Chapter 3

Modernization with the Spark Framework

Modifying the UI

Purpose

This chapter will attempt to outline various forms of modifications that can be made to the front-end using the current wizard pattern, in conjunction with KendoUI. Each step will gradually build from minor changes to a full screen and then a complete template. But first, we need to discuss what is being built with this tool.

Arriving at Modern Web Development

The traditional architecture of a web site was (and mostly still is) the use of distinct pages for your site. Each page has a specific purpose, and navigating to a new URL means loading (requesting, fetching, returning) a separate and distinct HTML document from the web server. The idea has been modified over the years to include more dynamic output via ASP, PHP, SpeedScript & WebSpeed, and other scripting languages. So, for a "current" dynamic website some context is passed around, and each requested page can use that snippet of info to determine the content to fill the page upon request. This was good, but then we all discovered AJAX and blew that whole idea out of the water…

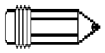
With the advent of AJAX and "Rich Internet Applications" it became more possible to construct a single, mostly-static page that now made partial requests for data as needed. Slowly this produced more problems that needed to be addressed: for instance, if you could request data without reloading the page then that meant you would need to potentially redraw or alter the existing page based on new values. This gave rise to toolkits like jQuery that provide DOM selectors and methods to manipulate elements on the page. It also meant that we needed to maintain page-level context, to know when some piece of data existed and the content needed to be changed—now we've introduced data binding and frameworks such as AngularJS to deal with this need. And if we can manipulate the elements what's to stop us from building pre-constructed widgets? So now we have libraries such as KendoUI, and together these give us a potentially new way of building applications.

The Rise of Single-Page Applications

It should be strongly implied as to where this path leads. The idea behind a single-page application (hereby shortened to SPA) is to provide a single HTML document [the page] to act as the basis for our application, through the use of the various toolkits and frameworks above. In this case, the "page" is more like a shell–it contains only what is necessary to get our application up and running, then it's the result of data requests to tell that application what to do next. Let's walk through a quick scenario:

1. End-user requests a URL on your server, specifically the index page of the application.
2. The HTML document (page) loads files for jQuery, KendoUI, custom CSS, etc.
3. Some kind of session logic creates/reestablishes the identity of the requesting user.
4. At some point, it requests code to apply some initial elements such as navigation.
5. The end-user selects some menu item, and specific content is loaded for that option.
6. Data is requested and displayed to the user within the newly created widgets.
7. User continues requesting new content without reloading the original page.

That's pretty much the setup, but we'll need to examine this concept of additional content. To do that, we need to introduce the concept of a Model-View-Controller pattern that will keep our code segregated by feature and more maintainable.


NOTE

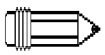
You may hear another term "*Single Point of Authentication*" that is also abbreviated as "SPA". This is a reference to what's also called a Single Sign-On (SSO) and represents a service that authenticates users in 1 location but is utilized for multiple services.

"But wait!" you say. **"What if I want to use KUIB or another framework for my user interface?"**

The good news is that PMFO introduces an SPA approach that is similar to Angular, KUIB, or just about any other modern framework. The following features are present in our framework and could be extended or built using nearly any other current framework:

1. Establish a dedicated authentication page for logins.
2. Establish a "landing" page for authenticated users.
3. Provide a basic navigation system for loading content.
4. The concept of load-on-demand content (views, modals, etc.).
5. UI state-management and cleanup of unused artifacts.

We will explore in more detail the concepts that go into each of these features, beginning with the overall idea of a Model-View-Controller (MVC) pattern. Once you begin to see the patterns and learn the common names of each piece, you will be able to adapt quickly to the same concepts found in other frameworks.


NOTE

Eventually the WebWizard's features for generating UI content may be deprecated in favor of the KUIB, though portions may still remain only for generating **BusinessEntity** classes from custom templates. In the end, use of any solution that works best and assists you comfortably in creating your application is up to you!

