

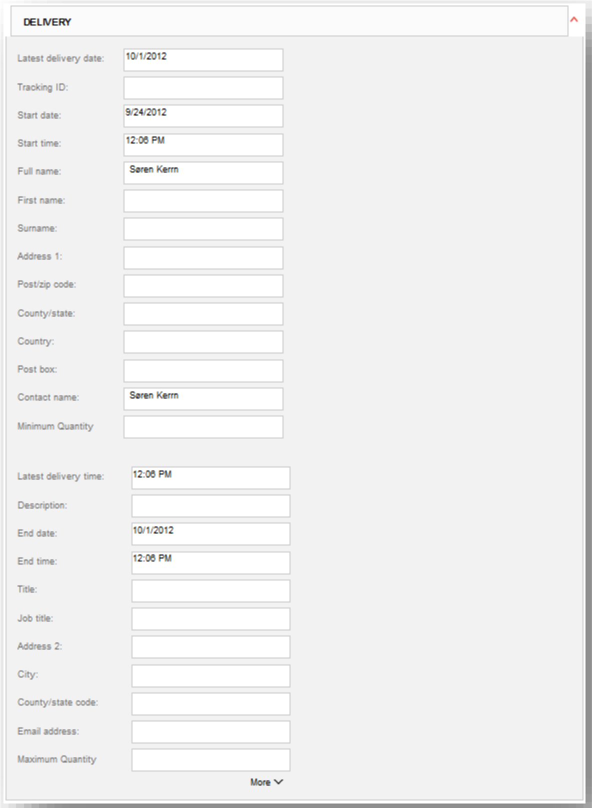
Title: Minimum Quantity

Name [unversioned, shared]: DefaultDelivery.MinimumQuantity

Tooltip:

4. Insert the second **Editor field**, call it **Maximum Quantity** and then Change the value of the **Name** field to *DefaultDelivery.MaximumQuantity*.
5. Insert the third **Editor field**, call it **Quantity** and then Change the value of the **Name** field to *DefaultDelivery.Quantity*.

6. In the **Order Manager** application, navigate to the task page to see the editor fields you have just created.



DELIVERY

Latest delivery date:	10/1/2012
Tracking ID:	
Start date:	9/24/2012
Start time:	12:06 PM
Full name:	Søren Kerrn
First name:	
Surname:	
Address 1:	
Post/zip code:	
County/state:	
Country:	
Post box:	
Contact name:	Søren Kerrn
Minimum Quantity	
Latest delivery time:	12:06 PM
Description:	
End date:	10/1/2012
End time:	12:06 PM
Title:	
Job title:	
Address 2:	
City:	
County/state code:	
Email address:	
Maximum Quantity	

The following table describes the properties of the details item:

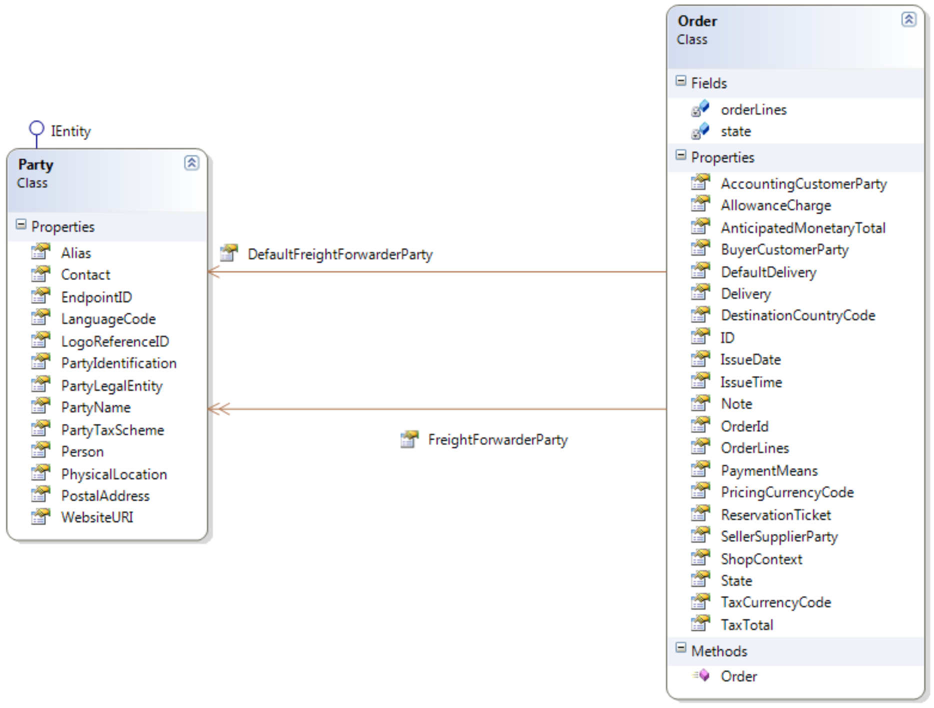
Property	Description
Field Type	Describes the type of the field, for example, Text, Multi-Line Text and Date.
Importance	The importance of the field. You can assign it one of the following values: • *High* — the values of this property appear even if you collapse the accordion group. • *Normal* — the default appearance of the fields. • *Low* — the values of this property only appear when you click on more.
ReadOnly	Specifies if the field is Read-only.
Title	The text to be displayed.
Name	The property of the model that is that is used in the data source that is assigned to the field editor.
Tooltip	The tooltip for the field.

3.4.5 Extending the Order Manager to Show Multi-valued Fields

The order details task page does not contain all of the fields that belong to an order. The following are some examples of this limitation:

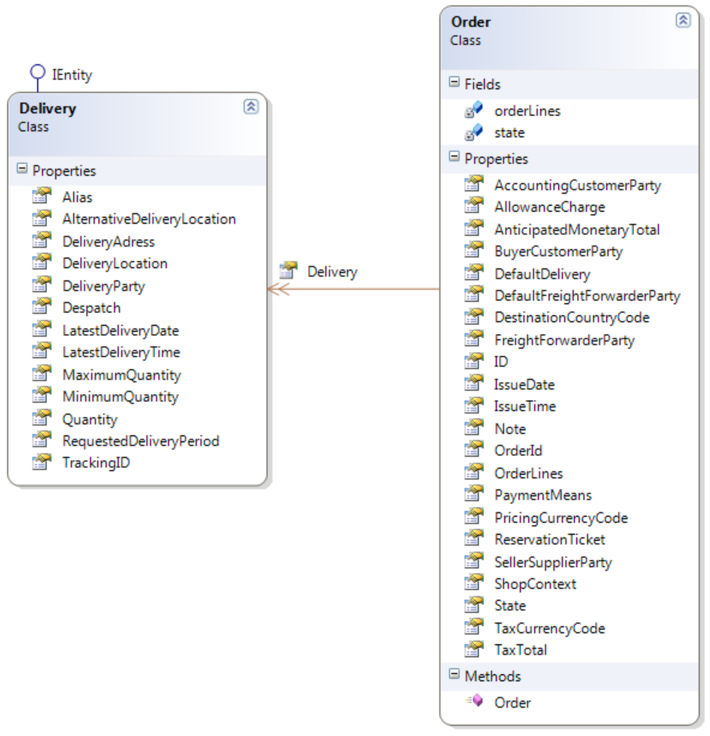
FreightForwarderParty

The Order Manager application shows only one freight forwarder party for each order. The following class diagram shows the `FreightForwarderParty` relationship in the data model.



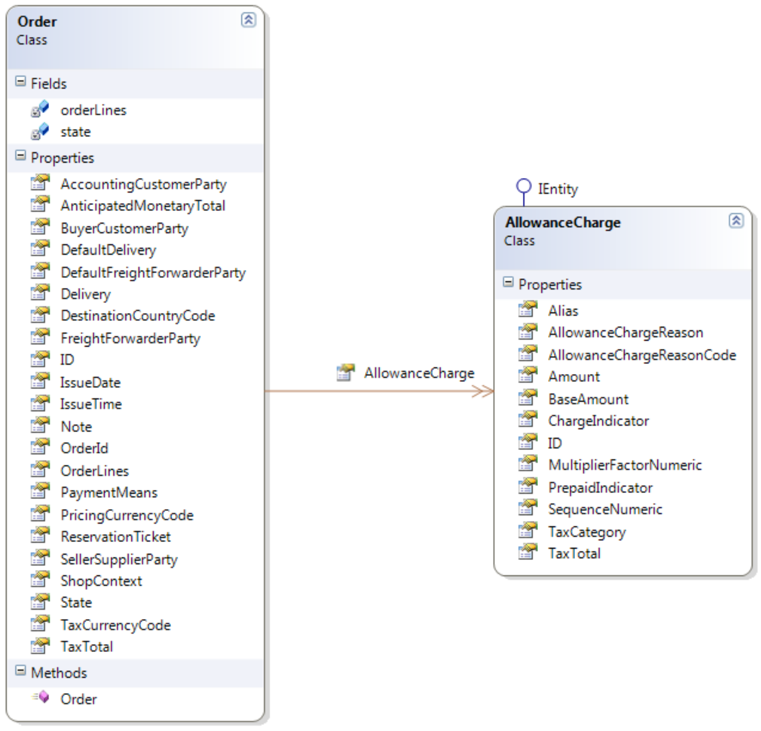
Delivery

The Order Manager application shows only one delivery destination for each order. The following class diagram shows the `Delivery` relationship in the data model.



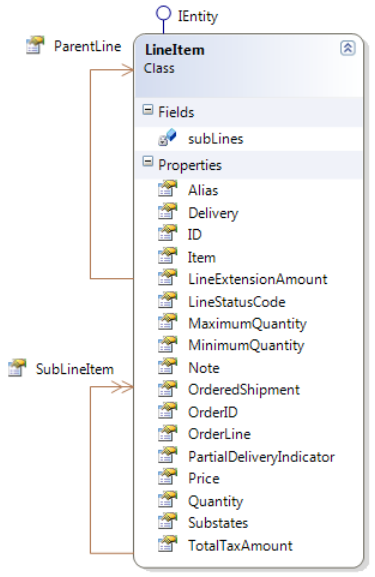
AllowanceCharge

The Order Manager application shows only one delivery destination for each order. The following class diagram shows the `AllowanceCharge` relationship in the data model.



SublineItem

The Order Manager application shows only one delivery destination for each order. The following class diagram shows the `SublineItem` relationship in the data model.



If you want to show all the freight forwarder parties, delivery destinations, allowance charges, and subline items for the order, you must:

1. Extend the Order Manager application to base the corresponding items on **Order details list** instead of **Field editor**
2. Add a **Smart Panel**. The following section describes how to configure the **Smart Panel**.

