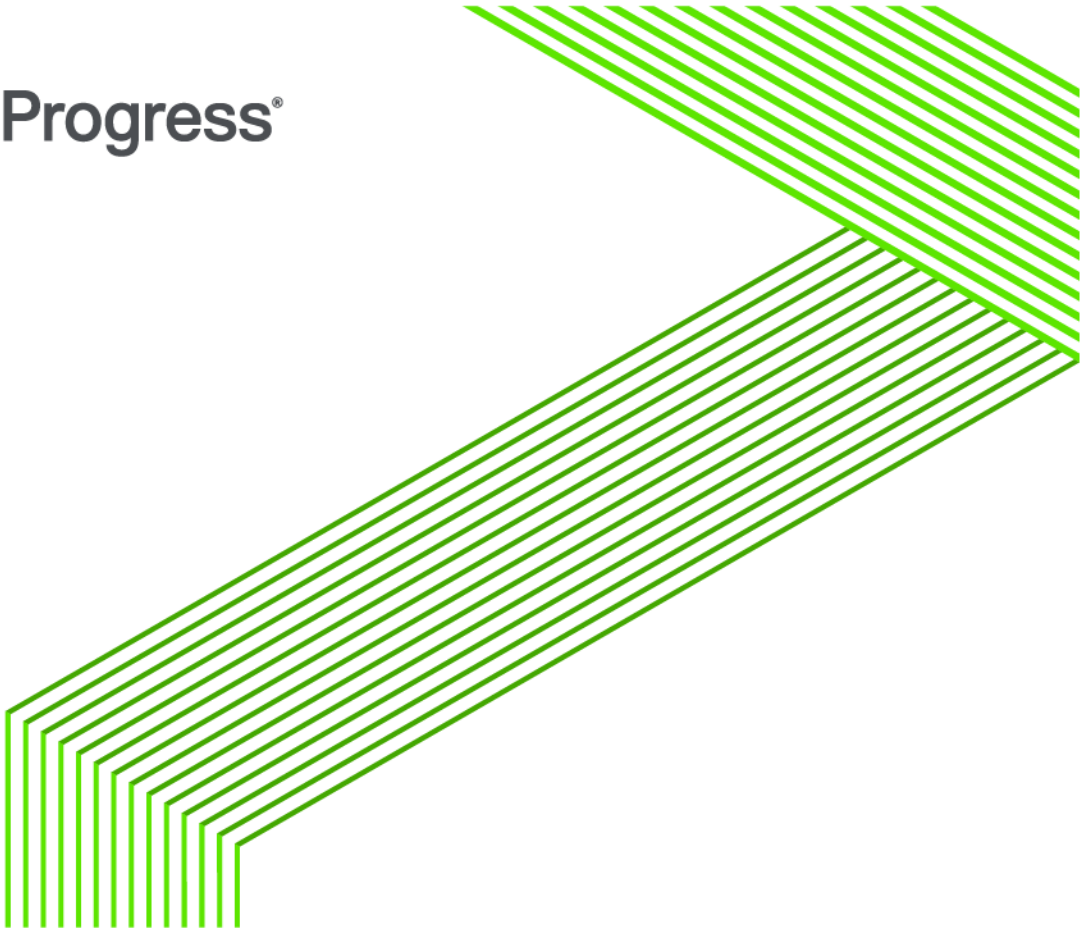
Progress®



Kendo UI® Builder by Progress®:

Modernizing OpenEdge Applications

Progress®
Kendo UI® Builder

Copyright

December 2017

Last updated with new content: Version 2.1

Updated: 2017/12/22

Table of Contents

1

Overview and Architecture

The Kendo UI® Builder by Progress® facilitates the modernization of existing Progress® OpenEdge® desktop business applications by moving the application user interface (UI) to the web using Kendo UI. These OpenEdge applications can be simple to more complex applications containing multiple, feature-specific modules. Applications that already conform to the OpenEdge Reference Architecture (OERA) are especially well suited for modernization using the Kendo UI Builder. However, you can modernize any OpenEdge application with its UI separated from its business logic running on an OpenEdge application server, which can be either an instance of Progress Application Server for OpenEdge or the classic Progress® OpenEdge® AppServer®.

Kendo UI Builder tooling supports the design and development of a modern and responsive web UI in the form of a deployable OpenEdge web app that accesses one or more ABL application services implemented as OpenEdge Data Object Services. This tooling supports UI upgrades for future versions of the initial web app over time, with little or no additional coding, using customizable templates and meta-data from which the deployable web app is generated.

For details, see the following topics:

- Steps to modernize an OpenEdge application
- Architecture and components

Steps to modernize an OpenEdge application

For OpenEdge, the Kendo UI Builder includes components from several Progress products that you use in an iterative fashion to modernize an OpenEdge application as follows:

1. Ensure that your application UI is separate from its business logic, with the ABL business logic tailored to run on an OpenEdge application server. The OpenEdge Reference Architecture provides a methodology

for accomplishing this. For more information on the OERA, see *OpenEdge Getting Started: Guide for New Developers*.

For more information on tailoring ABL business logic to run on your choice of OpenEdge application server, see the overview and application development documentation for:

- **Progress Application Server for OpenEdge** — *Progress Application Server for OpenEdge: Introducing PAS for OpenEdge* and *Progress Application Server for OpenEdge: Application Migration and Development Guide*
- **OpenEdge AppServer** — *OpenEdge Getting Started: Application and Integration Services* and *OpenEdge Application Server: Developing AppServer Applications*

You can then deploy and run your ABL application with its UI running in a separate ABL client that accesses its business logic as an ABL application service.

2. Implement a new service interface for your business logic in the form of an OpenEdge Data Object Service. An OpenEdge Data Object Service provides web access to your ABL business logic through one or more OpenEdge Data Objects implemented as ABL Business Entities. ABL Business Entities are annotated ABL class or procedure-based objects that provide a standard web interface to your data and business logic. An OpenEdge web app can then access this standard interface using a JavaScript Data Object (JSDO) that hides the underlying details of the network request and response protocol from the app. A Data Object Service then manages all web access between an instance of the JSDO in the web app and a given Data Object running on the OpenEdge application server.

   For an overview of OpenEdge Data Object Services and how to implement ABL Business Entities as Data Objects, see the information on Data Object Services in *OpenEdge Development: Web Services*.

   For information on creating, editing, testing, and deploying Data Object Services for both PAS for OpenEdge and the classic OpenEdge AppServer, see the *Progress Developer Studio for OpenEdge Online Help* and the administration documentation for each OpenEdge application server.

   Note: Also see the *New Information* documentation for recent service packs of your supported OpenEdge Release.

   For information on the JSDO and how it can be used in web apps to access Data Object Services, see the *Progress Data Objects: Guide and Reference*

3. Design and build the OpenEdge web app that contains the web UI for your OpenEdge application using the Kendo UI Designer. This is a Electron-based, Kendo UI Builder tool that can optionally install into your OpenEdge environment. The Kendo UI Designer is thus an on-premise, visual design tool that accelerates web app development based on selected Data Object Service meta-data and UI templates for supported Kendo UI components.

   The initial result is a set of UI meta-data that you can customize in a prescribed fashion. You can then invoke the integrated Kendo UI Generator to build and preview the web app directly from this meta-data in your default web browser, allowing you to test the UI and its data access from within the Kendo UI Designer itself. A Node.js-based webpack-dev-server continuously updates your initially built and previewed app as you save further changes to it in the Designer using the Kendo UI Generator or in customizable source code using your choice of editor.

   The present documentation provides an overview of the Kendo UI Designer and how to work with it to build and test a web app with access to OpenEdge data and business logic. For more detailed information on using the options of the Kendo UI Designer, see *Kendo UI Builder by Progress: Using the Kendo UI Designer*.

4. Optionally, use Progress Developer Studio for OpenEdge to deploy each stage of completion for both Data Object Services and the client OpenEdge web app. In Kendo UI Designer, you can configure the web app location so the Kendo UI Generator automatically builds the web app within a Web UI project of Developer Studio. From this project, you can deploy the app to a development instance of PAS for OpenEdge to test

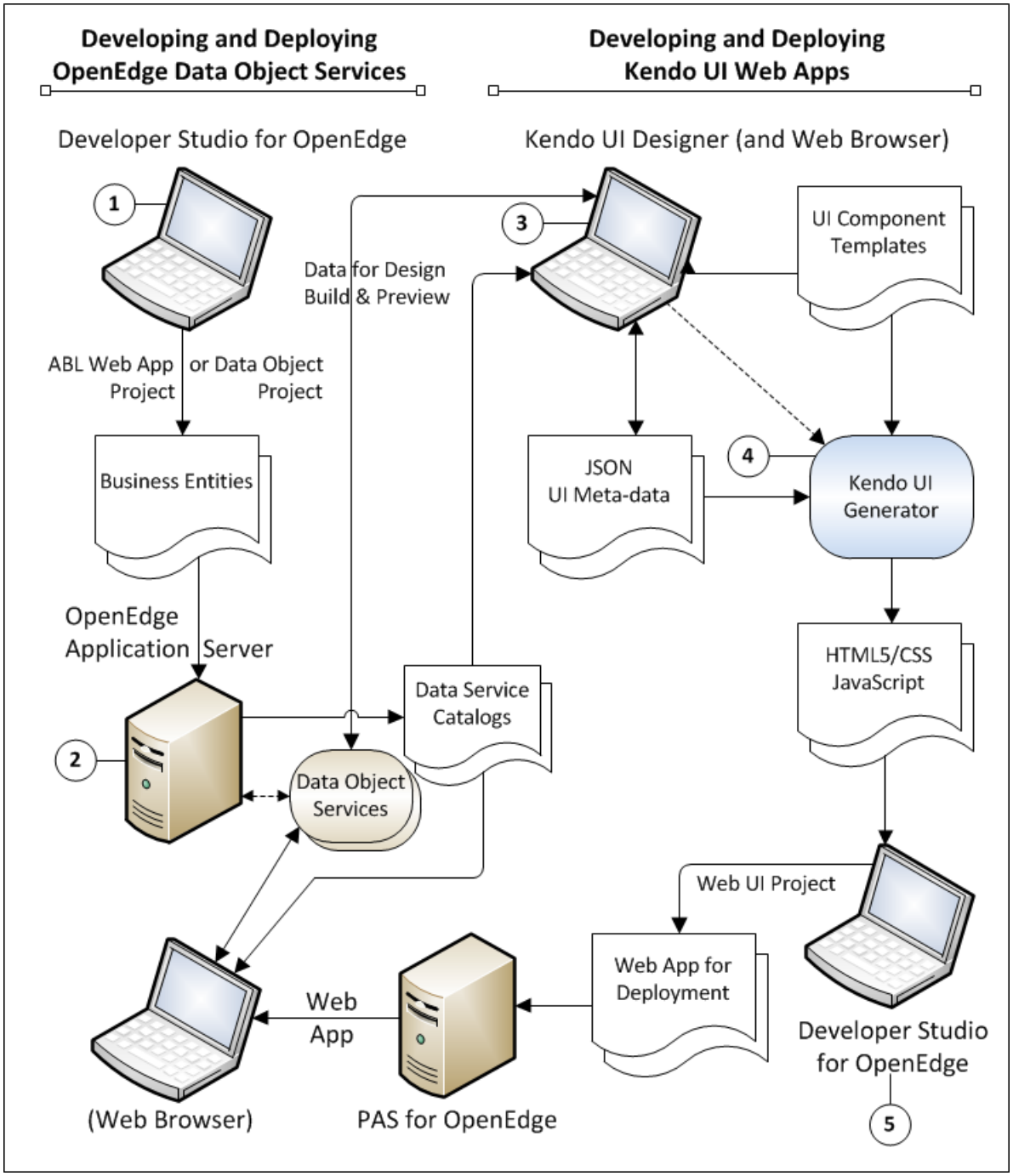
general web access. Ultimately, you can deploy the completed web app for release on any production web server of your choice.

For a brief walk-through of building an OpenEdge application, end-to-end, with the modern web UI provided by an OpenEdge web app, see Kendo UI Builder by Progress: Sample Workflow.

Architecture and components

The following figure shows the overall architecture of the Kendo UI Builder components and their relationship for modernizing OpenEdge applications:

Figure 1: Kendo UI Builder components



Refer to the numbered call outs in the following description:

1. Use an appropriate project in **Progress Developer Studio for OpenEdge** to develop Data Objects from **Business Entities** that you create and package together as one or more Data Object Services for deployment

to an **OpenEdge Application Server**. Use an **ABL Web App** project to build and deploy **Data Object Services** to an instance of the Progress Application Server for OpenEdge. Use a **Data Object** project to build and deploy **Data Object Services** to the classic OpenEdge AppServer.

2. The deployment of **Data Object Services** includes **Data Service Catalogs**. Each Data Object Service is defined by an associated Data Service Catalog. This Catalog is a JSON file that contains meta-data describing the schema and operations supported by each Data Object managed by the Data Object Service. The JavaScript Data Objects (JSDOs) that the Kendo UI Designer and its generated web apps create to access Data Object resources rely on the Catalog for each Data Object Service that provides these resources to manage access to the data across the network.
3. The Kendo UI Designer runs as an Electron application installed on your local system. Use the Designer to create an OpenEdge web app, for which you specify a name and folder location for the files. An OpenEdge web app in the Kendo UI Builder is built from a template that consists of one or modules with access to table resources provided by one or more Data Object Services. You create each module from one or more views that you specify using selected **UI Templates** that support a variety of Kendo UI layouts, such as a grid and form. You bind data to each view by associating one or more Data Object resource tables as data sources, depending on the view. You can define app function and presentation by setting properties on the app, each module, and its views, then preview the result in your default web browser with real data from the data sources that are bound to the views. At any point in your design, you can save the current state of the web app to **JSON UI Meta-Data** that, together with the selected **UI Templates**, define the UI and behavior of the app. Note that the UI meta-data is itself independent of the Kendo UI implementation, and is used to generate a Kendo UI-based web app based on the **UI Templates** that you select.
4. At any stage that you are ready to preview and test the app, you can build the app by invoking the **Kendo UI Generator**. This code generator takes the saved **JSON UI Meta-Data** and referenced **UI Templates** as input, and generates a deployable OpenEdge web app containing the functionality you have designed. In addition, the Generator builds your web app in the context of Bootstrap and AngularJS, which provides a responsive UI for your app. The generated **HTML5/CSS and JavaScript** files are then saved to the app location you have specified, which can be a **Web UI Project** of Progress Developer Studio for OpenEdge. Note that a Node.js-driven webpack-dev-server (running in the background) continuously and incrementally updates your previewed app as you save further changes to it in the Designer using the Kendo UI Generator or in customizable source code using your choice of editor.
5. By creating a **Web UI Project** in Developer Studio, you can save your generated **Web App for Deployment** either as a development build for round trip testing and debugging, using a development instance of PAS for OpenEdge, or as a release build for delivery on a production instance of PAS for OpenEdge. In addition, you can export the **Web UI Project** as a **Web UI Application**, which creates a WAR file for your web app that you can deploy to any compatible web server of your choice.

2

Kendo UI Designer Overview

The Kendo UI Designer allows you to visually design, build, and preview OpenEdge web apps with a responsive UI based on Kendo UI, Bootstrap, and AngularJS and with access to OpenEdge Data Object Services. These are one-page apps that you design with the app and data definition stored in JSON meta-data that is separate from the Kendo UI implementation.

This meta-data is then used by the integrated Kendo UI Generator to generate the HTML5/CSS and JavaScript files that you build for an app. The app generation also allows you to immediately preview the app in a browser using live data. You can then eventually deploy the app to separate web servers, including OpenEdge application servers, for further development testing and production.

You design a web app from inputs that include a set of Kendo UI Builder templates from which you can create functional views within one or more app modules. Each module can contain one or more user-created views, and each view can be bound to one or more data source tables that you select, depending on the view. You select each data source from a data provider that you define for one or more OpenEdge Data Object Services. You can have multiple data providers defined for an app, and depending on the view, you can select one or more of these data providers from which to then bind data sources to that view. Some user-created views bind to only one data source at a time, while others allow binding to multiple data sources.

Modules and their views, data providers and their data sources, can all be configured with corresponding properties. These property settings then help to define the meta-data for your app, which is saved separately for each module and data provider that you define.

Finally, you can customize each view with code extensions for both view event handlers and custom sections that are available in the layout of every view. There are also additional extension points available for both basic and more advanced app customization.

The following topics further describe these components of an OpenEdge web app and how you can use them in the Kendo UI Designer to design and build the app. For more information on options for using the individual wizards and dialog boxes that are provided to complete this work in the Designer, see *Kendo UI Builder by Progress: Using the Kendo UI Designer*. For more information on the extension points available for app customization, see Extension Points and Source Code Customization on page 103 in this document.

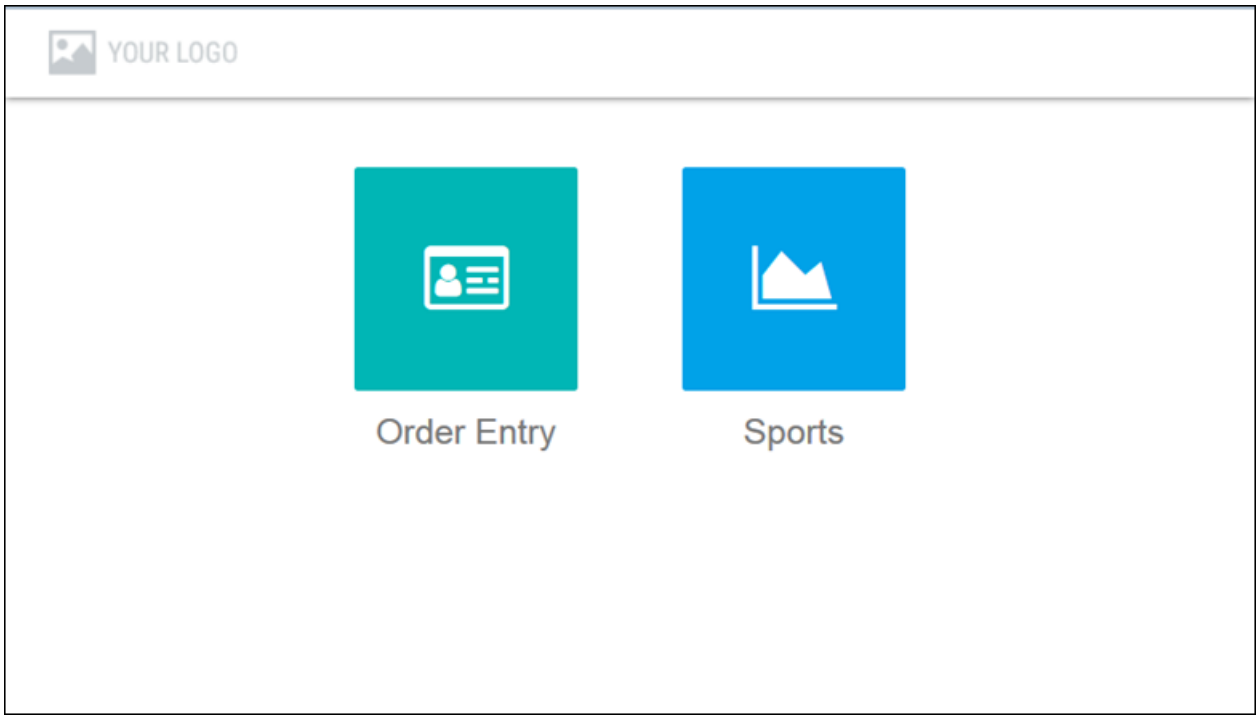
For details, see the following topics:

- App layout and components
- Creating and designing an app
- Data providers and data sources
- Modules and views
- Using roles to authorize user access
- Localizing the generated app
- App generation and deployment

App layout and components

When an OpenEdge web app first starts up, if no view in any module accesses data providers that require authentication, an app landing page displays when you first open the web app, which displays a built-in landing-page view that every Designer-generated web app contains, with a basic layout as shown in these example screens:

Figure 2: App layout example—landing page



If one or more views access data providers that require authentication, a built-in login view displays prompting for credentials to access each secure data provider in the app (see Editing the login view on page 39). If all requested credentials are accepted, the landing page is then displayed.

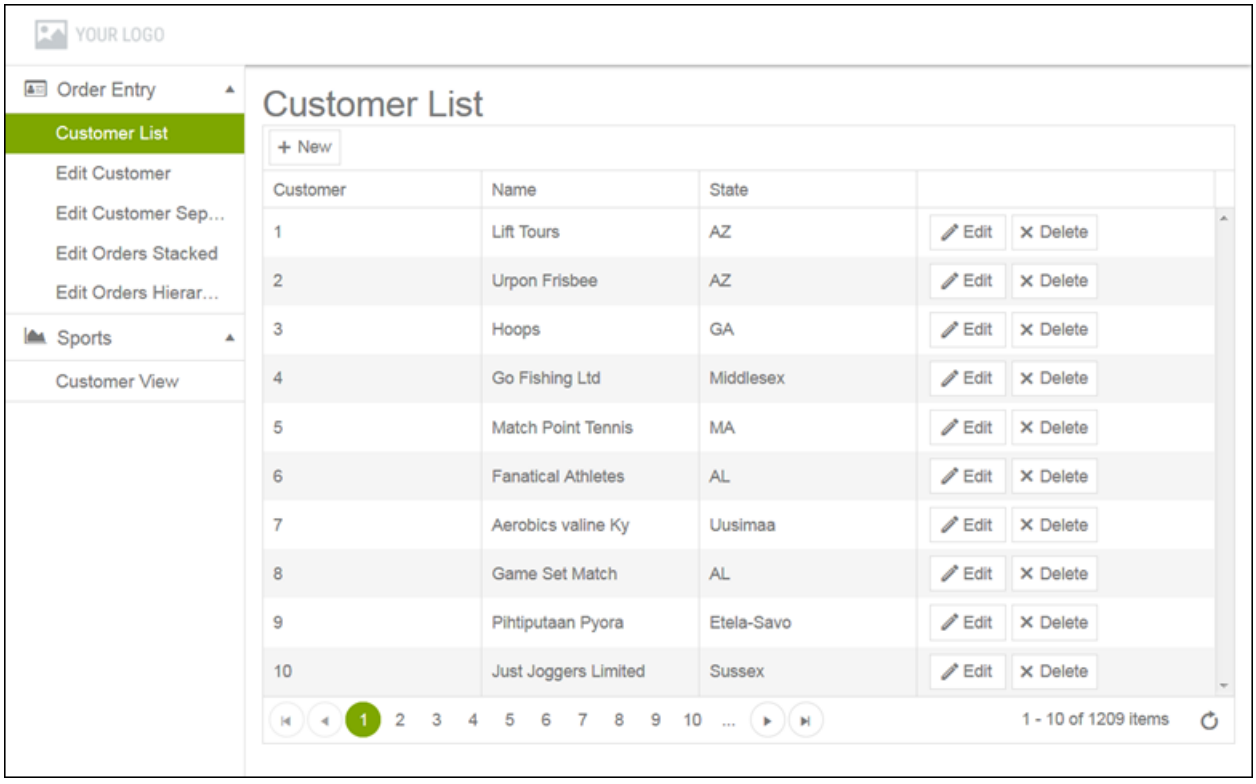
This landing page includes the following default components:

- **Header** — Showing the default logo in the example:
  - You can specify your own company logo using web app settings (described later).
  - This header can also provide a drop-down menu on the right (not shown) that includes a **Logout** option. This drop-down menu is hidden if no login was required to access data providers.
- **Module icon list** — Showing selectable icons for modules that are available in the app. If you select a module, its first available view displays in the views page (see below). Each module is identified with a run-time label that you can optionally specify at design time.

Note: You can bypass this landing page by providing a URL that goes directly to a view, such as the following **Customer List** view shown in the following **Order Entry** module example. For example, suppose this view is part of a web app, **OrderEntryWebApp**, that is hosted on a local Progress Application Server for OpenEdge instance. If the local URL for the landing page is `http://localhost:8810/OrderEntryWebApp/#!/home`, the URL to open this **Customer List** view directly is `http://localhost:8810/OrderEntryWebApp/#!/module/order-entry/customer-list`. Note that the module and view URIs are always provided in Kebab Case. This same kebab case is used in the URL regardless of how the components of the module and view names are specified in the Designer. For more information on specifying view names in the Designer, see Adding and editing a Data-Grid view on page 42.

This is an example app views page that might display if you select the **Order Entry** module in the example landing page above:

Figure 3: App layout example—views page



The app views page opens with the following components

- **Header** — Displays with the same elements as the landing page header described previously. If, at any time, you select the logo in the header, the app returns to the landing page.
- **Module/view list** — On the left, showing a list of the modules available in the app, each with its own drop-down list of views that it provides. The example shows the following modules:
  - **Order Entry** — After being selected in the app landing page, the views page is opened for this module, with its drop-down list shown and the first listed view in the module selected and displayed. You can then display a different view in the module by selecting the label for another view in the list.
  - **Sports** — An additional module, with its drop-down list displaying a single view.

  Note: Not shown for **Order Entry**, any module selected in the landing page has its initial view displayed along with the module description in an overlaying caption. The caption disappears with the next selection on the page.

- **View display area** — On the right, showing a single view from a module (initially, the first view in the module selected on the landing page). When you select and open a view in any available module, the opened view is then displayed, and replaces any view previously displayed in the view display area.

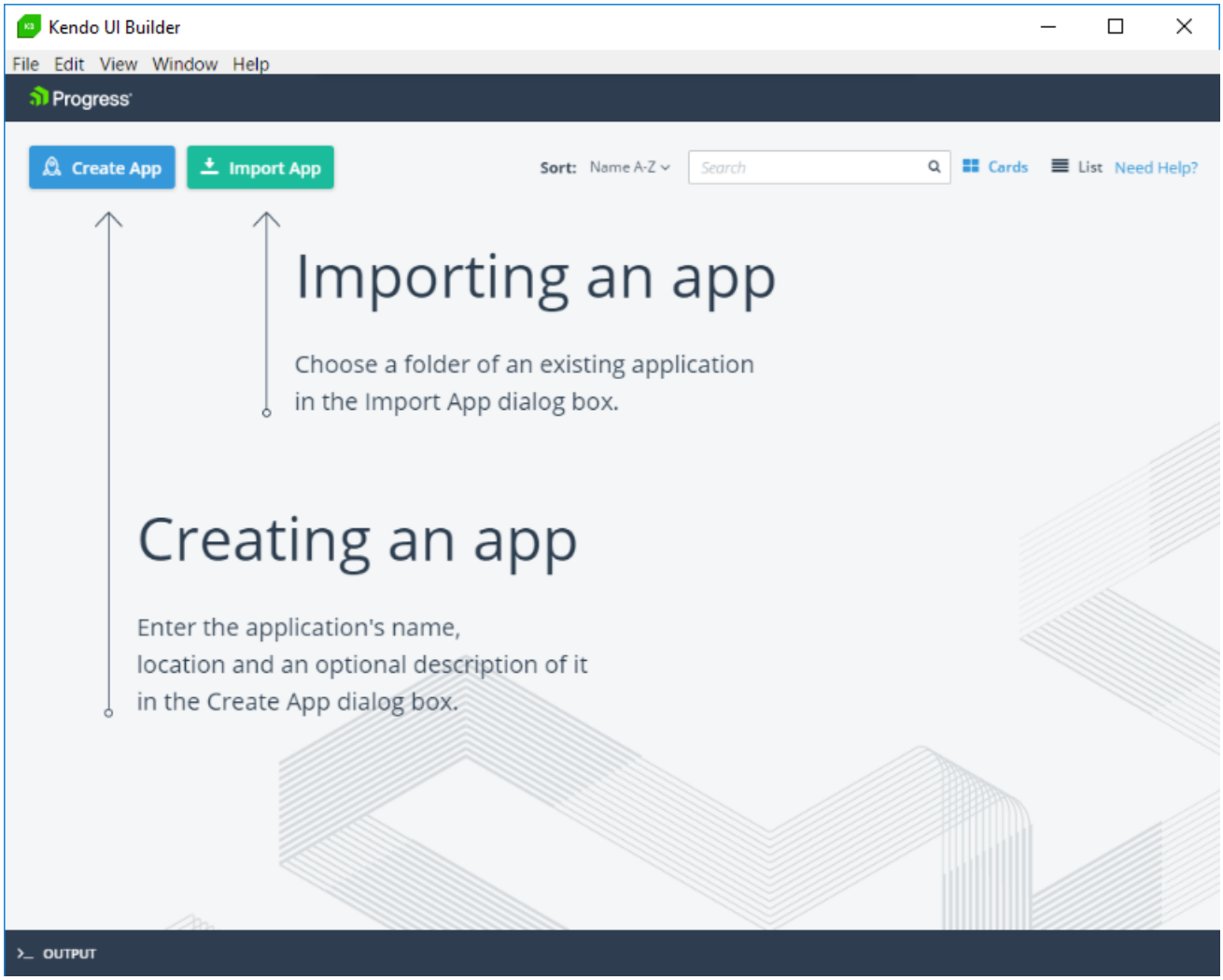
For more information on creating modules and views for an app, see Creating and designing an app on page 17.

Note: A web app built in the Kendo UI Designer is built with only a single HTML page. The app landing page and views page are not separate HTML pages, but represent the same HTML page displaying different types of views: 1) the landing page, which displays the built-in landing-page view, and 2) the views page, which displays a single view that is selected in a module.

Creating and designing an app

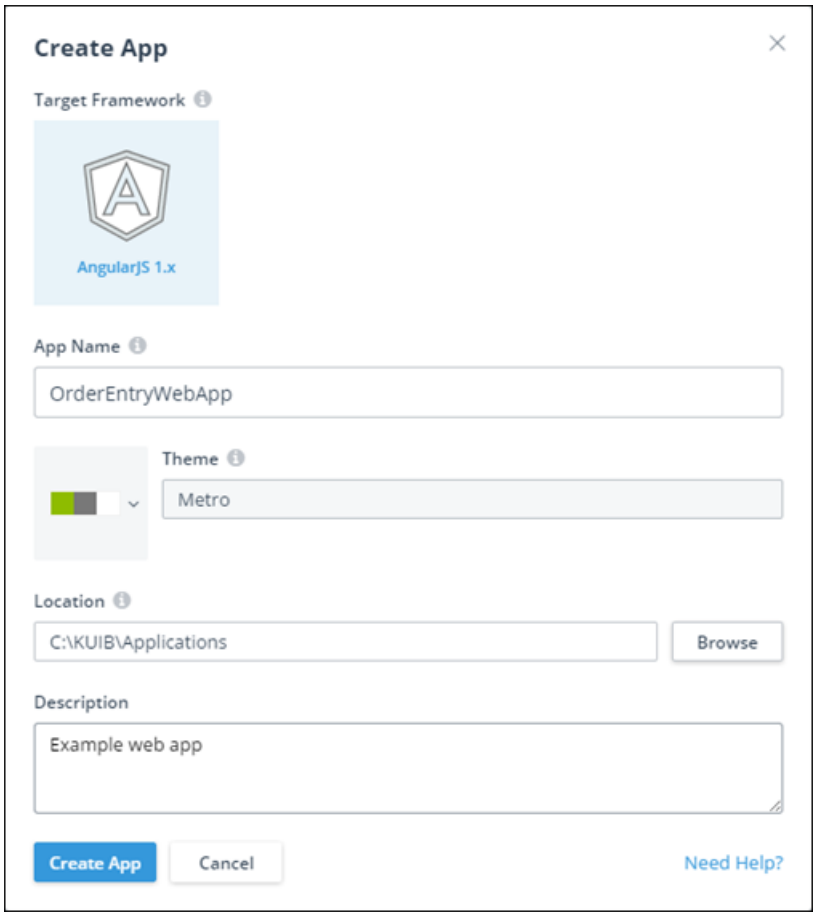
When you first start the Kendo UI Builder, it opens the Kendo UI Designer dashboard similar to this:

Figure 4: Kendo UI Designer dashboard



To create an app, click **Create App** as described on the page. This takes you to the **Create App** dialog box to create a web app, as in this example:

Figure 5: Create App dialog box



A read-only logo identifies the **Target Framework** for the app (only AngularJS 1.x is supported in this release). The value you choose for **App Name** becomes the name of the folder where your app is created. The **Theme** selector allows you to select a color scheme for the app from among several built-in options. The **Location** value is the path name of the folder where the app folder is to be created. You can optionally enter a **Description**, which appears along with the **App Name** on the dashboard after you create the web app.

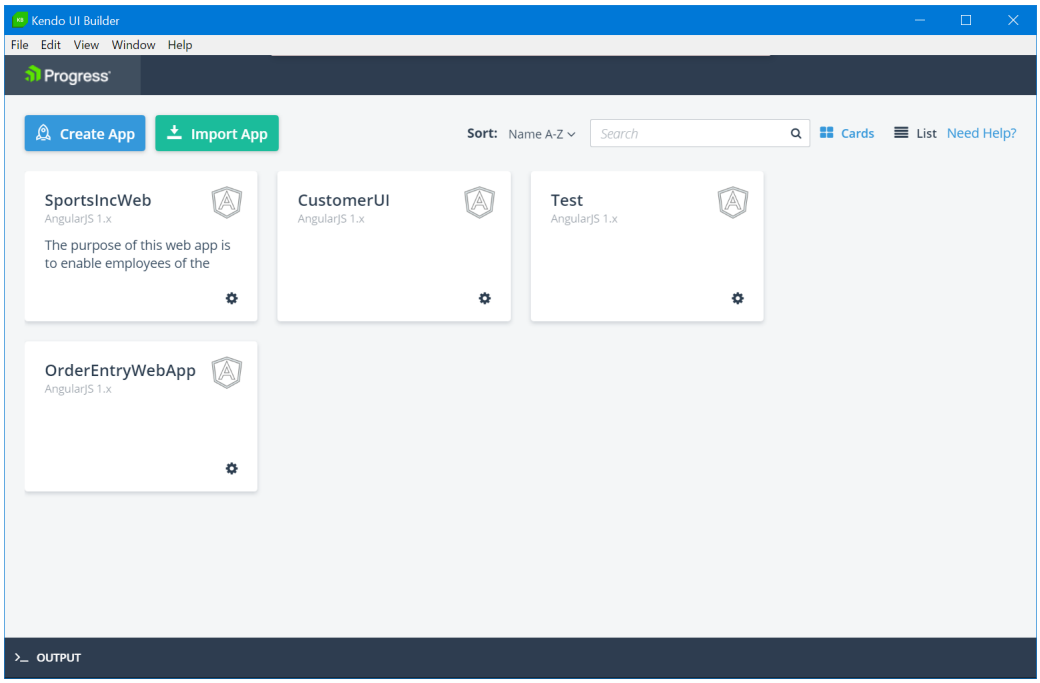
If you want your app development testing and production deployment to be managed from within a Web UI project of Progress Developer Studio for OpenEdge, specify **App Name** with the same name as the Web UI project, and specify **Location** as the pathname of the folder path name where you create the project as shown in the example. This **Location** can be either within the Developer Studio workspace or in any other location accessible to Developer Studio. Note also that you can create the Web UI project in Developer Studio to manage the web app either before or after you have created and built the app in the Kendo UI Designer as long as both the project and app names and locations for both Developer Studio and the Designer are the same.

After filling in the fields, you can create the app by clicking the **Create App** button, which opens the *app design page*, where you can design, build, and preview the app, as shown later in this topic.

When at any point after creating the app you return to the Kendo UI Designer dashboard by clicking the **Progress** logo in the header, a card or list item is displayed for the app by selecting **Cards** (the default) or **List** in the toolbar of the page.

This is an example card for the **OrderEntryWebApp** app, as created in the previous **Create App** example:

Figure 6: Dashboard with created app card



Note that if you are upgrading from an earlier build or version of Kendo UI Builder, the first time you open the Designer, it attempts to automatically import apps that you had on the dashboard of your previous version, displaying their app cards and an indication of the status of their import. You can also explicitly import external apps not currently on the dashboard by clicking **Import App**, which adds the app card and a similar indication of the import status.

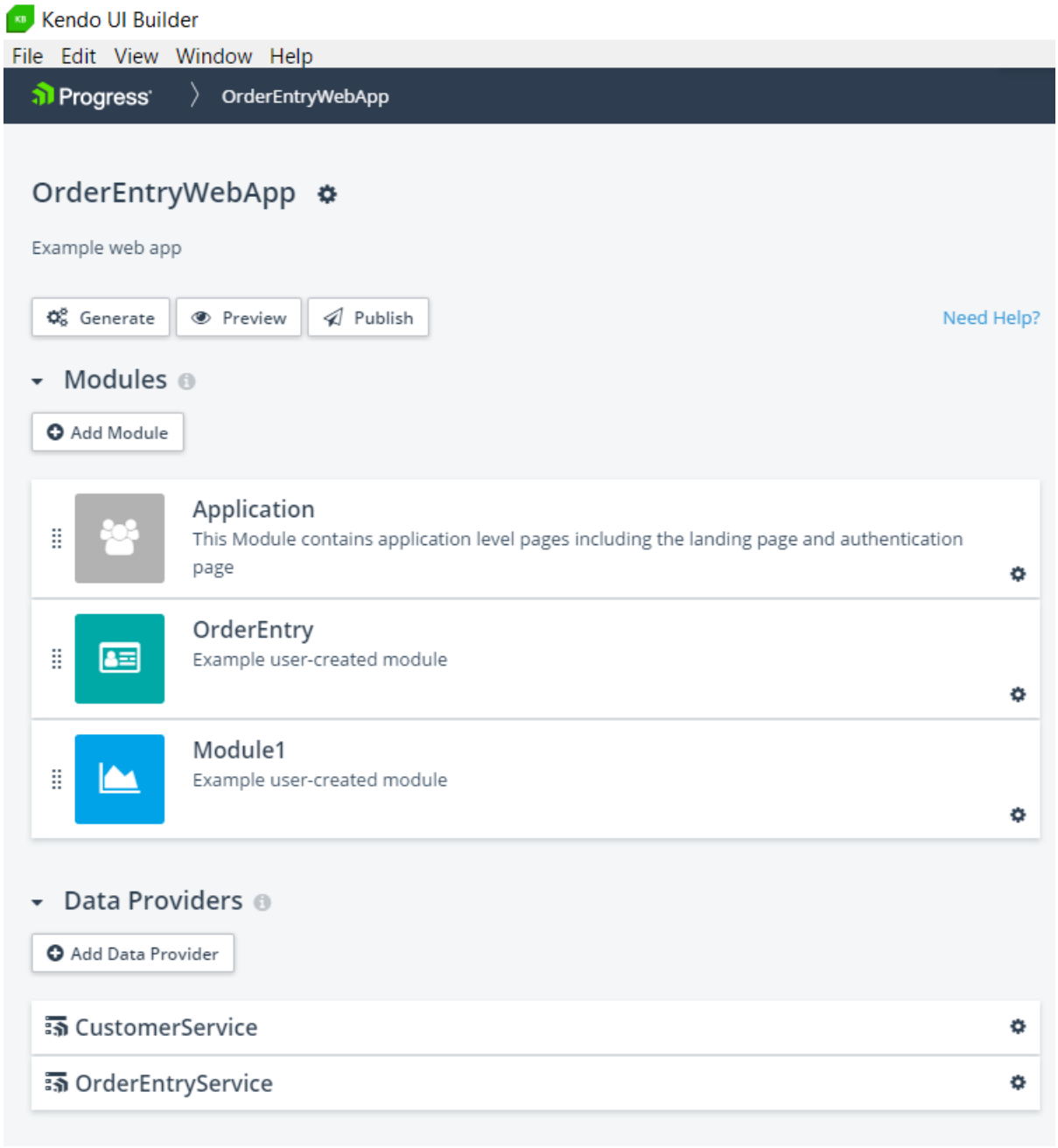
Clicking the gear control on the app card (or list item), provides several options, depending on any import status. Possible options include **Duplicate** (create a copy of the app with a different name and/or location), **Migrate** (migrate the meta data if an import was successful and migration is required), **Remove** (remove the app from the dashboard, leaving its file structure intact), and **Delete** (delete the app, including its file structure, from the Designer and from any Web UI project you have created for it in Progress Developer Studio for OpenEdge).

Note that to help locate one of many apps you might create, you can sort the apps or search for a given app card or list item using **Sort** or the search box, respectively. Note also, anywhere in the Designer where you see **Need Help?**, you can click to open a help topic for that page or dialog box in the Designer.

At this point, you can click the app card or list item to return to the app design page, which might appear similar to this example for **OrderEntryWebApp** showing app development already in progress:

The **Output** bar at the bottom displays logging information. Clicking the bar opens a console that displays log messages that are useful when you need to debug or troubleshoot a problem. The **Eraser** shaped icon on the left hand corner clears the console viewer and the **X** icon closes the console viewer.

Figure 7: App design page



The app design page is where all design, build, and preview activities are initiated for an app. This particular example shows some components of our **OrderEntryWebApp**, including modules and data providers, already designed to some extent.

The app design page contains the following elements, from top to bottom:

- **Header** — Similar to the header for the Kendo UI Designer dashboard, with the addition of breadcrumbs that track your path in the Designer. In the example, the design page is displayed at the top-level of its parent app.

- **App Title** — The name you gave the app when you created it (in this case, `OrderEntryWebApp`), followed by a gear menu. The menu provides the following options:
  - **Properties**: This opens the **Edit App** dialog box, where you can change the following app property settings:
    - **Theme**: To change the theme of your app, click the color-coded button and choose from a list of available themes. If you want to create your own custom theme, click the Go to Kendo UI ThemeBuilder link. This opens a tool called the Progress Sass ThemeBuilder which enables you to create custom themes. To import a custom theme, click **Import ThemeBuilder Theme** to import the CSS file for each custom theme you want to add to the list. Select the desired theme and click **Apply** to save your changes.
    - **App Logo**: The logo that appears in the header of the app main (landing) page.
    - **Language Label**: The default language label for the generated application.
    - **Language Culture**: The default language culture for the generated application. The language culture defines the number, date and time formats, etc, in the generated application. To know more about setting language cultures see Localizing the generated app on page 97.
    - **Description**: The description of the app for your own reference (not visible to the app user). This text appears both on the app card or list item on the dashboard and below the app title on the app design page.

    - **Note:** You cannot change the **App Name** or its **Location** after you have created the app.