Introducing an MVC Pattern

Let's first list and define a few helpful SPA terms going forward:

- **Page**: The top-level structure for the application, typically a static HTML file.
- **Screen**: An HTML/JS pairing (lacks HTML and BODY tags) loaded on demand.
- **View**: The HTML portion of a screen in the MVC pattern, describing the UI.
- **Model**: A description of schema or data as it will be used within a screen's view.
- **Controller**: The "glue" that connects the model to the view, and initializes a screen.

We have already discussed the role of the "page" in SPA, so let's focus on the concept of "screens". These are the pieces of content that are requested based on user interactions within the new application. Since we already have an HTML document loaded and present in the browser, these screens only need to contain the pieces of HTML that belong to a certain feature (such as the Customer Read-Only Grid that we created earlier). In terms of deliverable objects, a screen may be 1 or more physical file served on demand. If everything (HTML + JavaScript) is delivered in a single page, this is typically referred to as monolithic code. There are several reasons why this may be useful, the least of which is convenience in terms of development. However, the primary drawback is that the file can be extremely large if there are many HTML elements and a lot of JS logic present. To that end, the MVC pattern keeps key features separate so that they provide distinct purposes and enhance maintainability.

Within our MVC pattern the view is simply an HTML document that only describes the elements for the screen at hand. It does not contain any HTML, HEAD, BODY, SCRIPT, STYLE or similar tags. The controller is a JavaScript file, and contains the necessary code to create an instance of a model, or otherwise load a model on demand for the view and then bind it. In our framework the model is typically an instance of a Progress JSDO (JavaScript Data Object) that is bound to 1 or more widgets within the view and provides the data requests (CRUD + Invoke) to a REST back-end. Speaking of binding, let's talk briefly about this concept and how it is used in an SPA.

In some SPA's the predominant binding agent is a framework such as AngularJS that requires a similar setup with a view and controller. These are bound together in such a way that each element in the view has a corresponding JavaScript variable—that when changed from within the code will be reflected automatically within the view on screen. In the PMFO MVC pattern we luckily do not need to worry about AngularJS, as KendoUI already provides what is called an "observable object" which accomplishes this same data binding action. Additionally, a simple JavaScript object and not another framework provide our controller. And finally, the "page" portion of the SPA already has the necessary logic to load and launch screens by downloading the necessary files and executing them as needed.

In JavaScript terms, the controller used in the following MVC pattern is an IIFE: Immediately Invokable Function Expression. Each controller instance is created as a variable in the global namespace, as the result of executing the function expression. The return of this function is an object that contains all of the necessary properties and/or methods that we wish to expose for the controller. Within this function (or "closure") everything is scoped locally. This allows us to create as many internal variables as we wish, and they will only apply to this controller—they will not pollute the global space nor can they be confused with variables from any other controllers. It also allows us a way to immediately access and destroy this instance if we want to perform any DOM cleanup, which is something that is always needed but not always accounted for in an application (at least not initially).

Modifying Generated Output

Use of templates should get you within striking distance of your target, but it's always expected that with each generation of a screen you may need to provide customizations. So with the various components of the MVC pattern described we should now know what to modify if we wish to alter the output from the wizard. For this example, we will modify our existing screen to add another search field. The best way to work is from the UI first, then modify the controller as needed.

1. Navigate to **PASOEContent/WEB-INF/static/app/views/** in PDSOE and open the **CustomerGridSPA.html**
2. Look for the **input** field for the "**Name**" field, as we configured via the wizard.
    a. Copy and paste the entire input tag to the next line after the field.
    b. Change the name property to "`Balance`"
    c. Change the data-bind option to "`value:params.Balance`"
3. Next open the **CustomerGridSPA.js** file
    a. In the function expression is a variable for the primaryVM (aka. the primary View-Model). This is a Kendo observable which contains all of the properties to be data-bound to the UI. We need to add a new property to the params object here called "`Balance`" with a default value of 0.
    b. In the **doSearch** method below that object, look for the assignment of the **filter** variable to an array (referencing "`searchField1`"). We need to add more criteria by appending a new object to this array as follows:

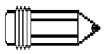
```
{
        field: "Balance",
        operator: "gt",
        value: parseFloat(params.Balance)
}
```

    c. Save your files and perform a **Publish** to the PAS instance.
4. In your browser, refresh the tab to update the application code with our changes.
    a. Test by filtering by name with a value of "**aaa**". This should return 1 record, and so far, is identical to our original filter criteria.
    b. The returned record should have a balance of 24,389.03, so providing a new search value of "**25,000**" for the balance should cause our query to exclude this result (as we're searching for customers that begin with "*aaa*" and have a balance greater than 25k).
        i. Note: We used the **parseFloat()** method to ensure our Balance value is typecast to the correct datatype. When passing JSON as parameters, we need to be certain that the datatypes match what is intended on the other side as field values will be converted according to their ABL datatype.
    c. If you search with these new criteria and no records are returned, then congratulations you've confirmed that your new criteria are in effect! Play around with some other values if you wish to further test this new criteria option.

Modifying Existing Templates

Next, we can take a quick look at the templates themselves, and discuss how you can modify them to produce consistent output. This of course differs from modifying the output as you're capturing your customization within the template itself, allowing all future generations of screens to include a particular change. Let's perform a very minor change, just to get you familiar with the process.

1. Navigate to the **C:/PASOE/TrainPAS/webapps/WebWizard/config/templates/** folder.
    a. Note that we have 3 different types of files present here: the CLS files represent templates for the business entities; the HTML files are either standalone pages or for SPA screens (as indicated by name); and the JS files are strictly for use with the SPA versions of the respective HTML files.
2. Open the **TemplateGridSPA.js** file, and begin searching for the exact text "*pageSize*".
3. Modify each occurrence of this property from **20** to **40**. This will change our default results size to 40 records per request to the server. There should be 2 occurrences in this file.
4. Save the file, and open your browser to the **http://localhost:8830/WebWizard/** URL.
    a. Select the "*UI from Catalog*" output option, and "*Read-Only Grid SPA*" template.
    b. Get the catalog, and select "*order*" as the master resource. (This should have been generated as part of an earlier exercise in Chapter 2. If the resource does not exist, please do so now, referring to the earlier section "*Generating Business Entities*".)
    c. Select "*ttOrder*" as the master table and "*Ordernum*" as the search field.
    d. Click the **Generate** button and note if the files were generated as expected.
    e. Remember to **Publish** your new files to the PAS instance via PDSOE.
5. Return to (or open) the application tab in your browser, and either refresh the screen or click on the "*Dashboard*" option to force a refresh of the menu.
6. Look under the "*Browse*" options for your new "*Order Read-Only Grid*" and click the option.
    a. You should see at the bottom of the screen the default selected page size is 40, and the initial results should include items "*1-40 of 3953*". If so, then you have successfully modified an existing template. If you move to the next page, it should reflect a new size of "41-80", and so on.



NOTE

The configuration file for the wizard, which also specifies options for the templates, is called "*wizard.json*" and lives in the **CATALINA_BASE/webapps/WebWizard/config/** directory. You are free to view and potentially modify the options here if needed, and there are 2 sections to know of: "*General*" config options and "*Template*" definitions. It would probably be wise to NOT modify the template definitions, unless you understand what you're affecting; but the general options should be tailored to your project as needed. Likewise, you can always view and modify the templates themselves to suite your project needs, or provide custom templates that reside in the like-named folder.

Creating Custom Screens

At this point we have seen what the templates can produce, and we know where the templates are located within our project. We will utilize the "Blank" template to generate a simple screen, and then fully implement some UI logic from the ground up within this shell of a screen.