3.5 Configuring the Smart Panel

The Smart Panel is a detail control that is used to view or edit list pages and task pages. You can use this control to edit a list, for example, orders and order lines. There is no inline functionality to edit a list. This is a flexible way to edit the selected items and to edit multiple entities at the same time.

This chapter describes how to:

- Enable the smart panel in your application
- Configure the actions panel
- Add a custom action

3.5.1 Enabling the Smart Panel in your Application

Showpopup

You can use the `ShowPopup` method in the Order Manager API to manipulate the smart panel.

The `ShowPopup` method has the following signature:

```
public static bool ShowPopup(
            this ScriptManager scriptManager,
            string url,
            object parameters,
            PopupType type,
            out string result)
```

Parameters:

- `url` — the URL of the page that you want to open as the popup content.
- `parameters` — the custom parameters that you want to pass to the popup page.
- `type` — the type of the popup content.

If the popup is not opened yet, it returns `false`. If the popup is closed, it returns `true.` The popup result must be processed if it returns `true`

There are also a few overloads for this method with fewer parameters.

The `Showpopup` method is available as an extension method in the `System.Web.UI.ScriptManager` class. To use it, add the `Sitecore.Marketing.Client.Web.UI.Controls` namespace to your code file.

```
var scriptManager = ScriptManager.GetCurrent(this.Page);
string result = null;
if (scriptManager.ShowPopup("/order manager/edit order",
        new { OrderId = "1V203E", DetailedView = true},
        PopupType.Smart, out result))
          {
            // the dialog was closed
            // do something with result here
          }
```

3.5.2 Configuring the Actions Panels

To configure the General or Order Details global actions in the **Action panel**:

3. In the `Sitecore.Ecommerce.Apps.Web.UI.WebControls.Actions` package, create a handler.
4. Navigate to the **Action Panels** folder /`sitecore/system/Modules/SPEAK/Order Manager/Repositories/Action Panels/Global actions`
5. Insert the new action in the folder that corresponds to the action place.

   The following table describes the different folders in the Action Panels folder.

Folder	Description
Global actions/General	Contains the general actions that are displayed in the **Action Panel** of the **List Page**.
Global actions/Order Details	Contains the order details actions that are displayed in the action panel of the **List Page**.
Order Details Actions/General	Contains the general actions that are displayed in the action panel of the **Task Page**.
Order Lines Actions/Order Line Functions	Contains the order line functions actions that are displayed in the action panel of the **Order Lines** accordion group.
OrderLine Info Actions/Order Line Functions.	Contains the order line functions actions that are displayed in the action panel of **Order Line Details**.

3.5.3 Adding a Custom Action

You can create a custom action in the Order Manager. The following code snippet is a Print Custom Action example:

```
namespace Sitecore.Ecommerce.Apps.Web.UI.WebControls.Actions
{
  using Speak.Extensions;
  using Diagnostics;
  using Ecommerce.OrderManagement.Orders;
  using Logging;
  using Sitecore.Web.UI.WebControls;

  /// <summary>
  /// Defines the print action class.
  /// </summary>
  public class PrintAction : ScriptManagedAction
  {
    /// <summary>
    /// Executes the specified order.
    /// </summary>
    /// <param name="order">The order.</param>
    protected override void Execute([NotNull] Order order)
    {
      Assert.ArgumentNotNull(order, "order");
      this.OrderId = order.OrderId;
      string url = string.Format("{0}ordermanager/printorder?orderid={1}&sc_lang={2}",
      Extensions.GetVirtualFolder(), this.OrderId, Sitecore.Context.Language.Name);
      string script =
      string.Format("window.open('{0}','PrintMe','resizable=yes,scrollbars=yes,
      location=no');", url);
      this.ScriptManager.RegisterStartupScript(script);
    }
```

```
            /// <summary>
            /// Performs the post steps.
            /// </summary>
            protected override void PerformPostSteps()
            {
              LogEntry logEntry = new LogEntry
              {
                Details = new LogEntryDetails(Constants.OrderPrinted),
                Action = Constants.PrintOrderAction,
                EntityID = this.OrderId,
                EntityType = Constants.OrderEntityType,
                LevelCode = Constants.UserLevel,
                Result = Constants.ApprovedResult
              };
             this.Logger.Write(logEntry);
            }
          }
        }
```

Chapter 4

Configuring the Order Report in Stimulsoft

The Stimulsoft reporting tool is used by the Order Manager to render the order confirmation and print the order.

This chapter is not a complete reference for how to configure the Stimulsoft reports, it just contains the customizations you may need for the Order Manager.

To configure the order reports that are defined in Stimulsoft, you can:

- Customize the order details report.
- Set up the data source.
- Create a variable and change its localization.

For information about how to configure the reports, see the *Report Designer Cookbook*.

4.1 Customizing the Order Details Report

To modify in the order details report:

1. In the folder `/sitecore modules/shell/Ecommerce/Reports`, create an Order Details `.mrt` file

Note
We recommend cloning the Order Details file to preserve all of the properties in the report as we do in our Examples packages which contains the `OrderDetailsExtended` report: `/sitecore modules/shell/Ecommerce/ReportsExtended/OrderDetailsExtended.mrt`.

2. To set the path of the new report file, open the new file that you created with the Stimulsoft Report Designer, apply your changes and then add the following elements to the `/App_Config/Unity.config` file:

```
<alias alias="StiReportFactory"
type="Sitecore.Ecommerce.Report.StiReportFactory, Sitecore.Ecommerce.Kernel" />
…
<register type="StiReportFactory">
  <property name="ReportFile" value="/sitecore
  modules/shell/Ecommerce/ReportsExtended/OrderDetailsExtended.mrt" />
</register>
```

4.2 Setting up the Data Source

By default, the data is read from the `OrderReportModel` class. This class contains a predefined set of properties that represent variables.

To set up additional data variables for the Stimulsoft report:

1. Create a class, call it `OrderReportModelExtended` that inherits the `OrderReportModel` class — see the `Sitecore.Ecommerce.Custom` assembly in the *Examples* package:

   In the following example, we extend the order report model with the information about the freight forwarder party.

```
public class OrderReportModelExtended : OrderReportModel
{
    // Getting the account ID of the default freight forwarder party.
    public virtual string FreightForwarderPartyIdentification
    {
        get
        {
            if ((this.Order != null) && (this.Order.DefaultFreightForwarderParty !=
            null))
            {
                return this.Order.DefaultFreightForwarderParty.PartyIdentification;
            }
            return string.Empty;
        }
    }
}
```

2. Add the following aliases to register the `OrderReportModelExtended` class in the `/App_Config/Unity.config` file:

```
<!—the default model-->
<alias alias="OrderReportModel" type="Sitecore.Ecommerce.Report.OrderReportModel,
Sitecore.Ecommerce.Kernel" />
<!—the new extended model ->
<alias alias="OrderReportModelExtended"
type="Sitecore.Ecommerce.Custom.Reports.OrderReportModelExtended,
Sitecore.Ecommerce.Custom" />
…
<!—Redirecting the mapping from default model to the extended model ->
<register type="OrderReportModel" mapTo="OrderReportModelExtended" />
<?xml version="1.0" encoding="utf-8"?>
```

3. Open the `OrderReportModel.mrt` file in Visual Studio and then add the `FreightForwarderPartyIdentification` data source variable:

```
<DataSources isList="true" count="10">
  <Order isKey="true" Ref="2"
  type="Stimulsoft.Report.Dictionary.StiBusinessObjectSource">
    <Name>Order</Name>
    <Dictionary isRef="1"/>
    <Alias>Order</Alias>
    <Parameters isList="true" count="0"/>
    <NameInSource>Order</NameInSource>
    <Columns isList="true" count="60">
      ...
      <value>FreightForwarderPartyIdentification, System.String</value>
    </Columns>
  </Order>
</DataSources>
```

4.3 Creating a Variable

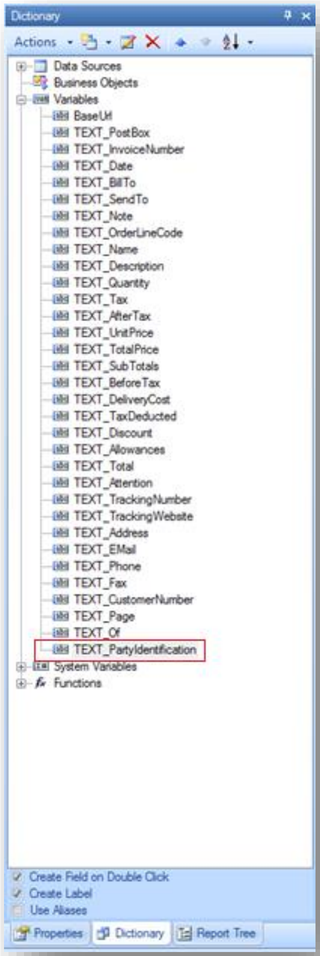
To start customizing the Stimulsoft report by inserting a variable, you should download and install the *Stimulsoft Report Designer* from the Stimulsoft website.

For more information about the Stimulsoft Report Designer, see the *Report Designer Cookbook*.

To customize a variable:

1. In the **Content Editor**, navigate to the *Dictionary* item: `/sitecore/system/Dictionary`
2. Right click *Dictionary*, click **Insert** and then select **Dictionary entry**.
3. Call the new entry *Party Identification*.
4. In the Stimulsoft Report Designer, open the `OrderDetailsExtended.mrt` report with Stimulsoft Report Designer.
5. Create a variable in the report, call it *TEXT_PartyIdentificatin* and in the **Value** field enter the name of the item that you have created — *Party Indentification*.