  - **Roles**: This opens an **Authorization Roles** dialog box that allows you to define the names of user roles that the app can use to authorize access to modules, views, and one or more layout components of a Blank view. You can then specify one or more of these user roles as part of the definition for each module and view, and for specific rows and columns of a Blank view layout. If an authenticated user is authorized for a given role, they are granted access to any module, view, or Blank view layout component that is also defined with that role. To know more about using roles, see Using roles to authorize user access on page 93.

  Below this **App Title** is any description you enter for the app, either when you first create it or using this **Edit App** dialog box.

- **Toolbar** — Provides the following tools:
  - **Generate** — Invokes the Kendo UI Generator to build the current state of the app ready for preview.
  - **Preview** — Either invokes the Kendo UI Generator to rebuild and preview the latest state of the app, or immediately preview the most recent generated build (if one exists). The preview opens in a tab of your default browser using a webpack-dev-server with live data from the data sources mapped to the app views.
  - **Publish** — Invokes the Kendo UI Generator to build a deployment version of the app either for testing (**Debug**) or production (**Release**) in their own respective locations.

  For more information on app builds, see App generation and deployment on page 101.

- **Modules** — A module is basically a container for one or more views. Every app is initially created with a built-in **Application** module that contains built-in views, as shown in the example. You can also create user-created modules that contain user-created views by clicking **Add Module**. In the example, two user-created modules are created, **OrderEntry** and **Module1**.

You can edit some identifying features of every created module by clicking its pencil control. You can add or edit the definitions of user-created views in a user-created module by clicking its **Edit** control, and you can delete a user-created module (with confirmation) by clicking its trash control. You can also rearrange the order of modules on the app design page by dragging and dropping them. For user-created modules, this also changes their order of appearance on both the landing page and views page at run time.

The built-in **Application** module can never be deleted and is created with the following built-in views, which also can never be deleted:

- **login** — Prompts the user for credentials and authenticates access to data providers that require it. This view only appears in the app at run time if one or more data providers require access using an authentication model other than Anonymous. For more information, see Data providers and data sources on page 22.
- **landing-page** — This is the first view to open for an app, and provides a page displaying a labeled icon for every user-created module in the app. You can then select any icon to continue app execution with the selected module (see App layout and components on page 14).

Note: At run time, the **Application** module itself never appears in the web app. Only its built-in views appear in he app according to how you define its user-created modules and views. The **Application** module serves only as a design-time container for these built-in views.

For more information, see Modules and views on page 36.

- **Data Providers** — Data providers define data services and their data sources for binding data to views. Each data provider can define one data service, the authentication model required to access that data service, and one or more data sources from that data service. Each data source represents a single table from its data service. You can create a new data provider by clicking **Add Data Provider**. In the example, two data providers are created, **CustomerService** and **OrderEntryService**.

  You can define data sources for a data provider either when you first create the data provider or by clicking its **Edit** control after you create it, which also allows you to change other properties of the data provider definition. You can delete a data provider (with confirmation) by clicking its trash control.

  For more information, see Data providers and data sources on page 22.

Using the controls on this page, you can create and update most components of a web app, except those that require extension point and source code customization, such as event handlers and custom CSS classes. Most of the remaining topics provide more information on working with these components. For more information on code customization for a web app, see Extension Points and Source Code Customization on page 103.

Data providers and data sources

A data provider defines a single data service and one or more data sources that represent tables provided for access by that data service, which you can bind to views. You can specify a data provider as one of the following:

- **Progress Data Provider** — Represents a single point of authorization with read-only or read-write access to an OpenEdge or Rollbase Data Object Service. This single point of authorization is the URI of a server web application that supports a specified Data Object Service. Each Data Object Service provides access to one or more Data Object resources that provide the tables that define data sources for the data provider. Each Data Object Service is defined by a Data Service Catalog, which is a JSON file on the web server that you can specify using its URI, and which specifies the tables for the Data Object Service. This data provider definition also includes the authentication model required to authenticate access to the data server (web application) that it supports, and which you must specify from the following options:
  - Anonymous

  - Basic
  - Form

- **OData Provider** — Represents a single point of authorization with read-write access to an OData service. This single point of authorization is the URI of an OData service root. An OData service provides access to one or more data sources and their definitions.
- **Generic REST Provider** — Represents a single point of authorization with read-write access to a generic REST service. This single point of authorization is the URI of a server web application that supports a specified REST service.

For a Progress Data or OData Provider, you can create and define data sources that you want the data provider to provide, either automatically, when you first create the data provider, or manually, by adding data sources to the data provider after you create it.

You can also create multiple data providers, especially if you need to access Progress Data Object Services hosted by multiple web applications.

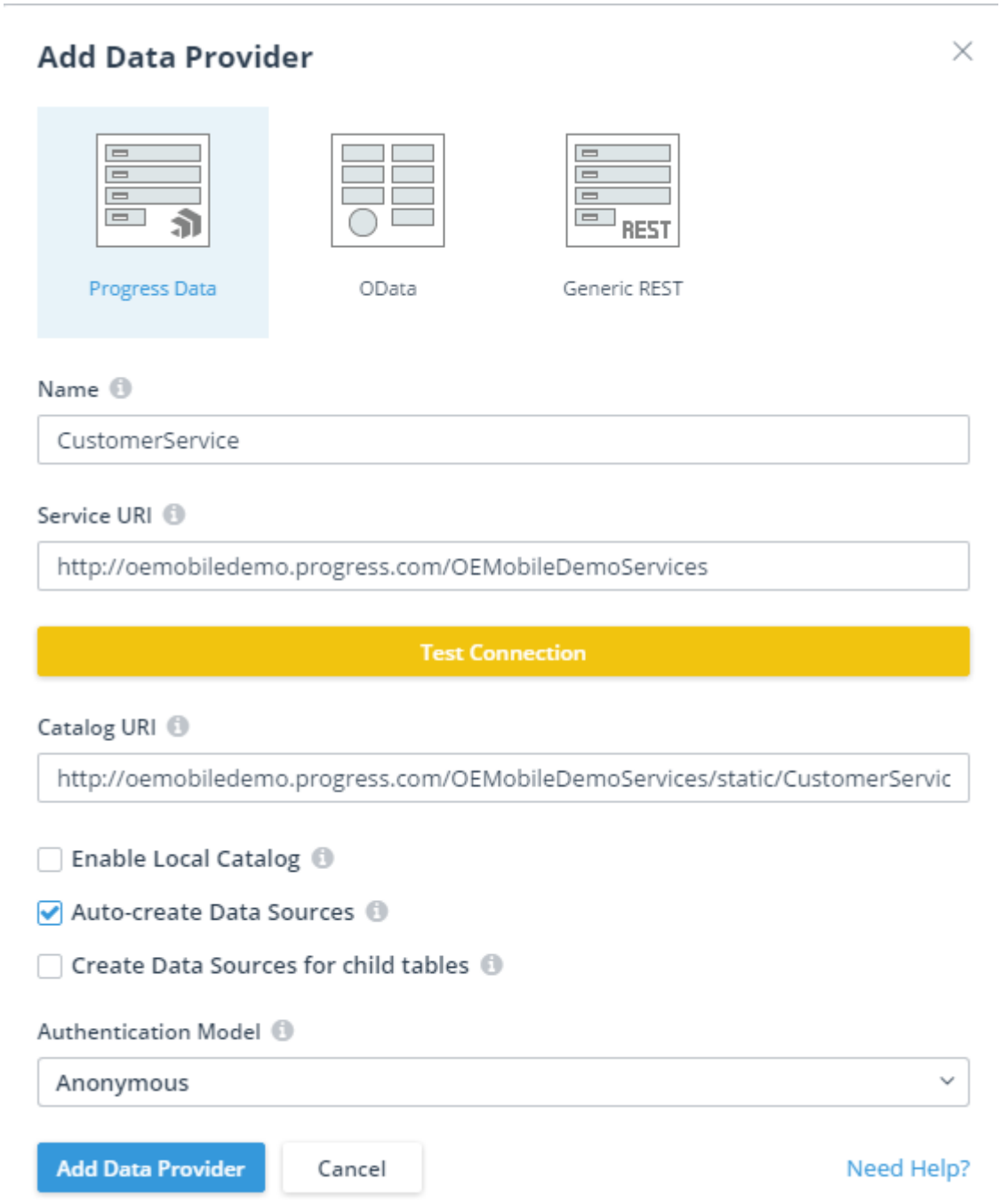
For details, see the following topics:

- Adding and editing a data provider on page 24
- Adding and editing a data source on page 27
- Editor and semantic types on page 31

Adding and editing a data provider

When you create a data provider, by clicking **Add Data Provider** on the app design page, the **Add Data Provider** dialog box displays for you to enter its initial definition, as shown in the following example:

Figure 8: Add Data Provider dialog box



In this example, the data provider is defined as a **Progress Data Provider**, with `CustomerService` entered as its **Name**, with the `OEMobileDemoServices` web application URI specified as its **Service URI**, and the Catalog URI for the single Data Object Service, `CustomerService.json`, specified in the **Catalog URI** field.

Note: If the **Service URI** specifies a secured web application, you must use an unsecured **Catalog URI** to work with it in the Designer. You can either specify an unsecured **Catalog URI** and change it to a secured URI during deployment (see App generation and deployment on page 101), or specify the secured URI for the **Catalog URI** and use a local Catalog for the Designer by selecting **Enable Local Catalog**, which allows you to browse for and select a Catalog file stored on your file system just for development using the Designer. When you deploy the app, it then uses the secured **Catalog URI** setting that you have also specified.

Once the **Add Data Provider** button is clicked, the specified data provider is created with all top-level table resources automatically created as data sources, and with the **Authentication Model** for the data provider specified as `Anonymous`. (The other supported options available for a secured data provider are `Basic` or `Form`.)

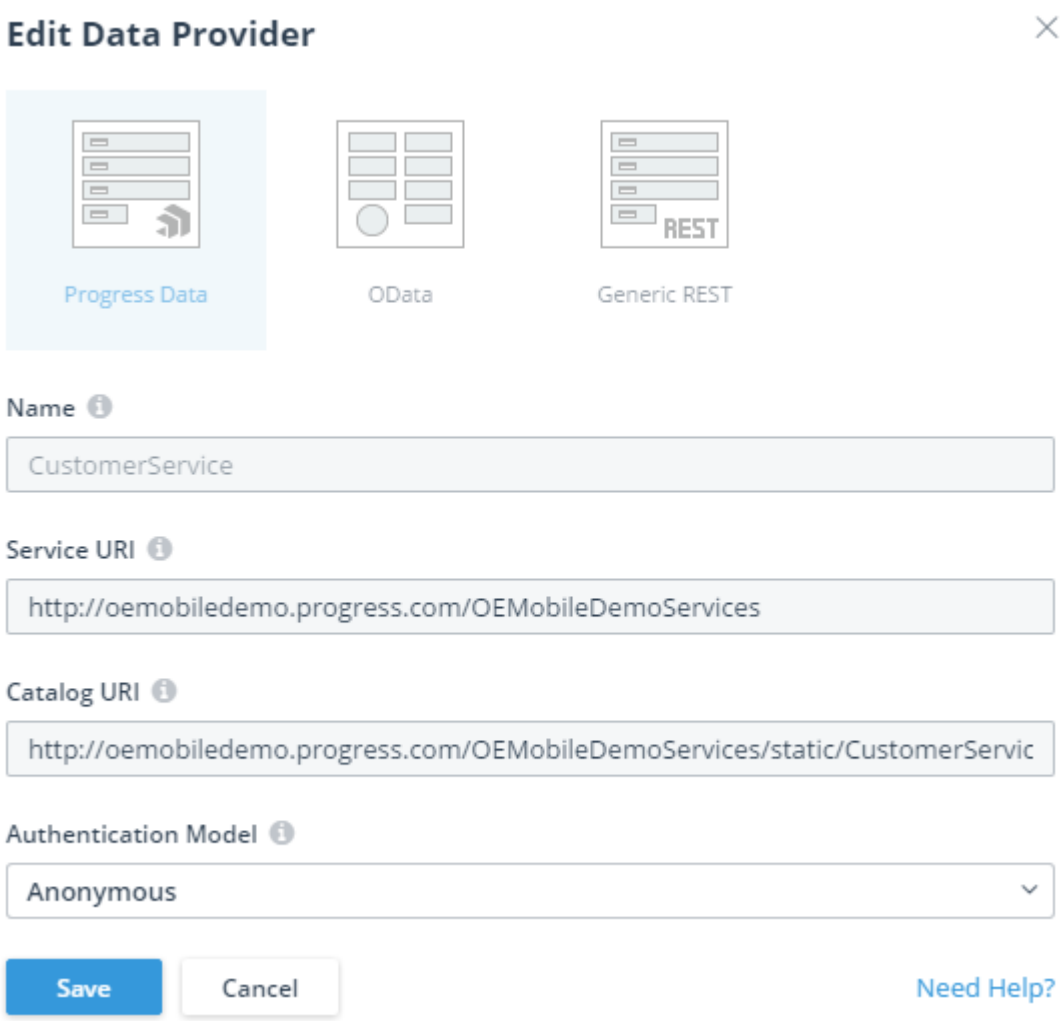
You can otherwise create an **OData Provider** or a **Generic REST Provider** as read-write data providers. Where as, a **Progress Data Provider** can offer read-only or full CRUD data access, with the possibility of transaction support.

Note: For a Progress Data Provider, the **Service URI** field always specifies the URI of the single server component that provides a connection to the Data Object Service whose Catalog URI is specified in the **Catalog URI** field, and that server component is always a single web application running on the web server for an OpenEdge application server. Note also that the authentication model specified in the **Authentication Model** field must be the same as the authentication model configured for the specified web application.

Once you create the data provider, its name appears in the **Data Providers** list as shown on the example app design page (see Creating and designing an app on page 17).

To edit the data provider's properties, such as the authentication model, click the gear icon next to the data provider's name and select **Properties**.

Figure 9: Edit data provider page



Note: After you create a data provider, you cannot change its specified **Provider Type**, **Name**, **Service URI**, or **Catalog URI** settings in the Designer. For production deployment and maintenance of your development and QA environments, you can update the specified web application and Data Service Catalog (or Catalogs) by changing appropriate settings in a JavaScript file generated for every generated app build before you access or deploy the web app in a given environment. For more information, see App generation and deployment on page 101.

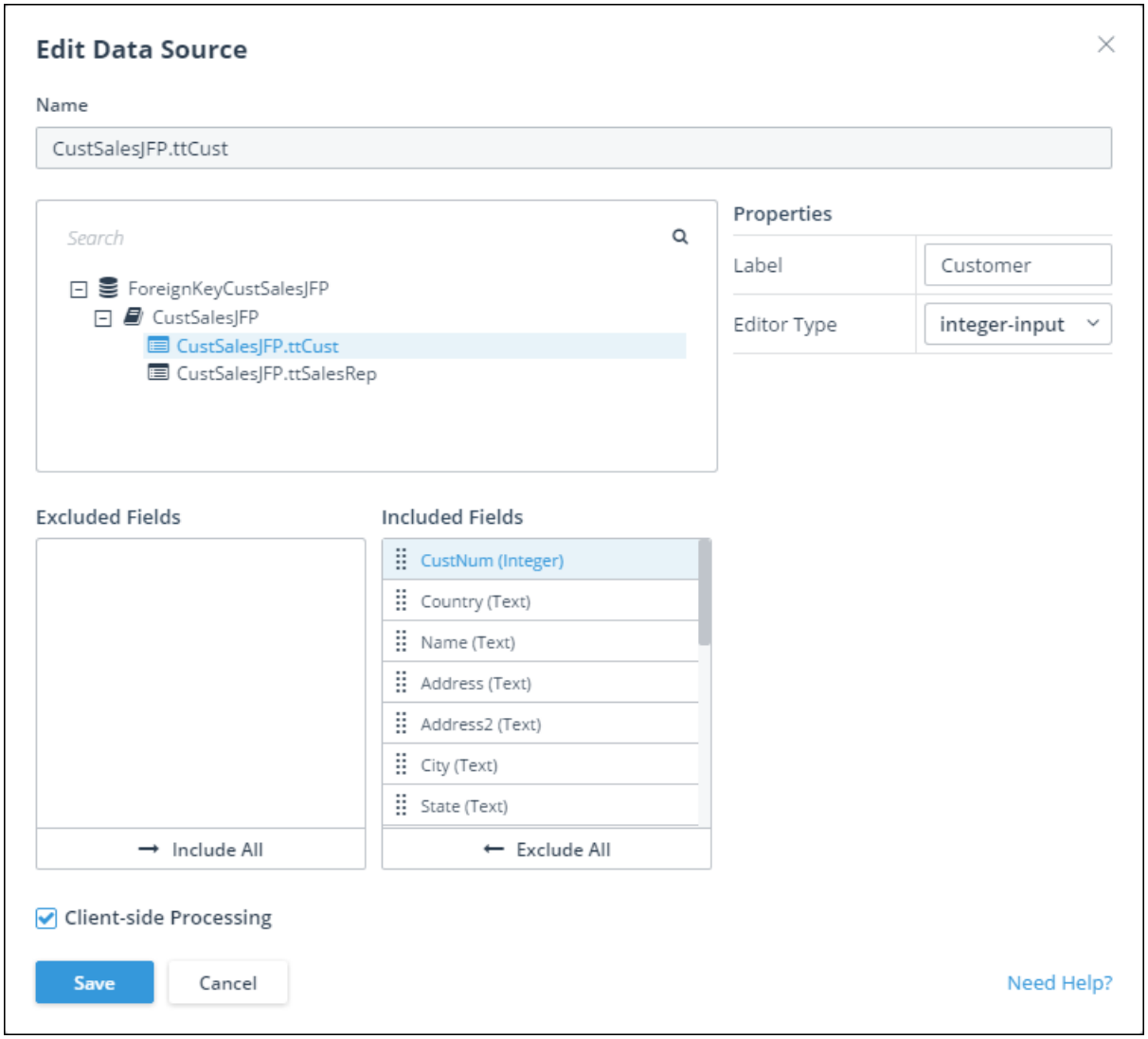
At this point, you might want to review and edit the data sources created for this data provider by clicking the data provider's name in the app design page. Any data sources automatically created for the data provider when you first create it appear in a list under **Data Sources**. If the Data Object Service specified for **Catalog URI** supports additional Data Object resource tables, you can manually create new data sources for them by clicking **Add Data Source**. For more information, see Adding and editing a data source on page 27.

Adding and editing a data source

When you click **Add Data Source** in the edit data provider page (see Adding and editing a data provider on page 24), the **Add Data Source** dialog box appears (not shown, but similar to the **Edit Data Source** dialog box shown below). In this dialog box, unlike when auto-creating data sources, you can select a resource table from the Data Object Service and define your own name for the new data source that is different from the name of the resource table that it represents. You can also set other options similar the **Edit Data Source** dialog box, with some differences as shown and explained, below.

After a data source is created, you can review and modify its definition by clicking the **Edit** control associated with the data source on the edit data provider page. This displays the **Edit Data Source** dialog box, as shown for the auto-created `CustSalesJFP.ttCust` data source in this example:

Figure 10: Edit Data Source dialog box



In the search box, the **Edit Data Source** dialog box highlights the resource table for which the data source is already created; in the **Add Data Source** dialog box, this same search box allows you to search through the available Data Object resource tables to select one for which to create the new data source. However, unlike in the **Add Data Source** dialog box, you cannot change the value specified for **Name** that has already been created for a data source in the **Edit Data Source** dialog box.

The **Excluded Fields** and **Included Fields** lists allow you to exclude or include all fields, and to drag-and-drop individual fields for exclusion from, or for inclusion in, the fields that are initially available to populate views from this data source. You can also rearrange the field order by dragging and dropping fields within a list. The order of fields in **Included Fields** is then the default order for the fields shown in views that you create with this data source.

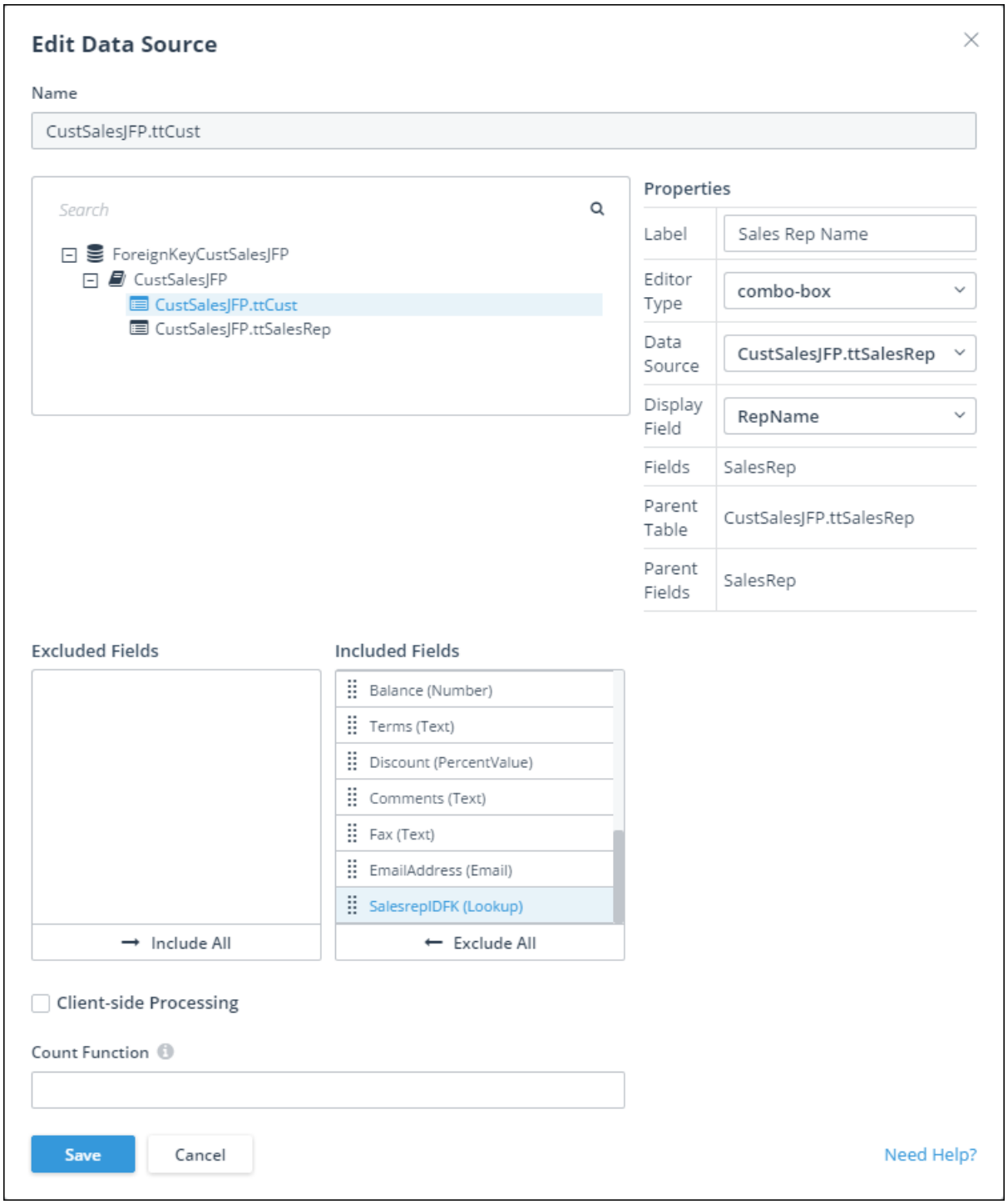
Note: Similar field list settings can also be changed for the individual UI components of a view (see Modules and views on page 36).

The **Properties** of the data source include the ability to define a label (using **Label**) for each field that is different from its field name shown in the **Included Fields** list, as shown for the entered `Customer` label defined for the selected `CustNum` field in the example. Using **Editor Type**, you can also select a UI-independent visualization for each field that is different from the initially displayed default, which for the selected `CustNum` field is changed from `integer-input` to `plain-text`. The options for selecting an **Editor Type** value depend on the field's semantic type shown in parentheses beside each field name in the list. For more information on editor types and semantic types, see Editor and semantic types on page 31.

If **Client-side Processing** is selected (the default), all filtering, paging, sorting, and grouping of fields in a view is managed by the Kendo UI widgets using data that is already retrieved in the client web app. If this check box is cleared, all filtering, paging, and sorting (but not grouping) is managed by the Data Object resource on the server as it responds to read requests from the client.

When you clear **Client-side Processing**, one additional field is displayed that requires a value, as shown in this example:

Figure 11: Setting the data source for filtering, paging, and sorting on the server



In **Count Function**, you must enter the name of an OpenEdge ABL routine in the corresponding Business Entity on the server that implements the Data Object resource. This routine (typically, an ABL class method) returns the total number of records returned by the resource Read operation when server paging is enabled. The data source must know this value in order to manage the paging of records in the client according to the page size of any grid view that is bound to the data source (see Modules and views on page 36). For more information on this routine, see the sections on "Updates to allow access by Kendo UI DataSources and Rollbase external objects" in *OpenEdge Development: Web Services*.

Also, in this example, note that the selected field in **Included Fields** is `SalesrepIDFK (Lookup)`. In this case, the semantic type for this `SalesrepIDFK` field is `Lookup`, which indicates that the field represents a foreign key into another data source where a unique record can be found ("looked up") by matching this foreign key to a primary key in the other (*parent*) data source. In addition to the **Label** and **Editor Type** properties available for all fields, the **Properties** of a foreign key field include:

- **Fields** — Lists the fields of the edited (*child*) data source (`CustSalesJFP.ttCust`) that define its foreign key, in this case `SalesRep`.
- **Parent Table** — Specifies the name of the parent data source table in which to look up the unique record that matches the foreign key, in this case `CustSalesJFP.ttCust`.
- **Parent Fields** — Lists the fields of **Parent Table** that define its primary key, whose values must match the corresponding fields of a given foreign key in order to look up a unique parent record associated with the child data source, in this case also `SalesRep`.

Note: In this release, the **Fields** and **Parent Fields** properties support only a single field to define any foreign or primary key, which happen to have the same name in this example.

You can also select values for the following foreign key **Properties**:

- **Data Source** — An available data source that maps to the specified **Parent Table**, in this case `CustSalesJFP.ttCust`, which also happens to have the same name specified for **Parent Table** in the example. If no more than one compatible data source is available for this mapping, the name of only one data source appears as a possible value.

  Note: The `name` is different in every generated view - it can be primeDS, jsdo resource name, or an auto generated name for **Foreign Key data sources**. This name can be seen in the private controller.

- **Display Field** — The field from the specified **Data Source**, in this case `RepName`, whose value you want to display from the unique record that is found for a given foreign key. This value is displayed in a form using the **Editor Type** that you select for the foreign key field. For more information on the editor types available to display a value for a foreign key (`Lookup` type) field, see Editor and semantic types on page 31.

Note: The Kendo UI Builder supports a built-in foreign key look up and display only for a foreign key field that you add to forms associated with a Data-Grid* view that is defined with forms. This built-in look up is not supported for a foreign key field that you might add to the grid associated with a Data-Grid* view, or that you might add for any UI component of a Blank view. For more information, see Modules and views on page 36.

When you have completed creating or updating a data source definition, you must click **Save** to save the changes and close the dialog box, then click **Save** on the edit data provider page to accept and save the changes to its parent data provider.

Editor and semantic types

A key feature of a data source is that it allows you to specify meta-data to represent each individual field, which defines its UI-independent function and visualization for update or read-only display. This meta-data includes an *editor type*, which identifies the visualization. The options for specifying an editor type depend on a field's semantic type, which also appears in the data source meta-data and is provided by the field's resource definition in the Data Service Catalog. The specified editor type for a field is then used by the Kendo UI Generator to identify the Kendo UI implementation to generate as the field's Kendo UI visualization in view forms (see Modules and views on page 36).

A *semantic type* specifies the functional usage for a field, such as to store currency or date and time values. In the **Add Data Source** or **Edit Data Source** dialog boxes (see Adding and editing a data source on page 27), the semantic type appears in parentheses next to its field name in the data source field list, such as `Integer` in `CustNum(Integer)` or `Text` in `Name(Text)`, as shown in the example.

For an OpenEdge Data Object resource (Business Entity), a semantic type is specified for each temp-table field as part of defining the service interface for the Data Object. This includes a default semantic type that is associated with each supported ABL field data type, if none other is specified. For more information, see the documentation on defining Data Object service interfaces in the *Progress Developer Studio for OpenEdge Online Help*.

The following table lists each supported editor type and its UI behavior, including the Kendo UI visualization created for it in forms by the Kendo UI Generator:

Table 1: Supported editor types and Kendo UI implementations

Editor type	Behavior	Kendo UI visualization
`bool-radio-set`	Describes the visualization and behavior of a boolean field using a choice between two selectable elements.	Boolean Radio Button List
`calendar`	Describes the visualization and behavior of a date field as a calendar control used to select its ISO-8601 value.	Calendar
`check-box`	Describes the visualization and behavior of a boolean field using a single element to reflect two different choices.	Check Box
`combo-box`	Describes the visualization and behavior of a control that is bound to a data source from which it displays a list of corresponding display fields that the user can both select and edit to specify the value of the control.	Combo Box
`currency-input`	Describes the visualization and behavior of a numeric field that represents currency data.	Currency Text Box

Editor type	Behavior	Kendo UI visualization
`date-input`	Describes the visualization and behavior of a date field with an ISO-8601 value.	Date Picker
`date-time-input`	Describes the visualization and behavior of a date and time field with an ISO-8601 value.	Date Time Picker
`drop-down-list`	Describes the visualization and behavior of a control that is bound to a data source from which it displays a list of corresponding display fields that the user can select to specify the value of the control.	Drop Down List
`editor`	Describes the visualization and behavior of a multi-line text field.	Editor
`email-input`	Describes the visualization of a text field that accepts well-formed email addresses.	Email Text Box
`integer-input`	Describes the visualization and behavior of an integer field.	Numeric Text Box
`numeric-input`	Describes the visualization and behavior of a numeric field for all supported numeric domains and formats.	Numeric Text Box
`numeric-slider`	Describes the visualization and behavior of a numeric field for all supported numeric domains and formats entered using a graphic control.	Slider
`password-input`	Describes the visualization and behavior of a text field that accepts a password with masking support.	Password Text Box
`percent-input`	Describes the visualization and behavior of a numeric field that represents a percentage value, where the percentage value is the field's value times 100 (e.g., `0.25 * 100`).	Percent Text Box

Editor type	Behavior	Kendo UI visualization
`percent-value-input`	Describes the visualization and behavior of a numeric field that represents a percentage value, where the percentage value is the field's actual value (e.g., `25.0`).	Percent Value Text Box
`phone-number-input`	Describes the visualization and behavior of a text field that represents a phone number.	Phone Text Box
`plain-text`	Provides the ability to display any semantic type as a read-only value.	Disabled Text Box (HTML5 text element (read-only))
`text-input`	Describes the visualization and behavior of a single-line text field.	Text Box
`url-input`	Describes the visualization and behavior of a text field that accepts well-formed URL values.	Url Text Box

The following table lists each semantic type, its designed behavior, and the available editor types that can represent it, including both the default editor type and compatible alternative editor types that you can select, if any:

Table 2: Default and compatible editor types available for each semantic type

Semantic type	Function	Default editor type	Compatible editor types
`Boolean`	Two (2) values	`check-box`	`bool-radio-set`

`Currency`	Decimal with currency symbol
	With localization override
		`currency-input`	`plain-text`
			`text-input`
			`numeric-slider`

`Date`	Date with no time
	With localization override
		`date-input` (date only)	`plain-text`
			`text-input`
			`calendar`

`Datetime`	Date and time with timezone support
	With localization override
		`date-time-input` (with time)	`plain-text`
			`text-input`
			`calendar`

`Email`	Text with single @ character delimiter	`email-input`	`plain-text` `text-input`

Semantic type	Function	Default editor type	Compatible editor types

`Integer`	Integer value
	With localization override
		`integer-input`	`plain-text`
			`text-input`
			`numeric-slider`

`Internal`	Fields marked as internal are not displayed to the user	—	—
`Lookup`	A value retrieved from a separate data source using a foreign key.	`combo-box`	`drop-down-list` `plain-text`

`Number`	Decimal with formatting options (separator, decimal points, etc.)
	With localization override
		`numeric-input`	`plain-text`
			`text-input`
			`numeric-slider`

`Password`	Text displayed as hidden characters	`password-input`	`plain-text` `text-input`

`Percent`	Decimal x 100 (with % sign)
	Example: value in database = 0.255, represented as 25.5%
		`percent-input`	`plain-text`
			`text-input`
			`numeric-slider`

`PercentValue`	Decimal (with % sign)
	Example: value in database = 25.5, represented as 25.5%
		`percent-value-input`	`plain-text`
			`text-input`
			`numeric-slider`

`PhoneNumber`	Numbers, alpha characters, parentheses, and dashes With localization override	`phone-number-input`	`plain-text` `text-input`
`RichText`	Multi-line, formatted text	`editor`	`plain-text` `text-input`

Semantic type	Function	Default editor type	Compatible editor types
Text	Single line of text **Note:** Data Object Services use UTF-8 as the content type.	text-input	plain-text
URL	Click through hyperlink with alternate text	url-input	plain-text text-input

For OpenEdge Data Object resources, the following table shows the semantic and editor type defaults for each supported OpenEdge ABL field data type:

Table 3: Default semantic and editor types for each ABL field data type

ABL field data type	Semantic type default	Editor type default
CHARACTER	Text	text-input
CLOB	RichText	editor
COM-HANDLE	Number	numeric-input
DATE	Date	date-input
DATETIME or DATETIME-TZ	DateTime	date-time-input
DECIMAL	Number	numeric-input
HANDLE	Number	numeric-input
INT64 or INTEGER	Integer	integer-input
LOGICAL	Boolean	check-box
ROWID	Text	text-input

Note: For more information the support for ABL field data types in Data Object resources, see the information on data type mapping in the *Progress Data Objects: Guide and Reference*.

Note that OpenEdge allows a table field to be defined as a one-dimensional array values with the specified ABL data type, which can be any primitive field data type, exception CLOB. The data source created for an OpenEdge resource table with an array field contains a separate individual data source field for each item in the original ABL array, where each data source field has the same semantic type as defined for the ABL array. This means that in the **Edit Data Source** dialog box, you can set a different compatible editor type for each data source array field, based on its semantic type.

The field name that displays in the data source definition for each data source array field conforms to the following convention:

Syntax:

```
ABLFieldName_idx
```

Where:

ABLFieldName

Specifies the name of the ABL array field from which each data source array field is derived.

idx

Specifies an integer that is the 1-based index of the ABL array item that this data source array field represents.

Note: This corresponds to OpenEdge ABL array indexes, which are always 1-based.

As with any data source field, you can specify a more useful label for each data source array field than the original ABL field name and *idx* value represent. For example, for a field named `Prize_1`, you might specify the label, `First Prize`.

When the Kendo UI Designer generates the UI for data source array fields, it then displays each field derived from the ABL array as a separate field in any view. This means that it creates a separate column in any grid for each array field, and creates a separate UI component for each array field in any editable form according to the individual array field's editor type. This means, for example, that an array of semantic `Text` fields can have all odd-numbered fields display as editable (`text-input` editor type) and all even-numbered fields display as read-only (`plain-text` editor type).

Modules and views

In a Kendo UI Designer web app, the *module* is the basic unit of application functionality. Each module contains one or more views that provide the functionality, typically for a common set of features. Therefore, a *view* provides the UI for a single application function or feature within a module.

As described in Creating and designing an app on page 17, when you first create a web app in the Designer, it creates one built-in **Application** module. You can then create as many additional user-created modules as you require to organize the features of your app.

The Designer supports the following module and view configurations:

- **Application module** — Created with two built-in views, the **login** and the **landing-page** view (see Creating and designing an app on page 17).

  At design time, you can edit properties of the login view on a *view design page*. However, the landing-page view has editable properties only for changing the default names of general event handler functions, which can also be changed for all views (see General view events on page 115). Otherwise, the run-time behavior of the landing-page view largely depends on the user-created modules that you create for an app (see App layout and components on page 14).

- **User-created modules** — Created with no default views, you can create as many user-created view instances as you need from a set of supported *view templates*. You can create two types of user-created views: predefined views and user-defined views. A *predefined* view has a responsive layout and content that is defined entirely by its template. Currently, the Kendo UI Builder supports three Data-Grid* templates

for predefined views that all contain a single grid or a combination of a single grid and form whose content depends on the data bound to the view. A *user-defined* view supports a responsive layout and content that you define entirely from components you drag-and-drop on a view design page. Currently, the Kendo UI Builder supports a single Blank template for defining user-defined views.

To display and update data in a user-created view instance, you must specify at least one data resource instance from any available data provider in order to bind data to the view and its components. The predefined Data-Grid* view templates allow you to specify a single data source per view instance and both the display and update of data in the view can be implemented with little or no additional programming.

The user-defined Blank view template allows you to specify multiple data source instances per view instance. Each data-bound content component can be bound, either directly or indirectly, to a single data source from the those you have specified for the view. However, for the Blank view, the display and update of data in the view requires additional programming, depending on the content components it contains and their data binding.

The six view templates that you can use to create views in a user-created module include:

- **Data-Grid** — This view is a predefined read-only or editable grid that displays rows of records on the app views page from a single view data source instance you specify on the view design page. If it is editable, this view provides a design-time choice of three edit modes for editing the grid either one row one at a time or one page of rows at a time.
- **Data-Grid-Form** — This view includes a predefined read-only grid, similar to the Data-Grid in read-only mode, and also includes a form with a design-time choice of two edit modes: a read-only or read-only-to-edit form that displays with the grid in a single split screen on the app views page. This form allows you to display and update fields from a single view data source instance you specify according to the editor type defined for each field in the data source. This includes built-in support for look-up editor types that enable foreign key access to field values in a parent data source. Note that for foreign key support, both the view data source and the parent data source used for look up must represent top-level tables in their respective Data Object resource (or resources), and **cannot** be part of a parent-child data-relation defined for any single Data Object resource.
- **Data-Grid-Separate-Form** — This view includes a read-only grid, similar to the Data-Grid in read-only mode, and also includes a form with a design-time choice of three edit modes: a read-only, directly editable, or read-only-to-edit form that overlays the grid in a separate screen on the app views page. This form allows you to display and update fields from a single view data source instance you specify according to the editor type defined for each field in the data source. This includes built-in support for look-up editor types that enable foreign key access to field values in a parent data source. Note that for foreign key support, both the view data source and the parent data source used for look-up must represent top-level tables in their respective Data Object resource (or resources), and **cannot** be part of a parent-child data-relation defined for any single Data Object resource.
- **Stacked-Data-Grids** — This is a parent-child view that displays two grids on a page, where the first grid displays records from a parent data source and the second grid displays records from a related child data source directly below the parent grid. The parent grid is read-only. When the user selects a parent row, the related records from the child data source are displayed in the rows of the child grid, which can be either read-only or editable, similar to the Data-Grid view. This view allows the user to display and edit child rows for only one parent row at a time. Note that the parent and child data sources accessed by this view **must** be part of a parent-child data-relation defined for a single Data Object resource.
- **Hierarchical-Data-Grid** — This is a parent-child view that displays a single hierarchical grid structure on a page containing both parent and child rows. Initially, the grid displays only rows from a parent data source, which are read-only. When the user clicks an expander on any parent row, a child grid opens indented under the parent row and displays records from the related child data source. The user can individually expand multiple parent rows, which allows the user to display multiple child grids under their respective parent rows at one time. The child grids can all be either read-only or editable, similar to the Data-Grid view. For editable child grids, the user edits and saves row changes for only one child grid at

a time. Note that the parent and child data sources accessed by this view **must** be part of a parent-child data-relation defined for a single Data Object resource.

- **Blank** — This is a user-defined view that allows you to define both its layout and content by dragging and dropping a variety of components, including layout (rows and columns), data management (e.g., Grid), editor (e.g., Text Box), scheduling (e.g., Calendar), charting (e.g., Bar Charts), navigation (e.g., Toolbar), and custom HTML components. The layout is based on a Bootstrap fluid grid system that you can define initially with row and column components. The content consists of individual UI components, many of which you can bind to data. You can add these components either to existing rows and columns or directly to any existing row, which implicitly creates a column for that component if necessary.

  The Blank view thus allows you to build a custom view that can be a variation on the predefined Data-Grid* views or something completely different, such as a stand-alone form with data navigation and without any associated grid. Unlike the Data-Grid* views, the Blank view allows you to bind multiple data sources from one or more data providers. You can then bind each content component in the view to any one of these data sources. To complete this data binding for a Blank view, you must code event functions as required for the content behavior you want in a JavaScript file that the Kendo UI Builder generates for the view.

Note: The editor components for a Blank view include Combo Box, Drop Down List, and Plain Text components that correspond to the editor types that support foreign key look-ups for Data-Grid views with forms. However, unlike in Data-Grid views with forms, this foreign key look-up support is not built-in for the corresponding editors of the Blank view. You must, instead, add custom code to implement a foreign key look-up for a Blank view. For an example, see Adding and editing a Blank view on page 66.

The design-time properties available to customize Data-Grid* view instances are all very similar from one view template to the next, with additional properties added for the more complex views with forms and parent-child views. In addition, for those views with forms or parent-child grids, only one design-time property setting is required to change from one supported edit mode to another.

The design-time properties and code to customize a Blank view instance depend entirely on the view content components and data sources that you use to define and bind data to the view. Some properties are available in many or all of the components, but many are unique to a subset of components.

Note that Kendo UI Builder supports two kinds of templates: built-in templates and custom templates. The *built-in templates* are installed and available for any project that you create in the Designer. Customization of built-in templates is limited to your settings of installed properties in the Designer, event function code that you write for existing events, and custom HTML sections that you code as prescribed for each built-in template.

Custom templates include both *custom view templates*, which are based on the five predefined view templates described here, and *custom component templates*, which are based on a supported selection of built-in components, and which you can use as part of the built-in Blank view template. You can customize the source code for custom templates in any way that works for your organization and use them in the Designer much like the built-in templates, but based on your modified source code. Note that unlike built-in templates whose features are available in every project, these custom templates are project based. This means that any source code customizations you make apply only to the web app project where they reside.

You can customize views that you create from both built-in and custom templates with additional code and settings at a number of extension points in your Kendo UI Builder installation and web app folders, including AngularJS files for coding view event functions and custom HTML files for coding custom HTML sections. For more information, see Extension Points and Source Code Customization on page 103.

For more information on accessing and editing the source code for custom templates, see Custom templates on page 106.