Generating the Blank Screen

1. From the WebWizard interface select "*UI from Catalog*".
2. Under the template options select "*Blank Screen (SPA)*".
    a. Only 3 options are shown: Entity Name Prefix, Custom CSS, and Master Resource.
    b. Let's use the name prefix of "*Custom*" for our new screen.
3. Click on the **Generate** button to output the new screen files and menu option. Remember to **Publish** your changes from PDSOE to make the files available.
4. Return to the application and either refresh the page or click on the "*Dashboard*" option to update the menu. There should be a [new] option for "*Transactional*" in the menu.
    a. Under this should be your new "**Custom Blank Screen**". It should have some basic formatting and look like part of the application, but contains no other content.
5. Return to PDSOE and expand the **PASOEContent/WEB-INF/static/app/views/** directory.
    a. Open the files **CustomBlank.html** and **CustomBlank.js** so that we can take a look at their contents and explore some deeper topics.

Examining the Output

First, let's examine the structure of the View (HTML) page. Everything is encapsulated within a "*section*" tag which contains some styles (making it a stretched horizontal layout), and an "*id*" property. This is a bit of important magic here: within a page's DOM there there should only be 1 occurrence of an ID, therefore this must be a unique value. (Conversely, you can use a "*name*" property value numerous times but we'll get into that more later.) In this case we have the name "*CustomBlankView*" which should uniquely identify this HTML block within the application, even if other screen views are loaded later—and that's part of our memory cleanup logic and screen state management. Further into the code we have an "*aside*" tag which helps to push content to the left (if more than 1 aside were present, and another "*section*" within that. Here is where we actually break up our screen into distinct header, content, and footer areas. You are free to keep these tags intact, or change/remove them as needed for your custom code. That's it for the UI portion at this time.

Second, we can take a look at the Controller (JS) file. As mentioned earlier in the material, our controller is the result of an IIFE (function expression) and is contained in a single global variable. It starts with the "*use strict*" declaration to hopefully stop us from doing "Very Bad Things ™" in our code. Within this function we have some "private" variables and functions and return an object to act as our controller. The first variable is "*viewName*" which represents the ID value of our view, very important for knowing what UI code we should be working with. The next variable is "primaryVM" which sets up our ViewModel—more on that in a moment. Next, we have the methods "*bindAll*" and "*loadTemplates*": the former sets up the binding of our observable object (VM) to the view, and the latter is simply a stub method. And finally, the returned object returns 2 methods called "*bindAll*" which is used to initialize the screen, and "*loadTemplates*" which can be executed to load external files at a particular time (prior to the screen initialization).

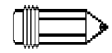
Understanding the ViewModel

Within the KendoUI framework, the concept of data-binding values to widgets is done through a Model-View-ViewModel (MVVM) pattern. In simplest terms, this means use of an instance of the "*kendo.data.ObservableObject*" JavaScript class, bound to a DOM element (via jQuery selector) in our HTML. The properties and methods defined by this object instance become values we can address and use from within our view to affect things like displayed text/values, hiding/showing of elements, enabling/disabling of elements, and executing events (such as click actions).

For example, in the following code we can control the visibility of the form via a property or method called "*showForm*" (if a method, it needs only to return true or false to trigger this event). We also bind the value of the "*SomeField*" input to an object "*params*", specifically a property in that object called "*SomeField*"). And finally we can perform an action "*doSomething*" when the user clicks on the button element. All of this is provided by the observable object instance that is bound to the "form" element in the document (in this case assuming only 1 such tag exists).

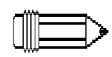
```
<script type="text/javascript">
	var primaryVM = kendo.observable({
		params: {
			SomeField: ""
		},
		showForm: function(){
			return true;
		},
		doSomething: function(){
			alert("Hello World! ");
		}
	});
	kendo.bind($("form"), primaryVM);
</script>

<form data-bind="visible:showForm">
	<input name="SomeField" data-bind="value:params.SomeField"/>
	<button data-bind="click:doSomething">Click Me</button>
</form>
```



NOTE

When binding an observable object to the DOM, you should bind to an element close enough to what encapsulates the content that should respond to the object, but not so high that it might collide with another screen's view. Ideally you should bind to your view's ID and never to the BODY tag, as the latter could cause multiple observables to be in contention with one another.