   The following images show the dictionary and the properties of the new variable:



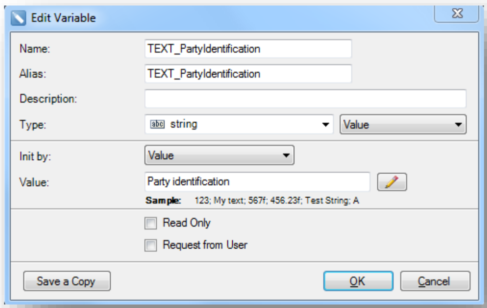
Dictionary

Actions

- Data Sources
- Business Objects
- Variables
  - BaseUrl
  - TEXT_PostBox
  - TEXT_InvoiceNumber
  - TEXT_Date
  - TEXT_BillTo
  - TEXT_SendTo
  - TEXT_Note
  - TEXT_OrderLineCode
  - TEXT_Name
  - TEXT_Description
  - TEXT_Quantity
  - TEXT_Tax
  - TEXT_AfterTax
  - TEXT_UnitPrice
  - TEXT_TotalPrice
  - TEXT_SubTotals
  - TEXT_BeforeTax
  - TEXT_DeliveryCost
  - TEXT_TaxDeducted
  - TEXT_Discount
  - TEXT_Allowances
  - TEXT_Total
  - TEXT_Attention
  - TEXT_TrackingNumber
  - TEXT_TrackingWebsite
  - TEXT_Address
  - TEXT_EMail
  - TEXT_Phone
  - TEXT_Fax
  - TEXT_CustomerNumber
  - TEXT_Page
  - TEXT_Of
  - TEXT_PartyIdentification
- System Variables
- Functions

✓ Create Field on Double Click
✓ Create Label
Use Aliases

Properties | Dictionary | Report Tree



Edit Variable

Name:	TEXT_PartyIdentification	
Alias:	TEXT_PartyIdentification	
Description:		
Type:	string	Value
Init by:	Value	
Value:	Party identification	

Sample: 123; My text; 567f; 456.23f; Test String; A

☐ Read Only
☐ Request from User

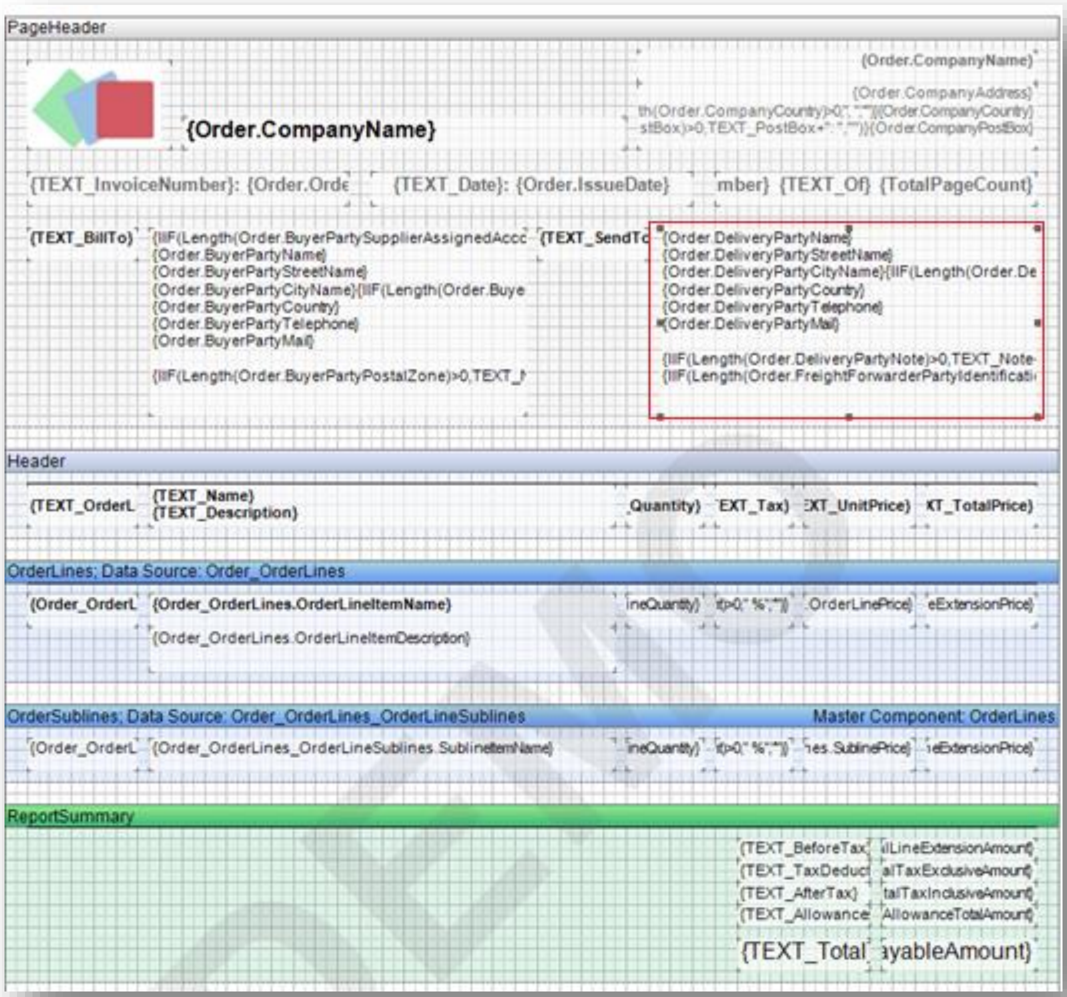
Save a Copy | OK | Cancel

In the code view, the following snippet is the variable representation:

```
<Variables isList="true" count="33">
  ...
<value>,TEXT_PartyIdentification,TEXT_PartyIdentification,System.String,Party_x0
020_identification,False,False</value>
</Variables>
```

The value of the constant in the `OrderDetailsExtended.mrt` file — `Party_x0020_identification` should be exactly the same as the key in the Sitecore dictionary — *Party identification* — where `_x0020_` is the code of the space.

6. In the designer view, extend the markup of the `OrderDetailsExtended.mrt` file with the information about freight forwarder party, as shown in the following image:



PageHeader

{Order.CompanyName}
{Order.CompanyAddress}
th(Order.CompanyCountry)>0", ""))(Order.CompanyCountry)
stBox)>0,TEXT_PostBox+": ",""))(Order.CompanyPostBox)
{Order.CompanyName}

{TEXT_InvoiceNumber}: {Order.Orde {TEXT_Date}: {Order.IssueDate} mber} {TEXT_Of} {TotalPageCount}

{TEXT_BillTo} {IIF(Length(Order.BuyerPartySupplierAssignedAccc
{Order.BuyerPartyName}
{Order.BuyerPartyStreetName}
{Order.BuyerPartyCityName}{IIF(Length(Order.Buye
{Order.BuyerPartyCountry}
{Order.BuyerPartyTelephone}
{Order.BuyerPartyMail}
{IIF(Length(Order.BuyerPartyPostalZone)>0,TEXT_

{TEXT_SendTo {Order.DeliveryPartyName}
{Order.DeliveryPartyStreetName}
{Order.DeliveryPartyCityName}{IIF(Length(Order.De
{Order.DeliveryPartyCountry}
{Order.DeliveryPartyTelephone}
{Order.DeliveryPartyMail}
{IIF(Length(Order.DeliveryPartyNote)>0,TEXT_Note
{IIF(Length(Order.FreightForwarderPartyIdentificati

Header

{TEXT_OrderL {TEXT_Name} {TEXT_Description} Quantity} EXT_Tax} EXT_UnitPrice} XT_TotalPrice}

OrderLines; Data Source: Order_OrderLines

{Order_OrderL {Order_OrderLines.OrderLineItemName} ineQuantity} t}>0,"%","")} OrderLinePrice} eExtensionPrice}
{Order_OrderLines.OrderLineItemDescription}

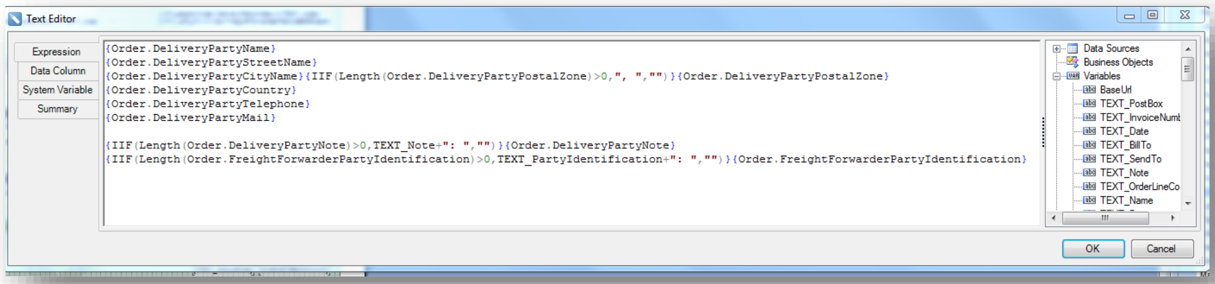
OrderSublines; Data Source: Order_OrderLines_OrderLineSublines Master Component: OrderLines

{Order_OrderL {Order_OrderLines_OrderLineSublines.SublineItemName} ineQuantity} t}>0,"%","")} nes.SublinePrice} neExtensionPrice}

ReportSummary

{TEXT_BeforeTax} ilLineExtensionAmount}
{TEXT_TaxDeduct alTaxExclusiveAmount}
{TEXT_AfterTax} talTaxInclusiveAmount}
{TEXT_Allowance AllowanceTotalAmount}
{TEXT_Total ayableAmount}

The following image shows the designer view of the text field definition with the *TEXT_PartyIdentification* variable:



```
{Order.DeliveryPartyName}
{Order.DeliveryPartyStreetName}
{Order.DeliveryPartyCityName}{IIF(Length(Order.DeliveryPartyPostalZone)>0,", ","")}{Order.DeliveryPartyPostalZone}
{Order.DeliveryPartyCountry}
{Order.DeliveryPartyTelephone}
{Order.DeliveryPartyMail}

{IIF(Length(Order.DeliveryPartyNote)>0,TEXT_Note+": ","")}{Order.DeliveryPartyNote}
{IIF(Length(Order.FreightForwarderPartyIdentification)>0,TEXT_PartyIdentification+": ","")}{Order.FreightForwarderPartyIdentification}
```

The following snippet is the code view of the text field definition with the *TEXT_PartyIdentification* variable:

```
<Text>
  {Order.DeliveryPartyName}
  {Order.DeliveryPartyStreetName}
  {Order.DeliveryPartyCityName}{IIF(Length(Order.DeliveryPartyPostalZone)&gt;0,",
  ","")}{Order.DeliveryPartyPostalZone}
  {Order.DeliveryPartyCountry}
  {Order.DeliveryPartyTelephone}
  {Order.DeliveryPartyMail}
  {IIF(Length(Order.DeliveryPartyNote)&gt;0,TEXT_Note+":
  ","")}{Order.DeliveryPartyNote}
  {IIF(Length(Order.FreightForwarderPartyIdentification)&gt;0,
  TEXT_PartyIdentification+": ","")}{Order.FreightForwarderPartyIdentification}
</Text>
```

You can create variables and fields that do not start with *Text_*.

4.3.1 Changing the Localization of the Variable

You can created a Stimulsoft report in different languages. By default, the customer receives the order confirmation in the same language that they created the order in. In the Order Detaila page, you can use the language value to change the language of the Buyer Customer Party. Sitecore OM currently supports English, Danish, German, and Japanese.

The `StiReportTranslator` class looks up the entry in the Sitecore dictionary and maps it to the corresponding variable in the order report.

To change the localization logic, create a custom version of the `StiReportTranslator` class and register it in the `/App_Config/Unity.config`. For more information about creating a custom version of a class, see the section *Setting up the Data Source*.