The following topics provide a more detailed overview of both the built-in login view and the user-created views and their templates. The description of each view builds to some extent on the next to provide a comparative overview of their capabilities.

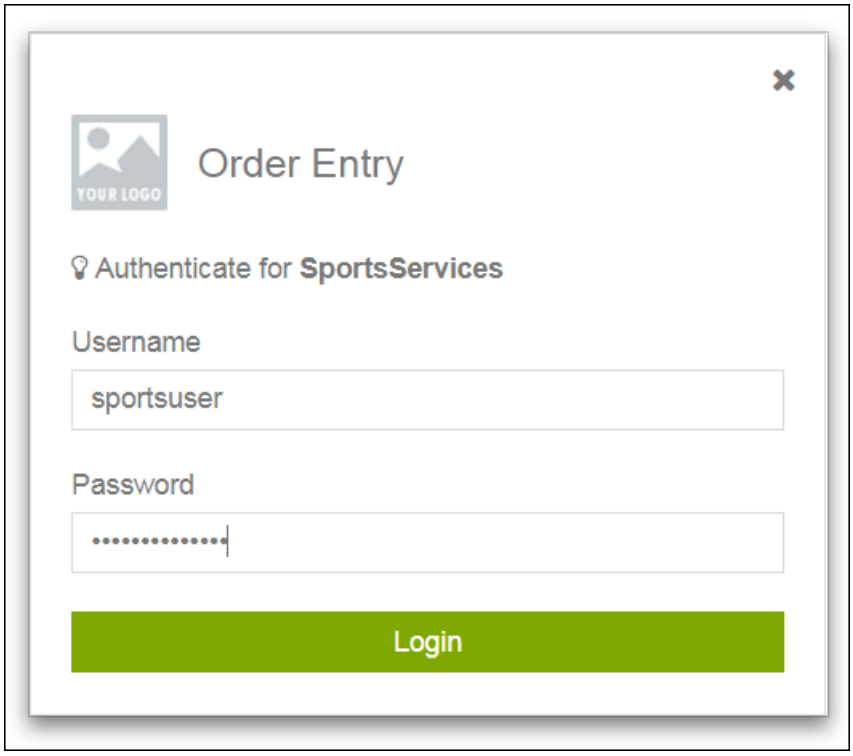
For details, see the following topics:

- Editing the login view on page 39
- Adding and editing a Data-Grid view on page 42
- Adding and editing a Data-Grid-Form view on page 49
- Adding and editing a Data-Grid-Separate-Form view on page 57
- Adding and editing a Blank view on page 66

Editing the login view

After running a web app that you create in the Kendo UI Designer, if any views you select are bound to a data provider that requires an authentication model other than Anonymous, the built-in **login** view in the **Application** module prompts the user to enter credentials, like this customized **OrderEntry** example:

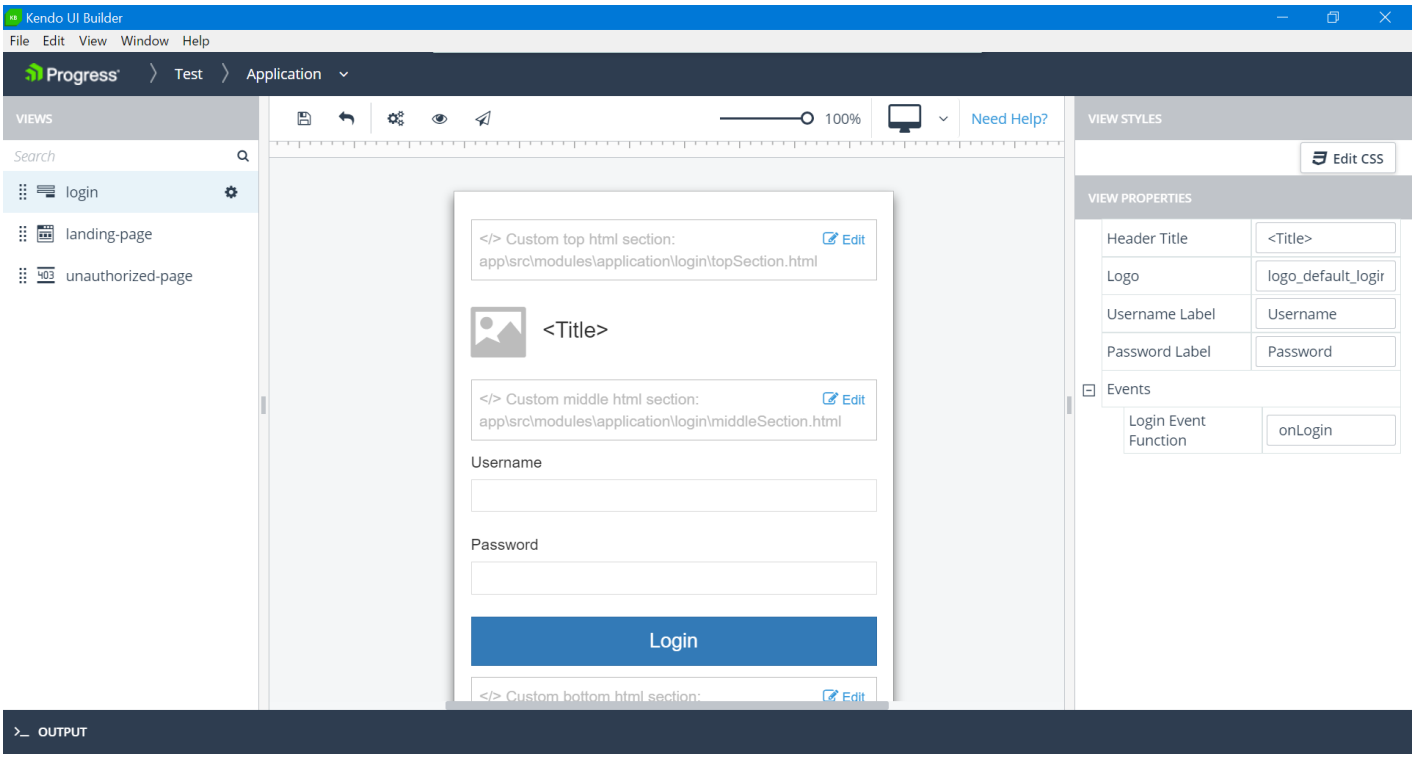
Figure 12: Login view running in app



After entering credentials and clicking **Login**, the app authenticates all data providers used by the app that require authentication. The login only succeeds if all such data providers have authenticated successfully using these same entered credentials.

Like any view, you can customize several of its features by modifying properties in the Kendo UI Designer. To customize the login view, edit the **Application** module. The *view design page* for the built-in login view opens similar to this example (with some properties already set):

Figure 13: Login view design page



The login view design page shows certain features common to all view design pages, top-to-bottom and left-to-right:

- **Header** — Similar to the header for the app design page, with the addition of breadcrumbs that track your path in the Designer to the current view design page.
- **Toolbar** — Provides the following tools:
  - **Save** — Saves any unsaved changes in the current view definition to the UI meta-data.
  - **Revert** — Cancels any unsaved changes and returns the current view definition to its state as of the most recent **Save**.
  - **Generate** — Invokes the Kendo UI Generator to build the current state of the app ready for preview.
  - **Preview** — Either invokes the Kendo UI Generator to rebuild and preview the latest state of the app, or immediately preview the most recent generated build (if one exists). The preview opens in a tab of your default browser using a webpack-dev-server with live data from the data sources mapped to the app views.
  - **Publish** — Invokes the Kendo UI Generator to build a deployment version of the app either for testing (**Debug**) or production (**Release**) in their own respective locations. For more information on app builds, see App generation and deployment on page 101.
  -  — Selects a device whose view you want to simulate in the view design page and which resembles the view as displayed on the physical device at run time. The displayed icon reflects the selected device from these supported devices: **Desktop** (default), **Laptop**, **Tablet landscape**, or **Tablet portrait**.

Note: The run-time preview that you open from the Designer using **Preview** always displays according to the physical device where you are running the Designer.

- **VIEWS** pane (in panel on the left) — Lists the views in the current module, which for the **Application** module include the built-in **login** and **landing-page** views.

  Note: For a user-created module, there is also an **Add** button for creating additional user-created views, as shown in following topics.

- ***View design panel*** **(in the middle)** — Shows a design simulation of the view, bounded on the left and top with a vertical and horizontal ruler. The view design panel contains the following elements that are common to both the built-in login and predefined views, but which might contain different content for each view type:

  - **Custom top html section** — Initially not implemented, this custom section appears in the view when you code the content of an HTML `<div>` element that displays at the top of the view.
  - **Header section** — In this case, an identifying title for the view (**Header Title** property) set to `OrderEntry` and a graphic image file (**Logo** property) set to the default, `logo_default.png`.

    Note: In the running example of this login view (shown above), no **Logo** image seems to appear because the graphic in the `logo_default.png` file is white in order to stand out in the black page header of the running web app (see App layout and components on page 14).

  - **Custom middle html section** — Initially not implemented, this custom section appears in the view when you code the content of an HTML `<div>` element that displays in the middle, between the header and data sections of the view.
  - **Data section** — A form containing username and password input fields with labels (**Username Label** and **Password Label** settings) that you can customize, and a **Login** button that the user must press to authenticate and login to the data provider bound to the selected view.
  - **Custom bottom html section** — Initially not implemented, this custom section appears in the view when you code the content of an HTML `<div>` element that displays at the bottom of the view.

  For more information on coding custom HTML sections, see Custom HTML sections on page 122 in this document.

  Note: The **landing-page** view has only two custom sections, **Custom top html section** and **Custom bottom html section**, as the middle is populated with the icons for whatever modules are created for the app.

  The design simulation in a view design panel changes as you modify some of the view properties that affect its appearance. For example, the **login** view title is set to **OrderEntry** using the **Header Title** property (see the **VIEW PROPERTIES** pane, described further as follows).

- **VIEW PROPERTIES pane (in panel on the right)** — Contains all the properties that you can set that change the design-time appearance and content of the view, as well as some settings that can affect view behavior, such as view-specific event handler settings.

  Additional properties of note include:

  - **Edit CSS** — Enables you to add and edit CSS styles for the view. clicking the **Edit CSS** button opens the view's `style.css` file in a Monaco editor. The `style.css`file is located in

application-folder\app\src\modules*module-folder**view-folder*, where *application-folder* is the path to the folder that contains and is named for your web app, *module-folder* is the name of the folder defining the module, and *view-folder* is the name of the folder defining the view.

Note: To apply CSS classes (defined in the *style.css* file) to the view, you must specify the class names in the **CSS Class List** property in the **VIEW PROPERTIES** pane.

- **Events** — Provides one or more properties to change the default name of the event handler function defined for each login view-specific event:
  - **Login Event Function** — Default value: `onLogin`. Executes for the `Login` event, which fires when the **Login** button is pressed.

Note: You can also use properties in the **Edit View** dialog box to change the default names of event handler functions for general events that apply to all views. You can access this dialog by clicking the drop-down menu for each view (as shown for the example **login** view listed in the **VIEWS** pane), then clicking the **Edit** option.

Caution: You must ensure that any change to the default name of an event handler function that you make using its view property you also make to the name of the actual JavaScript function in source code.

You must add custom code to each view event handler function in order to implement any useful behavior for it. The default behavior of these functions has no functional effect. Typically, you do not need to change the default names of event handler functions. However, the view properties exist to allow this name change if you have the need (for example, to avoid a naming conflict with existing JavaScript code you are using elsewhere in the app). For more information on coding both view-specific and general event handler functions (and changing their names in the source), see General view events on page 115 and View-specific events on page 119 in this document.

Adding and editing a Data-Grid view

The Data-Grid view is a read-only or editable grid that displays rows of records on the app views page from a single view data source instance you specify on the view design page. If it is editable, this view provides a design-time choice of three edit modes for editing the grid either one row one at a time or one page of rows at a time.

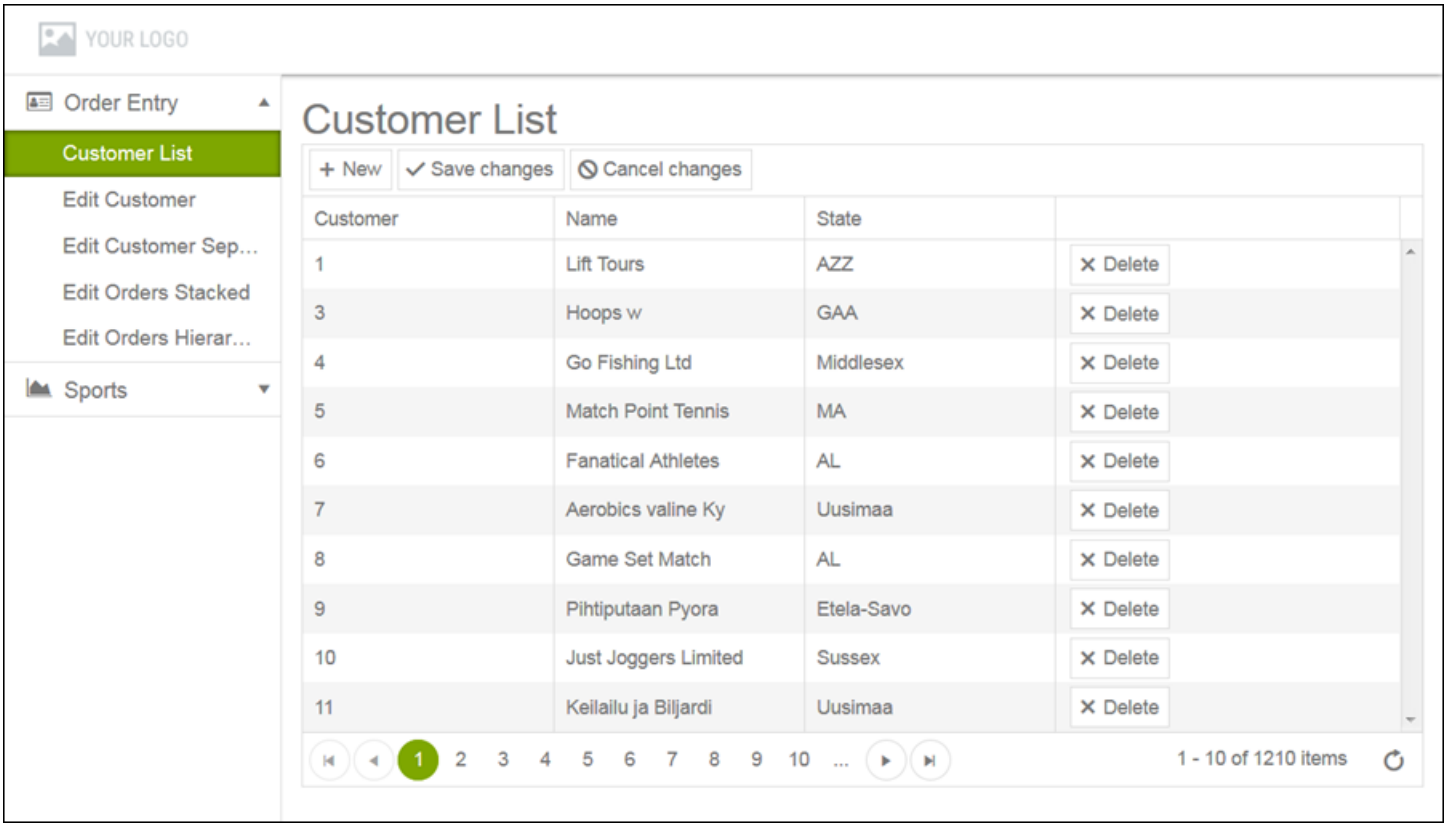
At run time, the grid initially displays by itself with no records selected, depending its edit mode. (For read-only mode, row selection in this grid has no built-in function except to highlight the selected row.) You can then navigate the rows of the grid a page at a time using a page selection control at the bottom of the grid. If the grid is editable, you can then edit the grid either one row at a time or multiple rows at a time on the current page, with a UI appropriate for the particular edit mode that is defined at design time

At design time, you can also customize what columns are displayed from the fields of the data source, as well as other properties that affect the display of the grid and its data. For example, you can enable design-time options to select the data source fields to display and edit in grid columns, modify the grid page size, filter the rows by column at run time, and sort the rows by column at run time.

This grid view provides the same basic capabilities that appear in grids displayed for other Data-Grid* views, some of which are read-only or editable, depending on the view and the grid that appears in the view. For example in Data-Grid* views with forms, the single grid is read-only and provides appropriate built-in behavior for displaying and optionally editing the columns of a single row in a form when you select the row. In the parent-child views, the parent grid is read-only and the child grid (or grids) can be defined as read-only or editable exactly like this Data-Grid view.

When you select an instance of this Data-Grid view in a module at run time, the app displays a page similar to the view labeled **Customer List** in this example:

Figure 14: Data-Grid view running in app



This particular view displays editable rows of records from its bound data source in a grid, showing field values for selected columns of each record. The rows are displayed in pages with a design-time specified size. You can navigate through the pages of the grid view by selecting the first, previous, specific, next or last page control.

In this case, the grid is defined with the **Incell** edit mode (described later in this section), which allows you to edit any and all writable columns in all rows displayed on the current page of the grid simply by clicking into a particular column on the page and modifying its contents. Buttons are also available to add and edit new rows (**New**), delete existing rows (**Delete**), and either save all changes (**Save changes**) or cancel all changes (**Cancel changes**) to the rows on the current page.

Note: For **Incell** edit mode to save changes for multiple rows at a time, the OpenEdge Data Object resource that underlies the data source bound to this view must be defined with a Submit operation to handle all write requests to the server. For more information, see the information on Submit operation support for Data Object Services in *OpenEdge Development: Web Services* and *Progress Developer Studio for OpenEdge: Online Help*. In addition, see *OpenEdge Getting Started: New Information* documentation that describes support for transactional Submit in OpenEdge 11.7 service packs.

If this view was defined with one of the other available modes for editing, selecting a row at run time allows you to edit the columns of the row, either directly within the selected row by clicking into the row's columns (**Inline** edit mode) or within a pop-up form that displays the columns of the row for editing using the editor type defined for each field (**Popup** edit mode). Corresponding buttons (not shown) are also displayed for each of these edit modes to add (**New**), edit (**Edit**), delete (**Delete**), and save (**Update**) or cancel (**Cancel**) changes to each row.

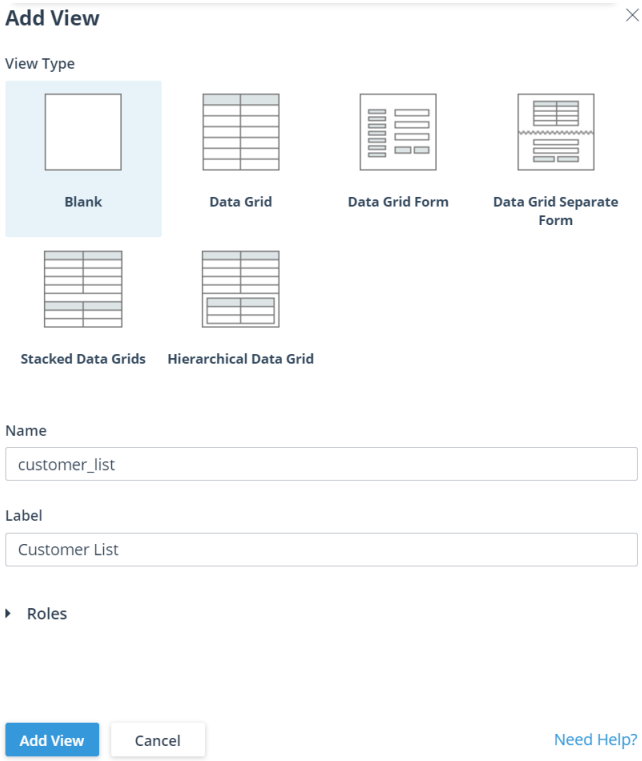
Note: Unlike the forms associated with the Data-Grid* views with forms, the pop-up form displayed for **Popup** edit mode does not support built-in look-ups using foreign key fields.

If this view was defined as read-only (**ReadOnly** edit mode), none of the editing buttons (such as **New** or **Delete**) are displayed, and selecting any single row highlights the row, but has no other default function.

To add a Data-Grid view to a user-created module, edit the module, which opens a view design page in the module (see Figure 16: Data-Grid view design page on page 45 for an example), then click **Add View** at the top of the **VIEWS** pane.

This opens an **Add View** dialog box, similar to this example:

Figure 15: Add View dialog box creating a Data-Grid view

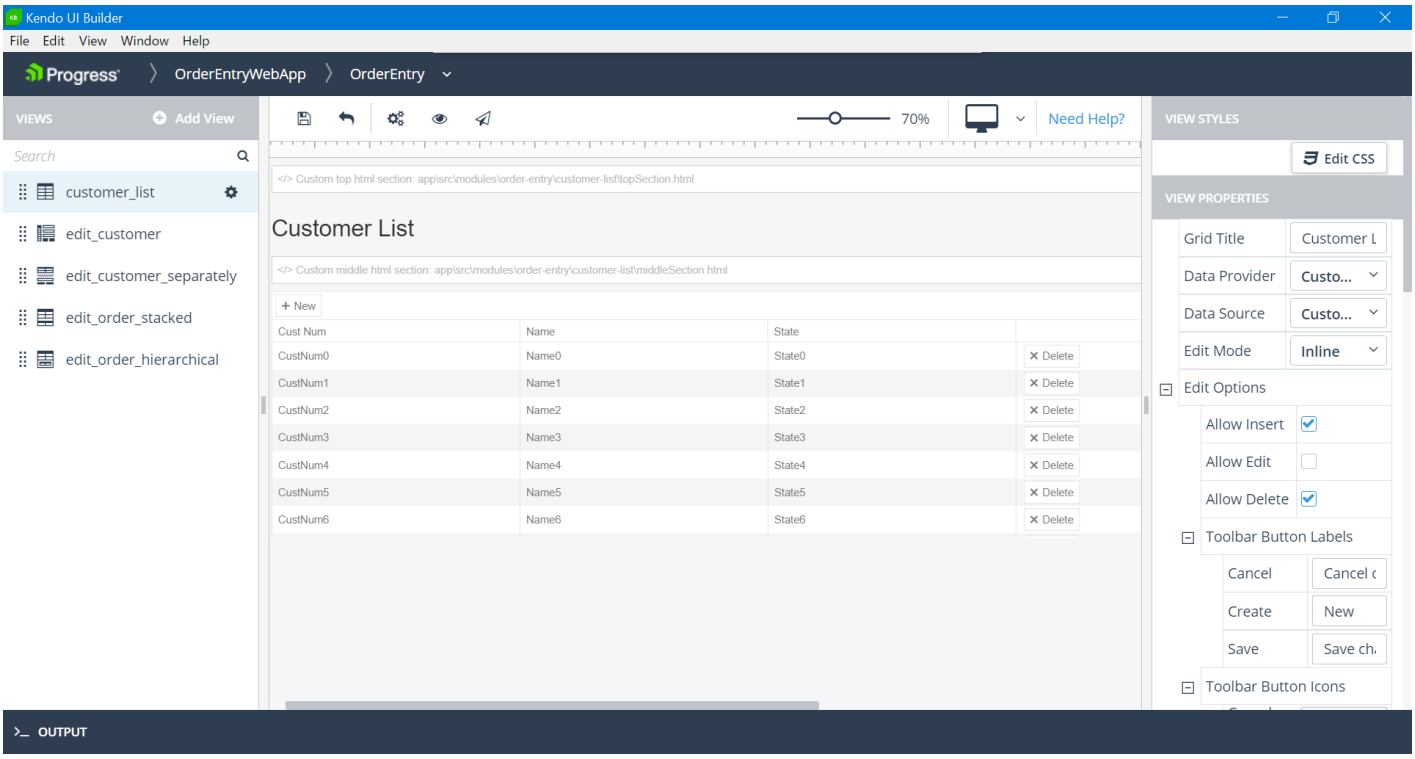


In this example, the dialog box has `customer_list` entered as the value of the view **Name**, `Customer List` entered as the value of the view **Label**, and **Data Grid** selected as the **View Type**. Note that the **Name** value cannot contain spaces to separate its components (along with other restrictions) and can never be changed once you create the view; it identifies the view both in the Designer view design page and in internal code generated for the view. However, the **Label** value can contain spaces and can be changed at any time; it identifies the view in the app run-time layout. The view types listed in this dialog box (shown with a corresponding icon) identify the available built-in view templates you can use to add a user-created view.

The **Roles**tab opens an **Authorization Roles** dialog box that allows you to select the user roles that the app can use to authorize access to this view. To know more about roles, see Using roles to authorize user access on page 93.

After specifying the **Name**, **Label**, and **View Type**, click **Add View** to create the specified view and display its view design page for editing, as shown for the **customer_list** Data-Grid view in this example:

Figure 16: Data-Grid view design page



The Data-Grid view design page shows certain features common to all predefined Data-Grid* view design pages, top-to-bottom and left-to-right:

- **Header** — Similar to the header for the app design page, with the addition of breadcrumbs that track your path in the Designer to the current view design page.
- **Toolbar** — Provides the following tools:
  - **Save** — Saves any unsaved changes in the current view definition to the UI meta-data.
  - **Revert** — Cancels any unsaved changes and returns the current view definition to its state as of the most recent **Save**.
  - **Generate** — Invokes the Kendo UI Generator to build the current state of the app ready for preview.
  - **Preview** — Either invokes the Kendo UI Generator to rebuild and preview the latest state of the app, or immediately preview the most recent generated build (if one exists). The preview opens in a tab of your default browser using a webpack-dev-server with live data from the data sources mapped to the app views.
  - **Publish** — Invokes the Kendo UI Generator to build a deployment version of the app either for testing (**Debug**) or production (**Release**) in their own respective locations. For more information on app builds, see App generation and deployment on page 101.
  -  — Selects a device whose view you want to simulate in the view design page and which resembles the view as displayed on the physical device at run time. The displayed icon reflects the selected device from these supported devices: **Desktop** (default), **Laptop**, **Tablet landscape**, or **Tablet portrait**.

Note: The run-time preview that you open from the Designer using **Preview** always displays according to the physical device where you are running the Designer.

- **VIEWS pane (in panel on the left)** — Lists the views in the current module, including the example **customer_list** view. There is also an **Add View** button (used to create this view) for creating additional views and a search box for locating a view in a long list of views in the module.
- ***View design panel*** **(in the middle)** — Shows a design simulation of the view, bounded on the left and top with a vertical and horizontal ruler. The view design panel contains the following elements that are common to all predefined Data-Grid* views, but which might contain different content for each predefined view type:
  - **Custom top html section** — Initially not implemented, this custom section appears in the view when you code the content of an HTML `<div>` element that displays at the top of the view.
  - **Header section** — In this case, an identifying title for the view (**Grid Title** setting shown below).
  - **Custom middle html section** — Initially not implemented, this custom section appears in the view when you code the content of an HTML `<div>` element that displays in the middle, between the header and data sections of the view.
  - **Data section** — In this case, showing a design simulation of the grid as currently configured in the example Data-Grid view instance for display.
  - **Custom bottom html section** — Initially not implemented, this custom section appears in the view when you code the content of an HTML `<div>` element that displays at the bottom of the view.

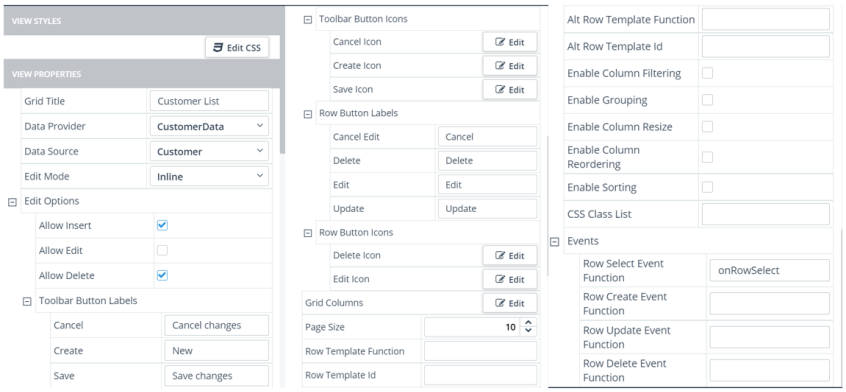
  For more information on coding custom HTML sections, see Custom HTML sections on page 122 in this document.

  The design simulation in a view design panel changes as you modify some of the view properties that affect its appearance, and even as you click around in the simulated view design panel, depending on these property settings. For example, the **customer_list** view grid title is set to `Customer List` using the **Grid Title** property and there are ten (`10.00`) rows on each page of the grid as set for the **Page Size** property (see the **VIEW PROPERTIES** pane description, below, for more information).
- The **VIEW STYLES pane (in panel on the right)** — provides an **Edit CSS** button that enables you to add and edit CSS styles for the view. Clicking the **Edit CSS** button opens the view's `style.css` file in a Monaco editor. The `style.css`file is located in *`application-folder`*`\app\src\modules\`*`module-folder`*`\`*`view-folder`*, where *`application-folder`* is the path to the folder that contains and is named for your web app, *`module-folder`* is the name of the folder defining the module, and *`view-folder`* is the name of the folder defining the view.

Note: To apply CSS classes (defined in the *`style.css`* file) to the view, you must specify the class names in the **CSS Class List** property in the **VIEW PROPERTIES** pane.

- **VIEW PROPERTIES pane (in panel on the right)** — Contains all the properties that you can set that change the design-time appearance and content of the view, as well as some settings that can affect view behavior, such as view-specific event handler settings.

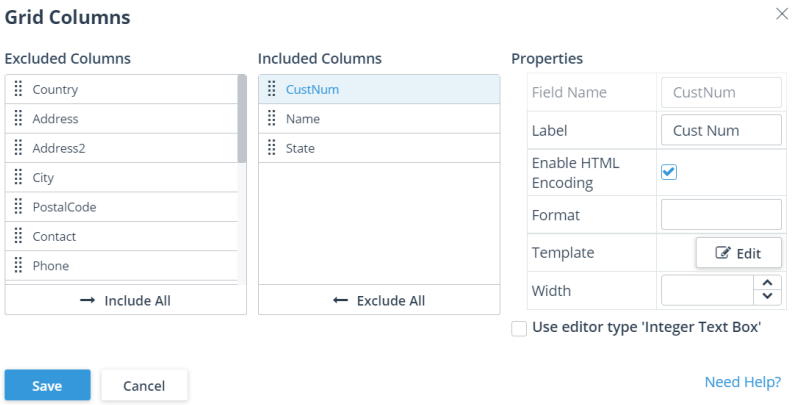
  Figure 17: Data-Grid view properties



  Additional properties and values of note include:

  - **Data Provider** and **Data Source** — Allow you to select an available data provider and data source instance to bind to the view. For more information, see Data providers and data sources on page 22.
  - **Edit Mode** — Allows you to define the grid as either read-only or editable using one of the following modes as described above: **ReadOnly**, **Incell**, **Inline**, or **Popup**.
  - **Edit Options** — For editable grids, these options allow you to customize the editing features available in the selected edit mode (whether adding new rows (**Allow Insert**), editing existing rows (**Allow Edit**), or deleting existing rows (**Allow Delete**) is allowed) and the labels for the buttons provided both in the editing toolbar (**Toolbar Button Labels**) and for each row (**Row Button Labels**), depending on the edit mode. The defaults for these options are shown.
  - **Grid Columns** — Clicking **Edit** for this property opens a **Grid Columns** dialog box that allows you to specify what data source fields appear as columns in the grid, and some features affecting how each field is displayed in the grid:

    Figure 18: Grid Columns dialog box



    As shown in this example, three data source fields are specified in the **Included Columns** list for display as columns in the grid, `CustNum`, `Name`, and `State`. You can also change the order of the displayed columns by dragging-and-dropping fields within **Included Columns**. All fields not displayed as columns in the grid are moved to the **Excluded Columns** list. You can move these fields between lists individually using drag-and-drop or move all fields from one list to the other by clicking **Include All** or **Exclude All**

The current **Properties** apply to the field that you select in the **Included Columns** list, as shown for the `CustNum` field. **Enable HTML Encoding** specifies whether HTML coding included in the field value applies to the column's displayed value. **Format** allows you to specify a Kendo UI notation for adding additional text to the column value or changing the display format of the column value, depending on its field data type. See the Kendo UI documentation for more information, for example, Number Formatting. **Template** allows you to customize the display of a given column using a Kendo UI column template that you can specify. For more information, see Column templates on page 125 in this document. **Label** allows you to specify or change the displayed column label. **Width** allows you to specify the initial column width in pixels; otherwise, the column width is responsive as needed. The **Use editor type <editor name>** property enables you to set a semantic type editor for the field. Kendo UI Designer assigns a default editor for every field based on the field's base data type or ABL data type. However, you can choose a semantic type editor instead by selecting this checkbox, which enables you to configure a few additional properties for the field.

- **Page Size** — Specifies the number of rows to display in each page of the grid. (The last page can have fewer rows, depending on the total number of records in the data source.)
- **Row Template Function** or **Row Template Id** — Specifies custom behavior for the display of every grid row. You can use one of these options to specify the behavior, but **Row Template Id** takes precedence if you specify both. You must write additional code to implement either one. Otherwise, the bound data source definition and the **Grid Columns** settings determine how each row is displayed.

  Row Template Function specifies a JavaScript function that you write to return template-formatted results to display for each row; **Row Template Id** specifies the `id` of a `<script>` tag that contains the actual HTML code for the row template to use to display each row. For more information, see Row templates on page 123 in this document.

- **Alt Row Template Function** or **Alt Row Template Id** — Specifies custom behavior for the display of every **other** grid row. Otherwise, these properties work similar to **Row Template Function** and **Row Template Id**, respectively. If you do not specify either one, all rows of the grid are formatted according to **Row Template Function** or **Row Template Id**.
- **Enable *** — Together with **Page Size**, these properties control the general presentation of data in the rows and columns of the grid, such as selecting a subset of the available data (**Enable Column Filtering**) and changing the order of rows (**Enable Sorting**) and columns (**Enable Column Reordering**).

Note: The **Page Size**, **Enable Column Filtering**, and **Enable Sorting** property values can be managed either by Kendo UI in the client web app or by the Data Object resource that implements the bound data source on the server. The choice of what data management facility responds to these property settings depends on the capabilities of the Data Object resource and whether you select the **Client-side Processing** option as part of the definition for the bound data source. For more information on the **Client-side Processing** option, see Adding and editing a data source on page 27.

- **Events** — Provides one or more properties to change the default name of the event handler function defined for each Data-Grid view-specific event:
  - **Row Select Event Function** — Default value: `onRowSelect`. Executes for the `Row Select` event, which fires when the selected row changes in the grid.
  - **Row Create Event Function** — Executes for the `Row Create` event, which fires before a row is created for a new data source record.
  - **Row Update Event Function** — Executes for the `Row Update` event, which fires before an existing data source record is updated in a row.
  - **Row Delete Event Function** —Executes for the `Row Delete` event, which fires before an existing data source record is deleted in a row.

Note: You can also use properties in the **Edit View** dialog box to change the default names of event handler functions for general events that apply to all views. You can access this dialog by clicking the drop-down menu for each view (as shown for the example **customer_list** view listed in the **VIEWS** pane), then clicking the **Edit** option.

Caution: You must ensure that any change to the default name of an event handler function that you make using its view property you also make to the name of the actual JavaScript function in source code.

You must add custom code to each view event handler function in order to implement any useful behavior for it. The default behavior of these functions has no functional effect. Typically, you do not need to change the default names of event handler functions. However, the view properties exist to allow this name change if you have the need (for example, to avoid a naming conflict with existing JavaScript code you are using elsewhere in the app). For more information on coding both view-specific and general event handler functions (and changing their names in the source), see General view events on page 115 and View-specific events on page 119 in this document.

Adding and editing a Data-Grid-Form view

The Data-Grid-Form view is a read-only grid that offers a design-time choice of two edit modes using a form: a read-only or read-only-to-edit form that displays with the grid in a single split screen on the app views page.

At run time, the grid initially displays with the first row selected and a read-only form displayed wherever it fits on the page (to the right or below the grid). Fields from the selected record are displayed in the read-only form as read-only, plain text. These fields can be organized into field groups that appear in labeled tabs within the form. You can then navigate the rows of the grid and select any other record, and the read-only form displays the selected record fields accordingly.

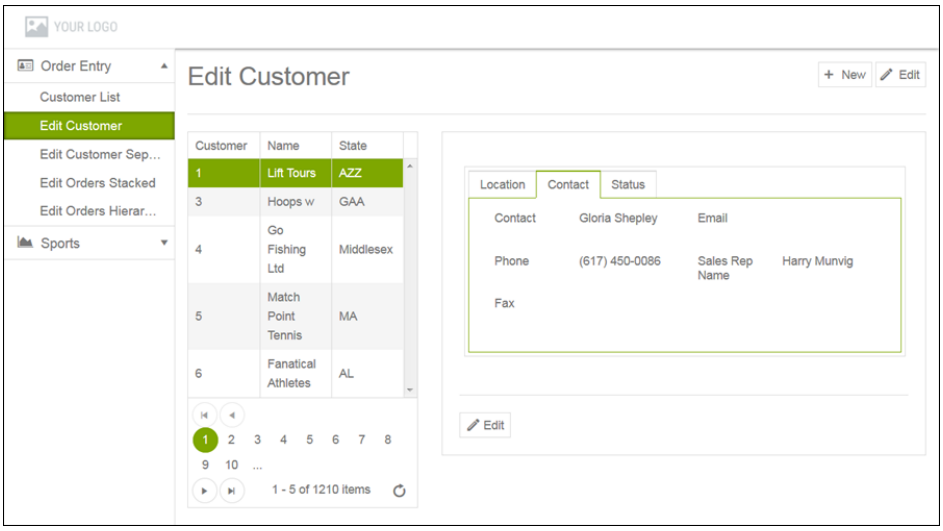
The same is true for a read-only-to-edit form, but you also have the option to edit the selected record or to add a new record to the bound data source. If you choose to edit the selected record, the view overlays the read-only form with a form that displays the fields for editing according to the editor types selected for the fields in the data source. This editable form also provides options to save or cancel the changes you make, or to delete the record from the data source that is displayed in the editable form.

If you select the option to add a new record, the view overlays the read-only form with a similarly editable form that displays the fields for a new record with initial values that you can change before adding the record to the data source. For any editable form, you can either save the changes or cancel the changes and return to the grid with the row selected with the most recently edited or added record, or with the first row selected in the current grid page after canceling a new record add.

At design time, you can separately customize what columns are displayed in the grid and what fields are displayed on the form, as well as other properties that affect the display of the grid, the form, and its data.

When you select an instance of this Data-Grid-Form view in a module at run time, the app displays a page similar to the view labeled **Edit Customer** in this read-only-to-edit mode example:

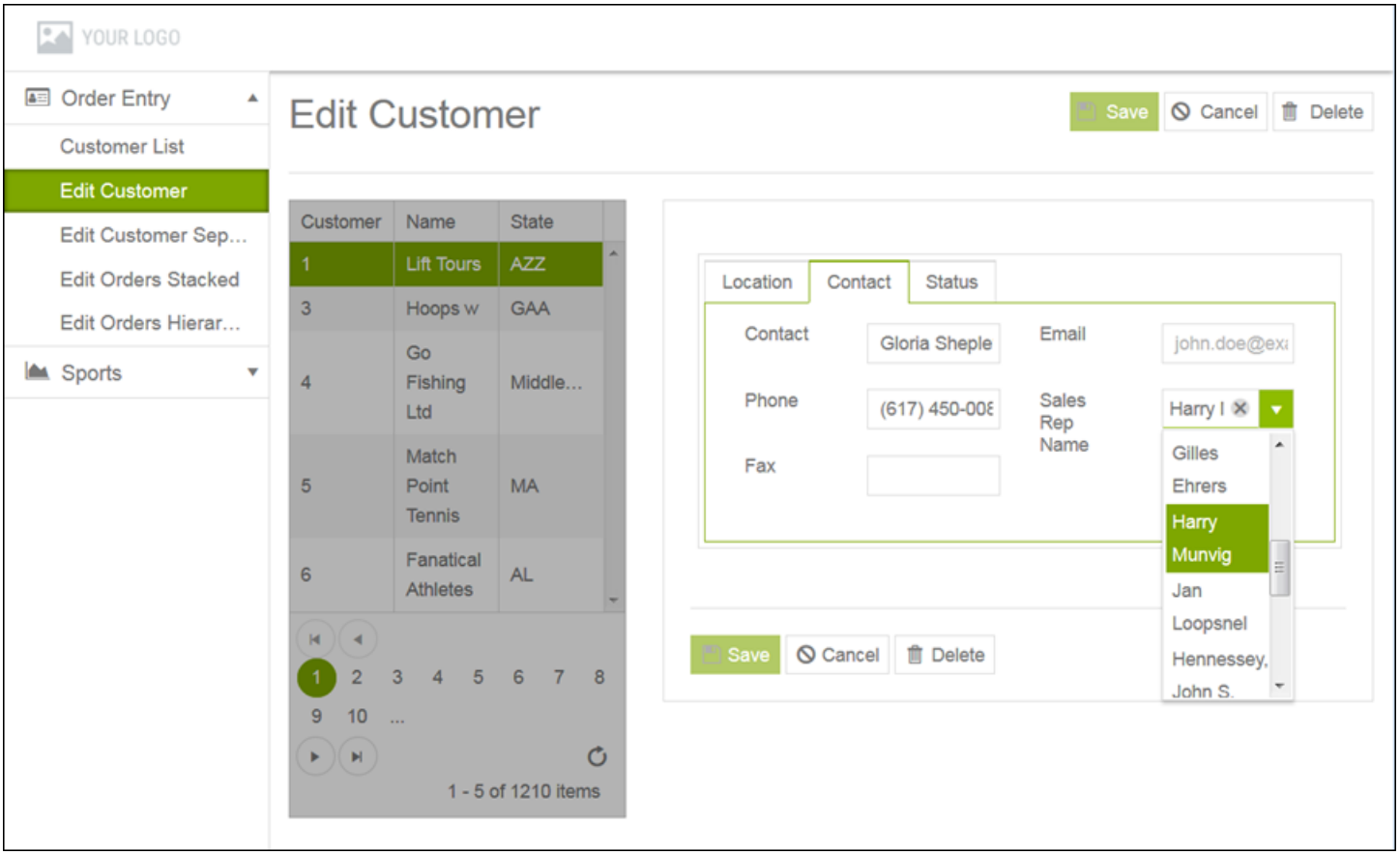
Figure 19: Data-Grid-Form view running in app with read-only form



The view opens with a read-only grid similar to the previous **Customer List** Data-Grid view example (see Adding and editing a Data-Grid view on page 42), but presented in a split screen with a read-only form containing a tab folder for three field groups, with the **Contact** tab selected. These form fields are displayed from the record (initially) from the first row on the first page of the grid, then from any grid row that you select.