NOTE

What about the built-in Kendo SPA pattern? While true that Kendo has a pattern with Views and Layouts, it makes heavy use of templates which involve more escaping of special characters and any JS code. So in terms of simplicity, the MVC pattern outlined here has less of a learning curve—though that's not to say that it's perfect. Our SPA does make use of Kendo's Router class for navigation and launching all screens, so that much is still in common with the Kendo SPA.

Making the Changes

With the exposition done, we can focus on actually creating something. First, we're going to cheat a little bit and use another template to give us a majority of what we need for our new screen. To do this, use the wizard to create a new screen via the "*Grid Over Grid (SPA)*" template using "*order*" and "*orderline*" as the resources (making sure to set your search and key fields), and generate the screen. Confirm that your screen was created and working, then move on to the next steps.

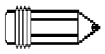
We will first need the HTML and JS files for our generated blank screen open in PDSOE. Also, open the files for the new "*OrderOrderLineGridOverGridSPA*" screen (yes, it's a mouthful). The easy solution here is to copy a good bit of the JS content as-is, then we will update our local code as needed. The idea is that we will create a tab-based layout that still uses a master and detail grid, but with those grids on separate tabs (instead of stacked or nested). This is not necessarily a typical pattern, but should at least be an example of using portions of existing screens to create new functionality.

JavaScript Changes

1. Copy the variables at the top of the Grid-over-Grid screen to the same area in our "blank" screen. These will set the necessary table names, etc. for the new screen.
    a. Remove the copied **viewName** variable, as we already have that defined.
2. Copy the entire block of code from the "*primaryVM*" declaration all the way through the end of the "*bindAll*" method definition. Replace the same 2 items (inclusive) in the new JS file, with the copied content.
3. Change the height properties on the grids (master and detail) with a value of 400.
    a. Hint: Replace the string stating a % with just the new integer value.
4. In the "*bindAll*" method, add the following just after the "kendo.bind" line. This will bind the new widget that will be added to the HTML file, and convert it to a TabStrip widget:

```
$(viewName + " div[name=tabstrip]").kendoTabStrip({animation: false});
```

5. Save the file, and move on to the HTML view.


NOTE

Not covered in depth here is the availability of the client-side JavaScript library. This is already included in your code via **/vendors/spark/lib/spark.min.js** as a minified file. It creates a singular "*spark*" object as a global variable, meaning you can access it from anywhere in your code. It can provide some commonly-used code shortcuts for certain Kendo operations as well as converting elements to specific widgets (such as an "M/D/Y" date field, or a formatted phone number field). The easiest way to see what is available is by typing "spark" within your browser's Console window to look at the returned object. For more details, look at the **/vendors/spark/src/** directory under **PASOEContent/WEB-INF/static**.

HTML Changes

1. In the new screen's HTML view add this block of code within the **section** tags, between the **header** and **footer**:

```
<div name="tabstrip">
    <ul>
        <li class="k-state-active">Order</li>
        <li>OrderLine</li>
    </ul>
    <div>
        <form class="form-group" data-role="validator"
              name="searchForm" novalidate="novalidate">
            <div class="input-group m-t-sm m-b-md">
                <input class="form-control input-sm"
                       placeholder="Search by Order Num"
                       data-bind="value:params.searchValue"
                       style="width:300px;"/>
                <span class="btn btn-default btn-sm"
                      data-bind="click:doSearch">Search</span>
            </div>
        </form>
        <br/>
        <div name="MasterGrid"></div>
    </div>
    <div>
        <div name="DetailGrid"></div>
    </div>
</div>
```

2. Save the file, **Publish** to your PAS instance, and return to the application in the browser.
3. Navigate back to the "Custom Blank Screen", or otherwise refresh the page if already loaded.
    a. You should have two tabs available, "*Order*" and "*OrderLine*".
    b. The "*Order*" tab should display the search criteria/button and a grid.
    c. Selecting a record in the "*Order*" grid will put a record into context.
    d. Changing to the "*OrderLine*" grid should show lines associated with the order.