The `StiReportTranslator` class uses the Sitecore default localization method: `Sitecore.Globalization.Translate.TextByLanguage(key, language)`

The following snippet shows how the `Translate` method is implemented in the `StiReportTranslator`:

```
public virtual void Translate([NotNull] StiReport report, string languageCode)
    {
      Assert.ArgumentNotNull(report, "report");

      foreach (StiVariable variable in report.Dictionary.Variables)
      {
        if (this.MustBeTranslated(variable.Name))
```

```
            {
              variable.Value = this.TranslatePhraseByLanguage(variable.Value,
              languageCode);
            }
          }
        }
    protected virtual bool MustBeTranslated([NotNull] string key)
        {
          Assert.ArgumentNotNull(key, "key");
          return key.StartsWith("TEXT_");
        }
    protected virtual string TranslatePhraseByLanguage([NotNull] string phrase, [NotNull]
    string languageCode)
        {
          Assert.ArgumentNotNull(phrase, "phrase");
          Assert.ArgumentNotNull(languageCode, "languageCode");
          return Globalization.Translate.TextByLanguage(phrase,
          Language.Parse(languageCode));
        }
```

Chapter 5

Using the Order Manager API

The API of the Oder Manager application consists of three modules: Core Order Manager, Visitor Order Manager and Merchant Order Manager.

This chapter describes how to use the API of each these modules while processing the order.

This chapter contains the following sections:

- Using the Core Order Manager API
- Using the Visitor Order Manager API
- Using the Merchant Order Manager API

5.1 Using the Core Order Manager API

The Core Order Manager API (COM):

- Is a data manipulation layer.
- Is the first layer of abstraction above the actual data storage providers like the Entity Framework or simple Sitecore content items.
- Allows developers to work directly with the entire domain model
- Does not contain any business logic.
- Supports logging of order processing transactions and changing the order properties.

The following sections describe the most important classes in the COM API and how to use the API to import and export orders.

5.1.1 The COM API Reference

The main classes in the COM API are:

Sitecore.Ecommerce.Data.Repository<Order>

This class acts as a layer of abstraction above the actual data storage and allows you to interact with the Sitecore back-end. It checks security, sets the language, intercepts `Create`, `Read`, `Update`, `Delete` (CRUD) operations on orders and uses `OnOrderSaving` and `OnOrderSaved` events to performs some additional operations.

The `OnOrderSaving` and `OnOrderSaved` events are used to perform some additional actions during and after order saving. You can use the `logger` class to add the logging logic to these events.

Sitecore.Ecommerce.Logging.Logger

Almost all of the methods are intentionally marked as `protected internal`. This means that you can only work with them after you create a new implementation that is inherited from the `Repository<Order>` and make a new custom public API.

Developers should use MOM and VOM because they contain the security and business logic. They should not use COM because it has unrestricted access to the historical transaction data.

The `Logger` class has the following methods:

- `GetEntries():IQueryable<LogEntry>` —lists all the orders that the current user has access to.
- `Write(LogEntry):void` — writes information to the log immediately.
- `Log(LogEntry):void` — writes information to the `in-memory` buffer.
- `Flush():void` — moves the data from the buffer to the location where you store your data.

The last two methods support the transaction logic that manipulates an order.

When a visitor to the webshop updates some fields and creates a new order line in a session, this is regarded as a single unit of work and must be logged with the same transaction ID or completely rejected.

Sitecore.Ecommerce.OrderManagement.OrderProcessingStrategy

When an order is being processed, this layer performs robust logging of the operations during the orders processing.

By default, no logging occurs. SES developers should use the COM API to handle the log on their own.

The COM API has unrestricted access to the transaction data. You should be careful while using COM API because you can destroy existing orders. You should preferably use a more high-level API such as the VOM or MOM APIs.

5.1.2 Using the Core API to Import and Export Orders

One example of how to use the Core API is to extend the `OrdersPresenter` class in the MVP web store to support the import and export of orders. In this example, you should not use the VOM and MOM APIs, because the front-end validation and security checks that they perform degrades performance.

To extend the `OrdersPresenter` class:

1. Create the public class `SampleOrderManager` that wraps the protected internal methods of the `Repository<Order>` class.

   These methods are internal and protected to force developers to use the VOM and MOM API:

```
public class SampleOrderManager : Repository<Order>
{
  public SampleOrderManager(CoreOrderStateConfiguration orderStateConfiguration,
  Repository<Order> repository)
  {
    this.StateConfiguration = orderStateConfiguration;
    this.Repository = repository;
  }
  public virtual IQueryable<Order> GetAllOrders(Expression<Func<Order, bool>>
  expression)
  {
    return this.GetOrders(expression);
  }
  public virtual void SaveSingleOrder(Order order)
  {
    this.SaveOrder(order);
  }
}
```

2. Register the `SampleOrderManager` class in the `MvpWebStore.Unity.config` file.

3. Override the constructor of the `OrdersPresenter` class in the MVPWebStore so that you can receive the `SampleOrderManager` object as an additional parameter:

```
public OrdersPresenter(IOrdersView view, VisitorOrderRepositoryBase orderRepository,
SampleOrderManager orderManager): base(view)
{
}
```

   For more information, see the MVPWebstore on the marketplace.

4. To export the orders, you must use the Core API to retrieve all the orders.

   You can use the JSON.NET serializer to serialize them to the JSON format and write the data to a file:

```
var orders = this.orderManager.GetAllOrders(o => true).ToArray();
this.View.Model.SerializedOrders = JsonConvert.SerializeObject(orders,
Formatting.None, this.settings);
using (var file = new FileStream(this.filePath, FileMode.Create, FileAccess.Write))
{
  using (var stream = new StreamWriter(file))
  {
    stream.Write(this.View.Model.SerializedOrders);
  }
}
this.HttpContext.Response.Clear();
this.HttpContext.Response.ContentType = "application/json";
this.HttpContext.Response.AddHeader("content-disposition", "attachment;
filename=\"" + FileName + "\"");
this.HttpContext.Response.WriteFile(this.filePath);
this.HttpContext.Response.Flush();
this.HttpContext.Response.End();
```

5. To import the orders , you should:
   - upload the file to the server,
   - read its content,
   - deserialize the text from the JSON format to the collection of OIOUBL orders
   - save it in the database:

```
var orders =
JsonConvert.DeserializeObject<IEnumerable<Order>>(this.View.Model.SerializedOrders,
this.settings).AsQueryable();
foreach (var order in orders)
{
  this.orderManager.SaveSingleOrder(order);
}
```

5.2 Using the Visitor Order Manager API

When you create a Webshop, you should use the Visitor Order Manager (VOM) API to create and view orders. The VOM API gives you access to the entire domain model.

The main class in the VOM API is `Sitecore.Ecommerce.Visitor.OrderManagement.VisitorOrderRepository`.

The `VisitorOrderRepository` class:

- Is the current default implementation of the `Sitecore.Ecommerce.OrderManagement.VisitorOrderRepositoryBase` abstract class.
- Implements the `Sitecore.Ecommerce.Visitor.OrderManagement.IUserAware` interface that contains the definition of the `CustomerId` property.

  This value identifies the customer that created the order. You can also use the implementation of the `Sitecore.Ecommerce.Users.CustomerManager<T>` class to read the `CustomerId` property in the current user account.
- Manages the visitor who created the orders.

Note
You can also use the VOM API in the Sitecore *MVPWebStore* application which is based on the WebFormsMVP framework. For more information, see *the MVPWebStore Developer's Guide* on the Sitecore Market Place.

The following sections describe how to use the VOM API to:

- Read all orders for a specific customer.
- Cancel an order.
- Create an order.

The last section describes the limitations of the VOM API.

5.2.1 Reading all Orders for a Specific Customer

The MVP Webstore application contains some examples that use the VOM API. To allow visitors to list their orders on the MvpWebstore, you should use the `Sitecore.Ecommerce.MvpWebStore.Presenters.OrdersPresenter` class. It presents the orders page. It also handles the user interaction with this page.

To read all the orders that were created by a specific customer and display them in a page:

- The constructor of the `OrdersPresenters` class takes instances of `VisitorOrderRepositoryBase` and `IOrdersView` as initializing parameters and binds the `Load` handler to the `Load` event of the view:

```
private readonly VisitorOrderRepositoryBase orderRepository;
public OrdersPresenter(IOrdersView view, VisitorOrderRepositoryBase
orderRepository) : base(view)
{
  this.View.Load += this.Load;
  this.orderRepository = orderRepository;
}
```

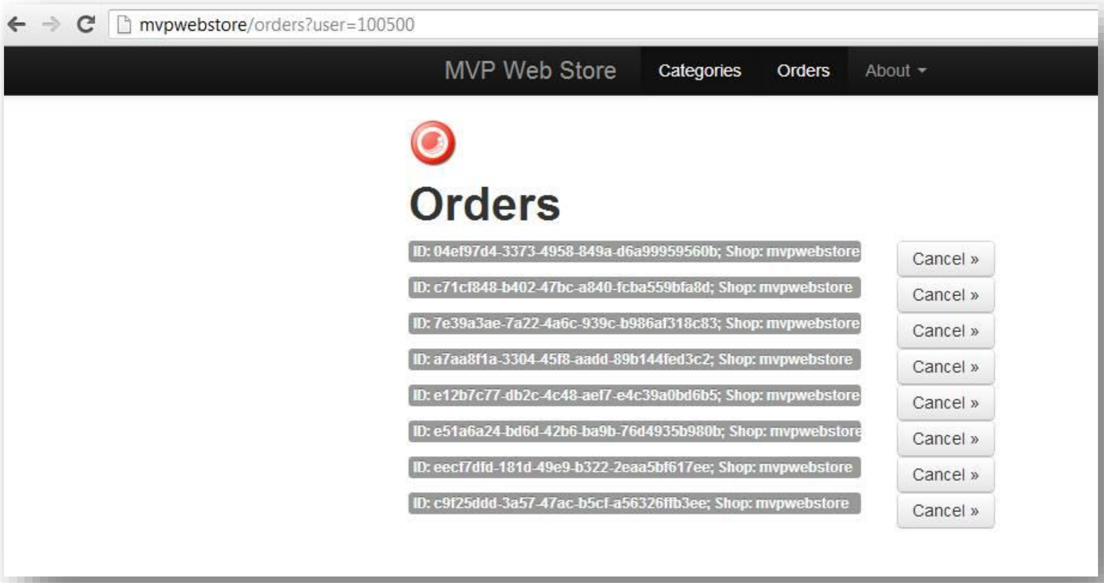
- When the user goes to the `~/orders?user=100500` page, the page parses the string value of the `user` parameter.

This is a sample. In a real implementation, you should not pass the ID in a query parameter because this could compromise security.