Clicking **Edit** above or below the form disables the grid and overlays the read-only form with an editable form, as in this example (a similar editable form displays to add a new record to the bound data source by clicking **New**):

Figure 20: Data-Grid-Form view running in app with editable form overlaying read-only form



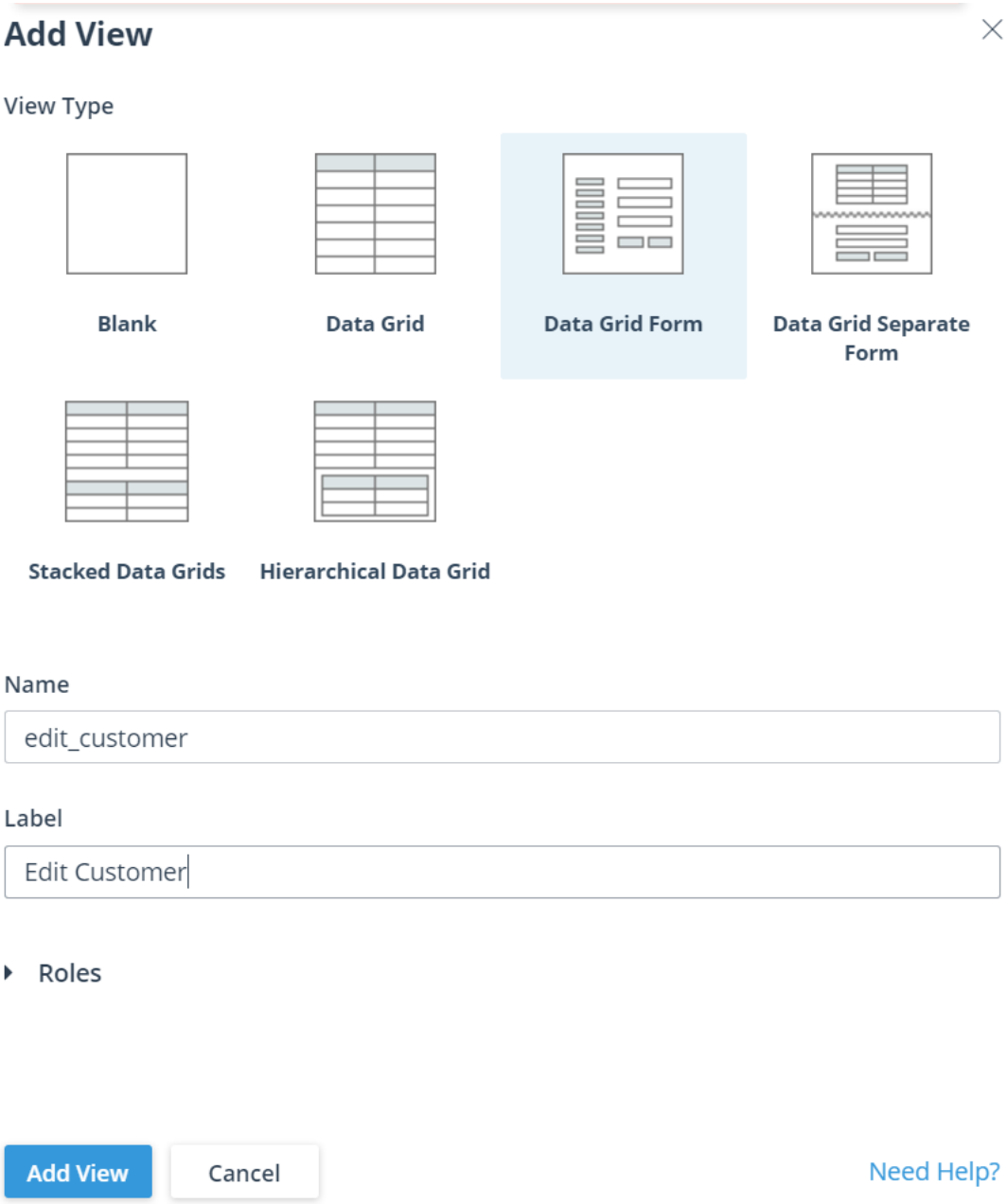
Each field in the editable form is displayed according to the editor type that has been selected for it in the bound data source. In this example, the displayed value of the form field labeled **Sales Rep Name** can be changed by selecting a new value from a list (`combo-box` editor type) that is populated from a foreign key look up. The Kendo UI Builder provides built-in support for foreign key look ups in the forms of Data-Grid* views with forms. For more information on foreign key support, as well as selecting editor types for fields, see Adding and editing a data source on page 27.

From here, the edited record can be saved (by clicking **Save**), deleted (by clicking **Delete**), or the edit canceled (by clicking **Cancel**), all of which return to the read-only form displaying fields from an appropriate record, with the grid enabled.

To add a Data-Grid-Form view to a user-created module, edit the module, which opens a view design page in the module (see Figure 22: Data-Grid-Form view design page on page 53 for an example), then click **Add View** at the top of the **VIEWS** pane.

This opens an **Add View** dialog box, similar to this example:

Figure 21: Add View dialog box creating a Data-Grid-Form view



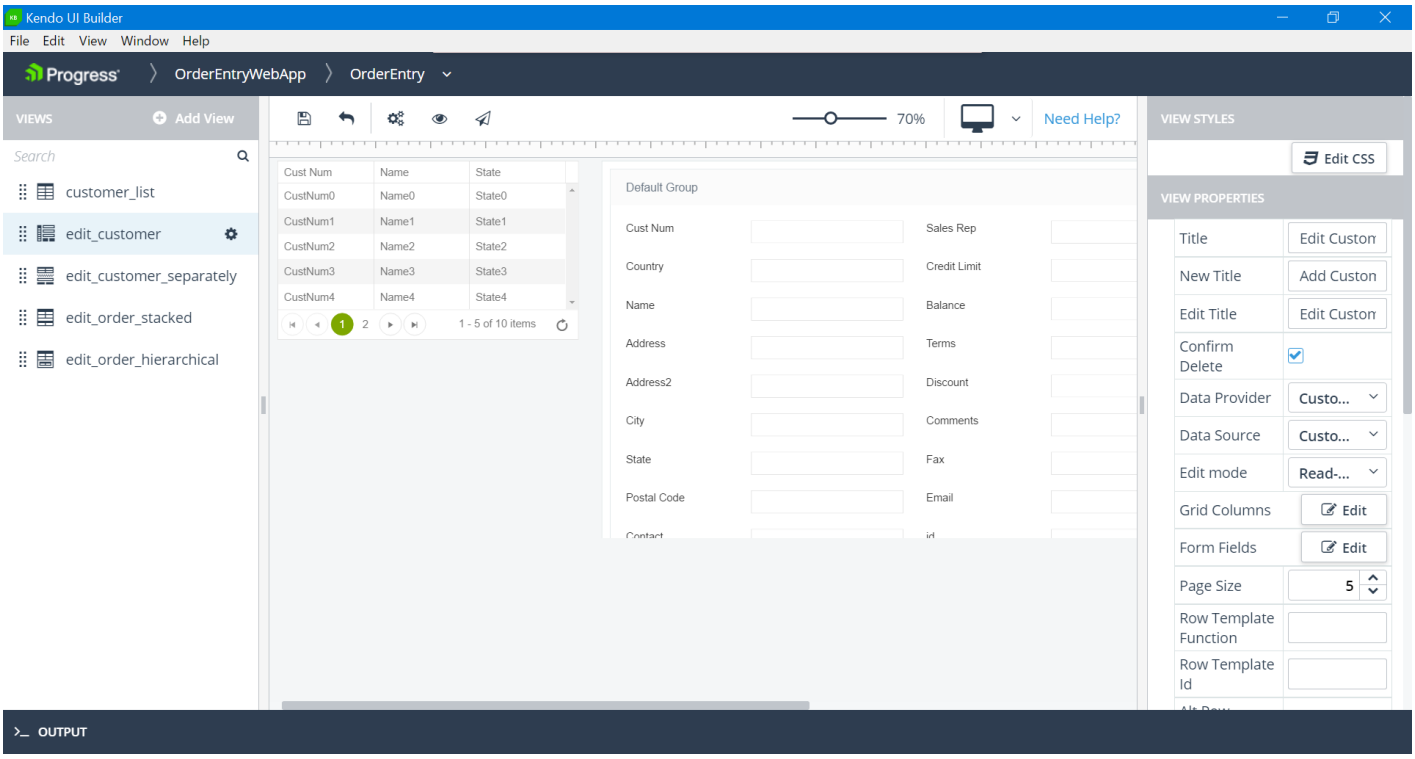
In this example, the dialog box has `edit_customer` entered as the value of the view **Name**, `Edit Customer` entered as the value of the view **Label**, and **Data Grid Form** selected as the **View Type**. The options for entering these values are the same for all user-created views.

The **Roles**tab opens an **Authorization Roles** dialog box that allows you to select the user roles that the app can use to authorize access to this view. To know more about roles, see Using roles to authorize user access on page 93.

For more information, see the **Add View** dialog description in Adding and editing a Data-Grid view on page 42.

After specifying the **Name**, **Label**, and **View Type**, click **Add View** to create the specified view and display its view design page for editing, as shown for the **edit_customer** Data-Grid-Form view in this example:

Figure 22: Data-Grid-Form view design page



The Data-Grid-Form view design page shows certain features common to all predefined Data-Grid* view design pages, top-to-bottom and left-to-right:

- **Header** — Similar to the header for the app design page, with the addition of breadcrumbs that track your path in the Designer to the current view design page.
- **Toolbar** — Provides the following tools:
  - **Save** — Saves any unsaved changes in the current view definition to the UI meta-data.
  - **Revert** — Cancels any unsaved changes and returns the current view definition to its state as of the most recent **Save**.
  - **Generate** — Invokes the Kendo UI Generator to build the current state of the app ready for preview.
  - **Preview** — Either invokes the Kendo UI Generator to rebuild and preview the latest state of the app, or immediately preview the most recent generated build (if one exists). The preview opens in a tab of your default browser using a webpack-dev-server with live data from the data sources mapped to the app views.
  - **Publish** — Invokes the Kendo UI Generator to build a deployment version of the app either for testing (**Debug**) or production (**Release**) in their own respective locations. For more information on app builds, see App generation and deployment on page 101.
  -  — Selects a device whose view you want to simulate in the view design page and which resembles the view as displayed on the physical device at run time. The displayed icon reflects the selected device from these supported devices: **Desktop** (default), **Laptop**, **Tablet landscape**, or **Tablet portrait**.

Note: The run-time preview that you open from the Designer using **Preview** always displays according to the physical device where you are running the Designer.

- **VIEWS pane (in panel on the left)** — Lists the views in the current module, including the example **edit_customer** view. There is also an **Add View** button (used to create this view) for creating additional views and a search box for locating a view in a long list of views in the module.
- ***View design panel*** **(in the middle)** — Shows a design simulation of the view, bounded on the left and top with a vertical and horizontal ruler. The view design panel contains the following elements that are common to all predefined Data-Grid* views, but which might contain different content for each predefined view type:
  - **Custom top html section** — Initially not implemented, this custom section appears in the view when you code the content of an HTML `<div>` element that displays at the top of the view.
  - **Header section** — In this case, an identifying title for the view (**Title** setting).
  - **Custom middle html section** — Initially not implemented, this custom section appears in the view when you code the content of an HTML `<div>` element that displays in the middle, between the header and data sections of the view.
  - **Data section** — In this case, showing a design simulation of the grid and form in split screen as currently configured in the example Data-Grid-Form view instance for display. Note that this includes a simulation of the button configuration above and below for either a read-only or editable form, depending on the edit mode selected for the view (each form style has a subset of these buttons at run time).
  - **Custom bottom html section** — Initially not implemented, this custom section appears in the view when you code the content of an HTML `<div>` element that displays at the bottom of the view.

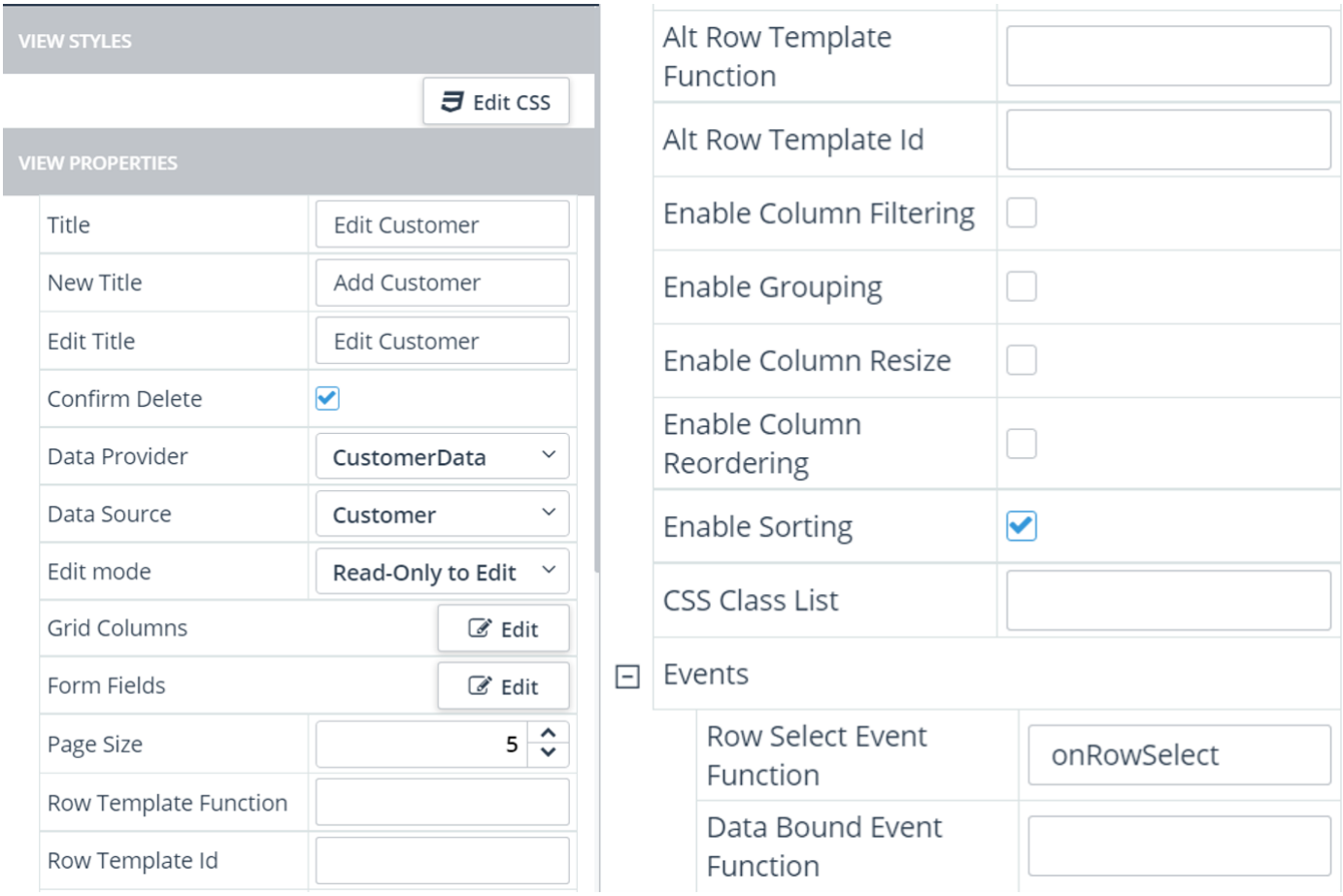
  For more information on coding custom HTML sections, see Custom HTML sections on page 122 in this document.

  The design simulation in a view design panel changes as you modify some of the view properties that affect its appearance, and even as you click around in the simulated view design panel, depending on these property settings. For example, the **edit_customer** view title is set to `Edit Customer` using the **Title** property, there are five (`5.00`) rows on each page of the grid as set for the **Page Size** property, and the buttons available with the read-only and editable forms are shown for the **Read-Only-to-Edit** setting of the **Edit mode** property (see the **VIEW PROPERTIES** pane description, below, for more information).
- The **VIEW STYLES pane (in panel on the right)** — provides an **Edit CSS** button that enables you to add and edit CSS styles for the view. Clicking the **Edit CSS** button opens the view's `style.css` file in a Monaco editor. The `style.css`file is located in *`application-folder`*`\app\src\modules\`*`module-folder`*`\`*`view-folder`*, where *`application-folder`* is the path to the folder that contains and is named for your web app, *`module-folder`* is the name of the folder defining the module, and *`view-folder`* is the name of the folder defining the view.

Note: To apply CSS classes (defined in the *`style.css`* file) to the view, you must specify the class names in the **CSS Class List** property in the **VIEW PROPERTIES** pane.

- **VIEW PROPERTIES pane (in panel on the right)** — Contains all the properties that you can set that change the design-time appearance and content of the view, as well as some settings that can affect view behavior, such as view-specific event handler settings.

Figure 23: Data-Grid-Form view properties

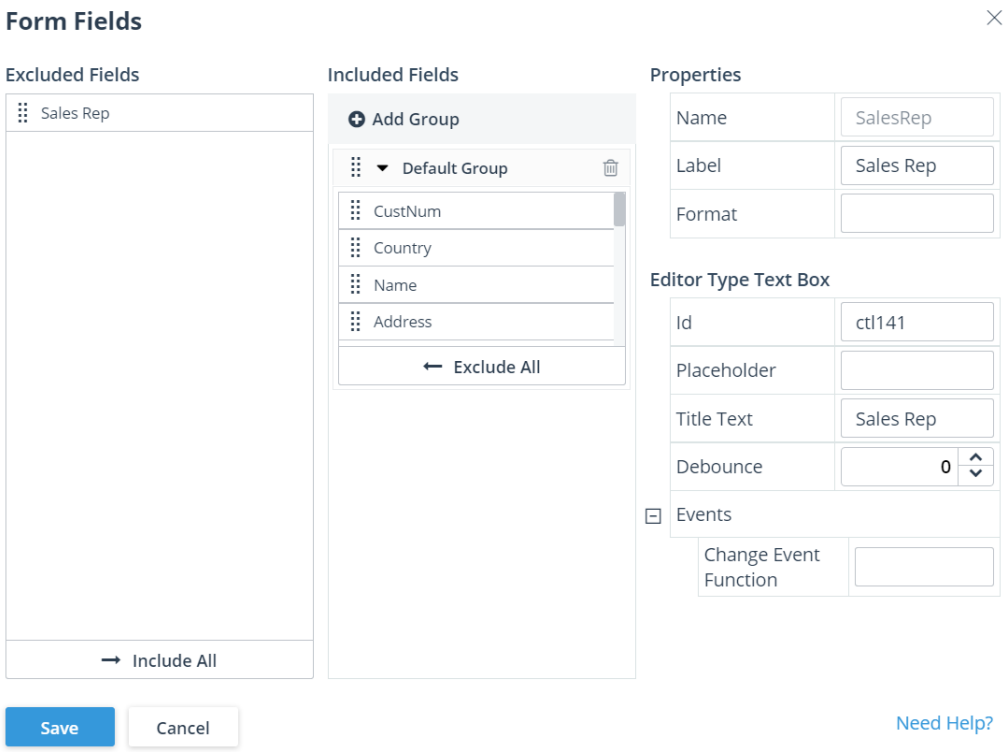


Additional properties and values of note include:

- **New Title** and **Edit Title** — Allow you to enter separate titles for an editable form displayed for adding a new record and an editable form displayed for editing an existing record, when clicking **New** and **Edit**, respectively, on the read-only form.
- **Data Provider** and **Data Source** — Allow you to select an available data provider and data source to bind to the view. For more information, see Data providers and data sources on page 22.
- **Edit Mode** — Allows you to select **Read-Only** or **Read-Only-to-Edit**. With **Read-Only** selected, only a read-only form is displayed with no buttons, since no editable form is available.
- **Grid Columns** — Clicking **Edit** for this property opens a **Grid Columns** dialog box that allows you to specify what data source fields appear as columns in the grid, and some features affecting how each field is displayed in the grid. This dialog box works the same for all Data-Grid* views. For more information, see the **Grid Columns** dialog box description in Adding and editing a Data-Grid view on page 42.

- **Form Fields** — Clicking **Edit** for this property opens a **Form Fields** dialog box that allows you to specify what data source fields appear as fields on a form, and some features affecting how each field is displayed in the form:

Figure 24: Form Fields dialog box



This dialog box provides an **Included Fields** and **Excluded Fields** list, similar to the **Included Columns** and **Excluded Columns** lists in the **Grid Columns** dialog box for selecting the data source fields that are displayed in a form. However, for the **Form Fields** dialog box the fields are listed by their field **Label** values instead of by their data source field names.

Initially, all the fields in the **Included Fields** list are listed in a field group labeled **Default Group** (which you can change) and you can exclude or include them all, or move them individually between the lists. Once you create additional field groups, you must work with each field or field group one at a time, dragging-and-dropping each field either between the lists or between one field group and another. You can also reorder both the individual fields in a field group and the field groups themselves by dragging-and-dropping them within the **Included Fields** list. The final order reflects both the run-time order in which fields appear in each field group and in which the tabs that select each field group appear on the form.

In this example, three fields groups are defined, labeled **Location**, **Contact**, and **Status**, with most data source fields distributed among them. The field labeled **Sales Rep** is excluded, because a foreign key look-up field (labeled **Sales Rep Name**) is used to display a corresponding value instead.

The **Properties** that affect how each field that you select in **Included Fields** is displayed in the form include **Name**, which displays the read-only data source field name (`Salesrep`, in the example), **Label**, which allows you to specify a field label for the form, and **Format**, which allows you to specify a format to augment the value that is displayed in the form (similar to the **Format** property in the **Grid Columns** dialog box).

Additional properties (**Editor Type ***) affect how the specified editor type for the selected field is implemented and displayed in the form. In this example, the editor type is **Combo Box**, which is selected (as **combo-box**) in the data source definition for the foreign key look-up field labeled **Sales Rep Name**. The **Id** property value must be (and is initially generated as) unique among all editor types in the view, and the **Title Text** property allows you to specify a value that displays when the user hovers over the field in the form. (**Note:** In addition to its data source definition, this is all that is required to implement a foreign key look-up field in the form of a Data-Grid* view with forms. For more information, see Adding and editing a data source on page 27.)

- **Page Size** through **Enable *** — Together, these properties control the general presentation of data in the rows and columns of the grid and work the same for all Data-Grid* views. For more information, see the description of these properties in Adding and editing a Data-Grid view on page 42.
- **Events** — Provides one or more properties to change the default name of the event handler function defined for each Data-Grid-Form view-specific event:
  - **Row Select Event Function** — Default value: `onRowSelect`. Executes for the `Row Select` event, which fires when the selected row changes in the grid.
  - **Data Bound Event Function** — Executes for the `Data Bound` event, which fires when the view is bound to its data source.

Note: You can also use properties in the **Edit View** dialog box to change the default names of event handler functions for general events that apply to all views. You can access this dialog by clicking the drop-down menu for each view (as shown for the example **edit_customer** view listed in the **VIEWS** pane), then clicking the **Edit** option.

Caution: You must ensure that any change to the default name of an event handler function that you make using its view property you also make to the name of the actual JavaScript function in source code.

You must add custom code to each view event handler function in order to implement any useful behavior for it. The default behavior of these functions has no functional effect. Typically, you do not need to change the default names of event handler functions. However, the view properties exist to allow this name change if you have the need (for example, to avoid a naming conflict with existing JavaScript code you are using elsewhere in the app). For more information on coding both view-specific and general event handler functions (and changing their names in the source), see General view events on page 115 and View-specific events on page 119 in this document.

Adding and editing a Data-Grid-Separate-Form view

Data-Grid-Separate-Form view is a read-only grid that offers a design-time choice of three edit modes using a form a read-only, directly editable, or read-only-to-edit form that overlays the grid in a separate screen on the app views page.

At run time, the grid initially displays by itself with no records selected. You can then navigate the rows of the grid without selecting a record. At any point during row navigation, you have the choice of selecting a row in the grid, or if the associated form is directly editable or read-only-to-edit, selecting an option to add a new record to the bound data source.

For a read-only or read-only-to-edit form, selecting a grid row automatically overlays the grid with a read-only form showing fields from the record displayed as read-only, plain text. These fields can be organized into field groups that appear in labeled tabs within the form. For a read-only-to-edit form, you also have the option to edit the record displayed in the read-only form or go back to the grid without making changes.

For a directly editable form, selecting a grid row overlays the grid with a form that displays the record fields for editing according to the editor types selected for the fields in the data source. These fields can be organized into field groups that appear in labeled tabs within the form. For a read-only-to-edit form, an identical editable form also overlays the initial read-only form when you select the option to edit the displayed record. This editable form provides options to save or cancel the changes you make, or to delete the record from the data source that is selected in the grid and displayed in the form.

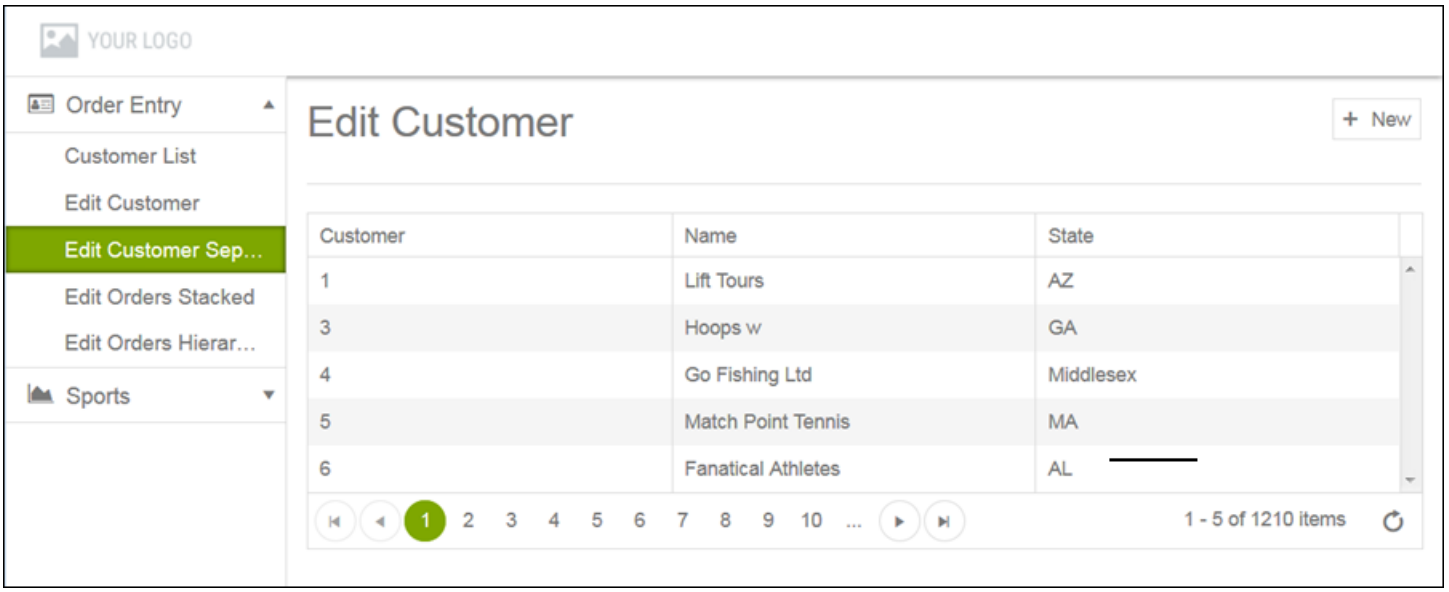
If you select the grid option to add a new record, the view overlays the grid with a similarly editable form that displays the fields for a new record with initial values that you can change before adding the record to the data source.

For any editable form, you can either save the changes or cancel the changes. For a read-only-to-edit form that is editing an existing record, the screen returns to the read-only form displaying the same record fields, with an option to go back to the grid. For a directly editable form, the screen goes directly back to the grid. For either edit mode, when the screen goes back to the grid, it either displays with the row highlighted (but not selected) for the most recently edited or added record, or displays with the first row highlighted in the current grid page after canceling a new record add.

At design time, you can separately customize what columns are displayed in the grid and what fields are displayed on the form, as well as other properties that affect the display of the grid, the form, and its data.

When you select a Data-Grid-Separate-Form view in a module at run time, the app displays a page similar to the view labeled **Edit Customer Separately** in this read-only-to-edit mode example:

Figure 25: Data-Grid-Separate-Form view running in app

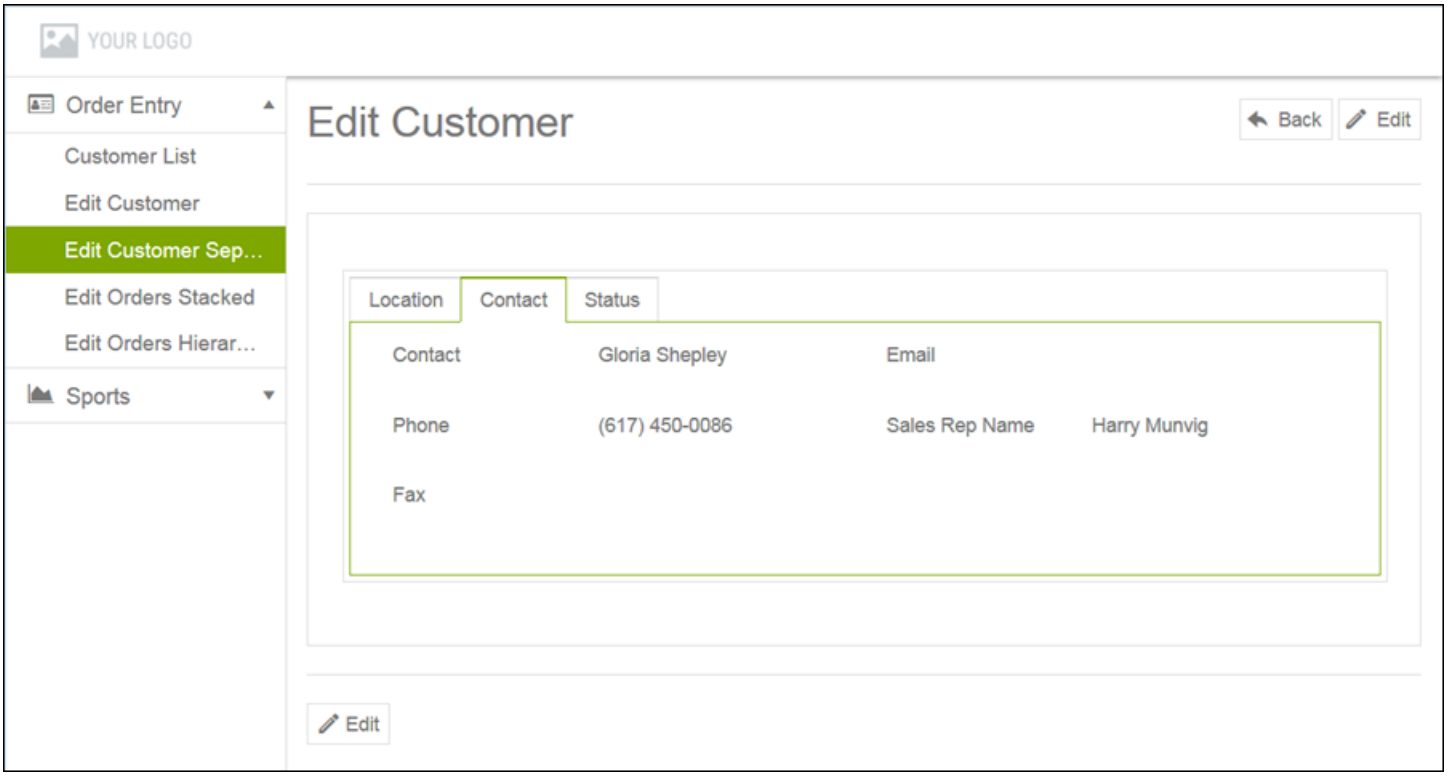


The view opens with a read-only grid similar to the previous **Edit Customer** Data-Grid-Form view example (see Adding and editing a Data-Grid-Form view on page 49).

Selecting any row, on any page of the grid immediately overlays the grid with a read-only form containing fields displayed from the bound data source record shown in that row, as in this example, shown with three field groups accessible using a tab folder:

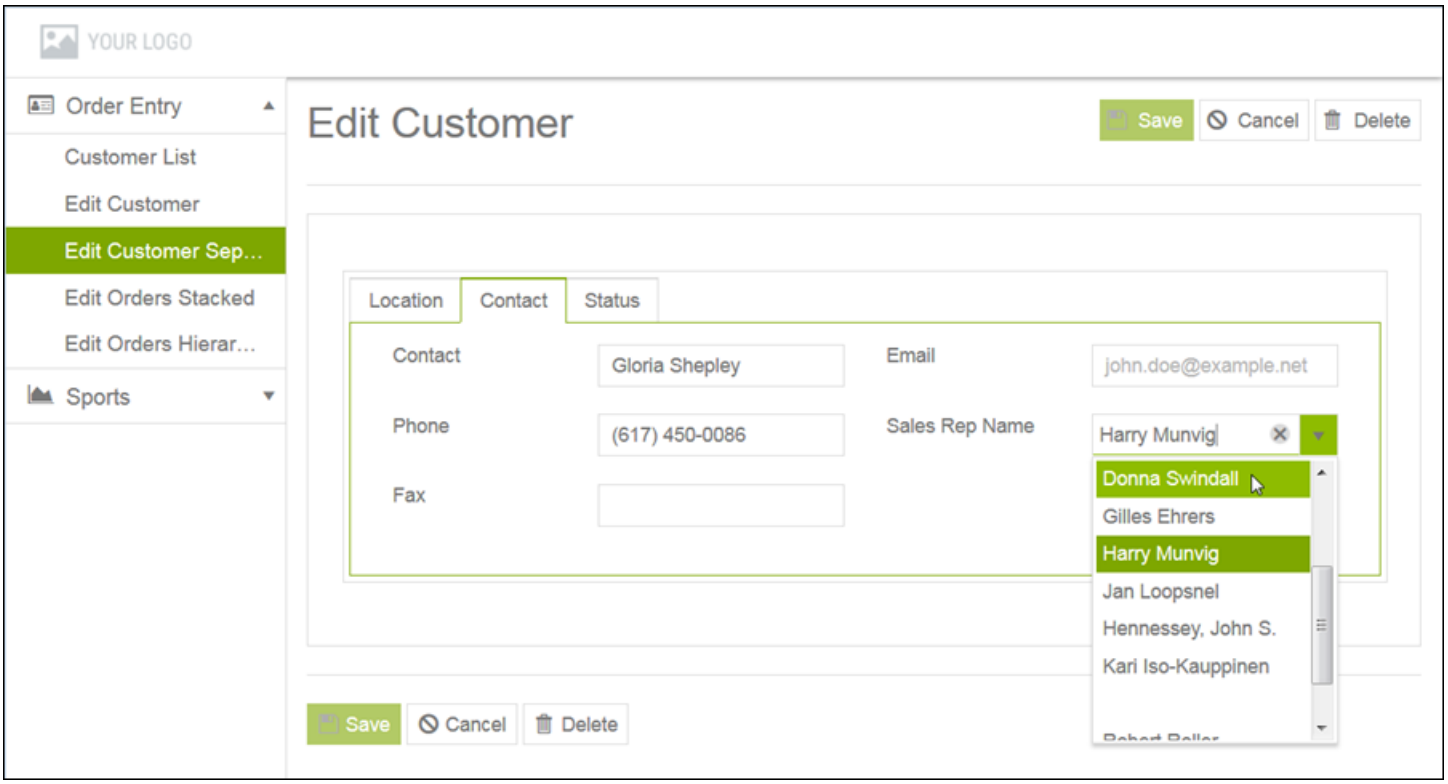
Note: Clicking **New** (above the grid) overlays the grid with an editable form for adding a new record to the bound data source, which is not shown.

Figure 26: Data-Grid-Separate-Form view running in app with read-only form overlaying the grid



Clicking **Back** (above the form) returns to the read-only grid with the same row highlighted; clicking **Edit** (above or below the form) overlays the read-only form with an editable form, as in the following example:

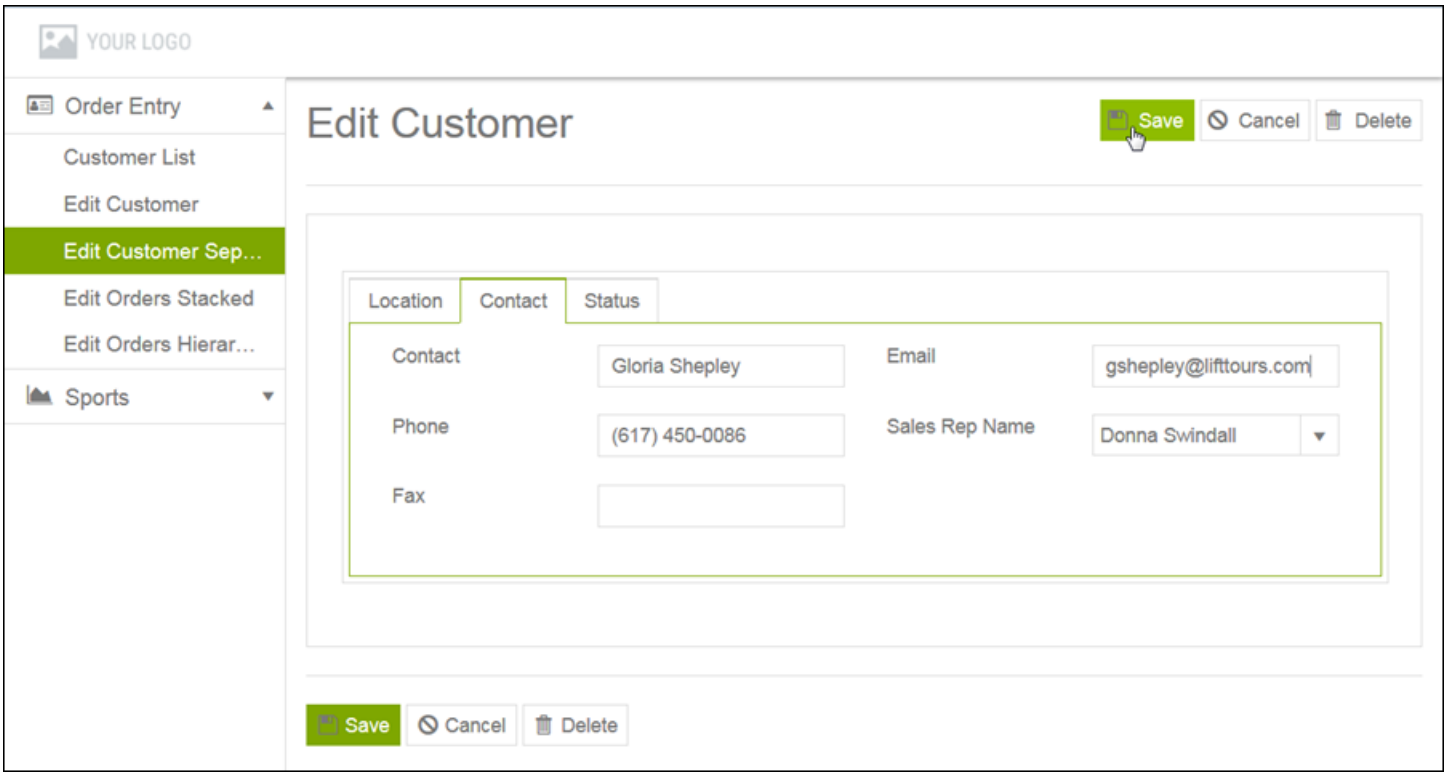
Figure 27: Data-Grid-Separate-Form view running in app with editable form overlaying the read-only form



Each field in the editable form is displayed according to the editor type that has been selected for it in the bound data source. In this example, the displayed value of the form field labeled **Sales Rep Name** can be changed by selecting a new value from a list (`combo-box` editor type) that is populated from a foreign key look up. The Kendo UI Builder provides built-in support for foreign key look ups in the forms of Data-Grid* views with forms. For more information on foreign key support, as well as selecting editor types for fields, see Adding and editing a data source on page 27.

This is the editable form after saving changes to the **Email** and **Sales Rep Name** fields:

Figure 28: Data-Grid-Separate-Form view running in app with editable form changes

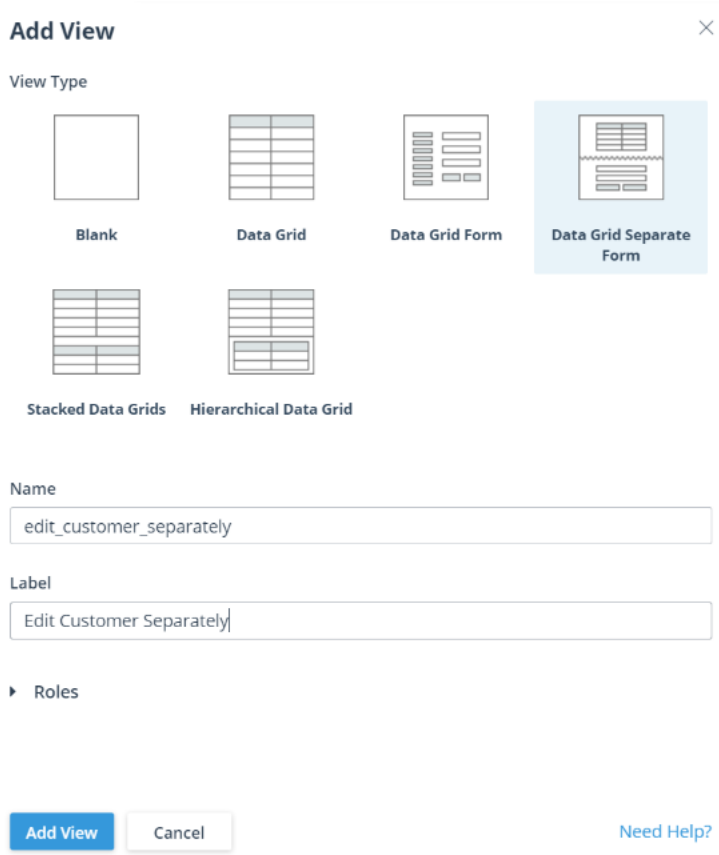


From here, the edited record can be saved (by clicking **Save**), deleted (by clicking **Delete**), or the edit canceled (by clicking **Cancel**), all of which return to the read-only form displaying fields from an appropriate record, with the grid enabled.