Creating a New Template

Now it's time to take things a bit further and learn how to convert our new screen into a template, just to illustrate how this can be done for your own application. But first we need to discuss the various tokens that are understood by the generator behind the scenes, especially if we intend to make use of any REST schema or resource options within our templates.

Spark Tokens

Tokens are found within the templates as values between "< >" tags and currently have a prefix of "*Spark_*". Below are the currently supported options within the template generator, and represent either values directly selected on-screen in the wizard or created within the generator logic as a result of those wizard selections.

Direct Substitutions

- **Spark_ResourceName –** Name of a single REST resource to be accessed via the JSDO.
- **Spark_MasterResource** – Alias for the Spark_ResourceName when working within a template that utilizes both a master and detail resource for data relations.
- **Spark_DetailResource** – A secondary resource for templates with master/detail data joins.
- **Spark_MasterTable** – The primary table name as utilized by the Spark_MasterResource.
- **Spark_DetailTable** – The secondary table name as utilized by the Spark_DetailResource. This is only utilized in templates that make use of a data relation between master/detail resources.
- **Spark_MasterKey1** – Denotes the field in the Spark_MasterTable that should be used for a data relation to the Spark_DetailTable.
- **Spark_DetailKey1** – Denotes the field in the Spark_DetailTable that should be used for a data relation to the Spark_MasterTable.
- **Spark_SearchField1** – The primary field to be used for search operations in most templates. This will be combined with the Spark_SearchField1Oper to create the necessary criteria for the Read operation on the specified REST resource.
- **Spark_CSSFile** – Custom CSS file to be incorporated into a page/screen.
- **Spark_EntityName** – Optional field to override the name of the "entity" being created. If not provided, a screen name will be assigned based on the selected resource and type of template.

Generated Values

- **Spark_CatalogURI** – Denotes the relative path to the JSDO catalog source.
- **Spark_ServiceURI –** Denotes the relative path to the primary service endpoint.
- **Spark_TemplateName** – A simplified name of the template, constructed at the time of generation and used internally to identify this particular screen (necessary for SPA's).
- **Spark_GenAuthor** – this token is used by the Template Business Entities to generate the name of the author of the file.
- **Spark_GenDate** – Dynamic token to provide the current date of generation.
- **Spark_Namespace** – The prefix (class path) to be added to any business entity class names to accurately reflect the location of the file. Default value is "Business." as files should be located in the /Business/ directory.
- **Spark_InheritedEntity** – The full class path and name of the common, inherited class as needed by a generated business entity.
- **Spark_SchemaFile** – A partially-calculated path file to the generated schema .i file as required by a business entity. This is based on the selected table for a resource.
- **Spark_SkipListArray** – Provides a comma-separated list of field names to be ignored in the create statement. Used in the generation of a business entity.
- **Spark_DatasetName** – A partially-calculated name of the dataset to be used by a resource or as part of a UI screen. In some cases, just the prefix "ds" plus the selected table name.
- **Spark_TempTableName** – A partially-calculated name of the temp-table to be used by a resource or as part of a UI screen. In some cases, just the prefix "tt" plus the selected table name.
- **Spark_SearchField1Oper** – A calculated data operator (eq, lt, gt, etc.) for the Spark_SearchField1 as based on the field data type (from schema information).
- **Spark_SearchField1Title** – Name of the search field as determined by the resource schema, specifically the label (or name if none available) of the selected field for Spark_SearchField1.
- **Spark_GridFields** – Generated list of fields (in JSON format) to be incorporated into a grid instance. Utilized on templates that make use of a KendoGrid as a master browse.
- **Spark_GridFieldsDetail** – Generated list of fields (in JSON format) to be incorporated into a grid instance. Utilized on templates that make use of a KendoGrid as a detail browse.
- **Spark_FormFields** – Generated HTML that provides INPUT fields in a form layout.
- **Spark_SearchFields** – Generated HTML that creates a form containing the 3 specified search fields within the "Run My Progress Code" template.
- **Spark_CodePath –** Special token that converts the class namespace into a directory path.