- In the page `Load` method, you should:
    - Cast the repository to the `IUserAware` interface and assign a value for the customer ID,
    - Retrieve all the orders that belong to this customer in the order repository,
    - Assign the `Orders` collection of the `ViewModel` to the retrieved orders.

```
private void Load(object sender, EventArgs e)
{
  var userId = this.HttpContext.Request.QueryString["user"];
  if (string.IsNullOrEmpty(userId))
  {
    return;
  }
  var aware = this.orderRepository as IUserAware;
  if (aware != null)
  {
    aware.CustomerId = userId;
  }
  var orders = this.orderRepository.GetAll(o => true).ToArray();
  this.View.Model.Orders = orders;
}
```

- The following image shows the result — it contains the order id, the shop context, and the link to the order cancelation page.



Orders

ID: 04ef97d4-3373-4958-849a-d6a99959560b; Shop: mvpwebstore
ID: c71cf848-b402-47bc-a840-fcba559bfa8d; Shop: mvpwebstore
ID: 7e39a3ae-7a22-4a6c-939c-b986af318c83; Shop: mvpwebstore
ID: a7aa8f1a-3304-45f8-aadd-89b144fed3c2; Shop: mvpwebstore
ID: e12b7c77-db2c-4c48-aef7-e4c39a0bd6b5; Shop: mvpwebstore
ID: e51a6a24-bd6d-42b6-ba9b-76d4935b980b; Shop: mvpwebstore
ID: eecf7dfd-181d-49e9-b322-2eaa5bf617ee; Shop: mvpwebstore
ID: c9f25ddd-3a57-47ac-b5cf-a56326ffb3ee; Shop: mvpwebstore

5.2.2 Using the Visitor API to Cancel an Order

In this example, we explain how to use the Visitor API to cancel an order if it is not already processed by the business logic of the webshop.

To cancel the order, you can use the `Sitecore.Ecommerce.MvpWebStore.Presenters.CancelOrderPresenter` class.

To use the visitor API to cancel an order:

- The constructor of the `CancelOrderPresenter` class takes instances of the `VisitorOrderRepositoryBase`, `ICancelOrderView` and `VisitorOrderProcessorBase` classes as initializing parameters and binds the `Load` handler to the `Load` event of the view:

```
private readonly VisitorOrderProcessorBase orderProcessor;
private readonly VisitorOrderRepositoryBase orderRepository;
public CancelOrderPresenter(ICancelOrderView view, VisitorOrderRepositoryBase
orderRepository, VisitorOrderProcessorBase orderProcessor) : base(view)
{
  this.View.Load += this.Load;
  this.orderRepository = orderRepository;
  this.orderProcessor = orderProcessor;
}
```

Note

In this example, we have created a simple page that accepts the IDs of the user and the order in the URL. In a real implementation, you should not pass the ID in a query parameter because this could compromise security.

- When the user goes to the `~/orders/cancelorder?id=zzz&user=100500` page, the page parses the value of the query string — the `user` and `id` parameters. If the order ID is not provided, the presenter stops to work. If the order ID is provided, it casts the repository to the `IUserAware` interface and sets the customer ID. The repository uses the order ID to retrieve the full order and then tries to use the `VisitorOrderProcessorBase` instance to cancel the order. At the end of the process, the presenter sets the label value of the result message:

```
private void Load(object sender, EventArgs e)
{
  try
  {
    var id = this.HttpContext.Request.QueryString["id"];
    if (string.IsNullOrEmpty(id))
    {
      this.View.Model.Result = Texts.TheOrderIdIsNotSpecified;
      return;
    }
    var userId = this.HttpContext.Request.QueryString["user"];
    var aware = this.orderRepository as IUserAware;
    if (aware != null)
    {
      aware.CustomerId = userId;
    }
    var order = this.orderRepository.GetAll(o => o.OrderId ==
    id).FirstOrDefault();
    this.orderProcessor.CancelOrder(order);
    this.View.Model.Result =
    string.Format(Texts.TheOrderHasBeenCancelledSuccessfully, id);
  }
  catch (Exception exception)
  {
    this.View.Model.Result = exception.Message;
  }
}
```

- In SES 2.0.0 and later, we store the state and sub-states as items in the content tree. However, to keep the MvpWebStore solution simple, we have not included them in the package.
- For simplicity, we use a custom implementation of the `VisitorOrderSecurity` and `ProcessingStrategy` classes that do not read the information from the content tree to check

whether or not the transition between the states is valid. They only read the `Order.State.Code` value to check that the order is not cancelled or closed.

- `VisitorOrderSecurity` applies security restrictions to the order and stops processing if the security restrictions are not satisfied.
- `ProcessingStrategy` changes the order state and performs other operations.

The implementations of these classes are registered in the `~/App_Config/MvpWebStore.Unity.config` file*:*

```
<unity xmlns="http://schemas.microsoft.com/practices/2010/unity">
  <alias alias="VisitorOrderSecurity"
  type="Sitecore.Ecommerce.Visitor.OrderManagement.VisitorOrderSecurity,
  Sitecore.Ecommerce.Visitor" />
  <alias alias="SampleOrderSecurity"
  type="Sitecore.Ecommerce.MvpWebStore.Domain.SampleOrderSecurity,
  Sitecore.Ecommerce.MvpWebStore" />
  <alias alias="ProcessingStrategy"
  type="Sitecore.Ecommerce.OrderManagement.ProcessingStrategy,
  Sitecore.Ecommerce.Core" />
  <alias alias="SampleOrderCancelationStrategy"
  type="Sitecore.Ecommerce.MvpWebStore.Domain.SampleOrderCancelationStrategy,
  Sitecore.Ecommerce.MvpWebStore" />
  <container>
    <register type="VisitorOrderSecurity" mapTo="SampleOrderSecurity" />
    <register type="ProcessingStrategy" mapTo="SampleOrderCancelationStrategy" />
  </container>
</unity>
```

- The following snippet implements the `SampleOrderSecurity` class to check if the order is in one of the following states:
  - *New*
  - *Open*
  - *InProcess*

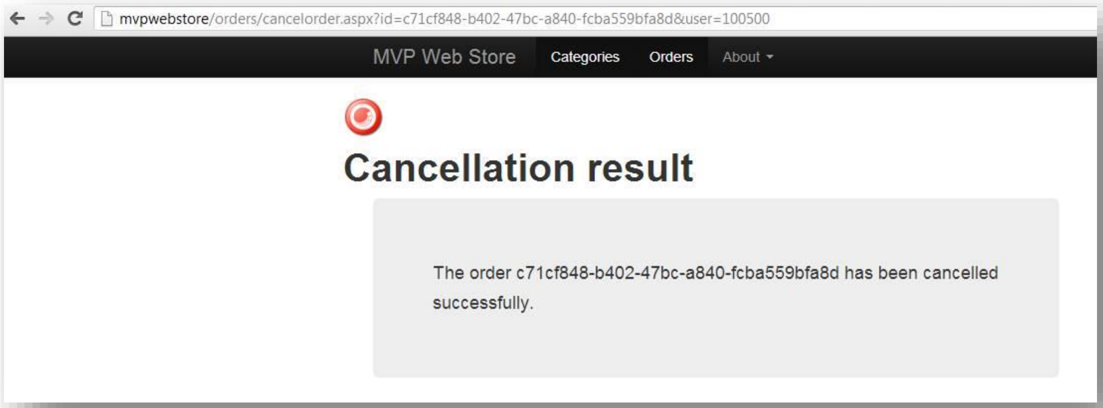
  If it returns *true*, you should allow the order to be cancelled. Otherwise, you should deny the cancelation.

```
// <summary>
// The overrided version of the VisitorOrderSecurity class.
// The 'CanCancel(Order):bool' method is simplified.
// It doesn't perform any sophisticated check like a default one and
// doesn't collaborate in any way with back-end.
// The decision whether to allow to cancel an order is taken when the State is not
// null
// and State.Code is within
// the following set: 'New', 'Open', 'InProcess'
// In opposite situation the cancellation is denied.
// The such is registered in the ~/App_Config/MvpWebStore.Unity.config.
// </summary>
public class SampleOrderSecurity : VisitorOrderSecurity
{
  // <summary>
  // Determines whether this instance can cancel the specified order.
  // </summary>
  // <param name="order">The order</param>
  // <returns>
  // <c>true</c> if this instance can cancel the specified order; otherwise,
  // <c>false</c>.
  // </returns>
  public override bool CanCancel(Order order)
  {
    if (order.State != null)
    {
      if ((order.State.Code == OrderStateCode.New) || (order.State.Code ==
      OrderStateCode.Open) || (order.State.Code == OrderStateCode.InProcess))
      {
        return true;
      }
    }
    return false;
  }
}
```

- You can then build the `SampleOrderCancellationStrategy` class:

```
    // <summary>
    // The simple implementation of the ProcessingStrategy abstract class.
    // It sets the State.Code of the provided order to the "Cancelled" value
    // without any collaboration with back-end.
    // </summary>
    public class SampleOrderCancelationStrategy : ProcessingStrategy
    {
      // <summary>
      // Gets or sets StateManager.
      // </summary>
      public virtual CoreOrderStateConfiguration StateManager { get; set; }

      // <summary>
      // Processes the order.
      // </summary>
      // <param name="order">The order.</param>
      public override void Process([NotNull] Order order)
      {
        Assert.ArgumentNotNull(order, "order");
        order.State.Code = OrderStateCode.Cancelled;
      }
    }
```

- If the order is successfully cancelled, you should see the following message:


mvpwebstore/orders/cancelorder.aspx?id=c71cf848-b402-47bc-a840-fcba559bfa8d&user=100500
MVP Web Store
Categories
Orders
About

Cancellation result

The order c71cf848-b402-47bc-a840-fcba559bfa8d has been cancelled successfully.

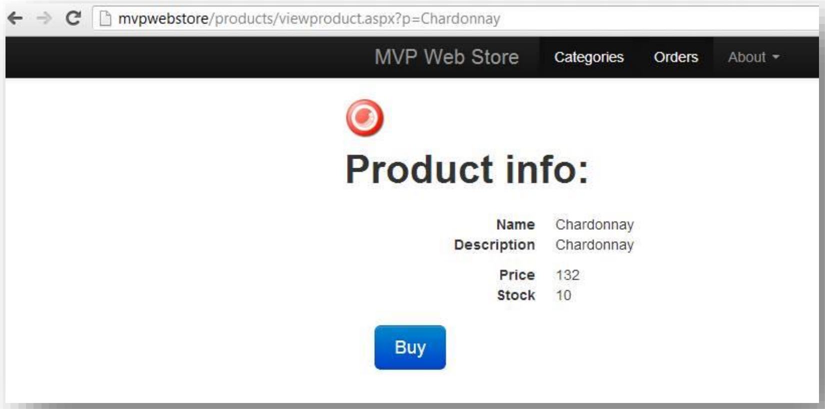
5.2.3 Using the Visitor API to Create an Order

To create an order, you must use the `Sitecore.Ecommerce.MvpWebStore.Presenters.ProductDetailsPresenter` class.

In SES 2.0.0 and later, we created an advanced checkout process. However, in MVPWebStore, you can use a single page with a **Buy** button. MVPWebStore contains instances of the `VisitorOrderRepositoryBase`, `IProductRepository`, `IProductStockManager`, `IProductPriceManager` and `VisitorOrderProcessorBase` classes as the required dependencies and binds the `Load` event handler and the `Buy` event handler of the view.

```
        public ProductDetailsPresenter(IProductDetailsView view, IProductRepository
        productRepository, IProductStockManager stockManager, IProductPriceManager
        priceManager, VisitorOrderRepositoryBase orderRepository) : base(view)
        {
          this.productRepository = productRepository;
          this.stockManager = stockManager;
          this.priceManager = priceManager;
          this.orderRepository = orderRepository;
          this.View.Load += this.Load;
          this.View.Buy += this.Buy;
        }
```

When it is loaded, the view is initialized with the *name*, *description*, *price*, *stock value* of the product and renders the **Buy** button:



If the stock value of the product is positive, the product is in stock and the order is created when you click the **Buy** button:

```
        private void Buy(object sender, EventArgs e)
        {
          ProductStockInfo productStockInfo = new ProductStockInfo { ProductCode =
          this.View.Model.Product.Code };
          ProductStock productStock = this.stockManager.GetStock(productStockInfo);
          // checking if the product is in stock
          if (productStock.Stock <= 0)
          {
            return;
          }
          // As a simple example, the value of the product stock is decremented
          this.stockManager.Update(productStockInfo, productStock.Stock - 1);
          // Initializing the order
          Order order = new Order { State = new State { Code = "New", Name = "New" },
          // Setting the shop context value of the order to mvpwebstore.
          ShopContext = "mvpwebstore",
          OrderId = Guid.NewGuid().ToString(),
          PricingCurrencyCode = "USD" };
          OrderLine orderLine = new OrderLine
          {
            Order = order,
            LineItem = new LineItem
            {
              Item = new Item { Code = this.View.Model.Product.Code },
              Price = new Price(new Amount(this.View.Model.Price, "USD")),
              Quantity = 1,
              TotalTaxAmount = new Amount(),
            }
          };
          order.OrderLines.Add(orderLine);
          // Setting the Supplier account ID to the predefined value "100500"
          order.BuyerCustomerParty = new CustomerParty { SupplierAssignedAccountID =
          Texts.MvpWebStoreCustomerId };
          // The order is sent to the visitor order repository that saves it in the
          // database.
          this.orderRepository.Create(order);
          this.HttpContext.Response.Redirect(this.HttpContext.Request.RawUrl);
        }
```

5.2.4 The Limitations of the Visitor API

The VOM API only allows customers to perform some high-level business operations with orders such as *Create*, *Read* and *Cancel*. The main aim is to give web shop visitors limited access to the orders stored in the database.

The most common operations are to:

- Read the existing orders that were created by the current customer.
- Create an order at the end of the checkout process.
- Cancel an order before completing the purchase if necessary.

The VOM API is an additional layer of abstraction on top of the Core Order Manager API. COM supports all CRUD operations, but VOM only supports Create, Read, and Cancel.

The VOM API is defined by the following class and interface definitions:

```
        public abstract class VisitorOrderProcessorBase
        {
          public abstract void CancelOrder(Order order);
        }

        public abstract class VisitorOrderRepositoryBase
        {
          public abstract void Create(Order order);
          public abstract IQueryable<Order> GetAll(Expression<Func<Order, bool>>  expression);
        }
```

The `VisitorOrderProcessor` class:

- Is the default implementation of the `VisitorOrderProcessorBase` abstract class.
- Implements the `VisitorOrderSecurity` class to check whether or not the order is in the appropriate state to be cancelled.
- Implements the `VisitorOrderCancelationStrategy` class to cancel the order.

The `VisitorOrderRepository.Create()` and `VisitorOrderProcessor.CancelOrder()` methods are marked with the custom `LogThis` attribute to make the IoC container intercept their work and log the creation and cancellation of the order in the `ActionLog` database.

```
        [LogThis(Constants.CreateOrderAction, Constants.UserLevel)]
        public override void Create(Order order)
        {
        }
        [LogThis("Cancel order", Constants.UserLevel)]
        public override void CancelOrder(Order order)
        {
        }
```

The `LogThis` attribute uses the `Sitecore.Ecommerce.Logging.LoggingHandler` and `Sitecore.Ecommerce.Core` classes. This class contains a reference to the Logger and calls it with the provided parameters. You can use the `unity.config` file to configure the interception:

```
        <register type="VisitorOrderManager" mapTo="DefaultVisitorOrderManager">
          <lifetime type="hierarchical" />
          <interceptor type="VirtualMethodInterceptor" />
          <policyInjection />
        </register>
        <register type="VisitorOrderProcessorBase" mapTo="VisitorOrderProcessor">
          <lifetime type="hierarchical" />
          <interceptor type="VirtualMethodInterceptor" />
          <policyInjection />
        </register>
```

Note
If you use the default implementation, you should not worry about logging.

You cannot use the Visitor API to remove an order. The only change that you can make with VOM is to change the `Order.State` to `Cancelled`. You must use the `VisitorOrderProcessor.CancelOrder` method to cancel the order.

You must use the `CancelOrder` method to:

- Statically provide a list of orders on the page or use the `XmlHttpRequest XHR` object and send the list in JSON format from the server to the client.

  `XHR` is a JavaScript object that is used to send asynchronous requests from the client code to the server.

- Create the custom checkout process. In the last stage, you should use the accumulated information to create the order.
- Cancel an existing order. For example, on the history page.

5.3 Using the Merchant Order Manager API

The Merchant Order Manager (MOM) API contains the business logic that is used by the OM web application to manage the orders.

For example, you can use the MOM API is used to:

- Create a new order.
- Validate an order.
- Update the order state.

5.3.1 Using the MOM API to Create an Order

You can use the MOM API to configure the UI of your OM application:

- Create a custom action class that contains the business logic.
- Create an action panel that is bundled with this class.

Creating a Custom Action Class

To implement the business logic for creating an order, you must create a custom action class:

1. Create a class that is based on the `Sitecore.Web.UI.WebControls.Actions` class and call it `CreateOrderAction`.

```
namespace Sitecore.Ecommerce.Apps.OrderManagement.Views
{
  using Sitecore.Web.UI.WebControls;

  /// <summary>
  /// The create order action.
  /// </summary>
  public class CreateOrderAction : Action
  {
    /// <summary>
    /// Executes the specified context.
    /// </summary>
    /// <param name="context">The context.</param>
    public override void Execute([CanBeNull] ActionContext context)
    {
     // To create an order using the MOM API, insert custom logic here
    }

    /// <summary>
    /// Queries the state.
    /// </summary>
    /// <param name="context">The context.</param>
    /// <returns>
    /// The state.
    /// </returns>
    public override ElementState QueryState([NotNull] ActionContext context)
    {
      return ElementState.Enabled;
    }
  }
}
```

2. To hide the action panel from users who are not members of the *Order Processor* role, override the `QueryState` method to return the *hidden* state.

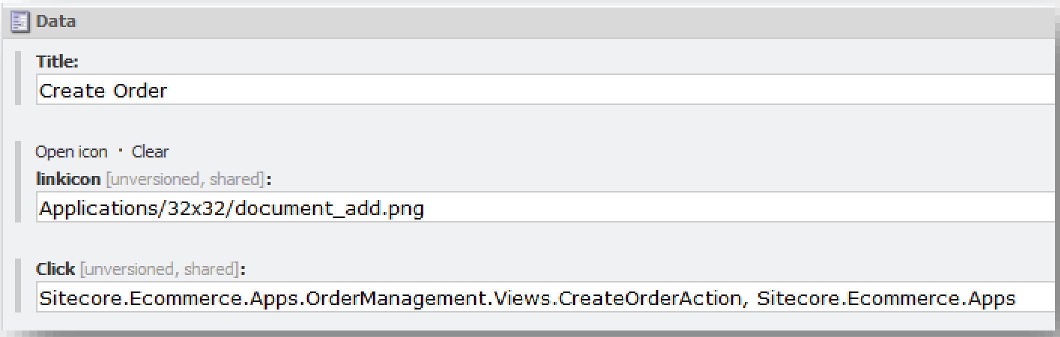
3. Use the following steps to override the `Execute` method with your custom logic:
   - Use the `DefaultOrderFactory` class to create an order and set the order State to `Open`.
   - Use the `OrderManager.save` method to save the order.
   - Redirect to the order details page with the Order ID of the new order in the query string.

```
//Create an order using DefaultOrderFactory.
var order = this.orderFactory.Create();

//Change order state from New to Open.
order.State = new State { Code = OrderStateCode.Open };

//Save order with MerchantOrderManager.
this.orderManager.Save(order);

//Redirect to order details page.
this.view.RedirectToOrderDetails(order.OrderId);
```

Create an Action Panel

The previous section describes how to create the custom action class that contains the business logic. The following a procedure describes how to create an action panel that calls the custom action class:

1. In the *Action Panels* folder, create an item and call it *Order Manager Actions* `sitecore\content\system\Modules\SPEAK\Order Manager\Repositories\Action Panels`
2. In the *Order Manager Actions* folder, add the create order action that you just created and enter values in the *Title*, *linkicon,* and *Click* fields. In the **Click** field, enter the Create Order Action class name, for example `Sitecore.Ecommerce.Apps.OrderManagement.Views.CreateOrderAction, Sitecore.Ecommerce.Apps.`



Title:
Create Order

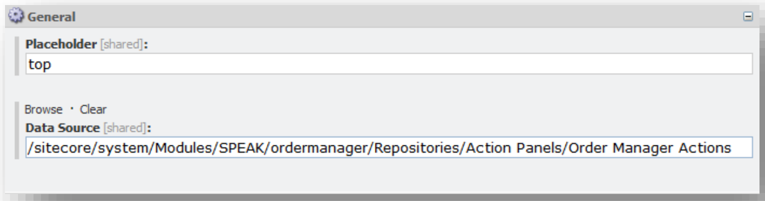
linkicon [unversioned, shared]:
Applications/32x32/document_add.png

Click [unversioned, shared]:
Sitecore.Ecommerce.Apps.OrderManagement.Views.CreateOrderAction, Sitecore.Ecommerce.Apps

3. Navigate to the *Order Manager* root item `sitecore\content\system\Modules\SPEAK\Order Manager`.
4. In the Content Editor, click the **Presentation** group and then in the **Layout** group, click **Details** and the **Layout Details** opens.

5. In the **Layout Details** dialog box, select the placeholder called *top* and in the **Data Source** field, enter the path to the *Order Manager Actions* item that you just created.

   You must assign the custom action class to the **Create Order** action item.


General

Placeholder [shared]:
top

Browse · Clear
Data Source [shared]:
/sitecore/system/Modules/SPEAK/ordermanager/Repositories/Action Panels/Order Manager Actions

5.3.2 Using the MOM API to validate Orders

SES contains a validation mechanism to avoid malicious behaviors, such as trying to create invalid orders. For example, a user may unexpectedly order excessive quantities of items or place orders in an unlikely fast pace, suggesting a denial of service attack (DOS). Manual checking of orders is time consuming and cumbersome.

The `orderCreated` pipeline is the right place to insert business logic for automatic order validation. This pipeline performs additional operations as part of the order creation process.

These additional operations are used to:

- Send order confirmation by mail to the customer. To confirm the order, the `NotifyCustomer` processor sends a notification email to the user. This processor is not be explained in this topic.
- Perform initial order validation or fraud checks

  By default, the `CheckProductQuantity` processor checks if the product quantity of any order line is greater than the declared maximum quantity. If yes, the order state is set to *Suspicious* with sub-state *Product Quantity* indicating the reason, see the section *Setting the Order State to Suspicious*.

SES uses the following validation mechanism:

1. After the order is created, the `orderCreated` pipeline starts. Prepare the order for manual inspection and fulfilment by the order manager.
2. According to the business logic of each pipeline processor, the order is validated. If the order is found suspicious, then it is the responsibility of the individual processor to mark the order as suspicious in the pipeline arguments and set the sub-states accordingly to indicate the nature of the suspicion, see the section *Setting the Order State to Suspicious*.
3. If there are no validation issues encountered by the previous processors, the order state is set to *Open* by the `TryOpenOrder` processor which is typically the last processor. If a suspicious activity has been encountered, the pipeline arguments take the suspicious order state and sub-state information and the order state is set to *Suspicious* and the sub-states are set accordingly.

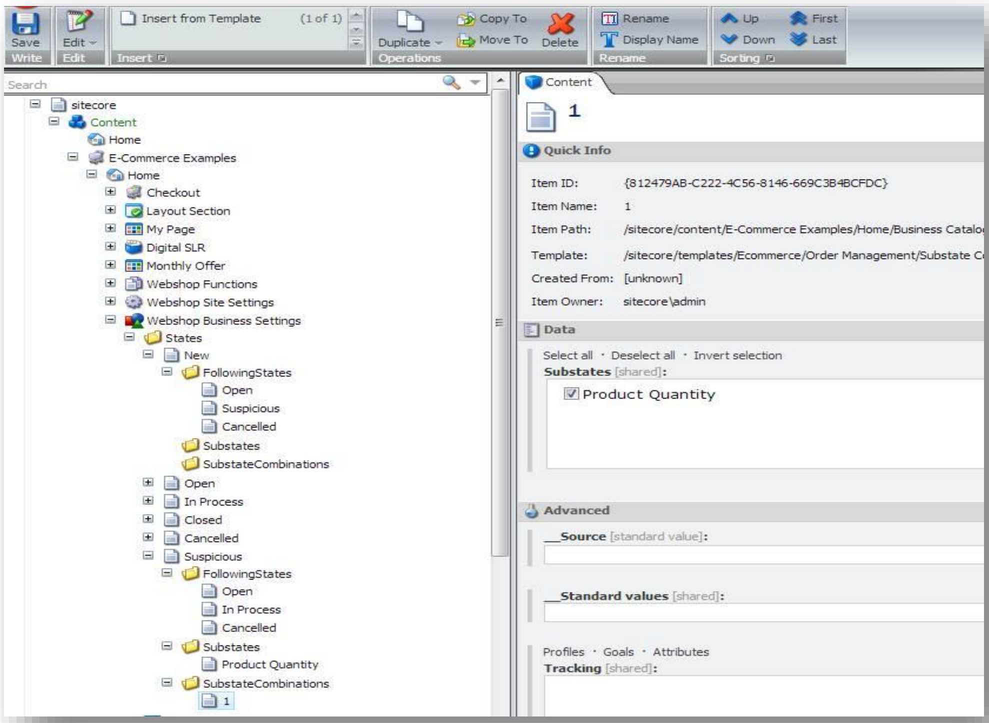
Note
The validation processors must not abort the pipeline as the order state is not set until the last processor `TryOpenOrder` is executed. Further processors might also find more evidence of suspicious behavior and set further sub-states accordingly.

Setting the Order State to Suspicious

SES 2.2 introduces the *Suspicious* state. By default, it only contains the *Product Quantity* sub-state, but you should extend the list of sub-states when you add further validation checks. In the *Suspicious* state, each sub-state represents the reason for setting the order state to *Suspicious*.

The predecessor of the *Suspicious* state is the *New* state, and the allowed successors are *Open*, *In Process* and *Cancelled*. See the *SES Configuration Guide*, the section *Configuring Sates*.

To specify the reason that the order is in the *Suspicious* state, the substate combinations are configured so that at least one *Suspicious* sub-state must be set in code or selected in the OM application



1

Item ID: {812479AB-C222-4C56-8146-669C3B4BCFDC}

Item Name: 1

Item Path: /sitecore/content/E-Commerce Examples/Home/Business Catalo

Template: /sitecore/templates/Ecommerce/Order Management/Substate C

Created From: [unknown]

Item Owner: sitecore\admin

Substates [shared]:

☑ Product Quantity

__Source [standard value]:

__Standard values [shared]:

Tracking [shared]:

Note
Processing a suspicious orders is restricted. Everyone who has access to the Order Manager application can see the suspicious order details, but editing is not allowed. Members of the Order Manager Administrators and the Order Manager Processing roles can change the order state to *Open*, *In Process* or *Cancelled*, but cannot change it while it is in the *Suspicious* state. An order that is considered valid by the Order Manager, must be changed to the state *Open* or *InProcess* and then it can be edited and processed.

The following are examples of a *suspicious* order that is examined in details in the following sections:

- Same Visitor within a Predefined Time Interval
- Quantity of the Order is Greater than a Certain Predefined Value

Same Visitor within a Predefined Time Interval

You can create a custom order validation processor which sets the order to *suspicious* if it is created by the same visitor within ten seconds.

The following code snippet describes the `CheckOrderProcessorBase` class. This class is provided by SES to help create validation processors that check the order:

```
namespace Sitecore.Ecommerce.Merchant.Pipelines.OrderCreated
{
  using System.Collections.Generic;
  using System.Linq;
  using Sitecore.Diagnostics;
  using Sitecore.Ecommerce.Merchant.OrderManagement;
  using Sitecore.Ecommerce.OrderManagement;
  using Sitecore.Ecommerce.OrderManagement.Orders;
  using Sitecore.Pipelines;

  /// <summary>
  /// Defines the CheckOrderProcessor type.
  /// </summary>
  public abstract class CheckOrderProcessorBase
  {
    /// <summary>
    /// Initializes an instance of the <see cref="CheckOrderProcessorBase" />
    /// class.
    /// </summary>
    protected CheckOrderProcessorBase()
    {
      this.OrderManager = Context.Entity.Resolve<MerchantOrderManager>();
    }

    /// <summary>
    /// Gets the order manager.
    /// </summary>
    /// <value>The order manager.</value>
    [NotNull]
    public virtual MerchantOrderManager OrderManager { get; private set; }

    /// <summary>
    /// Gets the order.
    /// </summary>
    /// <param name="args">The args.</param>
    /// <returns>The order.</returns>
    [NotNull]
    protected virtual Order GetOrder([NotNull] PipelineArgs args)
    {
      var orderNumber = args.CustomData["orderNumber"] as string;
      Assert.IsNotNull(orderNumber, "OrderNumber cannot be null.");

      var order = this.OrderManager.GetOrder(orderNumber);
      Assert.IsNotNull(order, "Order cannot be null.");

      return order;
    }

    /// <summary>
    /// Gets the suspicious sub states.
    /// </summary>
    /// <param name="args">The args.</param>
    /// <returns>The list of suspicious sub-states.</returns>
    [NotNull]
    protected IEnumerable<string> GetSuspiciousSubStates(PipelineArgs args)
    {
      return args.CustomData[OrderStateCode.Suspicious] as HashSet<string> ?? new
      HashSet<string>();
    }

    /// <summary>
    /// Marks the order as suspicious.
    /// </summary>
    /// <param name="args">The args.</param>
    /// <param name="suspiciousSubstateCode">The suspicious sub-state code.</param>
```

```
            protected virtual void MarkOrderAsSuspicious(PipelineArgs args, string
            suspiciousSubstateCode)
            {
              HashSet<string> hashSet = args.CustomData[OrderStateCode.Suspicious] as
              HashSet<string>;
              if (hashSet == null)
              {
                hashSet = new HashSet<string>();
                args.CustomData[OrderStateCode.Suspicious] = hashSet;
              }

              hashSet.Add(suspiciousSubstateCode);
            }

            /// <summary>
            /// Determines whether the specified args is suspicious.
            /// </summary>
            /// <param name="args">The args.</param>
            /// <returns>
            ///   <c>true</c> if the specified args is suspicious; otherwise, <c>false</c>.
            /// </returns>
            protected virtual bool IsSuspicious(PipelineArgs args)
            {
              HashSet<string> hashSet = args.CustomData[OrderStateCode.Suspicious] as
              HashSet<string>;

              return hashSet != null && hashSet.Any();
            }
          }
        }
```

To implement an order validation processor that sets the order to *suspicious*:

1. Create a class that inherits from the `CheckOrderProcessorBase` class and implement the `Process` method:

```
        namespace Ses.Samples.Merchant.Pipelines.OrderCreated
        {
          using System;
          using System.Linq;
          using Sitecore.Ecommerce.Merchant.Pipelines.OrderCreated;
          using Sitecore.Pipelines;

          // Create a CheckOrderFrequency processor to validate if order created with a proper
          // frequency.
          public class CheckOrderFrequency : CheckOrderProcessorBase
          {
            public TimeSpan Frequency { get; set; }

            public void Process(PipelineArgs args)
            {
              // Read the created order using order number stored in pipeline args.
              var order = this.GetOrder(args);

              // Read the customer ID from the new order.
              var customer = order.BuyerCustomerParty.SupplierAssignedAccountID;

              // Read the allowed order creating frequency which is set in pipeline processor
              // property and determine maximum allowed date.
              var recentAllowedOrderDate = DateTime.Now - this.Frequency;

              // Check if the customer has already placed an order recently and
              // determine if the order is suspicious or not.
              Var frequentOrders = this.OrderManager.GetOrders().Where(o =>
              o.BuyerCustomerParty.SupplierAssignedAccountID == customer &&
              o.OrderId != order.OrderId &&
              o.IssueDate >= recentAllowedOrderDate);

              // Mark the order as suspicious if there are some orders found.
              if (frequentOrders.Any())
              {
                this.MarkOrderAsSuspicious(args, "Order Frequency");
              }
            }
          }
        }
```

2. In the `Sitecore.Ecommerce.config` file, in the `orderCreated` section, create a pipeline processor entry as part of the `OrderCreated` pipeline before the `TryOpenOrder` processor:

```
<orderCreated>
 ...
  <processor
  type="Ses.Samples.Merchant.Pipelines.OrderCreated.CheckOrderFrequency,
  Sitecore.Ecommerce.Tests.Integration">
   <Frequency>00:00:10</Frequency>
  </processor>
...
</orderCreated>
```

Quantity of the Order is Greater than a Certain Predefined Value

To implement an order validation pipeline that sets the order to *suspicious* if the order line quantity is greater than a certain predefined value:

1. Create a class that inherits from the `CheckOrderProcessorBase` class and implement the `Process` method:

```
namespace Sitecore.Ecommerce.Merchant.Pipelines.OrderCreated
{
  using System.Linq;
  using Sitecore.Diagnostics;
  using Sitecore.Ecommerce.OrderManagement;
  using Sitecore.Pipelines;

  /// <summary>
  /// The product quantity validator.
  /// </summary>
  public class CheckProductQuantity : CheckOrderProcessorBase
  {
    /// <summary>
    /// Gets or sets the suspicious quantity.
    /// </summary>
    /// <value>
    /// The suspicious quantity.
    /// </value>
    public decimal MaximumQuantity { get; set; }

    /// <summary>
    /// Runs the processor.
    /// </summary>
    /// <param name="args">The arguments.</param>
    public virtual void Process([NotNull] PipelineArgs args)
    {
      Assert.ArgumentNotNull(args, "args");
      var order = this.GetOrder(args);
      foreach (var orderLine in order.OrderLines.Where(orderLine =>
      orderLine.LineItem.Quantity > this.MaximumQuantity))
      {
        this.MarkOrderAsSuspicious(args, OrderStateCode.SuspiciousProductQuantity);
      }
    }
  }
}
```

2. In the `Sitecore.Ecommerce.config` file, in the `orderCreated` section, create a pipeline processor entry as part of the `OrderCreated` pipeline:

```
        <orderCreated>
        ...
           <processor
           type="Sitecore.Ecommerce.Merchant.Pipelines.OrderCreated.CheckProductQuantity,
           Sitecore.Ecommerce.Merchant">
              <MaximumQuantity>100</MaximumQuantity>
           </processor>
        ...
        </orderCreated>
```

Setting the Order State to Open

If the order is not suspicious, the validation pipeline moves the order from the *New* state to the *Open* state.

You can use the following steps to replace the default implementation of the `TryOpenOrder` class:

1. Create a class that inherits from the `CheckOrderProcessorBase` class that is mentioned in the *Using the MOM API to validate* Orders

SES contains a validation mechanism to avoid malicious behaviors, such as trying to create invalid orders. For example, a user may unexpectedly order excessive quantities of items or place orders in an unlikely fast pace, suggesting a denial of service attack (DOS). Manual checking of orders is time consuming and cumbersome.

The `orderCreated` pipeline is the right place to insert business logic for automatic order validation. This pipeline performs additional operations as part of the order creation process.

These additional operations are used to:

- Send order confirmation by mail to the customer. To confirm the order, the `NotifyCustomer` processor sends a notification email to the user. This processor is not be explained in this topic.
- Perform initial order validation or fraud checks

  By default, the `CheckProductQuantity` processor checks if the product quantity of any order line is greater than the declared maximum quantity. If yes, the order state is set to *Suspicious* with sub-state *Product Quantity* indicating the reason, see the section *Setting the Order State to Suspicious*.

SES uses the following validation mechanism:

2. After the order is created, the `orderCreated` pipeline starts. Prepare the order for manual inspection and fulfilment by the order manager.
3. According to the business logic of each pipeline processor, the order is validated. If the order is found suspicious, then it is the responsibility of the individual processor to mark the order as suspicious in the pipeline arguments and set the sub-states accordingly to indicate the nature of the suspicion, see the section *Setting the Order State to Suspicious*.
4. If there are no validation issues encountered by the previous processors, the order state is set to *Open* by the `TryOpenOrder` processor which is typically the last processor. If a suspicious activity has been encountered, the pipeline arguments take the suspicious order state and sub-state information and the order state is set to *Suspicious* and the sub-states are set accordingly.

Note
The validation processors must not abort the pipeline as the order state is not set until the last processor `TryOpenOrder` is executed. Further processors might also find more evidence of suspicious behavior and set further sub-states accordingly.

5. Setting the Order State to Suspicious section:

```
namespace Sitecore.Ecommerce.Merchant.Pipelines.OrderCreated
{
  using System.Linq;
  using Sitecore.Diagnostics;
  using Sitecore.Ecommerce.Merchant.OrderManagement;
  using Sitecore.Ecommerce.OrderManagement;
  using Sitecore.Pipelines;

  /// <summary>
  /// The order validator.
  /// </summary>
  public class TryOpenOrder : CheckOrderProcessorBase
  {
    /// <summary>
    /// Gets or sets the order manager.
    /// </summary>
    /// <value>
    /// The order manager.
    /// </value>
    private readonly MerchantOrderManager orderManager;

    /// <summary>
    /// Initializes a new instance of the <see cref="TryOpenOrder" /> class.
    /// </summary>
    public TryOpenOrder()
    {
      this.orderManager = Context.Entity.Resolve<MerchantOrderManager>();
    }

    /// <summary>
    /// Runs the processor.
    /// </summary>
    /// <param name="args">The arguments.</param>
    public virtual void Process([NotNull] PipelineArgs args)
    {
      Assert.ArgumentNotNull(args, "args");
      var order = this.GetOrder(args);
      var states = this.orderManager.StateConfiguration.GetStates();
      if (!this.IsSuspicious(args))
      {
        order.State = states.Single(s => s.Code == OrderStateCode.Open);
      }
      else
      {
        var suspicionState = states.Single(s => s.Code == OrderStateCode.Suspicious);
        foreach (var suspicionSubStateCode in this.GetSuspiciousSubStates(args))
        {
          suspicionState.Substates.Single(s => s.Code == suspicionSubStateCode).Active
          = true;
        }
        order.State = suspicionState;
      }
      this.orderManager.Save(order);
    }
  }
}
```

6. In the `Sitecore.Ecommerce.config` file, in the `orderCreated` pipeline, the processor must be configured as the last processor after all the validation processors has been executed:

```
<orderCreated>
 ...
  <processor
  type="Sitecore.Ecommerce.Merchant.Pipelines.OrderCreated.TryOpenOrder,
  Sitecore.Ecommerce.Merchant"/>
</orderCreated>
```

5.3.3 Getting the Best-Selling Products

Besides the basic CRUD operations, the Merchant API can be used to query the orders.

To search for the best-selling products, use LINQ to:

1. Get all the orders.
2. Use the orders to get all the order lines.
3. Group the order lines by the product code and calculate the product quantity of each group.
4. Sort the groups by the calculated quantity in descending order.
5. Specify how many products of highest calculated quantity to be returned.

The following code snippet describes how to implement this:

```
// Specify the webshop to get the products from.
const string WebShopName = "example";

// Setup the environment for the webshop.
using (new SiteContextSwitcher(SiteContextFactory.GetSiteContext(WebShopName)))
{
// Get instance of MerchantOrderManager from IoCContainer.
MerchantOrderManager merchantOrderManager =
Context.Entity.Resolve<MerchantOrderManager>();

// Get instance of IProductRepository from IoCContainer.
IProductRepository productRepository = Context.Entity.Resolve<IProductRepository>();
// Defines number of top products to be selected.
const int SizeOfSelection = 5;
// Order product codes by total quantity and select products by ordered product codes.
IEnumerable<ProductBaseData> resultingProducts =
// Get orders first.
merchantOrderManager.GetOrders()
//Select all order lines from the orders.
SelectMany(order => order.OrderLines)
// Group order lines by product code and calculate total quantity for each of the
// group.
.GroupBy(orderLine => orderLine.LineItem.Item.Code, (productCode, orderLines) => new {
productCode, totalQuantity = orderLines.Sum(orderLine => orderLine.LineItem.Quantity)
})
// Order groups by calculated total quantity.
.OrderByDescending(pair => pair.totalQuantity)
// Take only the records we need
.Take(SizeOfSelection)
// Force query execution on subsequent operations
.AsEnumerable()
// and transform them to sequence of products.
.Select(pair => productRepository.Get<ProductBaseData>(pair.productCode));
}
```

5.3.4 Getting the Best Customers for a Webstore

To search for the best customers for the web store, use LINQ to:

1. Get all the orders.
2. Group the orders by the assigned account ID of the supplier and calculate the total price of the purchases for each group. It is assumed that all prices are of the same currency, otherwise the prices must be converted to a common currency.
3. Sort the groups by the calculated the total price in descending order.
4. Specify how many orders of highest calculated total price to be returned.

The following code snippet describes how to implement this:

```
        // Specify the webshop to get the orders from.
        const string WebShopName = "example";

        // Setup the environment for the webshop.
        using (new SiteContextSwitcher(SiteContextFactory.GetSiteContext(WebShopName)))
        {
        // Get instance of MerchantOrderManager from IoCContainer.
        MerchantOrderManager merchantOrderManager =
        Context.Entity.Resolve<MerchantOrderManager>();
        // Get instance of ICustomerManager from IoCContainer.
        ICustomerManager<CustomerInfo> customerManager =
        Context.Entity.Resolve<ICustomerManager<CustomerInfo>>();
        // Defines number of top customers to be selected.
        const int SizeOfSelection = 5;
        // Order customers by total price of purchased products and select CustomerInfo.
        IEnumerable<CustomerInfo> topBuyers =
        // Get orders first.
        merchantOrderManager.GetOrders()
        // Group orders by SupplierAssignedAccountID, and calculate total price of the
        // purchases
        // for each group. It is assumed that all prices are of the same currency, otherwise
        // the prices must be converted to some common currency.
        GroupBy(order => order.BuyerCustomerParty.SupplierAssignedAccountID, (customerId,
        orders) => new { customerId, totalPrice = orders.Sum(order =>
        order.AnticipatedMonetaryTotal.PayableAmount.Value) })
        // Order groups by calculated total price.
        .OrderByDescending(pair => pair.totalPrice)
        // Take only the records we need.
        .Take(SizeOfSelection)
        // Force query execution on subsequent operations
        .AsEnumerable()
        // and transform them to sequence of objects providing customer information.
        .Select(pair => customerManager.GetCustomerInfo(pair.customerId));
        }
```