Note: For a Data-Grid-Separate-Form view running in edit (as opposed to read-only-to-edit) mode, no read-only form is ever displayed. Instead, only an editable form is displayed for a selected row, and selecting any button (**Save**, **Delete**, or **Cancel**) completes the specified function and returns to the read-only grid with an appropriate row highlighted.

This opens an **Add View** dialog box, similar to this example:

Figure 29: Add View dialog box creating a Data-Grid-Separate-Form view



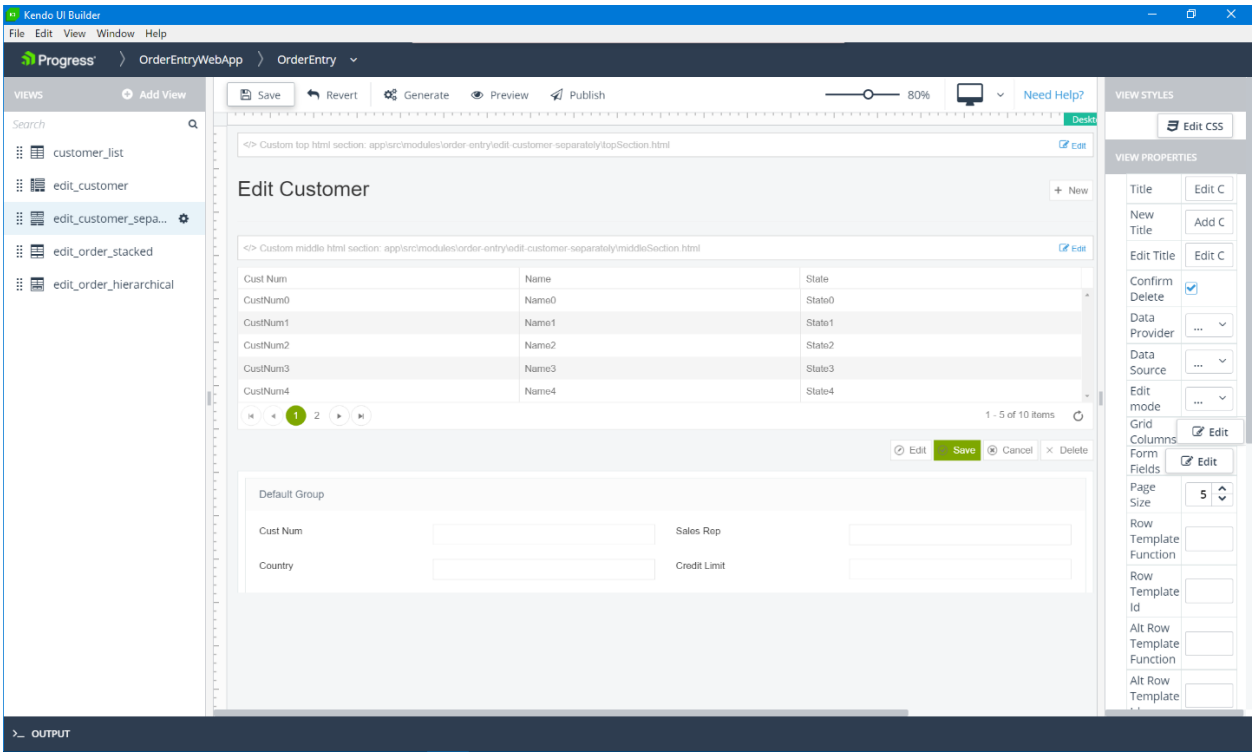
In this example, the dialog box has `edit_customer_separately` entered as the value of the view **Name**, `Edit Customer Separately` entered as the value of the view **Label**, and **Data Grid Separate Form** selected as the view type. The options for entering these values are the same for all user-created views.

The **Roles**tab opens an **Authorization Roles** dialog box that allows you to select the user roles that the app can use to authorize access to this view. To know more about roles, see Using roles to authorize user access on page 93.

For more information, see the **Add View** dialog description in Adding and editing a Data-Grid view on page 42.

After specifying the **Name**, **Label**, and **View Type**, click **Add View** to create the specified view and display its view design page for editing, as shown for the **edit_customer_separately** Data-Grid-Separate-Form view in this example:

Figure 30: Data-Grid-Separate-Form view design page



The Data-Grid-Separate-Form view design page shows certain features common to all predefined Data-Grid* view design pages, top-to-bottom and left-to-right:

- **Header** — Similar to the header for the app design page, with the addition of breadcrumbs that track your path in the Designer to the current view design page.
- **Toolbar** — Provides the following tools:
  - **Save** — Saves any unsaved changes in the current view definition to the UI meta-data.
  - **Revert** — Cancels any unsaved changes and returns the current view definition to its state as of the most recent **Save**.
  - **Generate** — Invokes the Kendo UI Generator to build the current state of the app ready for preview.
  - **Preview** — Either invokes the Kendo UI Generator to rebuild and preview the latest state of the app, or immediately preview the most recent generated build (if one exists). The preview opens in a tab of your default browser using a webpack-dev-server with live data from the data sources mapped to the app views.
  - **Publish** — Invokes the Kendo UI Generator to build a deployment version of the app either for testing (**Debug**) or production (**Release**) in their own respective locations. For more information on app builds, see App generation and deployment on page 101.
  -  — Selects a device whose view you want to simulate in the view design page and which resembles the view as displayed on the physical device at run time. The displayed icon reflects the selected device from these supported devices: **Desktop** (default), **Laptop**, **Tablet landscape**, or **Tablet portrait**.

Note: The run-time preview that you open from the Designer using **Preview** always displays according to the physical device where you are running the Designer.

- **VIEWS pane (in panel on the left)** — Lists the views in the current module, including the example **edit_customer_separately** view. There is also an **Add View** button (used to create this view) for creating additional views and a search box for locating a view in a long list of views in the module.
- ***View design panel*** **(in the middle)** — Shows a design simulation of the view, bounded on the left and top with a vertical and horizontal ruler. The view design panel contains the following elements that are common to all predefined Data-Grid* views, but which might contain different content for each predefined view type:
  - **Custom top html section** — Initially not implemented, this custom section appears in the view when you code the content of an HTML `<div>` element that displays at the top of the view.
  - **Header section** — In this case, an identifying title for the view (**Title** setting).
  - **Custom middle html section** — Initially not implemented, this custom section appears in the view when you code the content of an HTML `<div>` element that displays in the middle, between the header and data sections of the view.
  - **Data section** — In this case, showing a design simulation of the grid and separate form as currently configured in the example Data-Grid-Separate-Form view instance for display. Note that this includes a simulation of the button configuration for the view, including the **New** button above the grid, as well as the **Edit**, **Save**, **Cancel**, and **Delete** buttons that appear above and below either a read-only or editable form, depending on the edit mode selected for the view (each form style has a subset of these buttons at run time)
  - **Custom bottom html section** — Initially not implemented, this custom section appears in the view when you code the content of an HTML `<div>` element that displays at the bottom of the view.

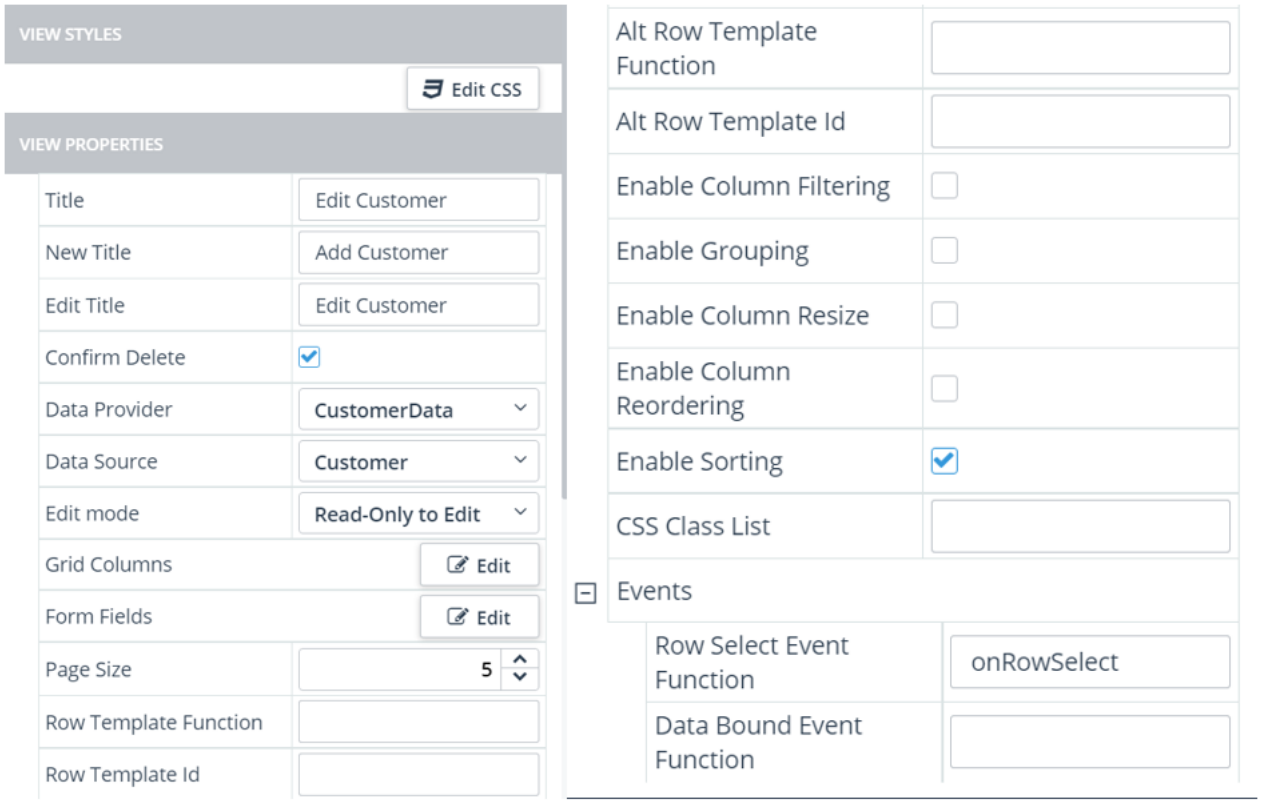
  For more information on coding custom HTML sections, see Custom HTML sections on page 122 in this document.

  The design simulation in a view design panel changes as you modify some of the view properties that affect its appearance, and even as you click around in the simulated view design panel, depending on these property settings. For example, the **edit_customer_separately** view title is set to `Edit Customer Separately` using the **Title** property, there are five (`5.00`) rows on each page of the grid as set for the **Page Size** property, and the buttons available with the read-only and editable forms are shown for the **Read-Only-to-Edit** setting of the **Edit mode** property (see the **VIEW PROPERTIES** pane description, below, for more information).
- The **VIEW STYLES pane (in panel on the right)** — provides an **Edit CSS** button that enables you to add and edit CSS styles for the view. Clicking the **Edit CSS** button opens the view's `style.css` file in a Monaco editor. The `style.css`file is located in *`application-folder`*`\app\src\modules\`*`module-folder`*`\`*`view-folder`*, where *`application-folder`* is the path to the folder that contains and is named for your web app, *`module-folder`* is the name of the folder defining the module, and *`view-folder`* is the name of the folder defining the view.

  Note: To apply CSS classes (defined in the *`style.css`* file) to the view, you must specify the class names in the **CSS Class List** property in the **VIEW PROPERTIES** pane.

- **VIEW PROPERTIES pane (in panel on the right)** — Contains all the properties that you can set that change the design-time appearance and content of the view, as well as some settings that can affect view behavior, such as view-specific event handler settings.

  Figure 31: Data-Grid-Separate-Form view properties



  Additional properties and values of note include:

  - **New Title** and **Edit Title** — Allow you to enter separate titles for an editable form displayed for adding a new record and an editable form displayed for editing an existing record, when clicking **New** and **Edit**, respectively, above the grid and on the read-only form (depending on the edit mode).
  - **Data Provider** and **Data Source** — Allow you to select an available data provider and data source to bind to the view. For more information, see Data providers and data sources on page 22.
  - **Edit mode** — Allows you to select **Read-Only**, **Edit**, or **Read-Only-to-Edit**. With **Read-Only** selected, the grid has no **New** button and **only** a read-only form is displayed with no **Edit** button, since no editable form is available. With **Edit** selected, the grid has a **New** button and **only** an editable form is displayed with appropriate buttons for editing either an existing selected record (**Save**, **Cancel**, **Delete**) or a new record (**Save**, **Cancel**).
  - **Grid Columns** — Clicking **Edit** for this property opens a **Grid Columns** dialog box that allows you to specify what data source fields appear as columns in the grid, and some features affecting how each field is displayed in the grid. This dialog box works the same for all Data-Grid* views. For more information, see the **Grid Columns** dialog box description in Adding and editing a Data-Grid view on page 42.
  - **Form Fields** — Clicking **Edit** for this property opens a **Form Fields** dialog box that allows you to specify what data source fields appear as fields on a form, and some features affecting how each field is displayed in the form. This dialog box works the same for all Data-Grid* views with forms. For more information, see the **Form Fields** dialog box description in Adding and editing a Data-Grid-Form view on page 49.

- **Page Size** through **Enable *** — Together, these properties control the general presentation of data in the rows and columns of the grid and work the same for all Data-Grid* views. For more information, see the description of these properties in Adding and editing a Data-Grid view on page 42.
- **Events** — Provides one or more properties to change the default name of the event handler function defined for each Data-Grid view-specific event:
  - **Row Select Event Function** — Default value: `onRowSelect`. Executes for the `Row Select` event, which fires when the selected row changes in the grid.
  - **Data Bound Event Function** — Executes for the `Data Bound` event, which fires when the view is bound to its data source.

Note: You can also use properties in the **Edit View** dialog box to change the default names of event handler functions for general events that apply to all views. You can access this dialog by clicking the ⚙ drop-down menu for each view (as shown for the example **edit_customer_separately** view listed in the **VIEWS** pane), then clicking the **Edit** option.

Caution: You must ensure that any change to the default name of an event handler function that you make using its view property you also make to the name of the actual JavaScript function in source code.

You must add custom code to each view event handler function in order to implement any useful behavior for it. The default behavior of these functions has no functional effect. Typically, you do not need to change the default names of event handler functions. However, the view properties exist to allow this name change if you have the need (for example, to avoid a naming conflict with existing JavaScript code you are using elsewhere in the app). For more information on coding both view-specific and general event handler functions (and changing their names in the source), see General view events on page 115 and View-specific events on page 119 in this document.

Adding and editing a Blank view

The Blank view is a user-defined view that allows you to define both its layout and content by dragging and dropping a variety of components, including:

- Layout components (rows and columns)
- Data management components (e.g., Grid)
- Editor components (e.g., Text Box)
- Chart components (e.g., Bar Charts)
- Scheduling components (e.g., Calendar)
- Navigation components (e.g., Toolbar)
- Media Components (e.g., Images)
- Custom HTML components that you define

The layout is based on the Bootstrap fluid grid system, which manages how you can use row and column components to define it. The content consists of individual UI components, most of which can be bound to data.

To define a view instance you have created, you drag-and-drop these two types of components onto the view design panel: initially, the *layout* components (rows and columns), then the *content* components (such as editable grid and field input components), which you can drop into existing layout columns or directly into existing rows (where a column is automatically created for the component). You can combine these layout and content components in various ways to define views with a variety of content presentations.

The Blank view template thus allows you to build a custom view with virtually any arrangement of the supported content components. For example, you can build a view that is a variation on the predefined Data-Grid* views or something completely different, such as a stand-alone form without any associated grid, but with another style of data navigation.

Also, unlike the Data-Grid* views, the Blank view allows you to specify multiple data source instances from one or more data providers to bind data to the view. To complete the data binding, you code view factory event functions in a JavaScript file that is generated for the view with initial AngularJS code, depending on the components and events you have added to the view.

For example, the following running Blank view instance (labeled **Customer View**) is a form constructed from a toolbar for data navigation and seven content components:

Figure 32: Blank view running in app with toolbar and form

The Toolbar component is configured with four buttons for navigating the records of a `CustomerDS` data source instance. This data source is in turn related to a `SalesrepDS` data source instance by a custom-coded foreign key look-up on the `SalesRep` field in both data sources. (**Note:** You must maintain such relationships in the event function code you write. The built-in foreign key support for Data-Grid* views with forms is not available for the Blank view.)

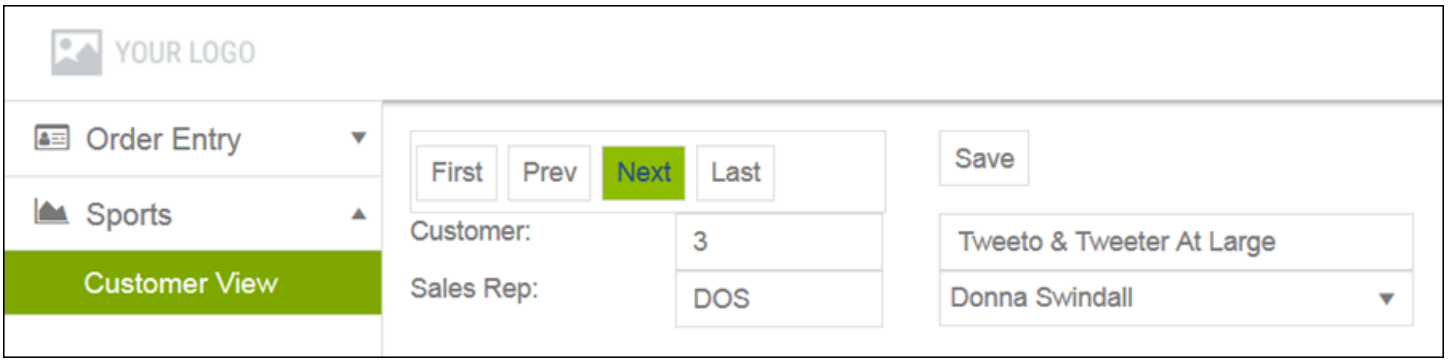
The **Customer:** and **Sales Rep:** labels are hard coded and displayed through Label components. The customer number, **1**, is a `CustNum` field value displayed from the `CustomerDS` data source using a Disabled Text Box component, which displays data in a read-only mode. **Alpha and Omega Services** is the name of a customer uniquely identified by this `CustNum` field value and is a `Name` field value displayed from the `CustomerDS` data source using a Text Box component, which allows editing of the displayed value to update it in the data source.

The sales rep code, **GPE**, is the value displayed from the `SalesRep` field of the `SalesrepDS` data source using a Disabled Text Box. **Gilles Ehrers** is the name of the sales rep uniquely identified by the custom sales rep look-up code and is displayed from the `RepName` field of the `SalesrepDS` data source using a Drop Down List component, which allows any other sales rep to be selected from the `SalesrepDS` data source to update its association with the current `CustomerDS` record.

If any changes have been made to the customer name or to the sales rep associated with the current `CustomerDS` record, they can be saved by clicking **Save**, which is a Button component.

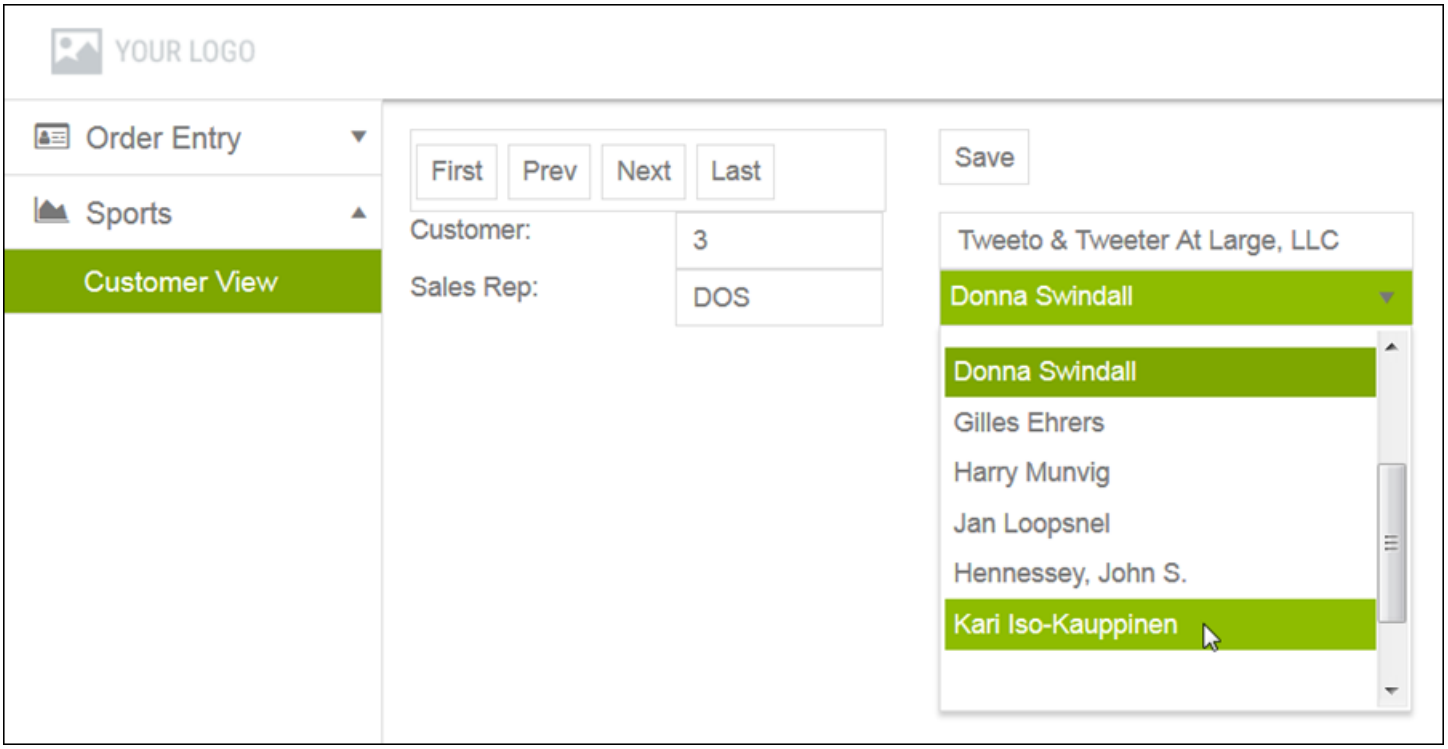
For example, clicking **Next** in the toolbar a couple of times displays field values from the following `CustomerDS` and associated `SalesrepDS` records:

Figure 33: Blank view running in app clicking toolbar item



You can change the customer name from **Tweeto & Tweeter At Large** to **Tweeto & Tweeter At Large, LLC**) and the associated sales rep from **Donna Swindall** to **Kari Iso-Kauppinen**, as shown:

Figure 34: Blank view running in app with change to form fields



Then save the new customer name and the `SalesRep` field values in the `CustomerDS` data source record by clicking **Save**, as shown:

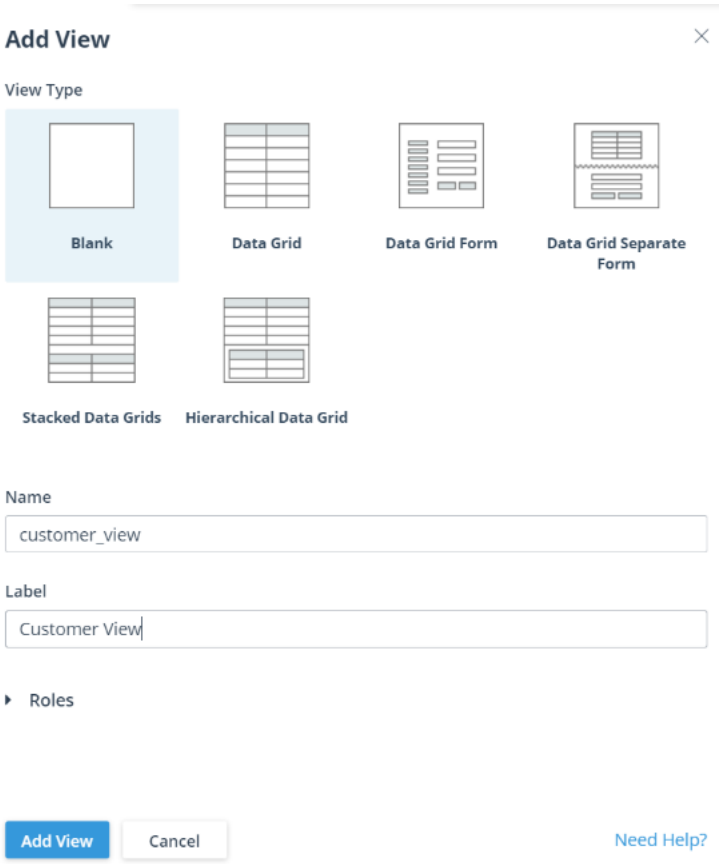
Figure 35: Blank view running in app with changes saved

In this case, the `RepName` field does not also need to be saved to the `CustomerDS` record because it is only displayed from the available `SalesrepDS` records in order to return the associated `SalesRep` field value based on the selected `RepName` field value that the user selects.

To add a Blank view to a user-created module, edit the module, which opens a view design page in the module (see Figure 37: Blank view design page on page 70 for an example), then click **Add View** at the top of the **VIEWS** pane.

This opens an **Add View** dialog box, similar to this example:

Figure 36: Add View dialog box creating a Blank view



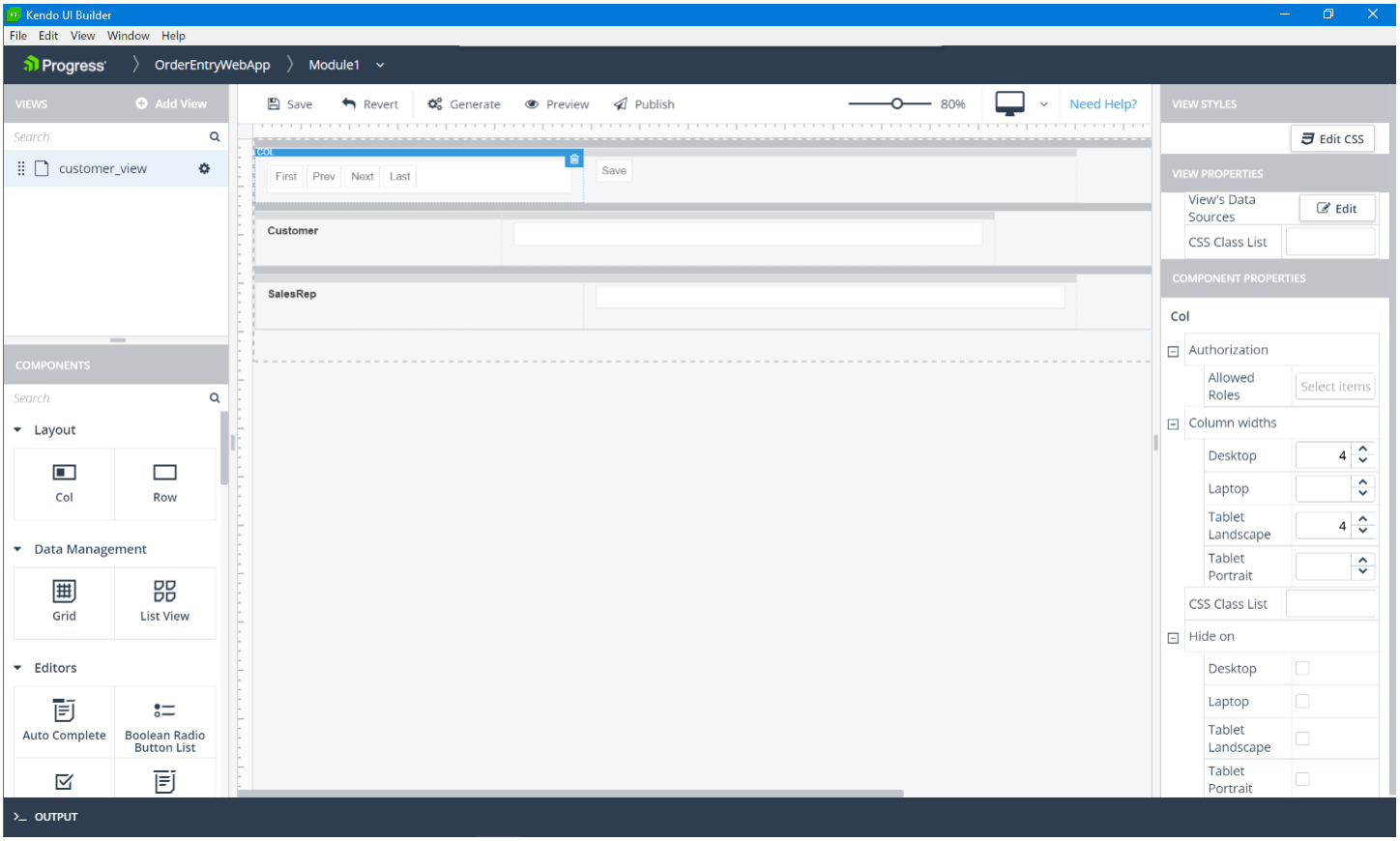
In this example, the dialog box has `customer_view` entered as the value of the view **Name**, `Customer View` entered as the value of the view **Label**, and **Blank** (the default) selected as the view type. The options for entering these values are the same for all user-created views.

The **Roles**tab opens an **Authorization Roles** dialog box that allows you to select the user roles that the app can use to authorize access to this view. To know more about roles, see Using roles to authorize user access on page 93.

For more information, see the **Add View** dialog description in Adding and editing a Data-Grid view on page 42.

After specifying the **Name**, **Label**, and **View Type**, click **Add View** to create the specified view and display its view design page for editing, as shown for the **customer_view** Blank view in this example:

Figure 37: Blank view design page



The Blank view design page displays the following features, top-to-bottom and left-to-right:

- **Header** — Similar to the header for the app design page, with the addition of breadcrumbs that track your path in the Designer to the current view design page.
- **Toolbar** — Provides the following tools:
  - **Save** — Saves any unsaved changes in the current view definition to the UI meta-data.
  - **Revert** — Cancels any unsaved changes and returns the current view definition to its state as of the most recent **Save**.
  - **Generate** — Invokes the Kendo UI Generator to build the current state of the app ready for preview.
  - **Preview** — Either invokes the Kendo UI Generator to rebuild and preview the latest state of the app, or immediately preview the most recent generated build (if one exists). The preview opens in a tab of your

default browser using a webpack-dev-server with live data from the data sources mapped to the app views.

- **Publish** — Invokes the Kendo UI Generator to build a deployment version of the app either for testing (**Debug**) or production (**Release**) in their own respective locations. For more information on app builds, see App generation and deployment on page 101.
-  — Selects a device whose view you want to simulate in the view design page and which resembles the view as displayed on the physical device at run time. The displayed icon reflects the selected device from these supported devices: **Desktop** (default), **Laptop**, **Tablet landscape**, or **Tablet portrait**.

Note: The run-time preview that you open from the Designer using **Preview** always displays according to the physical device where you are running the Designer.

- **VIEWS pane (in panel on the left)** — Lists the views in the current module, including the example **customer_view** view. There is also an **Add View** button (used to create this view) for creating additional views and a search box for locating a view in a long list of views in the module.
- **COMPONENTS pane (in panel on the left)** — Lists the supported layout and content components that you can drag-and-drop into a Blank view, including the following component categories:
  - **Layout** — Including **Col** and **Row** for adding layout columns and rows, respectively.
  - **Data Management** — Content components that bind to a data source and support both navigation through its records and editing of the fields of selected records, for example, a **Grid** or **List View**.
  - **Editors** — Content components that bind to a single field of a selected record based typically on the field's semantic type, and that either support editing of the field (for example, a **Text Box**) or prevent editing of the field (for example, a **Disabled Text Box**), or that support no data binding, but provide labels for other content components (for example, a **Label**). **Editors** support the largest variety of content components for field values of various types.
  - **Charts** — Content components that bind to and chart data points for a single data source instance based on one or more fields along a value axis and a single field along a category axis (or the equivalent, depending on the chart type). Supported charts include area, bar, donut, line, and pie charts.
  - **Scheduling** — Content components that bind to and provide a graphical representation for scheduling data (for example, a **Calendar**).
  - **Navigation** — Content components that provide no data binding, but that support fully customizable navigation through data, UI, other functional elements of an app (for example a **Toolbar** or **Button**).
  - **Media** — Content components that enable you to add images in BMP, JPG, GIF, and PNG format.
  - **Custom** — Content components that support customizable content (for example, **Custom Html**, which allows you to enter a single `<div>` element).

  Following is more information on working with layout and content components based on this **customer_view** example. For more information on the individual properties of all supported Blank view components, see the Blank view topics in *Kendo UI Builder by Progress: Using the Kendo UI Designer*.
- ***View design panel*** **(in the middle)** — Shows a design simulation of the view, bounded on the left and top with a vertical and horizontal ruler. The Blank view design panel initially contains a single **Row** component containing two empty **Col** components (a single layout row with two empty columns).

  To further define the view instance, you can drag-and-drop additional rows onto the view design panel, then drag-and-drop columns into each row or directly drag-and-drop content components into a row without columns. If you directly drag-and-drop content components into layout rows, the column for the content component is created automatically.

Once you have at least one empty layout column, you can **either** drag-and-drop a single content component into it, **or** you can drag-and-drop one or more layout rows into which you can drag-and-drop their own columns or content components. In this way, you can have layout columns that contain rows, each of which contain their own columns or content components with columns created for them, and each empty column of which can contain either a single content component or its own rows, and so on to any depth within the layout grid. However, the Bootstrap fluid grid system does impose some limitations in how layout columns can be distributed in rows.

An layout column that you add, either directly or as part of adding a content component directly to a row, represents one or more horizontal *slots* of screen space, whose size depends generally on the device and orientation. After add it, you can specify for each column how many slots of space it represents, and Bootstrap dynamically determines how much space that actually is within a given range, based on the device and the remaining available space in its row. The main limitation with Bootstrap is that the sum total of slots for the columns in a row can take up no more than 12 slots of space in their row. Otherwise, excess columns can be bumped to the next row. Also, note that the actual screen size of slots in a row depends on whether the row is nested within a column of a longer row. Any nested row always has less space available than a row at the top level of a view layout. For more information on setting slot sizes for each column, see the description of the **COMPONENT PROPERTIES** pane, below.

The components already configured in the example **customer_view** view design panel include:

- **Three layout rows** — With two columns in the first row and three columns in each of the next two rows.
- **In the first row** — A **Toolbar** configured with four buttons for `CustomerDS` data source navigation in the first column and a separate **Button** (with the text **Save**) in the second column.
- **In the second row** — A **Label** with the text **Customer:** in the first column, a **Disabled Text Box** in the second column for displaying the read-only value of the `CustNum` field from the current record in the `CustomerDS` data source, and a **Text Box** in the third column for displaying and editing the value of the Name field in the current record of the `CustomerDS` data source.
- **In the third row** — A **Label** with the text **Sales Rep:** in the first column, a **Disabled Text Box** in the second column for displaying the read-only value of the `SalesRep` field from the current record in the `CustomerDS` data source (custom coded as a foreign key to the `SalesrepDS` data source), and a **Drop Down List** in the third column for displaying the `RepName` field from the corresponding record of the `SalesrepDS` data source and for selecting a new `SalesrepDS` record (based on its `RepName` field value) in order to change the sales rep associated the current `CustomerDS` record by changing its `SalesRep` field value to the value in the selected `SalesrepDS` record.

How the data source and field values are specified using the example content components and how their values and relationships are updated is described later in this topic.

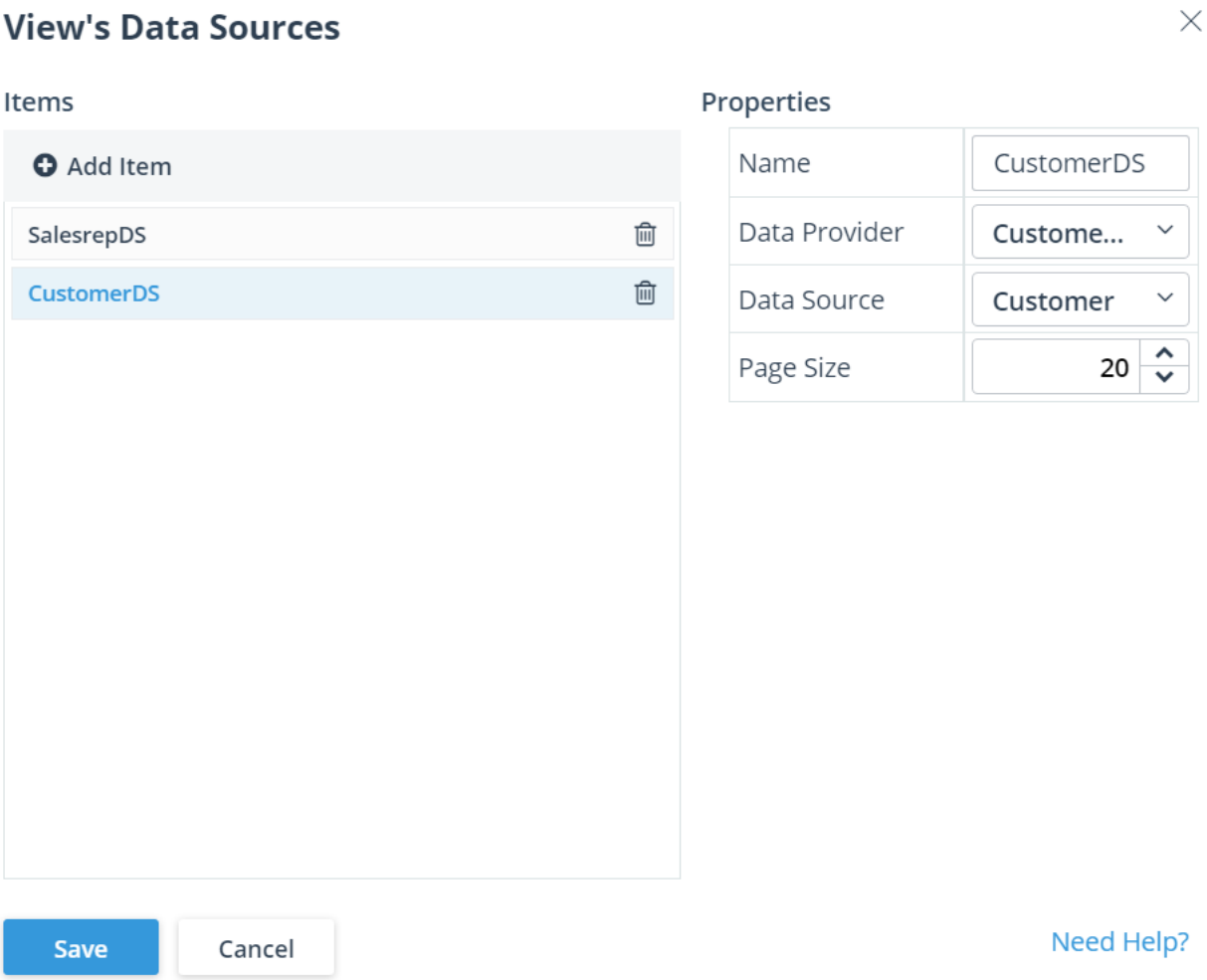
Note: The Blank view design panel also has a single custom HTML top section for coding UI templates for content components that can use them. However, unlike the custom top sections available for predefined views, this custom top section is not available for displaying visible HTML in the layout of the Blank view. For more information on custom HTML sections, see Custom HTML sections on page 122 in this document.

- The **VIEW STYLES pane (in panel on the right)** — provides an **Edit CSS** button that enables you to add and edit CSS styles for the view. Clicking the **Edit CSS** button opens the view's `style.css` file in a Monaco editor. The `style.css`file is located in *`application-folder`*`\app\src\modules\`*`module-folder`*`\`*`view-folder`*, where *`application-folder`* is the path to the folder that contains and is named for your web app, *`module-folder`* is the name of the folder defining the module, and *`view-folder`* is the name of the folder defining the view.

Note: To apply CSS classes (defined in the *style.css* file) to the view, you must specify the class names in the **CSS Class List** property in the **VIEW PROPERTIES** pane.

- **CSS Class List**: The CSS classes that need to be applied to the view. The classes must be defined in the view's *style.css* file, which can be opened and edited by clicking the **Edit CSS** button in the **View Styles** pane on top.
- **VIEW PROPERTIES pane (in panel on the right)** — Contains the properties that you can set for a Blank view instance, regardless of the components it contains. These properties specify the data sources for binding data to the view and are available through **View's Data Sources** by clicking **Edit**, which displays a dialog box similar to what might be shown for the **customer_view** example, as follows:

Figure 38: Adding and editing Blank view data sources



The specified data sources represent instances of data sources you have already defined in data providers that you have created on the app design page (see Data providers and data sources on page 22). (In OpenEdge, a *data source instance* is analogous to an ABL *temp-table*.) In this example, two data source instances have been added for the view, `CustomerDS` and `SalesrepDS`, which names are specified using the **Name** property. As noted, you select the data source definition for each instance using the **Data Provider** and **Data Source** drop down lists. The **Page Size** property specifies the number of records that the data source instance maintains in client memory at a time. This value is also used by data management components, such as the Grid, to specify how many records to display in each page of records to which you can navigate using the component (similar to the **Page Size** property for the predefined Data-Grid* views).

In addition, each data source instance you specify is automatically created with a data model that represents the current record of the data source table. (In OpenEdge, a *data model* is analogous to a *buffer* of an ABL temp-table.) The name of the model is always in the form of *`DataSourceName`*`Model`, where *`DataSourceName`* is the instance name you specify in the **View's Data Sources** dialog box. For example, the `CustomerDS` data source instance has a data model named `CustomerDSModel`.

You can bind any data that you specify for a Blank view to each content component in the view, not to the view itself. Depending on the component, you can then bind the component to any one of the specified data source instances or to a specific field in the current record specified by its data model. For some components, you bind both a data source and its model, and for others, you bind only a data model.

Note that you must implement appropriate event functions to move data between a data source instance and its data model as described later in this section.

Note: You can also use properties in the **Edit View** dialog box to change the default names of event handler functions for general events that apply to all views. You can access this dialog by clicking the ⚙ drop-down menu for each view (as shown for the example **customer_view** view listed in the **VIEWS** pane), then clicking the **Edit** option. Note also that the default behavior for general event functions has no functional effect. You must add custom code to each event handler function in order to implement any useful behavior for it. Typically, you do not need to change the default names of the general event handler functions. However, the view properties exist to allow this name change if you have the need (for example, to avoid a naming conflict with existing JavaScript code you are using elsewhere in the app). For more information on coding general event handler functions (and changing their names in the source), see General view events on page 115 in this document. An example of custom coding for the general `onShow` event function appears in the sample Table 4: controller.public.js file for customer_view on page 80, which is described later in this section.

Caution: You must ensure that any change to the default name of a general event handler function that you make using its view property you also make to the name of the actual JavaScript function in source code.

- **COMPONENT PROPERTIES pane (in panel on the right)** — Contains all the properties that you can set for the single layout or content component you currently have selected in the view design panel. (Property settings are available only for components you have added to the view.) In the **customer_view** view, for example (see Figure 37: Blank view design page on page 70), the first **Col** component is selected in the first row, with its two groups of properties shown as follows:

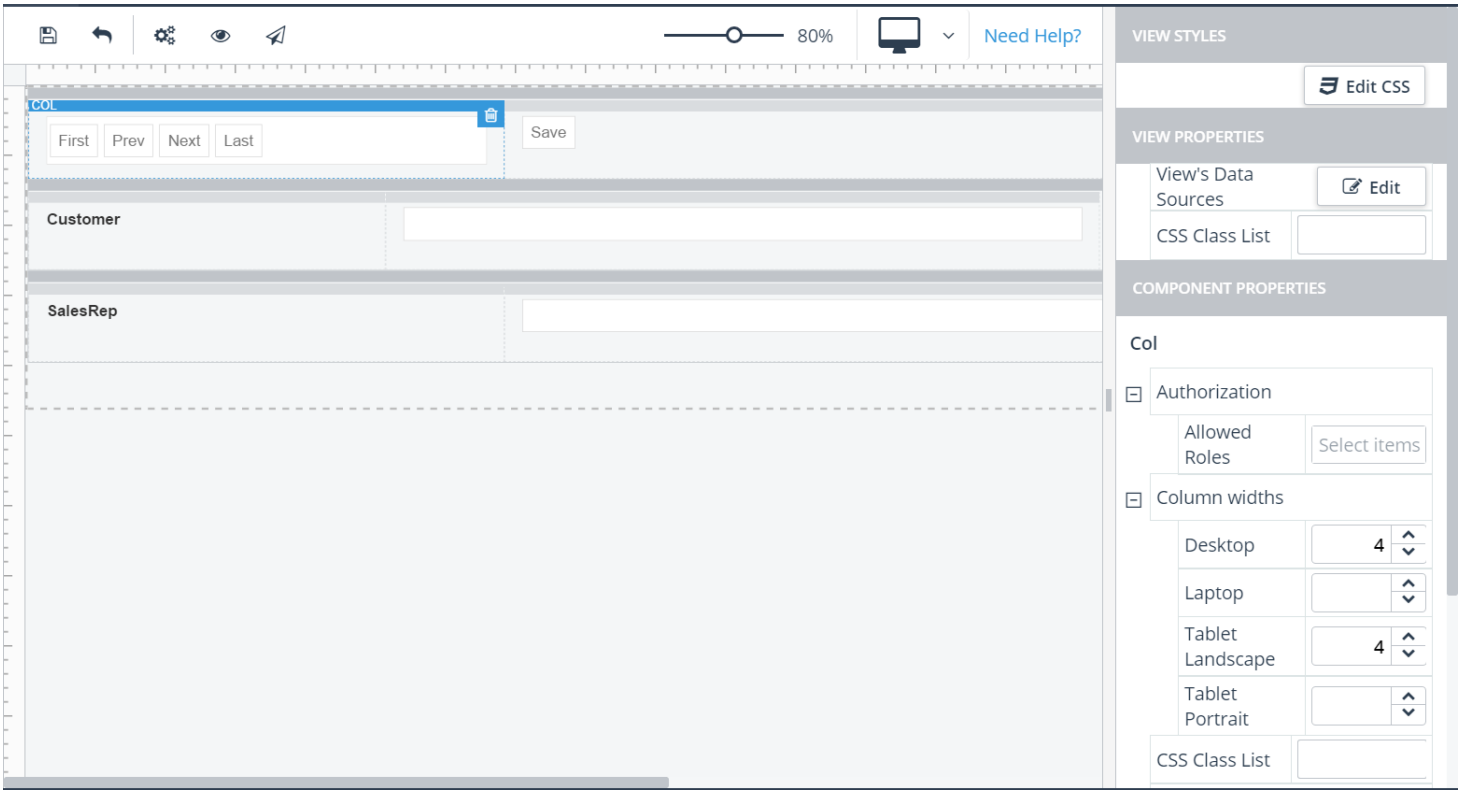
  - **Screen widths** — Properties that you can set to specify the number of slots of horizontal screen space for a given device and orientation. A setting for the **Tablet Landscape** property is required, because this value is used by default for any other device and orientation properties you do not set for the column. Note also that for reliable layout management, the total number of slots set for all columns in the same row cannot exceed 12. For more information, see the description of the *view design panel*, above.

  - **Hide on** — Check-box properties that you can select to hide the column on a given device and orientation.

This section provides more information below on setting properties for content components based on this **customer_view** example. For information on setting the properties of all supported Blank view components, see the Blank view topics in *Kendo UI Builder by Progress: Using the Kendo UI Designer*.

This is the **customer_view** view design panel showing the Toolbar selected with its property settings displayed:

Figure 39: Editing Blank view navigation using a toolbar

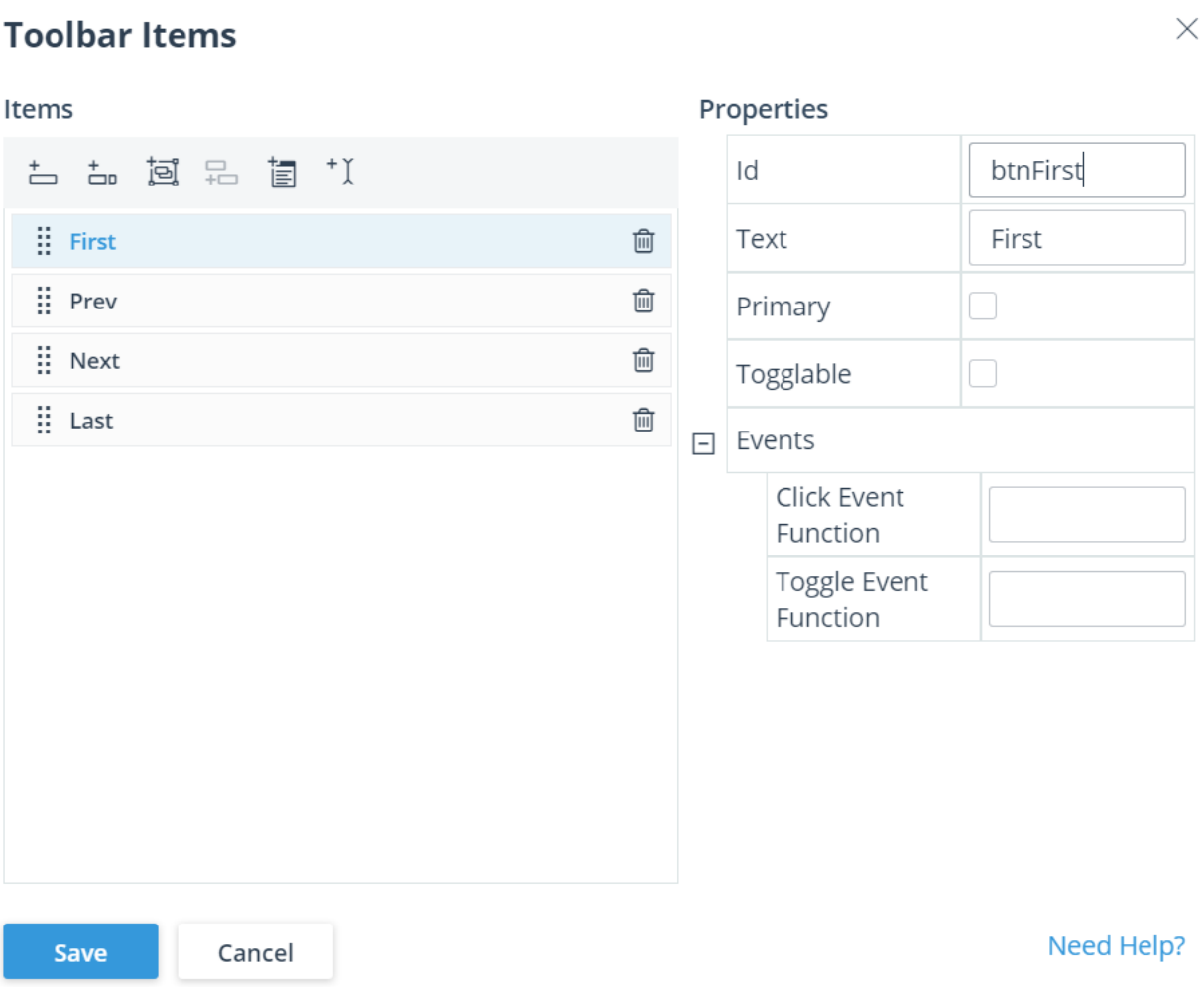


These settings include:

- **Id** — The value `toolbar0`, which you can use to reference the Toolbar instance in event function code. (The Designer provides a default value to uniquely identify every content component in a view.)

- **Toolbar Items** — A dialog box that appears when **Edit** is clicked containing properties to define the toolbar navigation items displayed to the user, as shown:

  Figure 40: Adding and editing Blank view toolbar items



This dialog box allows you to define various combinations of toolbar **Items**, with **+**-icons listed left-to-right, respectively, for adding a:

- Button, similar to the example items
- Split button
- Button group
- Button you can add to any split button or button group you have defined by selecting an existing split button or button group, then clicking the **+**-icon for adding the button
- Button based on a template
- Separator to organize the toolbar items you are adding

The settings for this Toolbar instance include the icons and properties to add or edit the four button items, which are displayed with the label text **First**, **Prev**, **Next**, and **Last**. You can delete an existing item by clicking its trash icon, as shown for these buttons, and you can change the order of added items by dragging-and-dropping them in the **Items** list. You can use the **Properties** that appear for each selected item to complete its definition, then click **Save** to save the latest definitions. The **Properties** for each item, therefore, depend on the type of item that it is. For example, the selected **First** item (like the others for this example) is a button, has its **Id** set to access it in code to `btnFirst` and its **Text** property set to `First`.

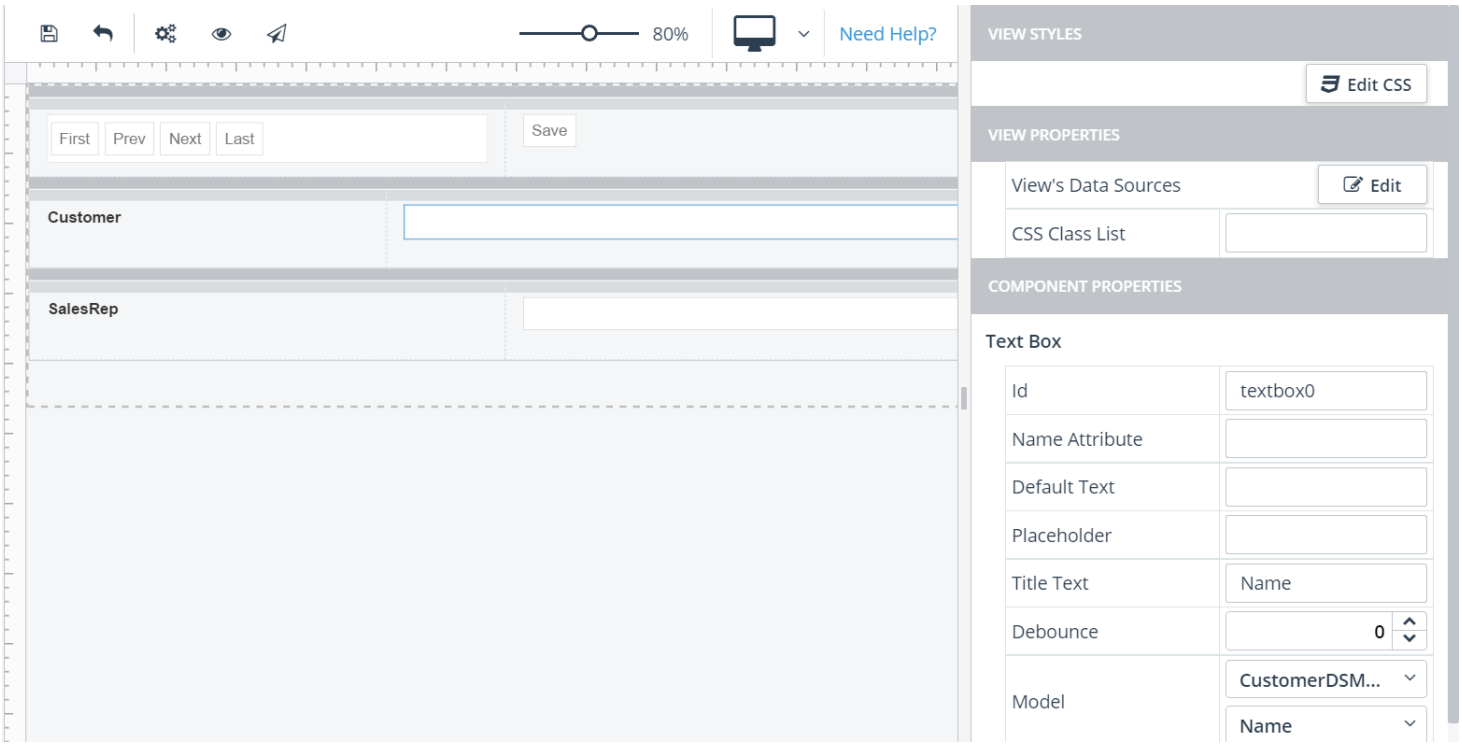
Other properties that are not set in this example include **Primary**, which specifies an item to be highlighted, **Togglable**, which activates the behavior defined by any specified **Toggle Event Function** for the selected item, and also the option to specify a **Click Event Function** for the selected item independent of the toolbar instance itself.

- **Resizable** — Selected in the example, this allows the toolbar to move items into a drop-down list if resizing its container hides some of its items.
- **Events** — Including:
  - **Click Event Function** — Set to `onClickToolbar`, a JavaScript function that you define to execute when the user clicks a toolbar item. This function is where you can reference the ID values you have set for this component to implement its behavior. A sample implementation for this event function appears later in this section.
  - **Toggle Event Function** — A JavaScript function that you define to execute when the user clicks togglable toolbar item.

Note: If you select the **Togglable** check-box for any Toolbar item, you need to set the **Toggle Event Function** to implement the toggle behavior. Note also that none of the navigation components, such as the Toolbar, have a data binding. Instead, you use an appropriate event function to implement navigation through existing data sources and displayed UI elements.

This is the **customer_view** view design panel showing the selected Text Box with its property settings displayed:

Figure 41: Editing Blank view data binding for a component using a model

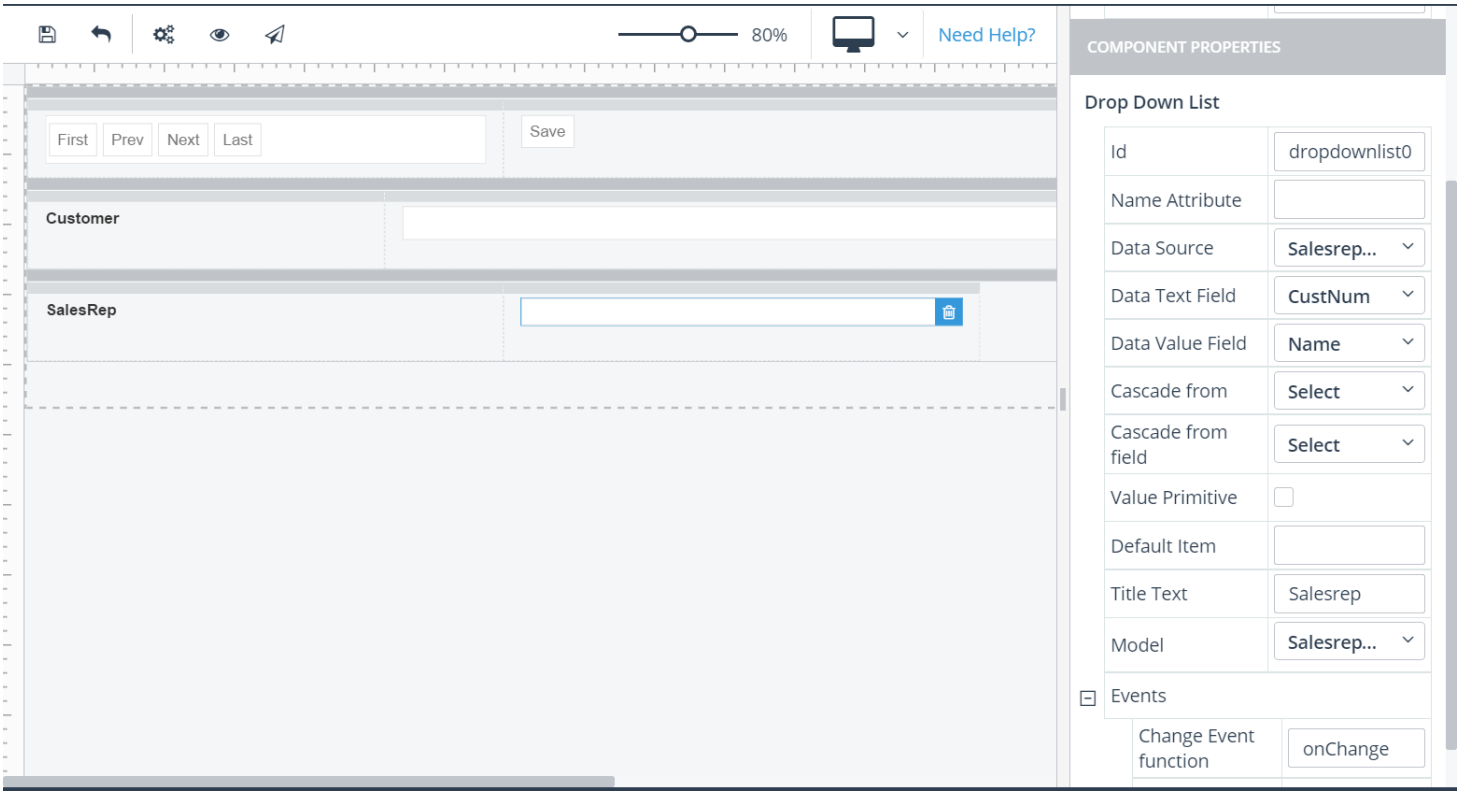


These settings include:

- **Id** — The value `textbox0`, which you can use to reference the Text Box instance in event function code. (The Designer provides a default value to uniquely identify every content component in a view.)
- **Title Text** — The value `Name`, which is optional text that appears when the user hovers over the component.
- **Debounce** — (Default value: `0`) The delay, in milliseconds, between user input and execution of the **Change Event Function**.
- **Model** — Defines the data binding for the Text Box, which includes selection of the data model name for the current **Customer** record (`CustomerDSModel` in this example) and the name of the field in the model (`Name` in this example) whose value you want to display and edit in the Text Box.
- **Change Event Function** — No setting in this example, a JavaScript function that you can define to execute when the user changes the value in the Text Box. This function is one place where you can reference the **Id** you have set for this component to implement its behavior. However, in this **customer_view** example, handling the change in value for this component is entirely implemented using the event function set for the **Click Event Function** property (`onSaveClick`) of the **Save** Button component (**Id**: `button0`). A sample implementation for this event function appears later in this section.

This is the **customer_view** view design panel showing the selected Drop Down List with its property settings displayed:

Figure 42: Editing Blank view data binding for a component using a data source and model



These settings include:

- **Id** — The value `dropdownlist0`, which you can use to reference the Drop Down List instance in event function code. (The Designer provides a default value to uniquely identify every content component in a view.)
- **Data Source Name** — The value `SalesrepDS`, which is the data source selected from the view's data sources to set the component's data binding.
- **Data Text Field** — The value `RepName`, which is the name of the field selected from the specified data source both to display in the Drop Down List that corresponds to the current **Customer** record and to select in the Drop Down List in order to set a new value in the current **Customer** record.
- **Data Value Field** — The value `SalesRep`, which is the name of the field selected from the specified data source to use as the unique foreign key in the current **Customer** record to identify the `SalesrepDS` data source record from which to display the current **Data Text Field** value and to select and set a new current **Data Text Field** value from the available `SalesrepDS` data source records.
- **Cascade from**: The ID of another drop down list in the Blank view. You use this property when you want to build cascading drop down lists in the view. For example, one drop down list may display a list of product categories. When a user makes a selection, the selected category is cascaded to another drop down list, which then displays a list of products for that category. Click here to see an example of cascading drop down lists.

Note: To build cascading drop down lists, you must ensure that the data sources of the drop down lists are related. Kendo UI Builder extracts this relationship from the JSDO catalog of the data provider.

- **Cascade from field**: A field in the data source of the drop down list from which a selection is cascaded to the current drop down list.
- **Value Primitive**: Restricts the model to one field of the record.
- **Default Item**: The item initially displayed in the drop down list. The user can then change the selected item as needed.
- **Title Text** — The value `Salesrep`, which is optional text that appears when the user hovers over the component.
- **Model** — Selects the name of the data model (`SalesrepDSModel` in this example) used to display the current record from the `SalesrepDS` data source or to select a new current record from the key value specified by **Data Value Field** that corresponds to the value the user selects in the list specified by **Data Text Field**.
- **Change Event Function** — Set to `onChange`, a JavaScript function that you can define to execute when the user changes the value selected for the Drop Down List. This function is one place where you can reference the **Id** you have set for this component to implement its behavior. A sample implementation for this event function appears later in this section.

This is an implementation of the `controller.public.js` file where you can code the event functions for this view, with property settings from the **customer_view** example shown in bold:

Table 4: controller.public.js file for customer_view

```
import BaseController from './controller.js'

class Module1CustomerViewCtrl extends BaseController {
    constructor($scope, $injector, stateData) {
        super($scope, $injector);
    }

    // Fired when custom html section is loaded
    includeContentLoaded() {

    }

    // Fired when custom html section loading failed
    includeContentError(e) {

    }

    // Fired when view content is loaded
    onShow($scope) {
        var that = this;
        this.customerDS = $scope.vm.$ds.CustomerDS;
        this.salesrepDS = $scope.vm.$ds.SalesrepDS;
        this.customerModel = $scope.vm.$viewModels.CustomerDSModel;
        this.salesrepModel = $scope.vm.$viewModels.SalesrepDSModel;

        this.customerIdx = 0;

        this.customerDS.read().done(function(){
            var data = that.customerDS.data();

            that.displayCustomer(0);
            $scope.$apply();

        });
    }

    displayCustomer(operation){
        var data = this.customerDS.data();
```

```
        switch(operation) {
        case 0:
            this.customerIdx = 0;
            break;
        case -1:
            if (this.customerIdx) {
                this.customerIdx -= 1;
            }
            else {
                this.customerIdx = 0;
            }
            break;
        case 1:
            if (this.customerIdx === (data.length - 1)) {
                this.customerIdx = data.length - 1;
            }
            else {
                this.customerIdx += 1;
            }
            break;
        default:
            this.customerIdx = data.length - 1;
        }

        this.customerModel.id = data[this.customerIdx].id;
        this.customerModel.CustNum = data[this.customerIdx].CustNum;
        this.customerModel.Name = data[this.customerIdx].Name;
        this.customerModel.SalesRep = data[this.customerIdx].SalesRep;

        this.displaySalesrep();

    }

    onClickToolbar(e){
        switch(e.id) {
        case "btnFirst":
            this.displayCustomer(0);
            break;
        case "btnPrev":
            this.displayCustomer(-1);
            break;
        case "btnNext":
            this.displayCustomer(1);
            break;
        case "btnLast":
            this.displayCustomer();
            break;
        }

    }

    displaySalesrep(e){
        if (this.customerModel.SalesRep) {
            this.salesrepDS.filter({
                field: "SalesRep",
                operator: "equals",
                value: this.customerModel.SalesRep });
            var view = this.salesrepDS.view();

            if (view.length) {
                this.salesrepModel.SalesRep = view[0].SalesRep;
                this.salesrepModel.RepName = view[0].RepName;
            }
            else {
                this.salesrepModel.SalesRep = "";
                this.salesrepModel.RepName = "";
            }
            this.salesrepDS.filter({});
```

```
            if (this.salesrepModel.RepName) {
                angular.element("#dropdownlist0"
                    ).data("kendoDropDownList"
                        ).search(this.salesrepModel.RepName);
            }
        } else {
            angular.element("#dropdownlist0").data("kendoDropDownList").value("");
        }

    }

    onChange(e){
        this.customerModel.SalesRep = angular.element("#dropdownlist0"
                                      ).data("kendoDropDownList").dataItem().SalesRep;

    }

    onSaveClick(e){
        var dataItem = this.customerDS.get(this.customerModel.id);
        var that = this;

        dataItem.set("Name", this.customerModel.Name);
        dataItem.set("SalesRep", this.customerModel.SalesRep);

        this.customerDS.sync().fail(function(){
            that.customerDS.cancelChanges();
        });
    }
}

Module1CustomerViewCtrl.$inject = ['$scope', '$injector', 'stateData'];

export default Module1CustomerViewCtrl
```

For more information on `controller.public.js` files and where to find the file that is generated for a given view, see the topics on coding view event functions in Extension Points and Source Code Customization on page 103.

The Kendo UI Builder also supports a `dsService` API that is inherited by but not used in the sample `controller.public.js` file for the **CustomerView** example.

In a Blank view, as described previously, you have data sources and data models. Every data source has a corresponding model property which represents the signature of an item (record) from a data source and its name is *`DataSourceName`*`Model`, where *`DataSourceName`* is the property that references the corresponding data source, for example `CustomerDSModel` for the `CustomerDS` data source. These data source and model properties live in `$scope.vm.$ds` and `$scope.vm.$viewModels` of the view, respectively. Note that these same properties can also be accessed as array references on `this.$ds` and `this.$viewModels`, for example, `this.$ds['CustomerDS']` and `this.$viewModels['CustomerDSModel']`, which are used in the following `dsService` API examples.

If you set the **Model** property of a data bound component, that model represents the selected (current) item in the data source. Having that, you can add input (editor) components bound to the same model and update the data source item (because the model is actually an item from its corresponding data source).

At some point you might want to update the changes you have made in the data source. You can achieve this by using the `dsService` API that is injected into every `controller.js` file in the generated application and inherited by its corresponding `controller.public.js` file. The `controller.public.js` file is the place where you add your event functions. For example, you might have five Button components that have the following five corresponding click event functions that each call an appropriate `dsService` function: `removeClick`, `cancelClick`, `saveClick`, `createClick`, and `createModelClick`. You can then update the changes in the data source by adding these event functions to the `controller.public.js` file of your view and setting the **Click Event Function** property of each Button to the name of the corresponding function.

This is a description of the available `dsService` functions your event functions can call:

Table 5: dsService API

```
// Returns a new pristine model. Use this function to set your model when
// you want to insert a new item.
createPristineModel: function(dataSource) {
},

// Adds the model to the dataSource and returns the model.
// By default, this operation updates only the local dataSource and
// does not sync with the server.
// If you want to immediately sync the dataSource, you need to pass
// this options value: var options = { sync: true};
create: function(dataSource, model, options) {
},

// Removes the model from the dataSource and returns a pristine model.
// By default, this operation removes the item only from the local dataSource
// and does not sync with the server.
// If you want to immediately sync the dataSource, you need to pass
// this options value:  var options = { sync: true};
remove: function(dataSource, model, options) {
    return new dataSource.reader.model();
},

// Syncs the local dataSource to the server.
save: function(dataSource) {
    dataSource.sync();
},

// Cancels all local dataSource changes, resets the dataSource to the server one,
// and returns a pristine model.
cancel: function (dataSource, model) {
}
```

This API resides in the `app/src/scripts/services/data-source-factory.js` file under each web app project folder. (Note also that the `extractErrorMessage(e)` function also included in this file is a utility function for the Kendo UI Builder and is not supported as part of this `dsService` API.) You can reference each of these API functions using `this.$dsService`, which is set in and inherited from the view's base `controller.js` file as follows: `this.$dsService = $injector.get('dsService');`

Following are the five example click event functions implemented to call the corresponding `dsService` functions:

Table 6: dsService API examples

```
removeClick: function () {
    this.$dsService.remove(this.$ds['CustomerDS'],
                           this.$viewModels['CustomerDSModel']);
},

cancelClick: function () {
    this.$viewModels['CustomerDSModel']
        = this.$dsService.cancel(this.$ds['CustomerDS'],
                                 this.$viewModels['CustomerDSModel']);
},

saveClick: function () {
    this.$dsService.save(this.$ds['CustomerDS']);
},

createClick: function () {
    this.$viewModels['CustomerDSModel']
```

```
        = this.$dsService.create(this.$ds['CustomerDS'],
                                 this.$viewModels['CustomerDSModel']);
},

createModelClick: function () {
    this.$viewModels['CustomerDSModel']
        = this.$dsService.createPristineModel(this.$ds['CustomerDS']);
}
```

These examples do not sync with the server, except for the call to `this.$dsService.save()` to explicitly sync the local data source with the server.

Stacked-Data-Grids view

Selecting the Stacked-Data-Grids view creates a vertically split screen with two separate grids. On the top, a parent grid arranges rows of included fields from a parent data source by column, and in the same order as they appear in the **Included Columns** list of the Grid Columns dialog for the parent grid (described below). On the bottom, a child grid arranges related rows of included fields from a child data source by column, and in the same order as they appear in the **Included Columns** list of the Grid Columns dialog for the child grid (described below).

The parent grid provides read-only access to its data source. When the user selects a row in the parent grid, the child grid displays the related rows from its child data source and provides read-only or editable access to that data source as specified by its properties. Like any grid, both the parent and child grid allow you to page through all their rows using their own page selection controls. In this way, you can page through all related child rows by paging and clicking every parent row, then paging through all the rows displayed in the child grid.

Note: Both the selected parent and child data sources must reference tables from the same OpenEdge Data Object Service, where both tables are bound by a parent-child relationship in the same Data Object resource. This resource must be created for an OpenEdge ProDataSet that includes both the two corresponding temp-tables and their data-relation as part of its definition. Therefore, no data sources from either an OData Provider or a Generic REST Provider are available to select for this view.

The Stacked-Data-Grids view has a number of customizable properties, listed on the right side of the view design page:

- **Data Provider**: The selected data provider for this view.
- **Parent Data Source**: The selected parent-grid data source from the chosen data provider.
- **Child Data Source**: The selected child-grid data source from the chosen data provider.
- **Parent Grid Properties**:
  - **Grid Title**: The title that appears for the parent grid when this view is rendered on the screen for the app.
  - **Grid Columns**: Click **Edit** to change the included columns and various properties of the columns.
  - **Page Size**: The number of rows that will be displayed per page in the grid.
  - **Row Template Function**: The name of a JavaScript function that returns the text for a Kendo UI rowTemplate. If specified, the row template is used to format all the rows in the grid unless you also define an altRowTemplate. For more information, see the entry for rowTemplate at http://docs.telerik.com/kendo-ui/api/javascript/ui/grid#configuration-rowTemplate, in the *Kendo UI by Progress Documentation and API Reference*. For more information on defining a rowTemplate for a Kendo UI Builder editable grid, see *Kendo UI Builder by Progress: Modernizing OpenEdge Applications*.

- **Row Template Id**: The ID of an HTML rowTemplate. If specified, the rowTemplate must be defined in either the *application-folder*\app\src\modules*module-folder**view-folder*\topSection.html file, the \middleSection.html file, or the \bottomSection.html file. If both a Row Template Function and a Row Template Id are specified, the Row Template Id takes precedence.
- **Alt Row Template Function**: The name of a JavaScript function that returns the text for a Kendo UI altRowTemplate. If specified, the altRowTemplate is used to format every other row in the grid. For more information, see the entry for altRowTemplate at http://docs.telerik.com/kendo-ui/api/javascript/ui/grid#configuration-altRowTemplate, in the *Kendo UI by Progress Documentation and API Reference*. For more information on defining a rowTemplate for a Kendo UI Builder editable grid, see *Kendo UI Builder by Progress: Modernizing OpenEdge Applications*
- **Alt Row Template Id**: The ID of an HTML altRowTemplate. If specified, the altRowTemplate must be defined in either the *application-folder*\app\src\modules*module-folder**view-folder*\topSection.html file, the \middleSection.html file, or the \bottomSection.html file. If both an Alt Row Template Function and an Alt Row Template Id are specified, the Row Template Id takes precedence.
- **Enable Column Filtering**: Provides the app user with the ability to filter the displayed information based on content, including filters for numeric values (e.g., is equal to, is not equal to) and text fields (e.g., starts with, does not contain, etc.).

  Note: Filtering, sorting, and paging operations execute based on the **Client-side processing** setting for the data source. Operations are performed on the client if **Client-side processing** is selected; otherwise, the operations are performed on the server side.

- **Enable Grouping**: Provides the app user with the ability to group the rows according to the value of a particular column. For example, in a grid with columns representing customer names and order numbers, grouping by customer name will arrange the rows so that all of the order numbers from each given customer are grouped together. Grouping should only be used with client-side processing.
- **Enable Column Resize**: Provides the app user with the ability to resize the columns by dragging the column divider(s).
- **Enable Column Reordering**: Provides the app user with the ability to reorder the columns by dragging and dropping the column headers.
- **Enable Sorting**: Provides the app user with the ability to reorder rows by ascending or descending value in a given column.
- **Events**:
  - **Row Select Event Function**: The name of a JavaScript function that runs when a parent row is selected by the app user.
  - **Data Bound Event Function**: The name of a JavaScript function that runs when the parent grid is bound to its data source.

  The code for these event functions must be included in the file, *application-folder*\app\src\modules*module-folder**view-folder*\controller.public.js. See *Kendo UI Builder by Progress: Modernizing OpenEdge Applications* for more information about extensions and event functions.

- **Child Grid Properties**:
  - **Grid Title**: The title that appears for the child grid when this view is rendered on the screen for the app.

- **Edit Mode**:
  - **ReadOnly**: This mode supports read-only access to data in the grid.
  - **Incell**: This mode supports edits to multiple rows at a time in the grid. Data editing is supported using the fields of the grid itself. You initiate editing by clicking a column within a row. Each row also provides a button to delete the associated row in the grid. In this mode, the toolbar above the grid provides buttons to create a new row, and to save all changes or cancel all changes to the rows in the grid.
  - **Inline**: This mode supports edits to a single row at a time in the grid. Data editing is supported using the fields of the row itself. You initiate editing by clicking an edit button on the row you want to update. Each row also provides a button to delete the associated row in the grid. In this mode, the toolbar above the grid provides a single button to create a new row in the grid.
  - **Popup**: This mode supports edits to a single row at a time in the grid. Data editing is supported using a pop-up form. You initiate editing by clicking an edit button on the row you want to update. Each row also provides a button to delete the associated row in the grid. In this mode, the toolbar above the grid provides a single button to create a new row in the grid.
- **Edit Options**:
  - **Allow Insert**: An option that allows new rows to be created in the grid.
  - **Allow Edit**: An option that allows existing rows to be edited in the grid.
  - **Allow Delete**: An option that allows existing rows to be deleted in the grid.
  - **Toolbar Button Labels**:
    - **Cancel**: A custom label for the button that cancels all pending edits in the grid. The default is "Cancel changes".
    - **Create**: A custom label for the button that creates a new row in the grid. The default is "New".
    - **Save**: A custom label for the button that saves all pending edits in the grid. The default is "Save changes".
  - **Row Button Labels**:
    - **Cancel Edit**: A custom label for the button that cancels all pending edits in the row. The default is "Cancel".
    - **Delete**: A custom label for the button that deletes the row. The default is "Delete".
    - **Edit**: A custom label for the button that initiates editing in the row. The default is "Edit".
    - **Update**: A custom label for the button that confirms any pending delete or edits to the row. The default is "Update".
- **Grid Columns**: Click **Edit** to change the included columns and various properties of the columns.
- **Page Size**: The number of rows that will be displayed per page in the grid.
- **Row Template Function Name**: The name of a JavaScript function that returns the text for a Kendo UI rowTemplate. If specified, the row template is used to format all the rows in the grid unless you also define an altRowTemplate. For more information, see the entry for rowTemplate at http://docs.telerik.com/kendo-ui/api/javascript/ui/grid#configuration-rowTemplate, in the *Kendo UI by Progress Documentation and API Reference*. For more information on defining a rowTemplate for a Kendo UI Builder editable grid, see *Kendo UI Builder by Progress: Modernizing OpenEdge Applications*.

- **Row Template Id**: The ID of an HTML rowTemplate. If specified, the rowTemplate must be defined in either the *application-folder*\app\src\modules*module-folder**view-folder*\topSection.html file, the \topParentSection.html file, the \middleSection.html file, the \topChildSection.html file, or the \bottomSection.html file. If both a Row Template Function and a Row Template Id are specified, the Row Template Id takes precedence.
- **Alt Row Template Function Name**: The name of a JavaScript function that returns the text for a Kendo UI altRowTemplate. If specified, the altRowTemplate is used to format every other row in the grid. For more information, see the entry for altRowTemplate at http://docs.telerik.com/kendo-ui/api/javascript/ui/grid#configuration-altRowTemplate, in the *Kendo UI by Progress Documentation and API Reference*. For more information on defining a rowTemplate for a Kendo UI Builder editable grid, see *Kendo UI Builder by Progress: Modernizing OpenEdge Applications*
- **Alt Row Template Id**: The ID of an HTML altRowTemplate. If specified, the altRowTemplate must be defined in either the *application-folder*\app\src\modules*module-folder**view-folder*\topSection.html file, the \middleSection.html file, or the \bottomSection.html file. If both an Alt Row Template Function and an Alt Row Template Id are specified, the Row Template Id takes precedence.
- **Enable Column Filtering**: Provides the app user with the ability to filter the displayed information based on content, including filters for numeric values (e.g., is equal to, is not equal to) and text fields (e.g., starts with, does not contain, etc.).
- **Enable Grouping**: Provides the app user with the ability to group the rows according to the value of a particular column. For example, in a grid with columns representing customer names and order numbers, grouping by customer name will arrange the rows so that all of the order numbers from each given customer are grouped together. Grouping should only be used with client-side processing.
- **Enable Column Resize**: Provides the app user with the ability to resize the columns by dragging the column divider(s).
- **Enable Column Reordering**: Provides the app user with the ability to reorder the columns by dragging and dropping the column headers.
- **Enable Sorting**: Provides the app user with the ability to reorder rows by ascending or descending value in a given column.
- **Selection Type**: Determines if rows in a grid can be selected:
  - **None**: No row selection is allowed (default).
  - **Single**: One row can be selected at a time.
  - **Multiple**: Multiple rows can be selected at one time.

Note: Row selection has no function except to invoke the Row Select Event for each selected row.

- **Events**:
  - **Row Select Event Function**: The name of a JavaScript function that runs when a child row is selected by the app user.
  - **Data Bound Event Function**: The name of a JavaScript function that runs when the child grid is bound to its data source.
  - **Row Create Event Function**: The name of a JavaScript function that runs before a row is created for a new data source record.

- **Row Update Event Function**: The name of a JavaScript function that runs before an existing data source record is updated in the row.
- **Row Delete Event Function**: The name of a JavaScript function that runs before an existing data source record is deleted in the row.

The code for these event functions must be included in the file, *application-folder*\app\src\modules*module-folder**view-folder*\controller.public.js. See *Kendo UI Builder by Progress: Modernizing OpenEdge Applications* for more information about extensions and event functions.

- **CSS Class List**: The CSS classes that need to be applied to the view. The classes must be defined in the view's *style.css* file, which can be opened and edited by clicking the **Edit CSS** button in the **View Styles** pane on top.

This view also includes the following custom HTML sections where you can include your own HTML for the view. Each section is identified in the view design page with a placeholder containing descriptive text:

- **Custom top html section:** Located in the area above the parent Grid Title, the text in this placeholder introduces the view path name of the `topSection.html` file where you can add your custom HTML.
- **Custom top parent html section:**: Located in the area below the parent Grid Title but above the parent grid itself, the text in this placeholder introduces the view path name of the `topParentSection.html` file where you can add your custom HTML.
- **Custom middle html section:**: Located in the area below the parent grid and above the child Grid Title, the text in this placeholder introduces the view path name of the `middleSection.html` file where you can add your custom HTML.
- **Custom top child html section:**: Located in the area below the child Grid Title but above the child grid itself, the text in this placeholder introduces the view path name of the `topChildSection.html` file where you can add your custom HTML.
- **Custom bottom html section:**: Located in the area below the child grid, the text in this placeholder introduces the view path name of the `bottomSection.html` file where you can add your custom HTML.

The Designer automatically generates these three files for you in *application-folder*\app\src\modules*module-folder**view-folder*. Add your custom HTML code to these files as required.

Hierarchical-Data-Grid view

Selecting the Hierarchical-Data-Grid view creates a single scrolling grid structure that contains a single parent grid containing rows from a parent data source. This parent grid then embeds a child grid below each parent row that has related records in the child data source. As you scroll the grid structure and click an expander on parent rows, any child grid containing related data source rows displays below its related row in the parent grid. You can also click the expander on an expanded child grid to collapse the grid into its parent.

Both the parent and its child grids arrange the included fields from their respective data sources by column, in the same order as they appear in the **Included Columns** list of the Grid Columns dialog for the parent its child grids (described below).

The parent grid provides read-only access to its data source. When the user selects the expander on a row in the parent grid, if this parent row has related child rows, a child grid displays the related rows from its child data source and provides read-only or editable access to that data source as specified by its properties. Like any grid, both the parent and its child grids allow you to page through all their rows using page selection controls on each grid. In this way, you can page through all related child rows by paging and expanding every parent row, then paging through all the rows of each displayed child grid.

Note: Both the selected parent and child data sources must reference tables from the same OpenEdge Data Object Service, where both tables are bound by a parent-child relationship in the same Data Object resource. This resource must be created for an OpenEdge ProDataSet that includes both the two corresponding temp-tables and their data-relation as part of its definition. Therefore, no data sources from either an OData Provider or a Generic REST Provider are available to select for this view.

The Hierarchical-Data-Grid view has a number of customizable properties, listed on the right side of the view design page:

- **Data Provider**: The selected data provider for this view.
- **Parent Data Source**: The selected parent-grid data source from the chosen data provider.
- **Child Data Source**: The selected child-grid data source from the chosen data provider.
- **CSS Class List**: The CSS classes that need to be applied to the view. The classes must be defined in the view's *style.css* file, which can be opened and edited by clicking the **Edit CSS** button in the **View Styles** pane on top.
- **Parent Grid Properties**:
  - **Grid Title**: The title that appears for the parent grid when this view is displayed on the screen for the app.
  - **Grid Columns**: Click **Edit** to change the included columns and various properties of the columns.
  - **Page Size**: The number of rows that will be displayed per page in the grid.
  - **Row Template Function**: The name of a JavaScript function that returns the text for a Kendo UI rowTemplate. If specified, the row template is used to format all the rows in the grid unless you also define an altRowTemplate. For more information, see the entry for rowTemplate at http://docs.telerik.com/kendo-ui/api/javascript/ui/grid#configuration-rowTemplate, in the *Kendo UI by Progress Documentation and API Reference*. For more information on defining a rowTemplate for a Kendo UI Builder editable grid, see *Kendo UI Builder by Progress: Modernizing OpenEdge Applications*.
  - **Row Template Id**: The ID of an HTML rowTemplate. If specified, the rowTemplate must be defined in either the *application-folder*\app\src\modules*module-folder**view-folder*\topSection.html file, the \middleSection.html file, or the \bottomSection.html file. If both a Row Template Function and a Row Template Id are specified, the Row Template Id takes precedence.
  - **Alt Row Template Function**: The name of a JavaScript function that returns the text for a Kendo UI altRowTemplate. If specified, the altRowTemplate is used to format every other row in the grid. For more information, see the entry for altRowTemplate at http://docs.telerik.com/kendo-ui/api/javascript/ui/grid#configuration-altRowTemplate, in the *Kendo UI by Progress Documentation and API Reference*. For more information on defining a rowTemplate for a Kendo UI Builder editable grid, see *Kendo UI Builder by Progress: Modernizing OpenEdge Applications*
  - **Alt Row Template Id**: The ID of an HTML altRowTemplate. If specified, the altRowTemplate must be defined in either the *application-folder*\app\src\modules*module-folder**view-folder*\topSection.html file, the \middleSection.html file, or the \bottomSection.html file. If both an Alt Row Template Function and an Alt Row Template Id are specified, the Row Template Id takes precedence.
  - **Enable Column Filtering**: Provides the app user with the ability to filter the displayed information based on content, including filters for numeric values (e.g., is equal to, is not equal to) and text fields (e.g., starts with, does not contain, etc.).

Note: Filtering, sorting, and paging operations execute based on the **Client-side processing** setting for the data source. Operations are performed on the client if **Client-side processing** is selected; otherwise, the operations are performed on the server side.

- **Enable Grouping**: Provides the app user with the ability to group the rows according to the value of a particular column. For example, in a grid with columns representing customer names and order numbers, grouping by customer name will arrange the rows so that all of the order numbers from each given customer are grouped together. Grouping should only be used with client-side processing.
- **Enable Column Resize**: Provides the app user with the ability to resize the columns by dragging the column divider(s).
- **Enable Column Reordering**: Provides the app user with the ability to reorder the columns by dragging and dropping the column headers.
- **Enable Sorting**: Provides the app user with the ability to reorder rows by ascending or descending value in a given column.
- **Events**:
  - **Row Select Event Function**: The name of a JavaScript function that runs when a parent row is selected by the app user.
  - **Data Bound Event Function**: The name of a JavaScript function that runs when the parent rows for the grid are bound to its parent data source.
  - **Detail Init Event Function**: The name of a JavaScript function that runs before the child rows for a parent expand.
  - **Detail Expand Event Function**: The name of a JavaScript function that runs after the child rows for a parent expand.
  - **Detail Collapse Event Function**: The name of a JavaScript function that runs after the user collapses the child rows for a parent.

  The code for these event functions must be included in the file, *application-folder*\app\src\modules*module-folder**view-folder*\controller.public.js. See *Kendo UI Builder by Progress: Modernizing OpenEdge Applications* for more information about extensions and event functions.

- **Child Grid Properties**:
  - **Grid Title**: The title that appears for each child grid when this view is displayed on the screen for the app.
  - **Edit Mode**:
    - **ReadOnly**: This mode supports read-only access to data in the grid.
    - **Incell**: This mode supports edits to multiple rows at a time in the grid. Data editing is supported using the fields of the grid itself. You initiate editing by clicking a column within a row. Each row also provides a button to delete the associated row in the grid. In this mode, the toolbar above the grid provides buttons to create a new row, and to save all changes or cancel all changes to the rows in the grid.
    - **Inline**: This mode supports edits to a single row at a time in the grid. Data editing is supported using the fields of the row itself. You initiate editing by clicking an edit button on the row you want to update. Each row also provides a button to delete the associated row in the grid. In this mode, the toolbar above the grid provides a single button to create a new row in the grid.

    - **Popup**: This mode supports edits to a single row at a time in the grid. Data editing is supported using a pop-up form. You initiate editing by clicking an edit button on the row you want to update. Each row also provides a button to delete the associated row in the grid. In this mode, the toolbar above the grid provides a single button to create a new row in the grid.

- **Edit Options**:
    - **Allow Insert**: An option that allows new rows to be created in the grid.
    - **Allow Edit**: An option that allows existing rows to be edited in the grid.
    - **Allow Delete**: An option that allows existing rows to be deleted in the grid.
    - **Toolbar Button Labels**:
        - **Cancel**: A custom label for the button that cancels all pending edits in the grid. The default is "Cancel changes".
        - **Create**: A custom label for the button that creates a new row in the grid. The default is "New".
        - **Save**: A custom label for the button that saves all pending edits in the grid. The default is "Save changes".
    - **Row Button Labels**:
        - **Cancel Edit**: A custom label for the button that cancels all pending edits in the row. The default is "Cancel".
        - **Delete**: A custom label for the button that deletes the row. The default is "Delete".
        - **Edit**: A custom label for the button that initiates editing in the row. The default is "Edit".
        - **Update**: A custom label for the button that confirms any pending delete or edits to the row. The default is "Update".

- **Grid Columns**: Click **Edit** to change the included columns and various properties of the columns.
- **Page Size**: The number of rows that will be displayed per page in the grid.
- **Row Template Function Name**: The name of a JavaScript function that returns the text for a Kendo UI rowTemplate. If specified, the row template is used to format all the rows in the grid unless you also define an altRowTemplate. For more information, see the entry for rowTemplate at http://docs.telerik.com/kendo-ui/api/javascript/ui/grid#configuration-rowTemplate, in the *Kendo UI by Progress Documentation and API Reference*. For more information on defining a rowTemplate for a Kendo UI Builder editable grid, see *Kendo UI Builder by Progress: Modernizing OpenEdge Applications*.
- **Row Template Id**: The ID of an HTML rowTemplate. If specified, the rowTemplate must be defined in either the *application-folder*\app\src\modules*module-folder**view-folder*\topSection.html file, the \topParentSection.html file, the \middleSection.html file, the \topChildSection.html file, or the \bottomSection.html file. If both a Row Template Function and a Row Template Id are specified, the Row Template Id takes precedence.
- **Alt Row Template Function Name**: The name of a JavaScript function that returns the text for a Kendo UI altRowTemplate. If specified, the altRowTemplate is used to format every other row in the grid. For more information, see the entry for altRowTemplate at http://docs.telerik.com/kendo-ui/api/javascript/ui/grid#configuration-altRowTemplate, in the *Kendo UI by Progress Documentation and API Reference*. For more information on defining a rowTemplate for a Kendo UI Builder editable grid, see *Kendo UI Builder by Progress: Modernizing OpenEdge Applications*

- **Alt Row Template Id**: The ID of an HTML altRowTemplate. If specified, the altRowTemplate must be defined in either the *application-folder*\app\src\modules*module-folder**view-folder*\topSection.html file, the \middleSection.html file, or the \bottomSection.html file. If both an Alt Row Template Function and an Alt Row Template Id are specified, the Row Template Id takes precedence.
- **Enable Column Filtering**: Provides the app user with the ability to filter the displayed information based on content, including filters for numeric values (e.g., is equal to, is not equal to) and text fields (e.g., starts with, does not contain, etc.).
- **Enable Grouping**: Provides the app user with the ability to group the rows according to the value of a particular column. For example, in a grid with columns representing customer names and order numbers, grouping by customer name will arrange the rows so that all of the order numbers from each given customer are grouped together. Grouping should only be used with client-side processing.
- **Enable Column Resize**: Provides the app user with the ability to resize the columns by dragging the column divider(s).
- **Enable Column Reordering**: Provides the app user with the ability to reorder the columns by dragging and dropping the column headers.
- **Enable Sorting**: Provides the app user with the ability to reorder rows by ascending or descending value in a given column.
- **Selection Type**: Determines if rows in a grid can be selected:
  - **None**: No row selection is allowed (default).
  - **Single**: One row can be selected at a time.
  - **Multiple**: Multiple rows can be selected at one time.

  Note: Row selection has no function except to invoke the Row Select Event for each selected row.

- **Events**:
  - **Row Select Event Function**: The name of a JavaScript function that runs when a child row is selected by the app user.
  - **Data Bound Event Function**: The name of a JavaScript function that runs when the child rows for the grid are bound to its child data source.
  - **Row Create Event Function**: The name of a JavaScript function that runs before a row is created for a new data source record.
  - **Row Update Event Function**: The name of a JavaScript function that runs before an existing data source record is updated in the row.
  - **Row Delete Event Function**: The name of a JavaScript function that runs before an existing data source record is deleted in the row.

  The code for these event functions must be included in the file, *application-folder*\app\src\modules*module-folder**view-folder*\controller.public.js. See *Kendo UI Builder by Progress: Modernizing OpenEdge Applications* for more information about extensions and event functions.

This view also includes the following custom HTML sections where you can include your own HTML for the view. Each section is identified in the view design page with a placeholder containing descriptive text:

- **Custom top html section:** Located in the area above the parent Grid Title, the text in this placeholder introduces the view path name of the `topSection.html` file where you can add your custom HTML.
- **Custom top parent html section:**: Located in the area below the parent Grid Title but above the parent grid itself, the text in this placeholder introduces the view path name of the `topParentSection.html` file where you can add your custom HTML.
- **Custom top child html section:**: Located in the area below each child Grid Title, immediately under the related row of the parent grid, but above the child grid itself, the text in this placeholder introduces the view path name of the `topChildSection.html` file where you can add your custom HTML.
- **Custom bottom child html section:**: Located in the area below each child grid but above either the next parent row or the bottom area of the parent grid, the text in this placeholder introduces the view path name of the `bottomChildSection.html` file where you can add your custom HTML.
- **Custom bottom html section:**: Located in the area below the parent grid, the text in this placeholder introduces the view path name of the `bottomSection.html` file where you can add your custom HTML.

The Designer automatically generates these three files for you in *`application-folder`*`\app\src\modules\`*`module-folder`*`\`*`view-folder`*. Add your custom HTML code to these files as required.

Using roles to authorize user access

Kendo UI Designer enables you to set up user roles to limit user access to modules, views, and Blank view rows and columns. To set up user roles with an OpenEdge backend, you need to have a data provider that is configured with form or basic authentication and a custom Business Entity that returns user roles on a service deployed on a Progress Application Server (PAS for OpenEdge) instance in the backend. You must then define roles at different levels in the app in Kendo UI Designer and write custom code to invoke the custom Business Entity.

When a user attempts to log in at runtime, a method in the Business Entity is invoked. If the user is authenticated, the method returns the user's roles. Kendo UI Builder then uses this role information to determine the user's access rights to modules, views, etc. The mapping of user roles must be defined in the custom Business Entity in the backend.

Defining roles in Kendo UI Designer

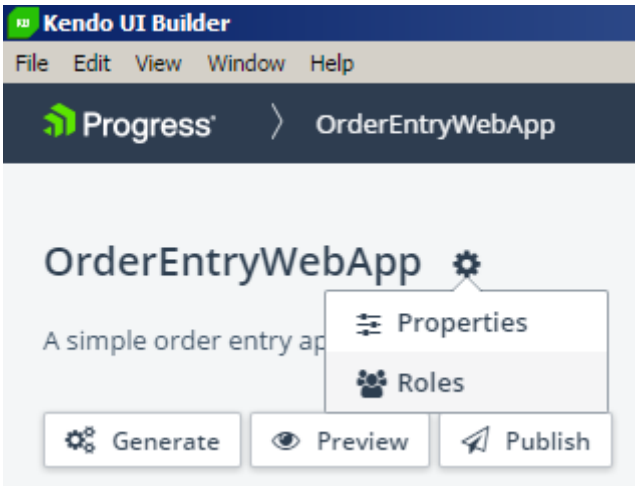
You can define roles at four levels of inheritance:

- *At the app level*: All users that require access to protected parts of the app must have a role defined at the app level.
- *At the module level*: Must be a subset of roles that are defined at the app level. Only those users that have these roles can access the module.
- *At the view level*: Must be a subset of roles that are defined at the module level. Only those users that have these roles can access the view.
- For Blank views only, *at the row or column level*: Must be a subset of roles that are defined for the Blank view. Only those users that have these roles can access the row or column.

Defining roles at the app level

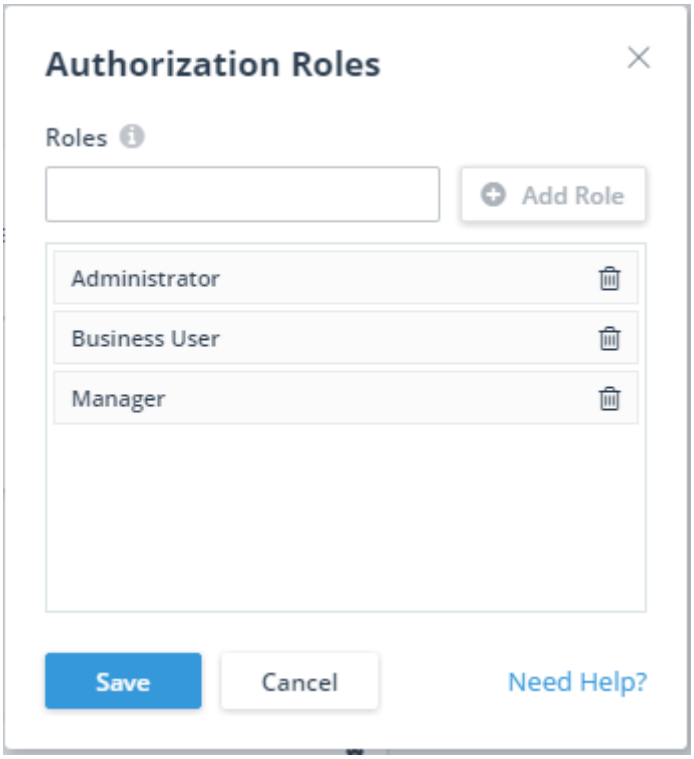
You must define the names of all roles that require access to protected parts of the app as follows:

1. In the *app design page*, click the gear icon next to the app name and select **Roles**.



This opens the **Authorization Roles** dialog box.

2. In the **Authorization Roles** dialog box, enter the name of a role and click **+ Add Role**. Repeat this step until you have added all the required roles as shown in this example:



3. Click **Save**.

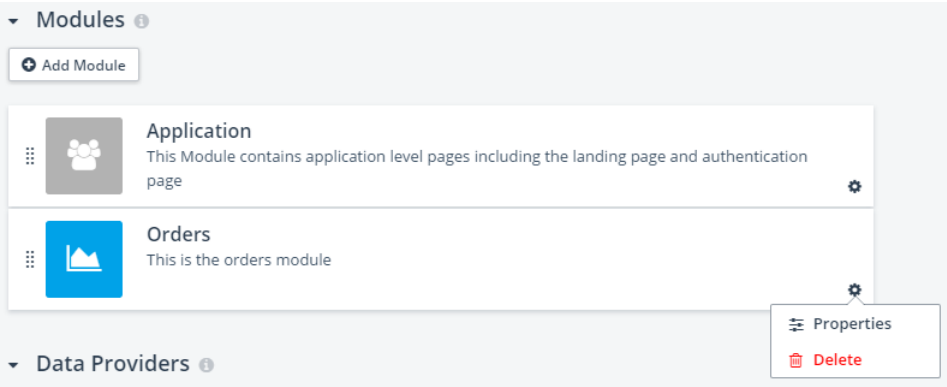
Defining roles at the module level

The next step is to define authorization at the module level. Only those users that have these roles will have access to the module.

Note: If you do not define any roles at the module level, all authenticated users will have access to its views.

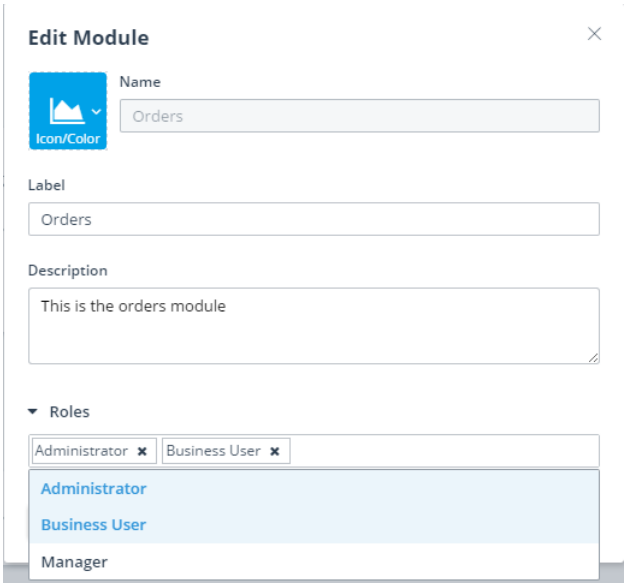
---

1. Click the gear icon in the module card and select **Properties**.



This opens the **Edit Module** dialog box.

2. In the **Edit Module** dialog box, expand **Roles**, choose the roles that require access to the module as shown in this example, and click **Save**. Note that you can choose only from roles that have been defined at the app level.



3. Repeat these steps for each module that requires authorization.

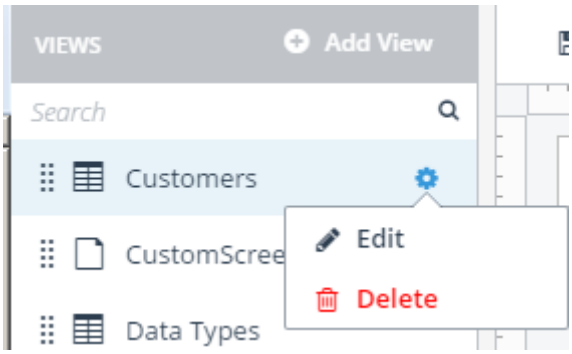
Defining roles at the view level

After you have defined roles at the module level, you can define roles for each view in the module. Roles at the view level must be a subset of the roles that are defined at the module level.

Note: If you do not define specific roles for a view, all roles that are defined for the view's module will have access to the view.

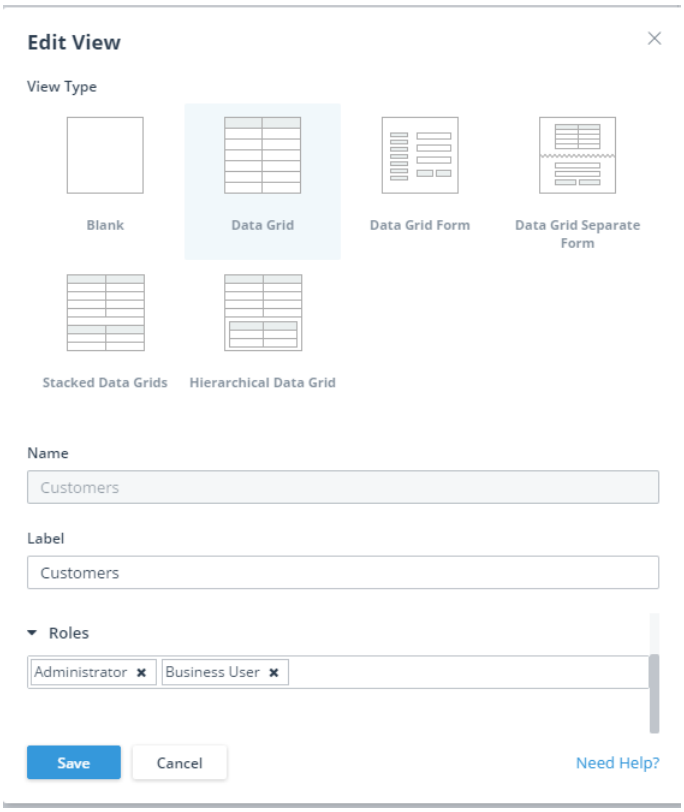
---

1. Click the gear icon next to the view name and select **Edit**.



This opens the **Edit View** dialog box.

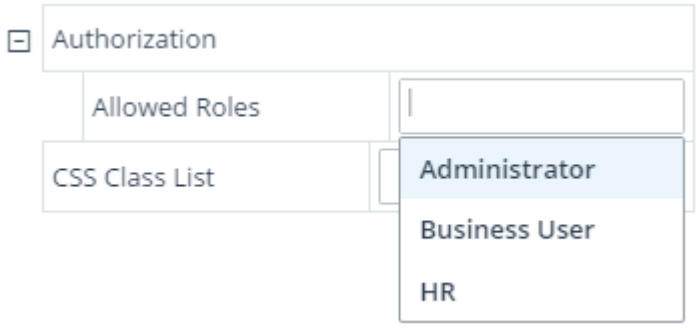
2. In the **Edit View** dialog box, expand **Roles**, choose the role names that require access to the view, and click **Save**.



3. Repeat these steps for every view that requires authorization.

Defining roles for Blank view rows and columns

To define roles for a row or a column in a Blank view, select the row or column in the Blank view layout and configure the **Authorization > Allowed Roles** property. Note that you can choose only from roles that are defined for the Blank view.



Writing custom code to obtain user roles from the OpenEdge business entity

The **login** view in the default **Application** module has a login event function named `onLogin()` . You must write custom code in this `onLogin()` function to invoke the method in the custom Business Entity that returns user roles. A stub for the `onLogin()` function is defined in the view's *`controller.public.js`* file, which you will find in `<app_dir>/app/src/modules/application/login`.

To learn more about creating an authorization service and to see sample code for the `onLogin()` function, visit https://community.progress.com/community_groups/openedge_kendo_ui_builder/w/openedgekendouibuilder/2926.kendo-ui-builder-faq#UserRoles.

Localizing the generated app

By default, the app that is generated by Kendo UI Designer uses a US-style date, time, and number format. However, you can change the default culture and label in the **Edit App** dialog box. This allows you to create your application for a specific culture.

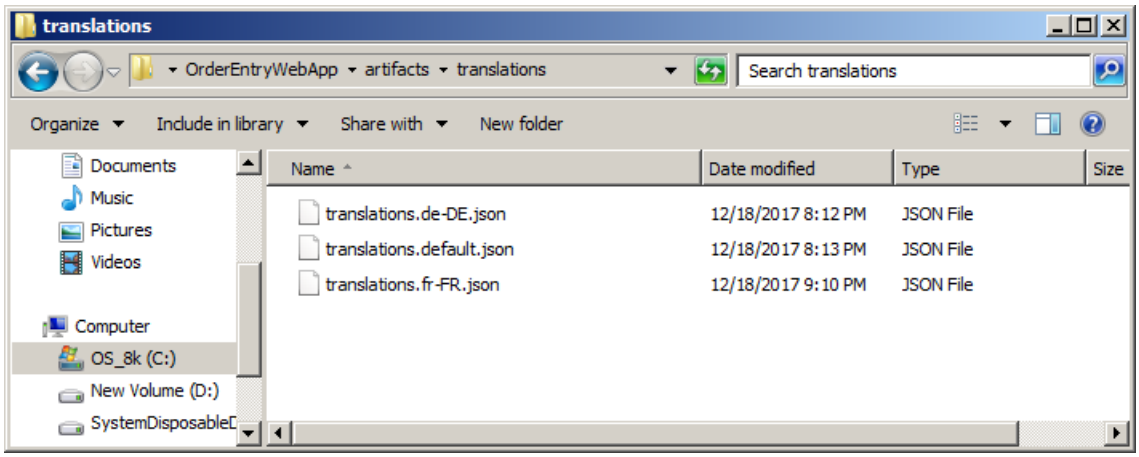
When the app is generated, a translation file--`<webapp_directory>/artifacts/translations/translations.default.json`--is created with all translatable strings in the application. It contains the default label and culture set in the **Edit App** dialog box.

After generation, you can add localization to the generated app to enable users to choose from two or more of the following language cultures:

- German-Germany (de-DE)
- English-UK (en-GB)
- English-US (en-US)
- Spanish-Spain (es-ES)
- French-France (fr-FR)

When you set up localization for any of these language cultures, the date, time, and number formats are automatically localized by Kendo UI Builder. In addition, you can manually add culture-specific translations for all translatable strings such as module and view labels.

To begin, make a copy of the *`translations.default.json`* file located in *`<webapp_directory>`*`/artifacts/translations` and replace the suffix *`default`* in the filename with the language culture that you want to add. For example, replace *`default`* with *`de-DE`* or *`fr-FR`* as shown here:



You must create one file for each language culture that you want to add. The *translations.default.json* file provides a definition of all translatable properties in JSON format. This enables you to add culture-specific translations in the copies. However, note that Kendo UI Designer updates only the default translation file when changes are made to modules or views. It does not automatically update the copies that you make, so it is recommended that you hold off on making the translations until your app is close to being ready for production.

The next step is to edit the file and add culture-specific translations. You must modify the **label** and **culture** properties as shown in this example:

```
"language": {
  "label": "French",
  "culture": "fr-FR",
  "order": 0
},
```

You can then add translations for any of the following properties:

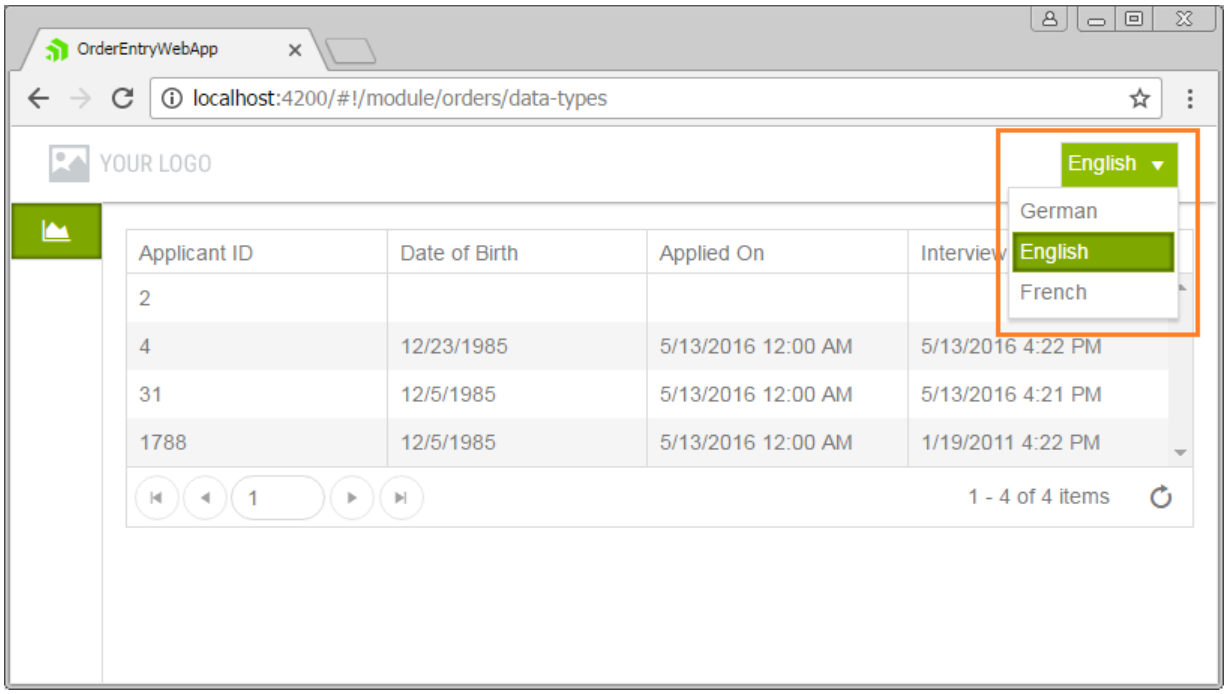
- **All modules**: label, description
- **Data Grid View**: label, title, columns, toolbarButtonLabels, rowButtonLabels
- **Data Grid Form View**: label, title, columns, newTitle, editTitle
- **Data Grid Separate Form View**: label, title, columns, newTitle, editTitle
- **Hierarchical Data Grid View**: label, parentTitle, childTitle, parentGridColumns, childGridColumns, childGridEditMessages
- **Stacked Data Grid View**: label, parentTitle, childTitle, parentGridColumns, childGridColumns, childGridEditMessages
- **Blank View**: label
- **Blank View Components**:

Component	Translatable Properties
grid	toolbarButtonLabels, rowButtonLabels, columns
boolean-radio-button-list	trueText, falseText
currency-text-box	placeholder
email-text-box	placeholder

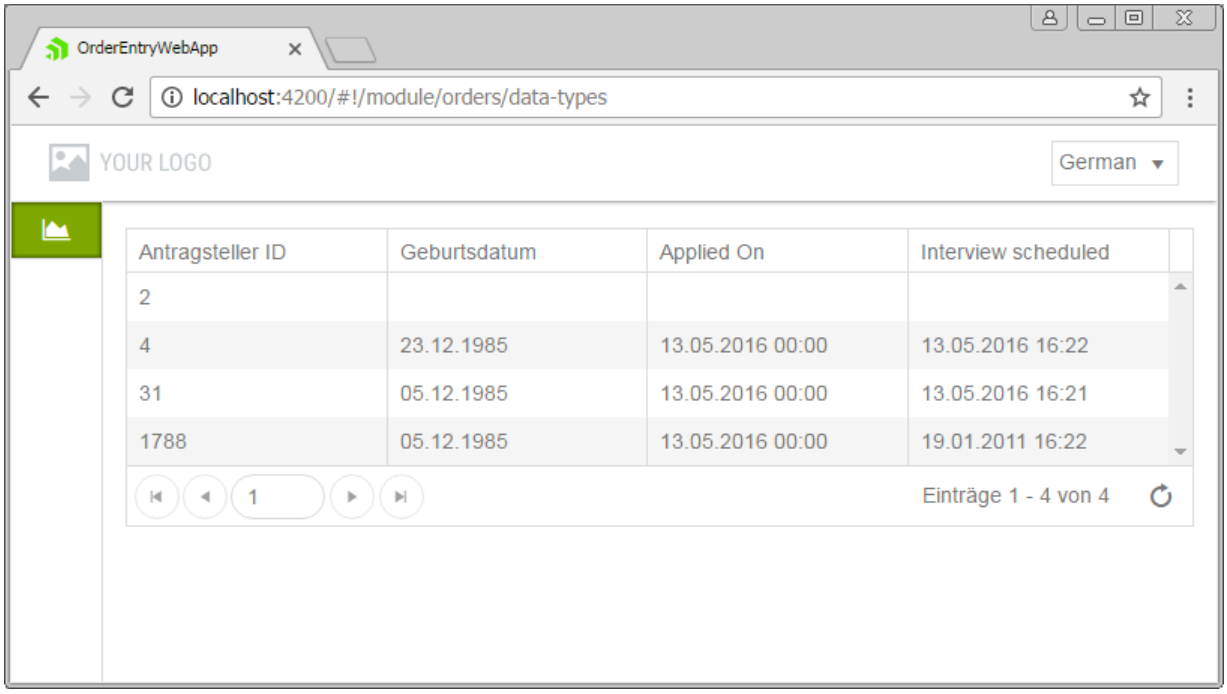
Component	Translatable Properties
integer-text-box	placeholder
numeric-text-box	placeholder
password-text-box	placeholder
percent-text-box	placeholder
percent-value-text-box	placeholder
text-area	placeholder
text-box	placeholder
url-text-box	placeholder
label	text
button	content
area-charts	title text
bar-charts	title text
donut-charts	title text
line-charts	title text
pie-charts	title text
expander	text, subcomponents
tab-strip	tab text, subcomponents
toolbar	button text, splitButton text, subitem text, buttonGroup button text

After you have added the translations, you must regenerate the app's files by clicking **Generate** in the *app design page* in Kendo UI Designer.

At runtime, the header of the generated app displays a dropdown containing a choice for the default language and for each language culture that you have configured:



Choosing a different language culture automatically switches the date, time, and number formats, and displays localized labels:



App generation and deployment

When you publish an app in the Kendo UI Designer, the Kendo UI Generator generates deployable HTML5, CSS, and JavaScript files for the app in the following folder:

```
webAppDir\build-output
```

Where *webAppDir* is the pathname of the root directory of your web app (see also Extension Points and Source Code Customization on page 103).

When you are ready to deploy the app to a test or production environment, or possibly to reconfigure existing environments, the values specified for the **Service URI** and **Catalog URI** used to define data providers for your app might need to change. You can do this without having to rebuild the app by modifying the following JSON file in your debug and/or release build and changing the corresponding URIs for each data provider as required:

```
webAppDir\build-output\debug\data-providers.json
webAppDir\build-output\release\data-providers.json
```

Caution: Be sure to make a backup copy of this file before making any changes to it.

If you have OpenEdge installed, you can then also use a Web UI project in Progress Developer Studio for OpenEdge that you have created with the same name as your web app to deploy the web app to other development or production instances of Progress Application Server for OpenEdge, or to export the web app to other Tomcat-based web servers. Note that this Web UI project must share the same `build-output` folder as your web app.

Alternatively, you can use other deployment tools to deploy the web app to a different type of web server, as its configuration and administration requirements prescribe.

3

Extension Points and Source Code Customization

There are many built in extension points for building and generating a web app with the Kendo UI Builder. Among some of the more basic of these extension points include some changes to:

- Static files
- Company logo

In addition, there are extension points, some of them in static files, that require advanced knowledge of HTML, JavaScript, Kendo UI, Extjs, and AngularJS 1.x:

- Custom templates
- General view events
- View-specific events
- Custom HTML sections
- Row templates

The following topics describe how to access and make changes to these extension points to customize your web apps. These topics refer to a number of file and folder pathnames that begin with the following root directory specifications:

- *`webAppDir`* — The pathname of the directory where Kendo UI Builder saves all files for a given web app, including the build folders and files that it generates for app preview and deployment. This directory takes the name of the web app that you specify in the Kendo UI Designer and can also be the same name as a Web UI project in Progress Developer Studio for OpenEdge where you also work with and deploy the web app. For example, the directory `C:\KUIBuilder\Applications\OrderEntryWebApp` for the web app named `OrderEntryWebApp`.

- *KUIBInstallDir* — The pathname of the directory where Kendo UI Builder is installed on your system, for example, `C:\Progress\KendoUIBuilder`.

For details, see the following topics:

- Static files
- Company logo
- Custom templates
- General view events
- View-specific events
- Custom HTML sections
- Row templates
- Column templates

Static files

The Kendo UI Builder creates a number of static files for any web app that you build with it. You can customize these files for different purposes, as described in many of the remaining topics on extension points and source code customization.

For details, see the following topics:

- Custom stylistic assets on page 104
- Custom code on page 105

Custom stylistic assets

Every Web app has a folder for static assets to be included with the app. These are initially populated with the default fonts, images, and styles available for an app. You can add your own static files or modify the ones provided. It is important to update any cross-references within the app to point to the static files. These files are located in the following directories:

```
webAppDir\app\src\fonts
webAppDir\app\src\images
webAppDir\app\src\styles
```

Note that you can define custom CSS classes for the entire app in the `app.custom.css` file located in *webAppDir*`\app\src\styles\app`.

Custom code

Every view has a folder that is automatically populated with customizable HTML and JavaScript files when you first generate the code for the view using the **Generate** or **Preview** option in the Designer. These customizable files (and some others that are not customizable) are created for each view in the following location:

```
webAppDir\app\src\modules\module-folder\view-folder\
```

The customizable files include *customSection*.html, router-events.js, and controller.public.js, where:

module-folder

Specifies a folder named for the module in which the view is created. If you name the module in the Designer with words separated using camel case or using any special character other than hyphen (-), the folder name always contains hyphens to separate all lower-case words.

view-folder

Specifies a folder named for the view in a similar manner as *module-folder*, and is where the custom HTML and JavaScript files are created.

customSection.html

Where *customSection* specifies the file name for one or more HTML files containing a custom HTML section for the view. Depending on the type of view, each of these files can have one of the following names, corresponding to the custom HTML section it can contain:

- topSection
- topParentSection
- middleSection
- topChildSection
- bottomChildSection
- bottomSection

The view design panel indicates the supported custom HTML sections and where they appear in the view.

router-events.js

Defines an ECMAScript 6 (ES6) class containing methods that define some default general event functions. You can change the names of these methods as long as you also change them for the corresponding property of the view, and otherwise add custom code to modify their behavior. For more information, see General view events on page 115.

controller.public.js

Defines an ES6 class containing methods that define some default general and view-specific event functions. You can change the names of these methods as long as you also change them for the corresponding property of the view, and otherwise add custom code to modify their behavior. This is also the file where you can add methods for all other view-specific event functions that you need

to define and whose names you also specify for corresponding properties of the view. You can similarly add methods for row template functions that you can use to define custom formats for grid rows. For more information, see General view events on page 115, View-specific events on page 119, and Row templates on page 123. You can also extend the view's data source by overriding the function `_$getDataSourceOptions()`, which is defined in the `controller.js` file. For example, you can modify the data source's page size as shown in this example:

```
_$getDataSourceOptions(name){
        this.$dsOptions[name].pageSize = 2;
        return this.$dsOptions[name];}
```

Note: The `name` is different in every generated view - it can be primeDS, jsdo resource name, or an auto generated name for Foreign Key data sources. This name can be seen in the private controller.

Company logo

You can specify the company logo for the web app and login page by specifying the name of an image located in the following directory:

```
webAppDir\artifacts\images
```

Custom templates

The Kendo UI Builder uses templates along with associated meta-data to generate the UI for a web app. Templates define the UI, data mapping, and UI logic in a specific framework (AngularJS currently), while the meta-data is framework agnostic. In addition to built-in templates, the Kendo UI Designer supports a set of templates (*custom templates*) for which you can customize the source for the template UI, data mapping, UI logic, and associated meta-data. These custom templates include two types of templates:

- **View templates** — Customizable versions of the built-in predefined templates for creating Data-Grid* views that you get using the **Add View** dialog in any user-created module.
- **Component templates** — Customizable versions of some of the built-in templates for components that you can add to a Blank view by dragging and dropping them from the **COMPONENTS** pane onto the view design panel.

These custom templates are project based compared to the built-in view and component templates, which are available in every project created in the Designer. Starter versions of these templates are provided as part of your Kendo UI Builder installation. You must copy any starter templates you want to customize into each web app project.

Within each project, you can then customize these templates as needed. However, these customizations apply only to the project where the custom templates reside. If you want to share these customizations with other projects, you must manually copy the custom templates to other projects that you want to share them.

Note: In order to customize templates, experience with JavaScript, ES6, Kendo UI, and AngularJS is required. Note also that customizing templates is an advanced activity and not typical for most OpenEdge ABL developers.

Note: You cannot customize the built-in view and component templates that are provided with every project that you create in the Designer. You can only customize instances of the starter templates in each project where you have copied them.

Caution: Do not customize the starter templates in your Kendo UI Builder installation, where they can be corrupted. Customize starter templates **only** in projects where you can test and verify your customizations.

Copying starter templates to a project

Kendo UI Builder supports a set of starter templates to use as a starting point for your own customizations.

To copy these starter templates to a project:

1. Locate the templates in your Kendo UI Builder installation:
   - *KUIBInstallDir* — The path name to your product installation folder.
     - `templates` — The folder containing starter templates.
       - `views` — A sub-folder containing all starter custom view templates. Each starter custom view template is defined in its own folder under `views`.
       - `components` — A sub-folder containing all starter custom component templates. Each starter custom component template is defined in its own folder under `components`.
2. If you want to copy all the starter templates to a project, copy the installed `templates` folder with its entire contents from *KUIBInstallDir* to *webAppDir* (the path name to your app project folder).
3. If you want to copy only selected starter templates to a project, create one or both of the `templates\views` and `templates\components` folders under *webAppDir*, then:
   - Under *webAppDir*`\templates\views`, copy any of the following sub-folders from *KUIBInstallDir*`\templates\views`, defining a supported custom view template:
     - `custom-data-grid`
     - `custom-data-grid-form`
     - `custom-data-grid-separate-form`
     - `custom-hierarchical-data-grid`
     - `custom-stacked-data-grids`

     For more information on customizing the contents of these custom view template folders, see Customizing templates on page 113.

     When you open the project in the Designer, your custom view templates, with their latest customizations, appear in the **Add View** dialog for any user-created module that you edit in the project. For more information, see Using custom templates in the Designer on page 114.
   - Under *webAppDir*`\templates\components`, copy any of the following sub-folders from *KUIBInstallDir*`\templates\components`, defining a supported custom component template:
     - `custom-auto-complete` — Editor component
     - `custom-calendar` — Scheduling component

- `custom-check-box` — Editor component
- `custom-combo-box` — Editor component
- `custom-currency-text-box` — Editor component
- `custom-date-picker` — Editor component
- `custom-email-text-box` — Editor component
- `custom-grid` — Data Management component
- `custom-masked-text-box` — Editor component
- `custom-slider` — Editor component

For more information on customizing the contents of these custom component template folders, see Customizing templates on page 113.

When you open the project in the Designer, your component templates, with their latest customizations, appear in the **COMPONENTS** pane of any Blank view design page in the project. For more information, see Using custom templates in the Designer on page 114.

Stacked-Data-Grids view

Selecting the Stacked-Data-Grids view creates a vertically split screen with two separate grids. On the top, a parent grid arranges rows of included fields from a parent data source by column, and in the same order as they appear in the **Included Columns** list of the Grid Columns dialog for the parent grid (described below). On the bottom, a child grid arranges related rows of included fields from a child data source by column, and in the same order as they appear in the **Included Columns** list of the Grid Columns dialog for the child grid (described below).

The parent grid provides read-only access to its data source. When the user selects a row in the parent grid, the child grid displays the related rows from its child data source and provides read-only or editable access to that data source as specified by its properties. Like any grid, both the parent and child grid allow you to page through all their rows using their own page selection controls. In this way, you can page through all related child rows by paging and clicking every parent row, then paging through all the rows displayed in the child grid.

Note: Both the selected parent and child data sources must reference tables from the same OpenEdge Data Object Service, where both tables are bound by a parent-child relationship in the same Data Object resource. This resource must be created for an OpenEdge ProDataSet that includes both the two corresponding temp-tables and their data-relation as part of its definition. Therefore, no data sources from either an OData Provider or a Generic REST Provider are available to select for this view.

The Stacked-Data-Grids view has a number of customizable properties, listed on the right side of the view design page:

- **Data Provider**: The selected data provider for this view.
- **Parent Data Source**: The selected parent-grid data source from the chosen data provider. This must be a top-level table in the dataset.
- **Child Data Source**: The selected child-grid data source from the chosen data provider. This must be a child table for the parent data source.
- **Parent Grid Properties**:

- **Grid Title**: The title that appears for the parent grid when this view is rendered on the screen for the app.
- **Grid Columns**: Click **Edit** to change the included columns and various properties of the columns. See Editing grid columns for more information.
- **Page Size**: The number of rows that will be displayed per page in the grid.
- **Row Template Function**: The name of a JavaScript function that returns the text for a Kendo UI rowTemplate. If specified, the row template is used to format all the rows in the grid unless you also define an altRowTemplate. For more information, see the entry for rowTemplate at http://docs.telerik.com/kendo-ui/api/javascript/ui/grid#configuration-rowTemplate, in the *Kendo UI by Progress Documentation and API Reference*. For more information on defining a rowTemplate for a Kendo UI Builder editable grid, see *Kendo UI Builder by Progress: Modernizing OpenEdge Applications*.
- **Row Template Id**: The ID of an HTML rowTemplate. If specified, the rowTemplate must be defined in either the *application-folder*\app\src\modules*module-folder**view-folder*\topSection.html file, the \middleSection.html file, or the \bottomSection.html file. If both a Row Template Function and a Row Template Id are specified, the Row Template Id takes precedence.
- **Alt Row Template Function**: The name of a JavaScript function that returns the text for a Kendo UI altRowTemplate. If specified, the altRowTemplate is used to format every other row in the grid. For more information, see the entry for altRowTemplate at http://docs.telerik.com/kendo-ui/api/javascript/ui/grid#configuration-altRowTemplate, in the *Kendo UI by Progress Documentation and API Reference*. For more information on defining a rowTemplate for a Kendo UI Builder editable grid, see *Kendo UI Builder by Progress: Modernizing OpenEdge Applications*
- **Alt Row Template Id**: The ID of an HTML altRowTemplate. If specified, the altRowTemplate must be defined in either the *application-folder*\app\src\modules*module-folder**view-folder*\topSection.html file, the \middleSection.html file, or the \bottomSection.html file. If both an Alt Row Template Function and an Alt Row Template Id are specified, the Alt Row Template Id takes precedence.
- **Enable Column Filtering**: Provides the app user with the ability to filter the displayed information based on content, including filters for numeric values (e.g., is equal to, is not equal to) and text fields (e.g., starts with, does not contain, etc.).

Note: Filtering, sorting, and paging operations execute based on the **Client-side processing** setting for the data source. Operations are performed on the client if **Client-side processing** is selected; otherwise, the operations are performed on the server side. See the info on enabling server-side processing in Adding data sources.

- **Enable Grouping**: Provides the app user with the ability to group the rows according to the value of a particular column. For example, in a grid with columns representing customer names and order numbers, grouping by customer name will arrange the rows so that all of the order numbers from each given customer are grouped together. Grouping should only be used with a data source that uses client-side processing.
- **Enable Column Resize**: Provides the app user with the ability to resize the columns by dragging the column divider(s).
- **Enable Column Reordering**: Provides the app user with the ability to reorder the columns by dragging and dropping the column headers.
- **Enable Sorting**: Provides the app user with the ability to reorder rows by ascending or descending value in a given column.

- **Events**:
  - **Row Select Event Function**: The name of a JavaScript function that runs when a parent row is selected by the app user.
  - **Data Bound Event Function**: The name of a JavaScript function that runs when the parent grid is bound to its data source.

  The code for these event functions must be included in the file, *application-folder*\app\src\modules*module-folder**view-folder*\controller.public.js. See *Kendo UI Builder by Progress: Modernizing OpenEdge Applications* for more information about extensions and event functions.

- **Child Grid Properties**:
  - **Grid Title**: The title that appears for the child grid when this view is rendered on the screen for the app.
  - **Edit Mode**:
    - **ReadOnly**: This mode supports read-only access to data in the grid.
    - **Incell**: This mode supports edits to multiple rows at a time in the grid. Data editing is supported using the fields of the grid itself. You initiate editing by clicking a column within a row. Each row also provides a button to delete the associated row in the grid. In this mode, the toolbar above the grid provides buttons to create a new row, and to save all changes or cancel all changes to the rows in the grid.
    - **Inline**: This mode supports edits to a single row at a time in the grid. Data editing is supported using the fields of the row itself. You initiate editing by clicking an edit button on the row you want to update. Each row also provides a button to delete the associated row in the grid. In this mode, the toolbar above the grid provides a single button to create a new row in the grid.
    - **Popup**: This mode supports edits to a single row at a time in the grid. Data editing is supported using a pop-up form. You initiate editing by clicking an edit button on the row you want to update. Each row also provides a button to delete the associated row in the grid. In this mode, the toolbar above the grid provides a single button to create a new row in the grid.
  - **Edit Options**:
    - **Allow Insert**: An option that allows new rows to be created in the grid.
    - **Allow Edit**: An option that allows existing rows to be edited in the grid.
    - **Allow Delete**: An option that allows existing rows to be deleted in the grid.
    - **Toolbar Button Labels**:
      - **Cancel**: A custom label for the button that cancels all pending edits in the grid. The default is "Cancel changes".
      - **Create**: A custom label for the button that creates a new row in the grid. The default is "New".
      - **Save**: A custom label for the button that saves all pending edits in the grid. The default is "Save changes".
    - **Row Button Labels**:
      - **Cancel Edit**: A custom label for the button that cancels all pending edits in the row. The default is "Cancel".
      - **Delete**: A custom label for the button that deletes the row. The default is "Delete".

    - **Edit**: A custom label for the button that initiates editing in the row. The default is "Edit".
    - **Update**: A custom label for the button that confirms any pending delete or edits to the row. The default is "Update".

- **Grid Columns**: Click **Edit** to change the included columns and various properties of the columns. See Editing grid columns for more information.
- **Page Size**: The number of rows that will be displayed per page in the grid.
- **Row Template Function Name**: The name of a JavaScript function that returns the text for a Kendo UI rowTemplate. If specified, the row template is used to format all the rows in the grid unless you also define an altRowTemplate. For more information, see the entry for rowTemplate at http://docs.telerik.com/kendo-ui/api/javascript/ui/grid#configuration-rowTemplate, in the *Kendo UI by Progress Documentation and API Reference*. For more information on defining a rowTemplate for a Kendo UI Builder editable grid, see *Kendo UI Builder by Progress: Modernizing OpenEdge Applications*.
- **Row Template Id**: The ID of an HTML rowTemplate. If specified, the rowTemplate must be defined in either the *application-folder*\app\src\modules*module-folder**view-folder*\topSection.html file, the \topParentSection.html file, the \middleSection.html file, the \topChildSection.html file, or the \bottomSection.html file. If both a Row Template Function and a Row Template Id are specified, the Row Template Id takes precedence.
- **Alt Row Template Function Name**: The name of a JavaScript function that returns the text for a Kendo UI altRowTemplate. If specified, the altRowTemplate is used to format every other row in the grid. For more information, see the entry for altRowTemplate at http://docs.telerik.com/kendo-ui/api/javascript/ui/grid#configuration-altRowTemplate, in the *Kendo UI by Progress Documentation and API Reference*. For more information on defining a rowTemplate for a Kendo UI Builder editable grid, see *Kendo UI Builder by Progress: Modernizing OpenEdge Applications*
- **Alt Row Template Id**: The ID of an HTML altRowTemplate. If specified, the altRowTemplate must be defined in either the *application-folder*\app\src\modules*module-folder**view-folder*\topSection.html file, the \middleSection.html file, or the \bottomSection.html file. If both an Alt Row Template Function and an Alt Row Template Id are specified, the Row Template Id takes precedence.
- **Enable Column Filtering**: Provides the app user with the ability to filter the displayed information based on content, including filters for numeric values (e.g., is equal to, is not equal to) and text fields (e.g., starts with, does not contain, etc.).
- **Enable Grouping**: Provides the app user with the ability to group the rows according to the value of a particular column. For example, in a grid with columns representing customer names and order numbers, grouping by customer name will arrange the rows so that all of the order numbers from each given customer are grouped together. Grouping should only be used with a data source that uses client-side processing.
- **Enable Column Resize**: Provides the app user with the ability to resize the columns by dragging the column divider(s).
- **Enable Column Reordering**: Provides the app user with the ability to reorder the columns by dragging and dropping the column headers.
- **Enable Sorting**: Provides the app user with the ability to reorder rows by ascending or descending value in a given column.
- **Selection Type**: Determines if rows in a grid can be selected:
    - **None**: No row selection is allowed (default).

- **Single**: One row can be selected at a time.
- **Multiple**: Multiple rows can be selected at one time.

Note: Row selection has no function except to invoke the Row Select Event for each selected row.

- **Events**:
  - **Row Select Event Function**: The name of a JavaScript function that runs when a child row is selected by the app user.
  - **Data Bound Event Function**: The name of a JavaScript function that runs when the child grid is bound to its data source.
  - **Row Create Event Function**: The name of a JavaScript function that runs before a row is created for a new data source record.
  - **Row Update Event Function**: The name of a JavaScript function that runs before an existing data source record is updated in the row.
  - **Row Delete Event Function**: The name of a JavaScript function that runs before an existing data source record is deleted in the row.

  The code for these event functions must be included in the file, *application-folder*\app\src\modules*module-folder**view-folder*\controller.public.js. See *Kendo UI Builder by Progress: Modernizing OpenEdge Applications* for more information about extensions and event functions.

- **CSS Class List**: The CSS classes that need to be applied to the view. The classes must be defined in the view's *style.css* file, which can be opened and edited by clicking the **Edit CSS** button in the **View Styles** pane on top.

This view also includes the following custom HTML sections where you can include your own HTML for the view. Each section is identified in the view design page with a placeholder containing descriptive text:

- **Custom top html section:** Located in the area above the parent Grid Title, the text in this placeholder introduces the view path name of the `topSection.html` file where you can add your custom HTML.
- **Custom top parent html section:**: Located in the area below the parent Grid Title but above the parent grid itself, the text in this placeholder introduces the view path name of the `topParentSection.html` file where you can add your custom HTML.
- **Custom middle html section:**: Located in the area below the parent grid and above the child Grid Title, the text in this placeholder introduces the view path name of the `middleSection.html` file where you can add your custom HTML.
- **Custom top child html section:**: Located in the area below the child Grid Title but above the child grid itself, the text in this placeholder introduces the view path name of the `topChildSection.html` file where you can add your custom HTML.
- **Custom bottom html section:**: Located in the area below the child grid, the text in this placeholder introduces the view path name of the `bottomSection.html` file where you can add your custom HTML.

The Designer automatically generates these three files for you in *application-folder*\app\src\modules*module-folder**view-folder*. Add your custom HTML code to these files as required.

Customizing templates

Everything a custom template needs is contained in the single sub-folder named for the template under *webAppDir*\templates\views or *webAppDir*\templates\components, depending on the type of template. This sub-folder contains additional folders and files required for both project design and code generation.

For example, the sub-folder for the custom-data-grid view template contains:

- angularjs — A folder containing everything required for a web app that is generated to run in the AngularJS framework.
- design-time — A folder containing everything required to visualize the template at design time, specifying what you see in the Designer.
- custom-data-grid.json — A JSON file that provides the schema definition for the view, what properties it has that will be stored as meta-data, and how they are presented in the **VIEW PROPERTIES** pane on a view design page.

The sub-folder for the custom-slider component template contains:

- angularjs — A folder containing everything required for a web app that is generated to run in the AngularJS framework.
- design-time — A folder containing everything required to visualize the template at design time, specifying what you see in the Designer.
- custom-slider.json — A JSON file that provides the schema definition for the component, what properties it has that will be stored as meta-data, and how they are presented in the **COMPONENT PROPERTIES** pane on a Blank view design page.

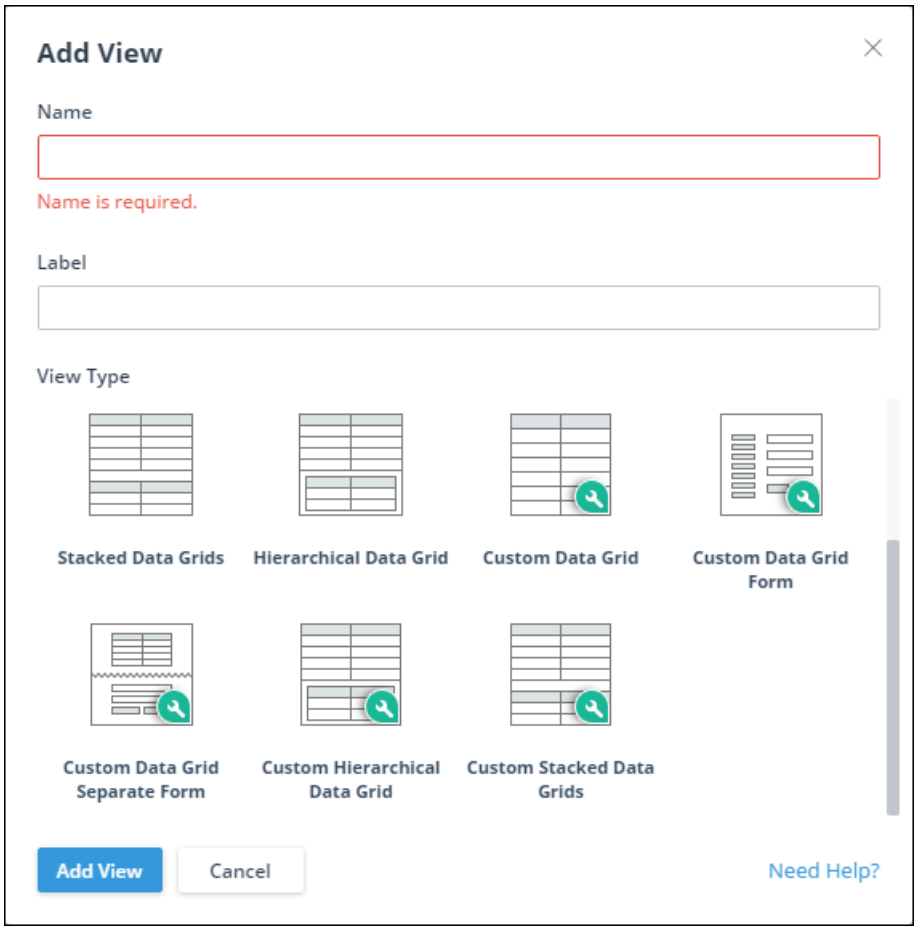
Note that custom templates are designed to support additional target web app frameworks in the future. For example, if Kendo UI Builder supports the generation of web apps using the Angular framework, an additional folder with an appropriate name (such as angular) will be included under the template sub-folder.

Caution: Custom templates must be edited with extreme care. Always make a backup copy of the template folder or ensure that the app project folder is under source code management before editing any of its files.

Using custom templates in the Designer

When you have custom view templates in a project and you open the **Add View** dialog to add a view to a module, it displays the custom view templates following the built-in templates similar to this:

Figure 43: Add View dialog box with custom templates

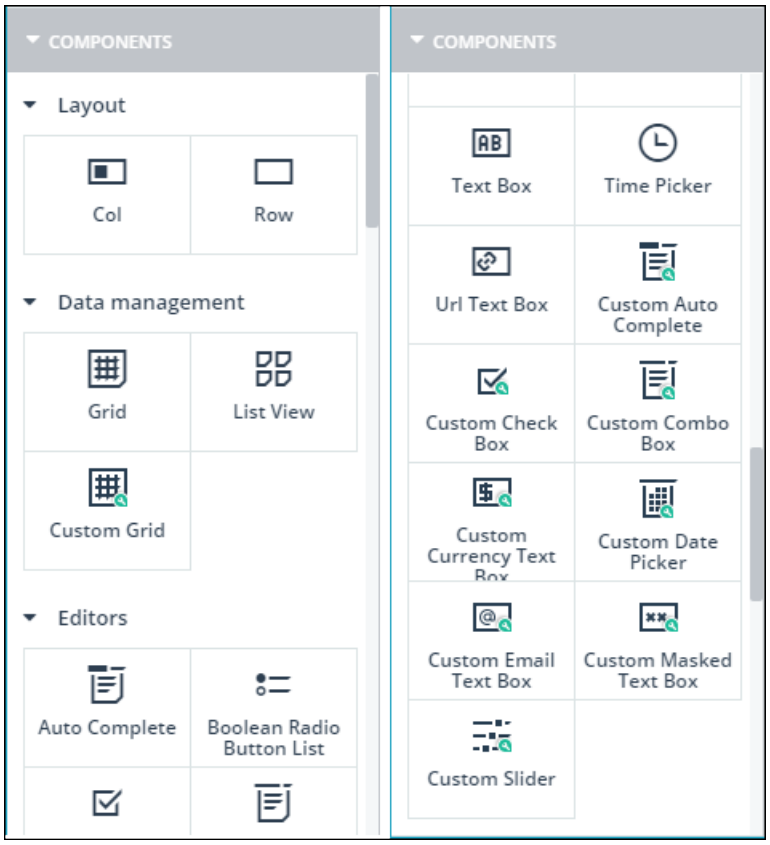


In this figure, the **Add View** dialog is scrolled to show all the custom view templates after you have copied them to the project. They are listed in alphabetical order following the last built-in view template, **Hierarchical Data Grid**. If you have copied only selected starter view templates, only the templates you copied are listed as custom templates.

Note that the Designer displays the all lower-case-hyphenated custom template names with all words in initial caps and the hyphens replaced with spaces, similar to the built-in template names. The icons shown for the custom templates are also customizable and are provided as a graphics file in the `design-time` folder of each template sub-folder.

Also, in any Blank view design page that you open, the custom component templates follow the built-in component templates in the **COMPONENTS** pane similar to this:

Figure 44: COMPONENTS pane with custom templates



In this figure the **COMPONENTS** pane on the left is scrolled to the top showing the **Custom Grid** template listed after the built-in templates for the Data Management components. The same **COMPONENTS** pane on the right is scrolled to the bottom of the **Editors** list showing the supported templates for all custom editor components starting with **Custom Auto Complete**, all listed in alphabetical order after the last built-in editor component template, **Url Text Box**.

Similar to the starter view templates, if you have copied only selected starter component templates, only the templates you copied are listed as custom templates. The icons for custom component templates are also customizable.

General view events

Events allow you to specify the name of a JavaScript function that runs when a specified event fires. General view events fire based on behaviors that are common to all views. The function that runs for an event are defined in either the `controller.public.js` or `router-events.js` file for the view, depending on the event. These files can be found for each view at the following location, and initially contains empty function definitions for these events with default names:

```
webAppDir\app\src\modules\module-folder\view-folder\
```

For more information on these files and its location, see Custom code on page 105.

The following table describes the supported general view events, the default names of the event functions that run for them when they fire, and the view static file location where they reside.

Table 7: Default general view event functions

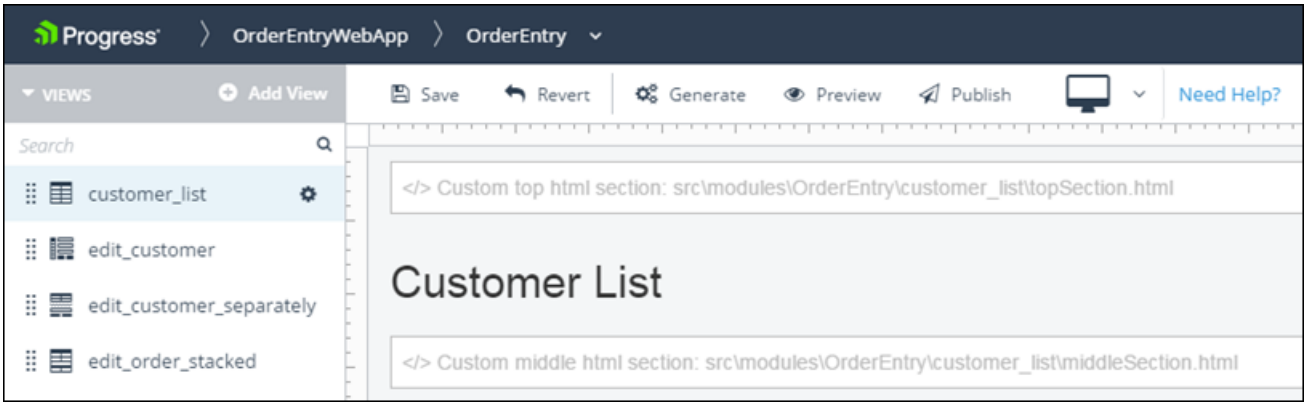
View event	Default function	View static file	Description
`Hide`	`onHide`	`router-events.js`	Fires when the view is hidden. You can find more information at http://docs.telerik.com/kendo-ui/api/javascript/view#events-hide.
`Init`	`onInit`	`router-events.js`	Fires when the view is initialized. You can find more information at http://docs.telerik.com/kendo-ui/api/javascript/view#events-init.
`Show`	`onShow`	`controller.public.js`	Fires when the view is made visible. You can find more information at http://docs.telerik.com/kendo-ui/api/javascript/view#events-show.

You can change the default names of these functions in its view static file for a view. One reason to change an event function name is to test alternative event behaviors using different event functions before deciding on the one you want to implement for the web app. If you do need to change the default name of a view event function in the file, you must also change the name of the event function as specified for the corresponding view in the Kendo UI Designer.

For the general view events, these function names are specified in the Edit View dialog box, which you can access in the Designer as follows:

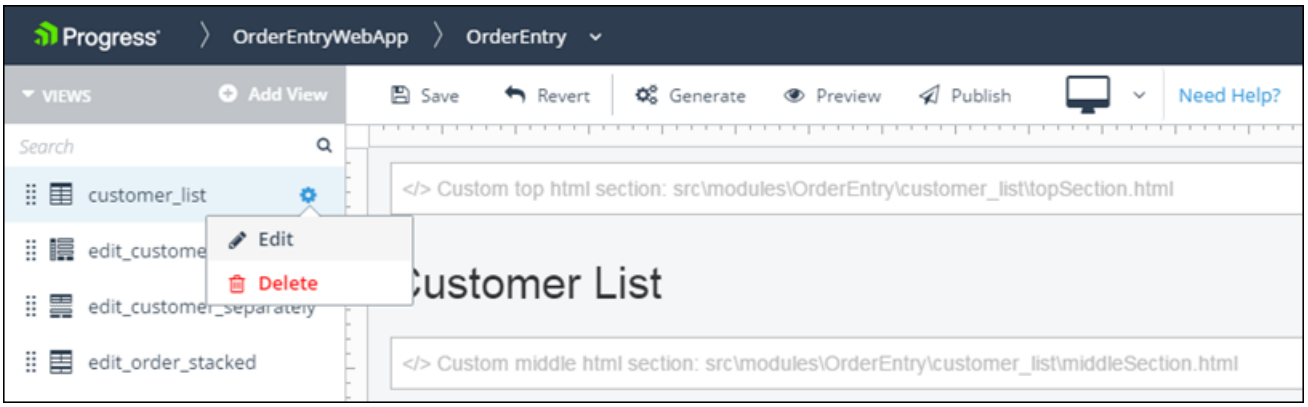
1. Edit the view by first editing its module in the app design page (see Creating and designing an app on page 17). This displays the view design page for the first view in the module and lists all its remaining views (for the example Data-Grid view, see Adding and editing a Data-Grid view on page 42).
2. Select the name of the view in the list you want to edit, displaying its view design page. This shows a gear control beside the selected view name, as shown for the **customer_list** view:

Figure 45: Editing general view events



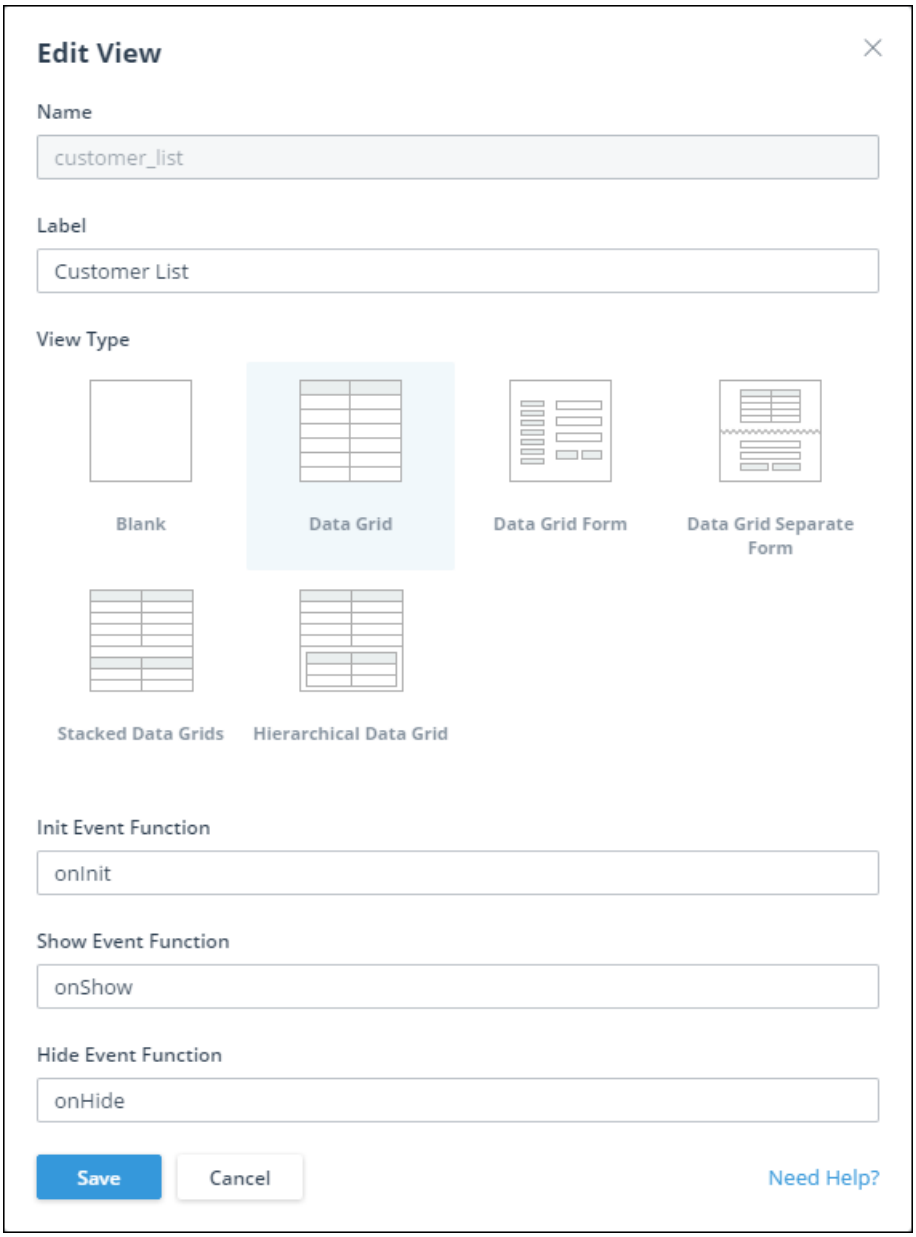
3. Click the gear control to display a drop-down menu, then click **Edit**:

Figure 46: Opening the Edit View dialog box



4. This opens the **Edit View** dialog box, where you can change the name of any general view event function by editing the field for the event as shown:

Figure 47: Changing the names of general view event functions



5. Click **Save** to save your changes and close the dialog box.

View-specific events

Built-in views offer context-specific events based on the function of the view. Like general view events, view-specific events allow you to specify the name of a JavaScript function that runs when the event fires. The function that runs for a view-specific event must also be defined in the `controller.public.js` file for the view. This file can be found for each view at the following location, and initially contains empty function definitions for any specific events for the view with default names:

```
webAppDir\app\src\modules\module-folder\view-folder\controller.public.js
```

For more information on this file and its location, see Custom code on page 105.

The following table describes the supported view-specific events for each view and component, and the default names of the default event functions that run for them when they fire.

Table 8: View-specific events and installed default functions

Event	View:Components	Default function	Description
`Cancel`	Blank:List View	–	Fires when the user cancels an operation.
`Change`	Data-Grid-Form:Editor Types Data-Grid-Separate-Form:Editor Types Blank:Editors Blank:Calendar	–	Fires when an underlying field value is changed in a Data-Grid view form editor type or in a supported Blank view component.
`Click`	Blank:Button Blank:Toolbar Button Blank:Charts	–	Fires when the user clicks on a button or on an item series in a chart.
`Data Bound`	Data-Grid-Form Data-Grid-Separate-Form Stacked-Data-Grids:Parent Stacked-Data-Grids:Child Hierarchical-Data-Grid:Parent Hierarchical-Data-Grid:Child Blank:Charts Blank:Grid Blank:List View	–	Fires when the view or component is bound to its data source.
`Detail Collapse`	Hierarchical-Data-Grid:Parent	–	Fires after the user collapses the child rows for a parent.

Event	View:Components	Default function	Description
`Detail Init`	Hierarchical-Data-Grid:Parent	–	Fires before the child rows for a parent expand.
`Detail Expand`	Hierarchical-Data-Grid:Parent	–	Fires after the child rows for a parent expand.
`Edit`	Blank:List View	–	Fires when the user edits an item in the list view.
`Filtering`	Blank:Auto Complete Blank:Combo Box	–	Fires after the user filters the results.
`Login`	Login	`onLogin`	Fires when the user clicks **Login** on the login view.
`Remove`	Blank:List View	–	Fires when the user removes an item from the list view.
`Row Create`	Data-Grid Stacked-Data-Grids:Child Hierarchical-Data-Grid:Child Blank:Grid	–	Fires before a row is created for a new data source record.
`Row Delete`	Data-Grid Stacked-Data-Grids:Child Hierarchical-Data-Grid:Child Blank:Grid	–	Fires before an existing data source record is deleted in a row.
`Row Select`	Data-Grid Data-Grid-Form Data-Grid-Separate-Form	`onRowSelect`	Fires when the selected row changes in the grid.
	Stacked-Data-Grids:Parent Hierarchical-Data-Grid:Parent	`onParentRowSelect`	
	Stacked-Data-Grids:Child Hierarchical-Data-Grid:Child	`onChildRowSelect`	
	Blank:Grid	–	
`Row Update`	Data-Grid Stacked-Data-Grids:Child Hierarchical-Data-Grid:Child Blank:Grid	–	Fires before an existing data source record is updated in a row.

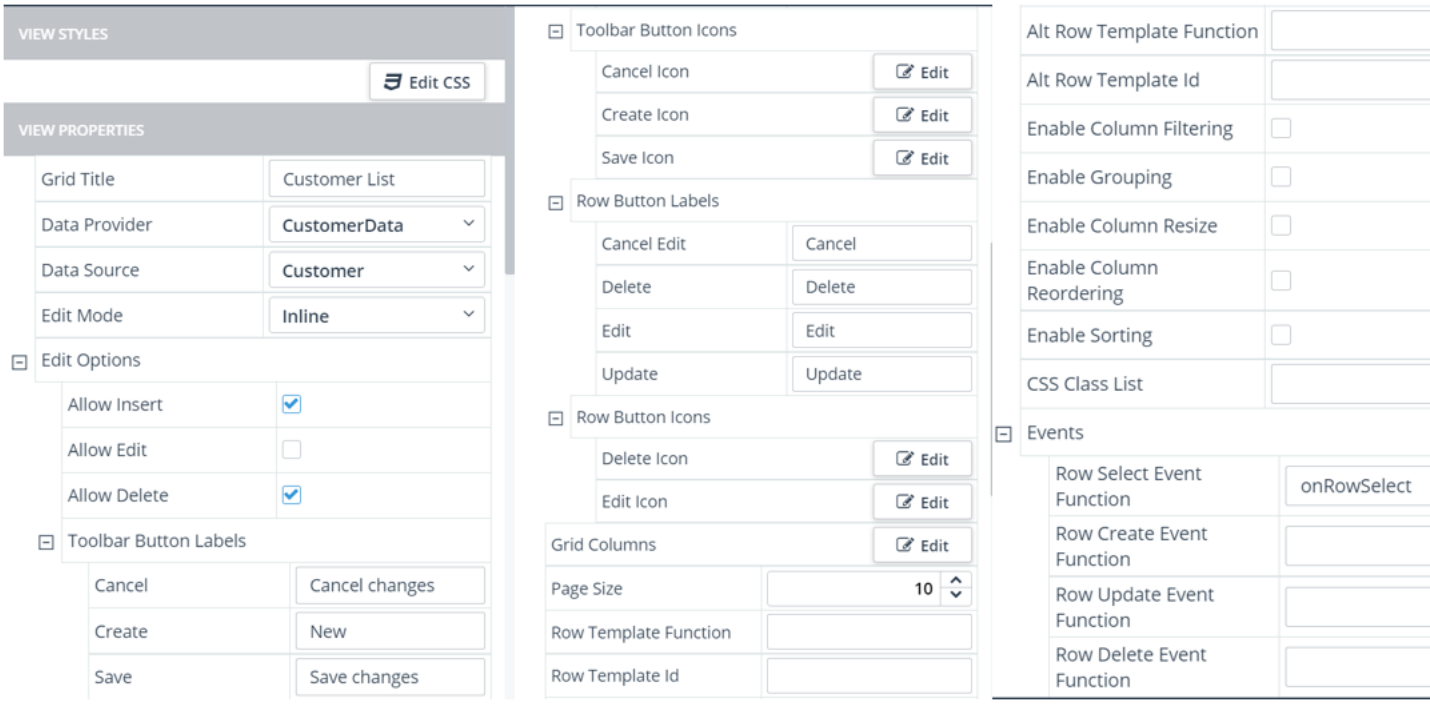
Event	View:Components	Default function	Description
`Save`	Blank:List View	–	Fires when the user saves a change to the list view.
`Select`	Blank:Auto Complete Blank:Combo Box Blank:Drop Down List Blank:List View	–	Fires when the user selects a value among the choices offered.
	Blank:Tab Strip	–	Fires when the user selects a tab in the tab strip.
`Show`	Blank:Tab Strip	–	Fires when the selected tab displays its contents.
`Slide`	Data-Grid-Form:Slider Editor Type Data-Grid-Separate-Form:Slider Editor Type Blank:Slider	–	Fires when the user moves the drag handle.
`Toggle`	Blank:Toolbar Button	–	Fires when the user toggles a button in the toolbar that has its **Togglable** property set.

You can change the default names of these functions in the `controller.public.js` file for a view just like general view event functions. One reason to change an event function name is to test alternative event behaviors using different event functions before deciding on the one you want to implement for the web app. If you do need to change the default name of a view event function in the `controller.public.js` file, you must also change the name of the event function as specified for the same view in the Kendo UI Designer.

For the view-specific events, these function names are specified in the **Properties** pane on the view design page, which you can access in the Designer when you edit the module that contains the view. For information on editing modules, see Creating and designing an app on page 17. For information on editing views, see Modules and views on page 36.

This is the **Properties** pane in the view design page for the example Data-Grid view, `customer_list`, showing the **Row Select Event Function** property setting under **Grid Events** (see also Adding and editing a Data-Grid view on page 42):

Figure 48: Editing view-specific events



Custom HTML sections

Built-in views have places (sections) for you to add your own hand-coded UI. The code must be placed in the correct file for a given section and must be valid HTML for a `<div>` section. For more information on `<div>` sections see http://www.w3schools.com/tags/tag_div.asp.

The custom section files for a view can be found as follows:

```
webAppDir\app\src\modules\module-folder\view-folder\customSection.html
```

For more information on the names and locations of these custom section files, see Custom code on page 105.

The following table lists each custom section supported for most views and the names of the corresponding file created for each section:

Table 9: Some custom sections and corresponding section files

Custom section	Corresponding file
Top	`topSection.html`

Custom section	Corresponding file
Middle	`middleSection.html`
Bottom	`bottomSection.html`

Note: The predefined landing-page view supports only top and bottom custom sections

If a given file for a custom section is empty or contains only comment elements (the default), the corresponding section does not appear in the view at run time.

Custom sections can also be used to contain row and column templates for grid views. For more information, see Row templates on page 123 and Column templates on page 125.

Row templates

Kendo UI has support for templates to display data in a certain format. Templates can be used in Kendo UI for both Grids and Forms. In Kendo UI Builder, this Kendo UI template functionality provides options to specify templates for the rows of Kendo UI Grids in grid views, such as the Data-Grid view.

A row template can be specified using the following properties in the **PROPERTIES** pane of any grid view in the Kendo UI Designer:

- **Row Template Id** — Specifies the ID of a template for grid rows defined in an HTML file.
- **Row Template Function** — Specifies the name of a function that returns a template for grid rows in the grid view's `controller.public.js` file.

  Note: If both the **Row Template Id** and **Row Template Function** properties are defined, **Row Template Id** takes precedence.

- **Alt Row Template Id** — Specifies the ID of a template for every other grid row defined in an HTML file.
- **Alt Row Template Function** — Specifies the name of a function that returns a template for every other grid row in the grid view's `controller.public.js` file.

  Note: If both the **Alt Row Template Id** and **Alt Row Template Function** properties are defined, **Alt Row Template Id** takes precedence.

For details, see the following topics:

- Row template format on page 123
- Row template ID on page 124
- Row template function on page 125

Row template format

The format of the row template is similar to the definition for an HTML table row.

It uses a combination of `<tr></tr>` to specify the row and `<td></td>` to specify the columns (table data).

The `data-uid` attribute is required to determine the data associated with the row.

You can use `#= FieldName #` or `#: FieldName #` for a column element to refer to a field in the data, or use `# JavaScriptCode #` to execute JavaScript code.

You can use HTML tags such as `<img>` or `<strong>` to define how to show the data.

The following code shows an example row template:

```
<tr data-uid="#= uid #">
    <td><b>#= EmpNum #</b></td>
    <td>#= LastName #</td>
    <td>#= FirstName #</td>
    <td>#= State #</td>
</tr>
```

For more information on working with row templates in Kendo UI, see http://docs.telerik.com/kendo-ui/api/javascript/ui/grid#configuration-rowTemplate and http://docs.telerik.com/kendo-ui/api/javascript/ui/grid#configuration-altRowTemplate.

Row template ID

The **Row Template ID** and **Alt Row Template ID** properties are used to specify a row template defined in an HTML file.

The row template definition uses the `<script>` tag with type `"text/x-kendo-template"`.

This code can be added to the same HTML files used for the custom sections (see Custom HTML sections on page 122).

When the web app code is generated, the Kendo UI Builder places the files into the `build-output` folder. For example, for the custom top section file in the `Column_List` view of the `OrderEntry` module:

```
webAppDir\build-output\debug\extensions\html\OrderEntry-Column_List\topSection.html
```

This means that after changing the template code, you need to rebuild the web app for the new template code to appear in the corresponding custom view section file.

Following is a sample template definition that can go in one of these files:

```
<script id="empTemplate" type="text/x-kendo-template">
<tr data-uid="#= uid #">
    <td><b>#= EmpNum #</b></td>
    <td>#= LastName #</td>
    <td>#= FirstName #</td>
    <td>#= State #</td>
</tr>
</script>
```

Row template function

The **Row Template Function** and **Alt Row Template Function** properties allow you to specify a function that processes the template and returns the result of executing the template for the data.

The code for this function is defined in the `controller.public.js` file for a given grid view found at the following location:

```
webAppDir\app\src\modules\module-folder\view-folder\
```

For more information, see Custom code on page 105.

There are three main parts to a row template function:

- The text for the template.
- The template function obtained by calling `kendo.template()`.
- The result of applying the template to the `dataItem`.

Following is a sample row template function that can be defined in a `controller.public.js` file:

```
rowTemplate(dataItem) {
    var template = kendo.template('<tr data-uid="#= uid #">
                                   <td><b>#= EmpNum #</b></td>
                                   <td>#= LastName #</td>
                                   <td>#= FirstName #</td>
                                   <td>#= State #</td></tr>');
    return template(dataItem);
}
```

Column templates

Kendo UI supports different ways to code templates for Grid columns, as described here: http://docs.telerik.com/kendo-ui/api/javascript/ui/grid#configuration-columns.template. These column templates are similar to specifying the column elements of a row column template (see Row template format on page 123).

You can specify the code for a column template as the value of the **Template** property for an included column selected in the **Grid Columns** dialog box that you open for editing the columns of a grid view. You can access this dialog box by clicking **Edit** for the **Grid Columns** property in the **PROPERTIES** pane of a grid view design page. For more information, see the topics on adding and editing grid views in Modules and views on page 36.

For example, to apply a column template that renders the value of a `Name` field in bold that appears in the corresponding column of a grid view, select the included column for that field and enter `#: Name #` (without quotes) as the value of the **Template** property.

Note: You must use the data source field name in a column template, **not** any label that has been defined for it. So, if the `Name` field has `Full Name` specified as its label, you cannot use `Full Name` in the column template.