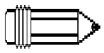
Adding Custom Templates

Here's where the real work is done to make things repeatable and easier. We will utilize the newly created screen to produce a template from this code, making it more generic and available for similar situations where this code is needed.

1. Copy the "*CustomBlank.html*" and "*CustomBlank.js*" files from your **PASOEContent/app/views/** directory to the directory at **C:/PASOE/TrainPAS/webapps/WebWizard/config/templates/custom/**
2. Rename the copied filed to "*TabbedGridsSPA.html*" and "*TabbedGridsSPA.js*", respectively.
3. Open the HTML document and make the following replacements as needed:
    a. "CustomBlankView" → "<Spark_TemplateName>View"
    b. Order → <Spark_MasterTable>
    c. OrderLine → <Spark_DetailTable>
    d. "Search by Order Num" → "Search by <Spark_SearchField1Title>"
4. Open the JS document and make the following replacements as needed:
    a. On the first line: var <Spark_TemplateName>Ctrl
    b. var masterResourceName = "<Spark_MasterResource>";
    c. var detailResourceName = "<Spark_DetailResource>";
    d. var searchField1 = "<Spark_SearchField1>";
    e. var searchOper1 = "<Spark_SearchField1Oper>";
    f. var datasetName = "ds<Spark_MasterTable>";
    g. var masterTableName = "tt<Spark_MasterTable >";
    h. var detailTableName = "tt<Spark_DetailTable >";
    i. var masterKeyName = "<Spark_MasterKey1>";
    j. var detailKeyName = "<Spark_DetailKey1>";
    k. var viewName = "#<Spark_TemplateName>View";
    l. var gridColumns = [<Spark_GridFields>];
    m. var detailColumns = [<Spark_GridFieldsDetail>];
5. Open the "*templates.json*" file so we can add the new template.
    a. Copy the object that defines the "*CustomSampleSPA*" template.
    b. Paste this as a new object in the **Template** array (remembering to use a comma).
    c. Increment the value for the **DisplayOrder** field (to ensure this falls as the last item).
    d. Change the **TemplateID** to "*TabbedGridsSPA*".
    e. Change the **TemplateLabel** to "*Tabbed Grids (SPA)*".
    f. Change the **TemplateUI** to "*TabbedGridsSPA.html*".
    g. Adjust the **NameTemplate** to "*[PREFIX]TabbedGridsSPA*".
6. Recompile your OpenEdge code, or change a file to trigger a Republish state on the PAS instance. Perform a **Publish** and this will restart our MSAS sessions. Refresh the WebWizard screen in Chrome to obtain the new list of templates.

This should allow us to use the new template as part of the wizard, so return to the open browser tab with the wizard and refresh the screen to obtain the new list of templates.

1. Select the new template, which should be at the bottom of the list.
2. Generate a new screen type using the "*customer*" and "*order*" resources, keyed on the "*CustNum*" field, and searchable on "*Name*". **Publish** your changes to the PAS instance.
3. Return to the application and either refresh or click the "Dashboard" item to refresh the menu.
4. The new screen should be under the "*Transaction*" section, with your "*Tabbed Grids (SPA)*" name.
5. If everything worked, you should have a Customer/Order hybrid screen similar to what we created by hand for the Order/OrderLine tables.



NOTE

This "*templates.json*" file is the key to adding new (read: custom) templates to the list of available options while not interfering with the framework-supplied options. This will be useful for future releases which may incorporate more templates but not collide with the custom templates. This is also why the **DisplayOrder** values begin at 50 for these types of templates. This must be a unique, incrementing value and allows for up to 49 standard templates and any number of custom templates (just so long as you keep the count at 50 or above).

NOTES: