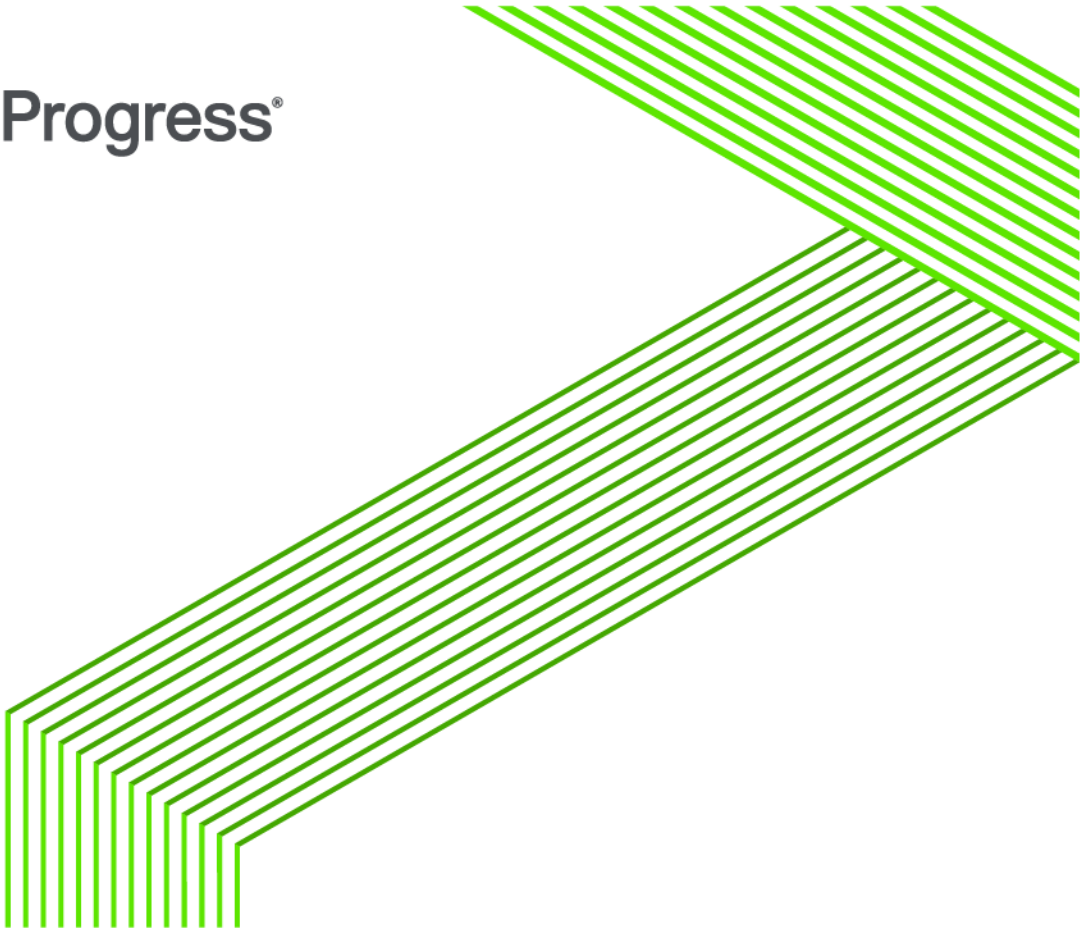
Progress®



Kendo UI® Builder by Progress®:
Modernizing OpenEdge Applications

Kendo UI® Builder by Progress

Notices

November 2016

Last updated with new content: Version 1.1

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 OpenEdge Release 11.6 (starting with Service Pack 11.6.3).

   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 Node.js-based, Kendo UI Builder tool that can 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 from this meta-data, allowing you to test the UI and its data access from within the Kendo UI Designer itself.

   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*.

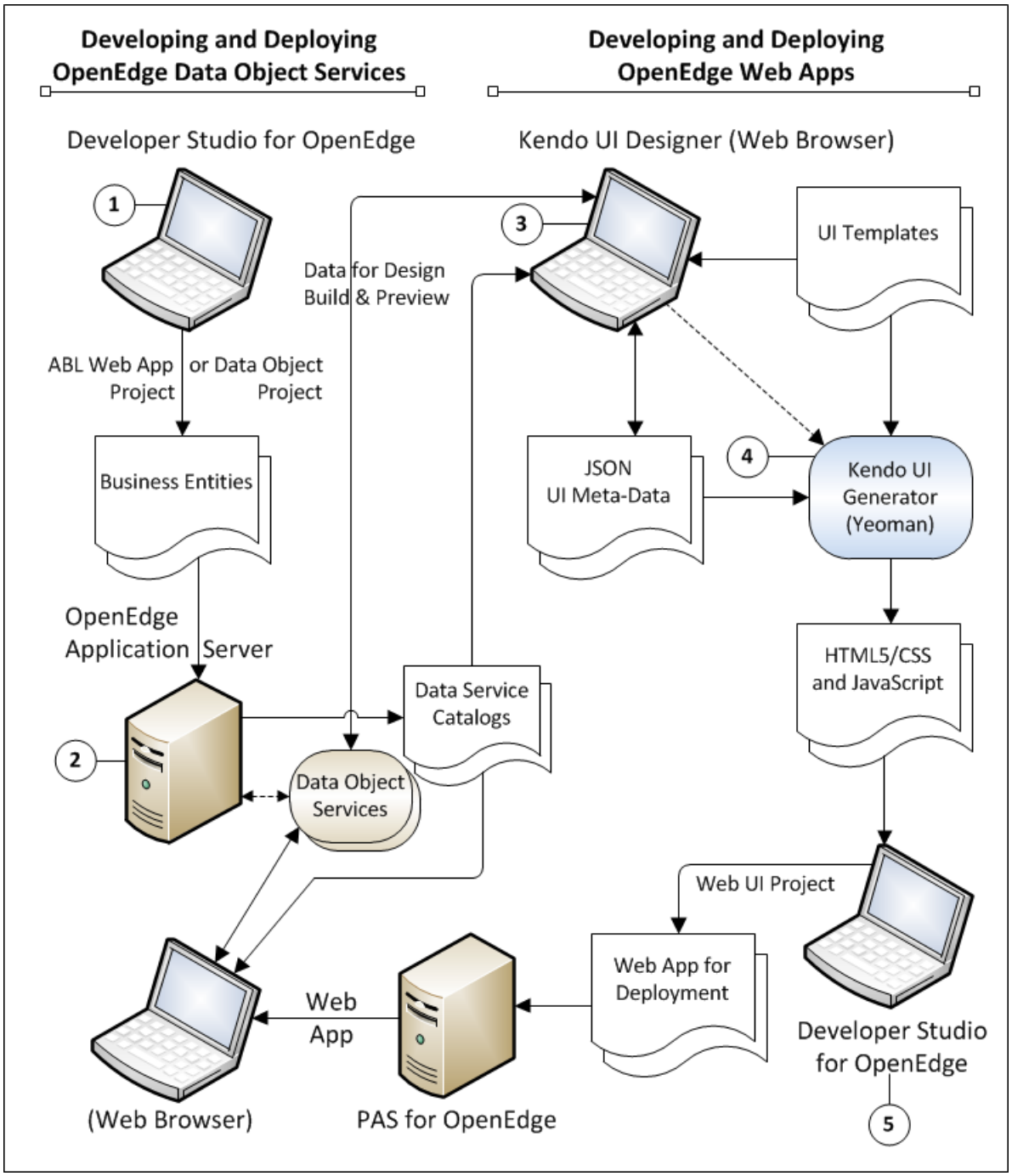
4. Optionally, use Progress Developer Studio for OpenEdge to deploy each stage of completion for both Data Object Services and the client OpenEdge web app. In Kendo UI Designer, you can configure the web app location so the Kendo UI Generator automatically builds the web app within a Web UI project of 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 a Node.js application installed on your local system using your supported default browser (for example, Chrome or Firefox) to display its UI. 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 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 is a Yeoman-based code generator that 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.
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 on 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 the Designer 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 templates from which you can select to 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 71 in this document.

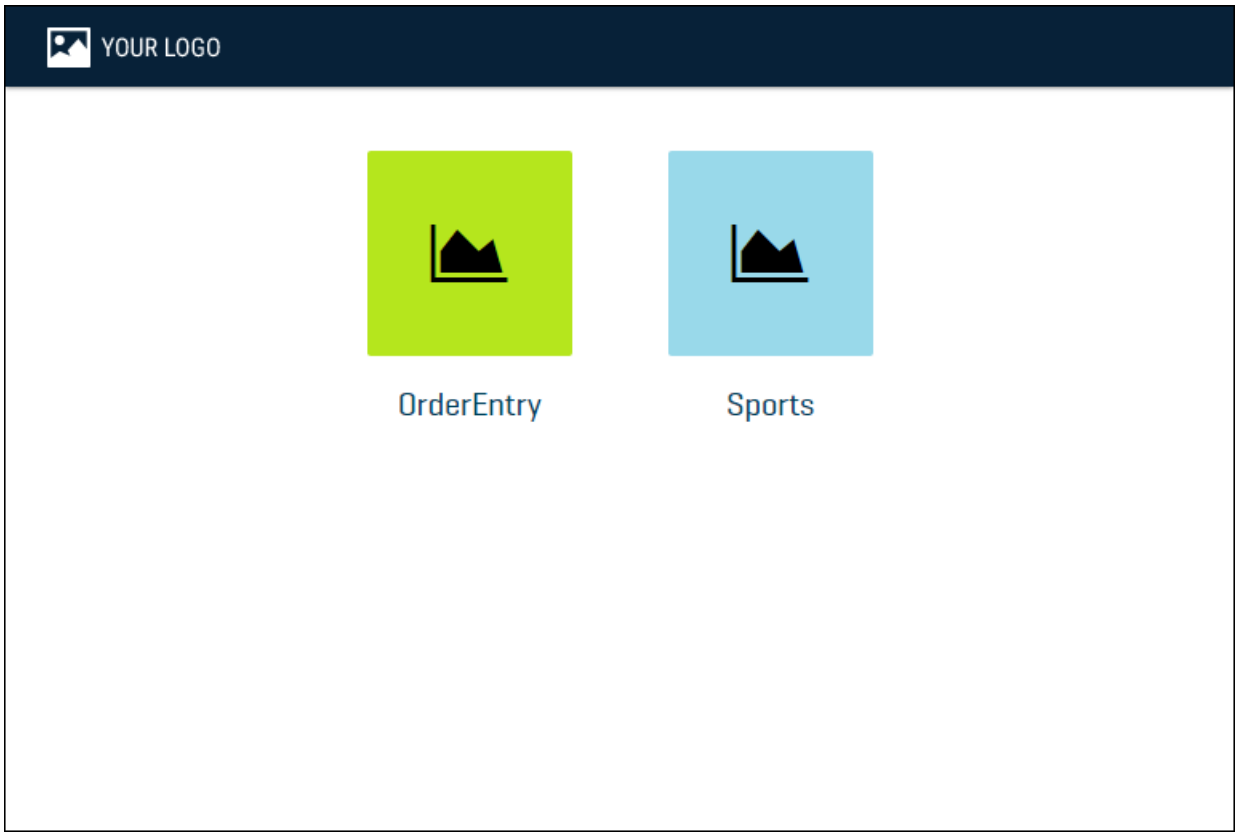
For details, see the following topics:

- App layout and components
- Creating and designing an app
- Data providers and data sources
- Modules and views
- App generation and deployment

App layout and components

At run time, each OpenEdge web app has the basic layout shown in these example screens, starting with the app landing page:

Figure 2: App layout example—landing page



The app landing page displays when you first open the web app and displays a built-in landing-page view that every Designer-generated web app contains. The landing page includes the following default components:

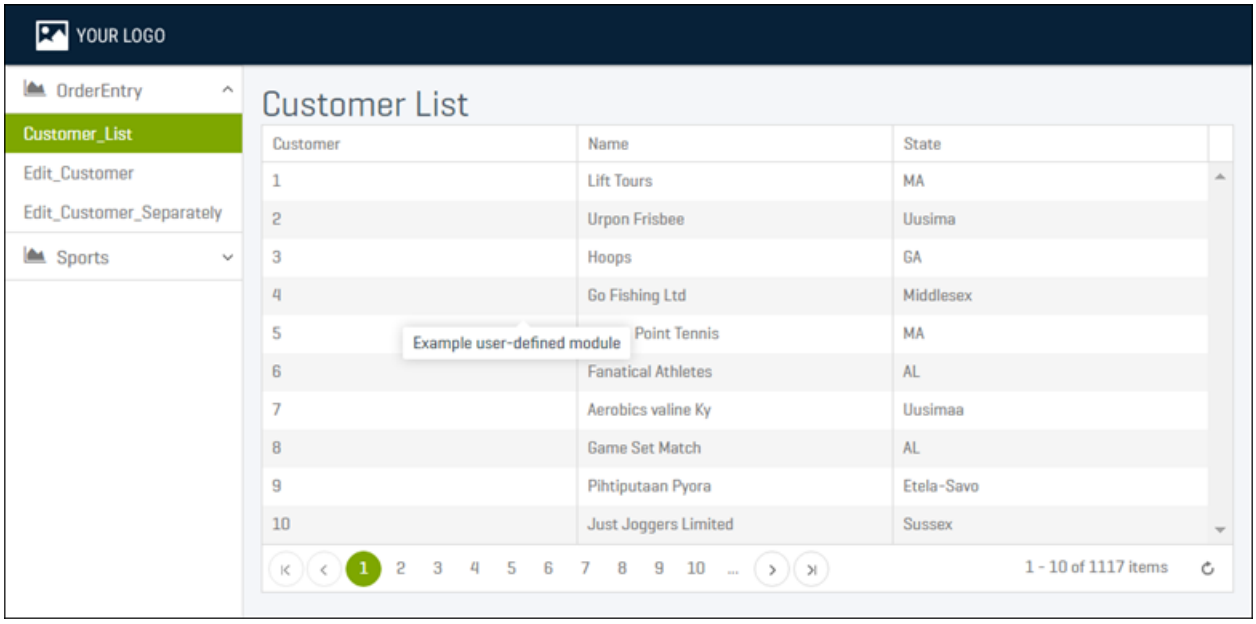
- **Header** — Showing the default logo in the example:

- 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 open the most recently displayed view.

- **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) after any required login. If the view requires a login for access, a built-in login view displays (described later) prompting for credentials to authenticate in order to open the view. If no login is required or the login succeeds, the views page opens displaying the view.

This is an example app views page that might display if you select the **OrderEntry** 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:
  - **OrderEntry** — After being selected in the app landing page, the views page is opened for this module, with its drop-down list shown with the first listed view in the module selected and displayed. You can then display a different view in the module by selecting the name of another view in the list.
  - **Sports** — An additional module, with its drop-down list yet to be selected.

  Note: As shown for **OrderEntry**, any module selected in the landing page has its initial view displayed along with the module description in an overlaying caption, in this case "Example user-defined module." 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. If the selected view

requires a login, the built-in login view is displayed to enter and authenticate credentials before displaying the view.

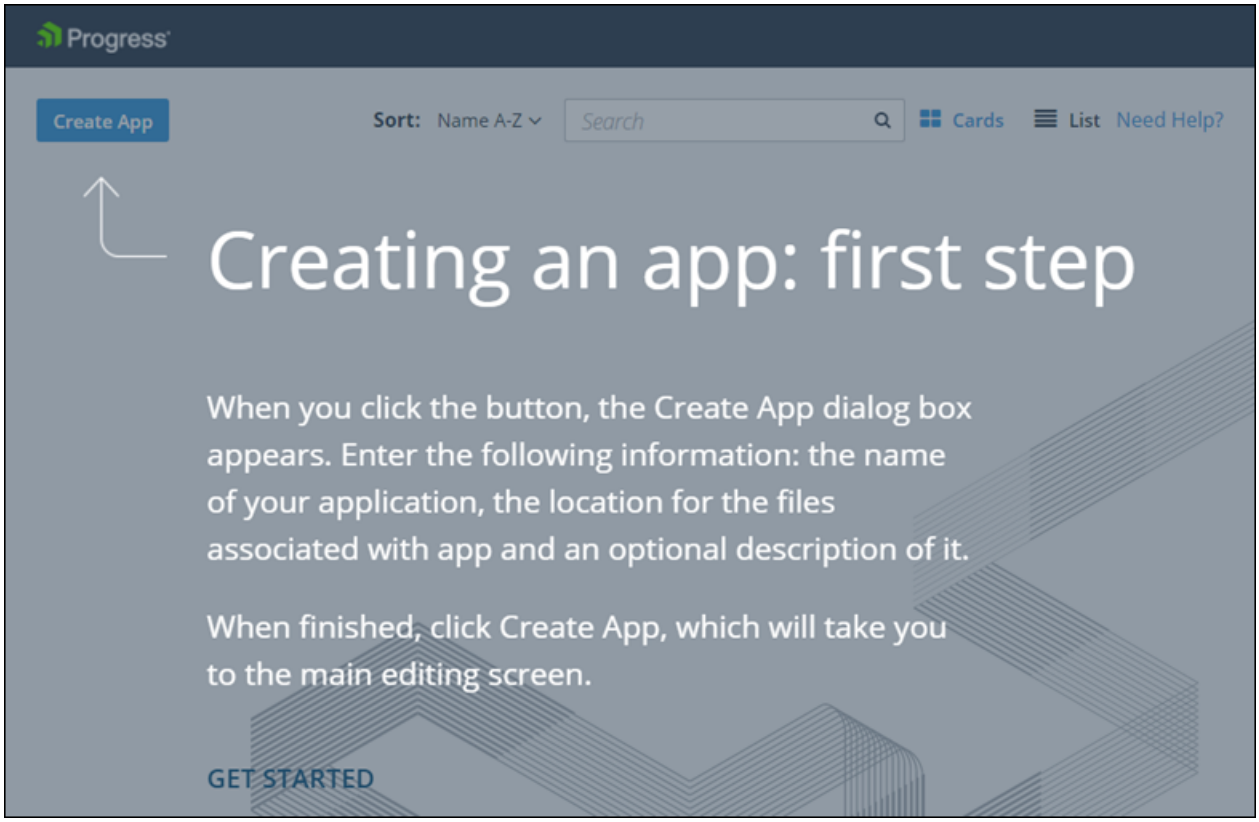
For more information on creating modules and views for an app, see Creating and designing an app on page 14.

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

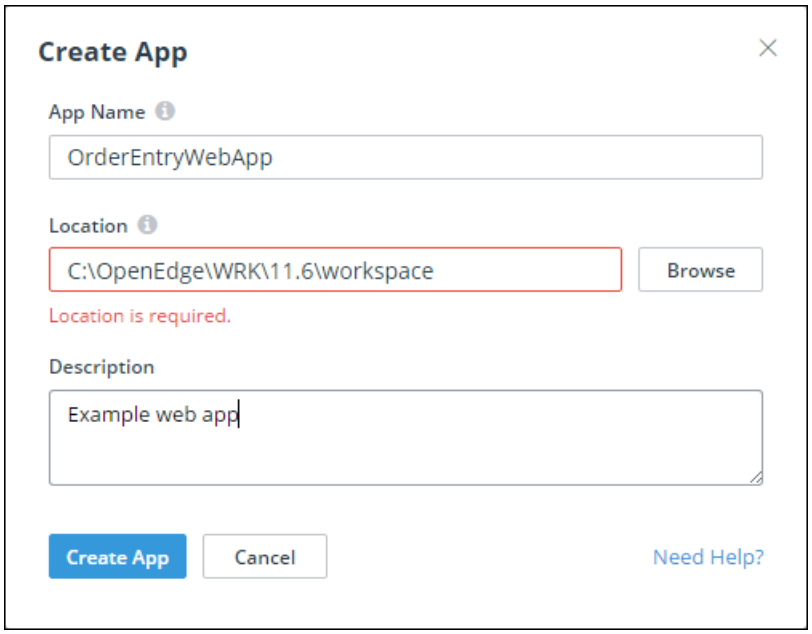
When you first start the Kendo UI Builder, it opens the Kendo UI Designer start page in your default browser similar to this:

Figure 4: Kendo UI Designer start page



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

Figure 5: Create App dialog box



The value you choose for **App Name** becomes the name of the folder where your app is created, and the **Location** is where the app folder is created. You can optionally enter a **Description**, which appears along with the **App Name** on the start page after you create the web app.

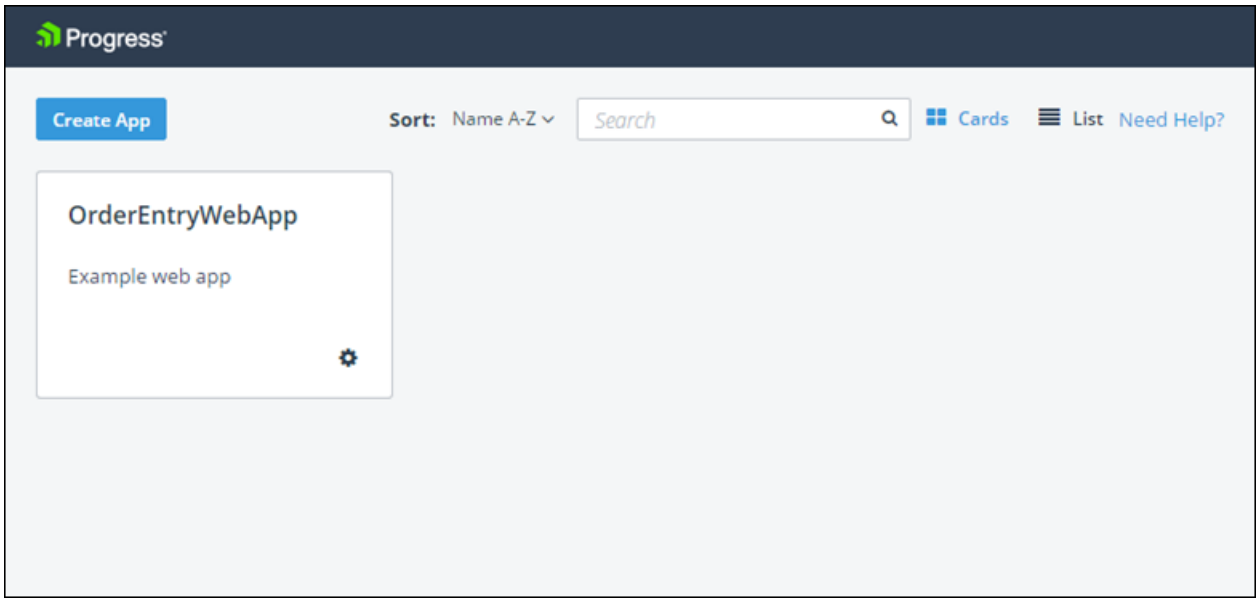
If you want your app development testing and production deployment to be managed from within a Web UI project of Progress Developer Studio for OpenEdge, specify **App Name** with the same name as the Web UI project, and specify **Location** as the folder of the Developer Studio workspace where you create the project as shown in the example. 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 project and app names and locations are the same.

After filling in the fields, you can create the app by clicking the **Create App** button, which immediately 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 start page 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: Start page with created app card

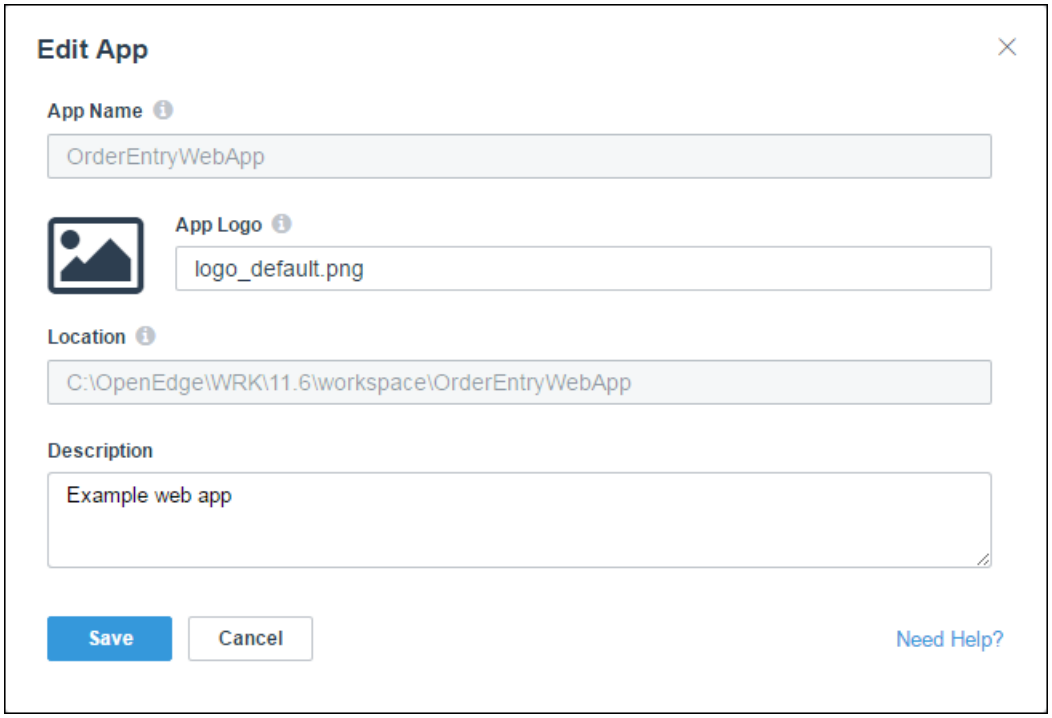


Note that to help locate one of many apps you might create, you can sort the apps or search for a given app card or list item using **Sort** or the search box, respectively. Note also, anywhere in the Designer where you see **Need Help?**, you can click to open a help topic for that page or dialog box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

Clicking the gear control in the app card (or list item), provides the option of either opening the **Edit App** dialog box (**Edit**), which allows you to edit certain properties of the app, or deleting the app from the Designer (**Delete**, with confirmation) and from any Web UI project you have created for it in Progress Developer Studio for OpenEdge.

If you open the **Edit App** dialog box, it displays similar to this example for **OrderEntryWebApp**:

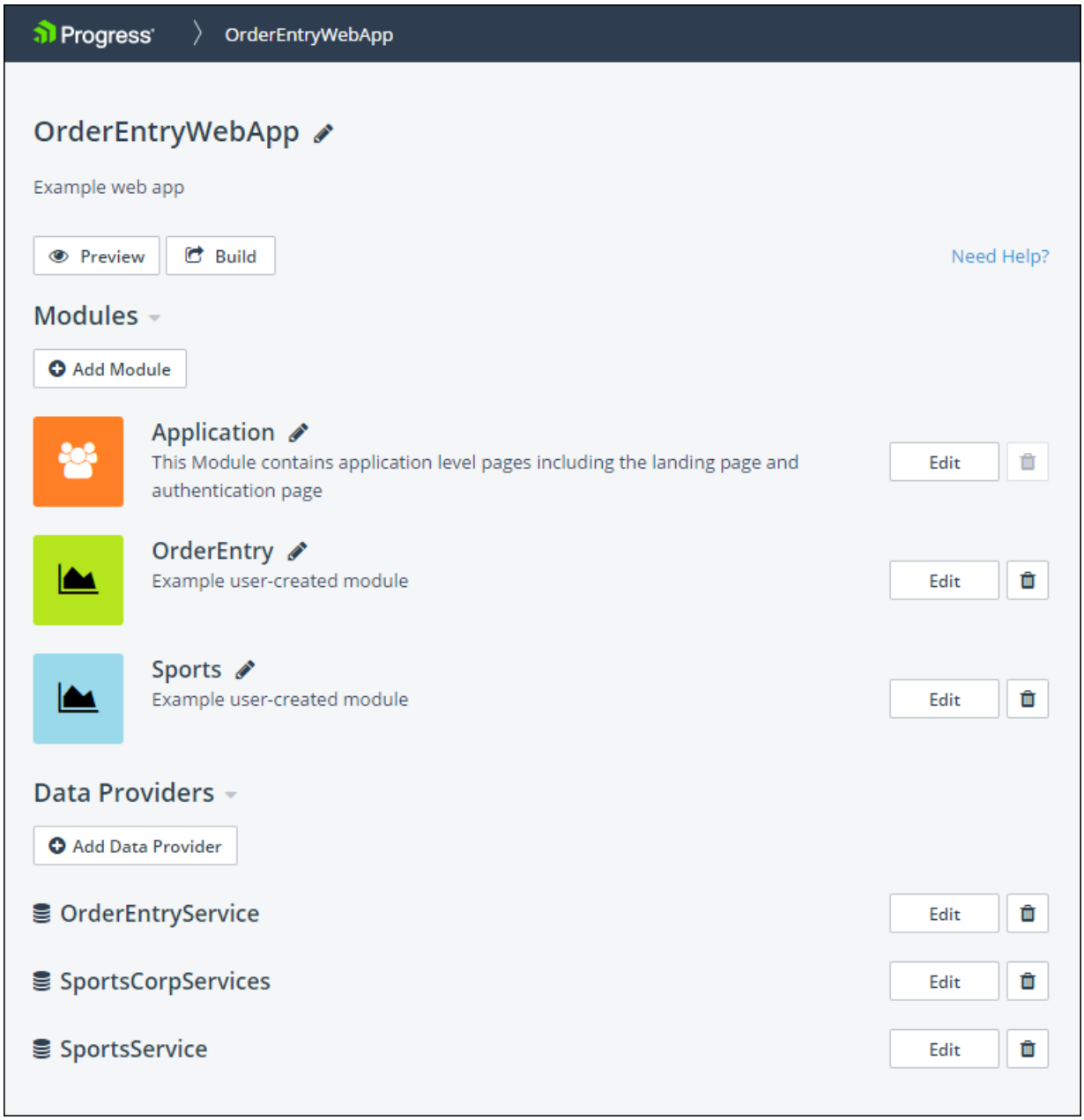
Figure 7: Edit App dialog box



For the **App Logo** property, you can replace the default `logo_default.png` image file with your company logo or other image file. (For information on where to save your own image file for access using this property, see Company logo on page 74 in this document.) The displayed image (shown beside the property setting) appears in the header of your web app at run time (see App layout and components on page 12). The **Description** property allows you to enter or edit any description you want to display for the app. The other properties are read-only and can only be set when you first create the app using the **Create App** dialog box (previously shown). Click **Save** to save any changes.

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:

Figure 8: 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 start page, 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, followed by a pencil control for editing certain properties of the app after it is created. Clicking this control opens the same **Edit App** dialog box previously described using the gear control for the app card on the Designer start page. 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:
  - **Preview** — Provides the option to either invoke the Kendo UI Generator to build and immediately preview the current state of the app or to preview the most recent build (if one exists). The preview opens in a separate browser tab using live data from the mapped data sources.
  - **Build** — Invokes the Kendo UI Generator to build the current state of the app without running a preview.

  For more information on app builds, see App generation and deployment on page 68.
- **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 **Sports**.

  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.

  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 19.
  - **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 12).

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

  For more information, see Modules and views on page 30.
- **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, three data providers are created, **OrderEntryService, SportsCorpServices**, and **SportsService**.

  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 19.

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. 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 71.

Data providers and data sources

A data provider defines a single data service and one or more data sources that represent tables from that data service, which you can bind to views. This data service can be a **Progress Data Service** that represents a single point of authentication for one or more OpenEdge Data Object Services. This single point of authentication is the URI of a server web application that supports the specified Data Object Service or Services. Each Data Object Service provides access to one or more Data Object resources that provide the tables you can specify to 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. The data provider definition also identifies the authentication model required to access the data server (web application) that it supports, which you can specify from the following options:

- Anonymous
- Basic
- Form

You can create and define data sources that you want the data provider to provide, both 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 Data Object Services hosted by multiple web applications.

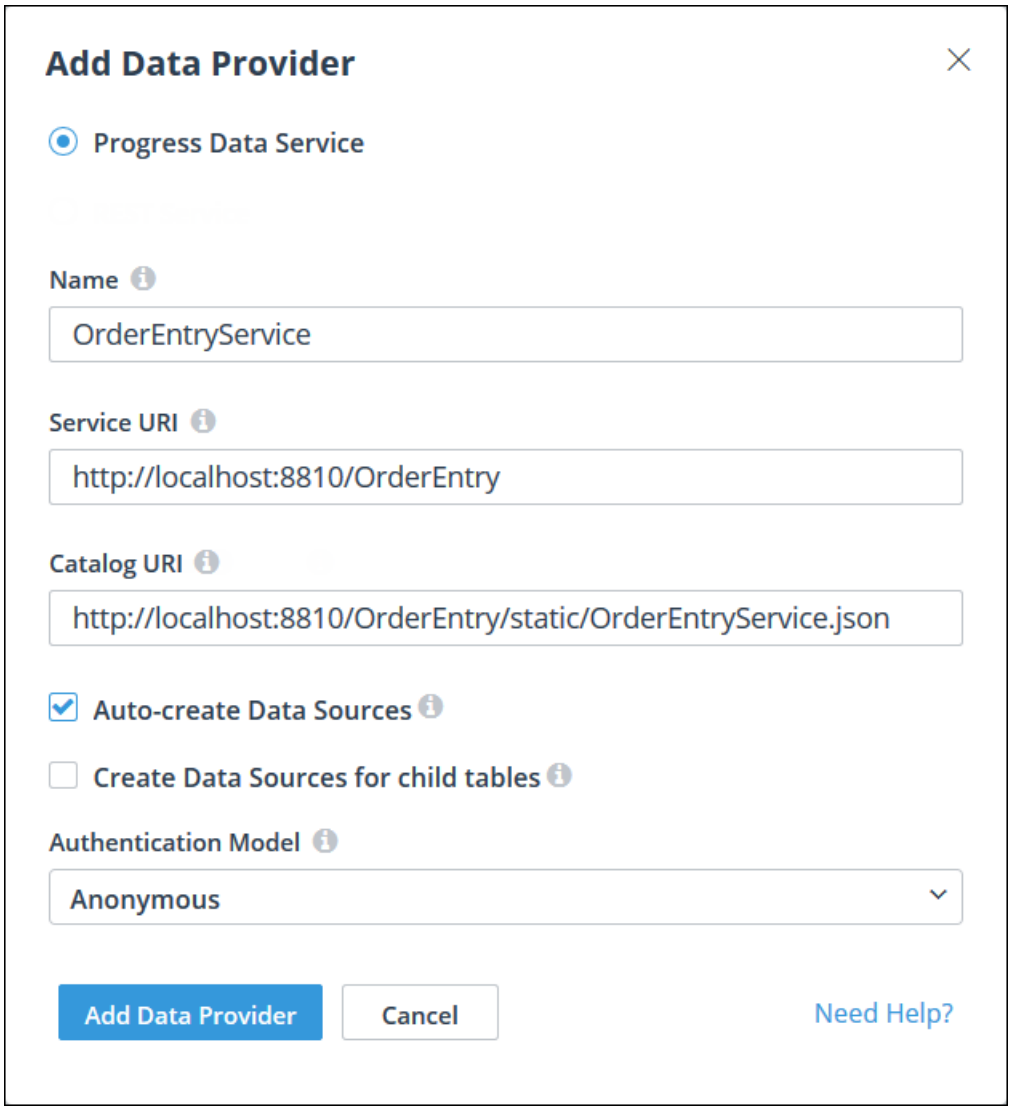
For details, see the following topics:

- Adding and editing a data provider on page 20
- Adding and editing a data source on page 22
- Editor and semantic types on page 25

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 9: Add Data Provider dialog box



In this example, the data provider is defined as a **Progress Data Service**, with `OrderEntryService` entered as its **Name**, with the OrderEntry web application URI specified as its **Service URI**, and the Catalog URI for the single Data Object Service, `OrderEntryService`, specified in the **Catalog URI** field. 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 the **Authentication Model** for the data provider specified as `Anonymous`.

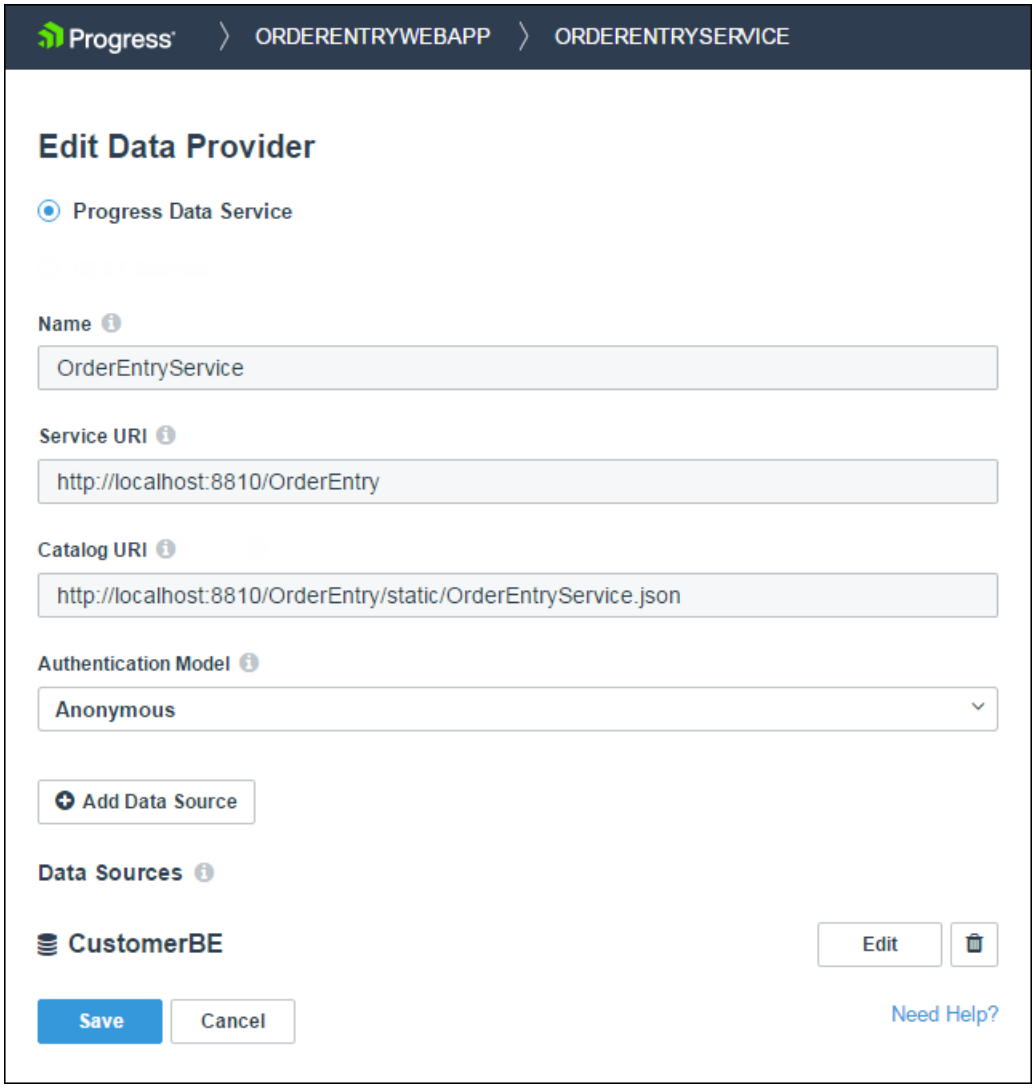
Note: The **Service URI** field always specifies the URI of the single server component that provides a connection to the Data Object Service whose Catalog URI is specified in the **Catalog URI** field, and that server component is always a single web application running on the web server for an OpenEdge 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 14).

Note: Once you create a data provider, you cannot change its specified **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 68.

At this point, you might want to review and edit the data sources created for this data provider by clicking the data provider's **Edit** control. For the OrderEntryService data provider just created, this displays an edit data provider page similar to this example:

Figure 10: Edit data provider page



Here, you can update the **Authentication Model** for the data provider definition, and create any additional data sources or edit data sources that have already been created.

Any data sources initially created for the data provider when you first create it appear in a list under **Data Sources**. In the example, there is only one data source created with the name, `CustomerBE`. For an auto-created data source like this, this is the name of a table provided by the associated Data Object resource.

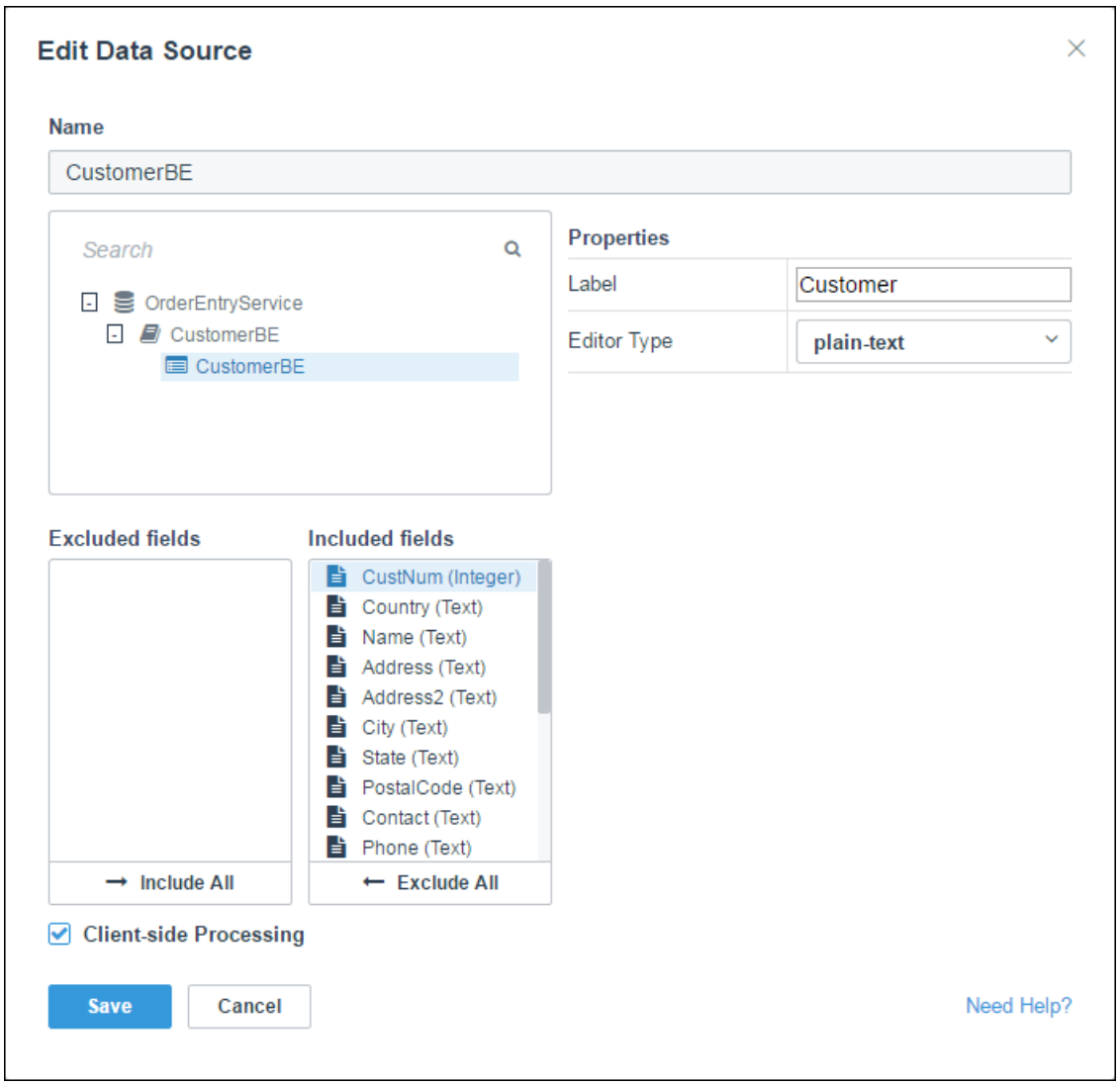
If the Data Object Service specified for **Catalog URI** supports additional Data Object resource tables, you can manually create new data sources for them by clicking **Add Data Source**. For more information, see Adding and editing a data source on page 22.

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 20), 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 `CustomerBE` data source in this example:

Figure 11: 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.

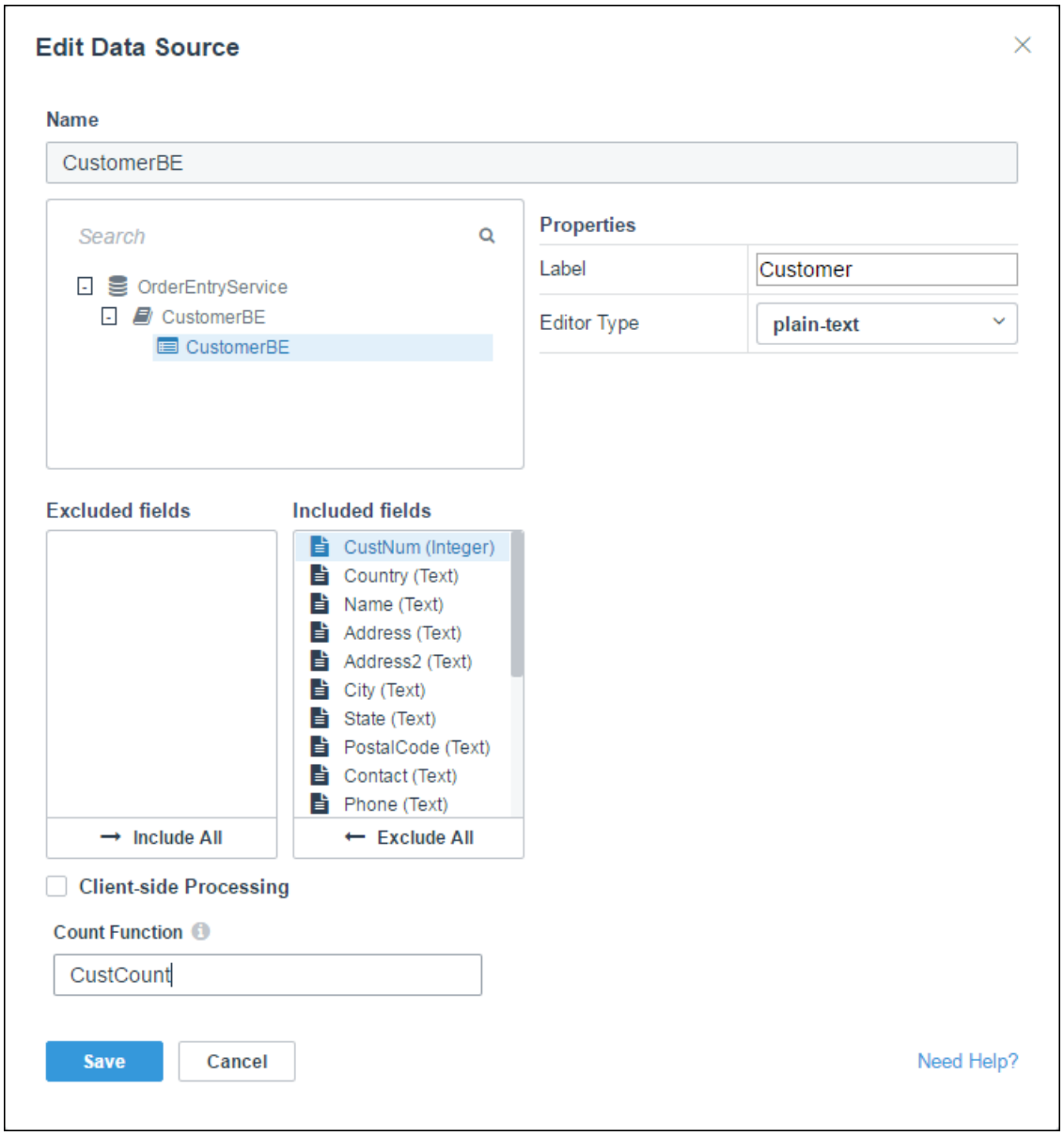
The **Excluded Fields** and **Included Fields** of the field list allow you to exclude or include all fields, and to drag-and-drop individual fields for exclusion from or inclusion in the fields that are initially available to populate views from this data source.

Note: Similar field list settings can also be changed for the individual UI components of a view (see Modules and views on page 30).

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 12: Setting the data source for filtering, sorting, and paging 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 30). For more information on this routine, see the sections on "updating Business Entities for access by Telerik DataSources and Rollbase external objects" in *OpenEdge Development: Web Services* and in *OpenEdge Service Pack 11.6.3: New Information*.

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 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 25.

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 30).

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 22), 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* and the *New Information* documentation for recent service packs of your OpenEdge Release 11.6 (starting with Service Pack 11.6.3).

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.	Two-option Radio Set
`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

Editor type	Behavior	Kendo UI visualization
`currency-input`	Describes the visualization and behavior of a numeric field that represents currency data.	Numeric Input
`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.	DateTime Picker
`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 Input
`integer-input`	Describes the visualization and behavior of an integer field.	Numeric Input
`numeric-input`	Describes the visualization and behavior of a numeric field for all supported numeric domains and formats.	Numeric Input
`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 Input
`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`).	Numeric Input
`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`).	Numeric Input

Editor type	Behavior	Kendo UI visualization
`phone-number-input`	Describes the visualization and behavior of a text field that represents a phone number.	Masked Input
`plain-text`	Provides the ability to display any semantic type as a read-only value.	HTML5 text element (read-only)
`text-input`	Describes the visualization and behavior of a single-line text field.	Text Input
`url-input`	Describes the visualization and behavior of a text field that accepts well-formed URL values.	Text Input

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`

`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	—	—

Semantic type	Function	Default editor type	Compatible editor types

`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`
`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`

ABL field data type	Semantic type default	Editor type default
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 f-or 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 14, 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 14).

  At design time, you can edit properties of the login view. 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 76). 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 12).

- **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 four templates that you can use to create views in a user-created module thus include:

  - **Data-Grid** — This is a simple predefined read-only grid that displays rows of records from the single view data source instance you specify on the app views page.

- **Data-Grid-Form** — This is a predefined read-only grid, similar to the Data-Grid, that 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 the single view data source instance you specify according to the editor type defined for each field in the data source.
- **Data-Grid-Separate-Form** — This is a read-only grid, similar to the Data-Grid, that that 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 the single view data source instance you specify according to the editor type defined for each field in the data source.
- **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), navigation (e.g., Toolbar), and custom HTML components. The layout is based on a Bootstrap fluid grid system that you define initially with row and column components. The content consists of individual UI components, many of which can be bound to data.

  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 view factory event functions as required for the content behavior you want in a JavaScript file that the Kendo UI Builder generates for the view.

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. In addition, for those views with both a grid and form, only one design-time property setting is required to change from one supported edit mode to another. You can also customize these views with additional code and settings at a number of extension points in your Kendo UI Builder installation and web app folders, including an AngularJS view factory file for coding view event functions.

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.

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.

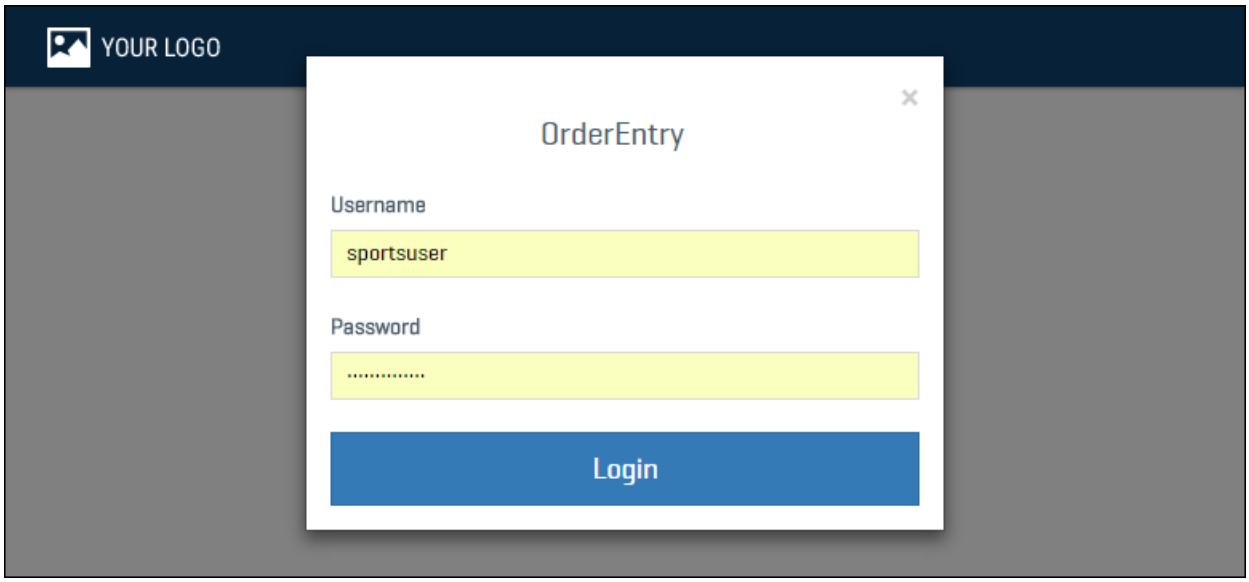
For details, see the following topics:

- Editing the login view on page 32
- Adding and editing a Data-Grid view on page 35
- Adding and editing a Data-Grid-Form view on page 40
- Adding and editing a Data-Grid-Separate-Form view on page 48
- Adding and editing a Blank view on page 54

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:

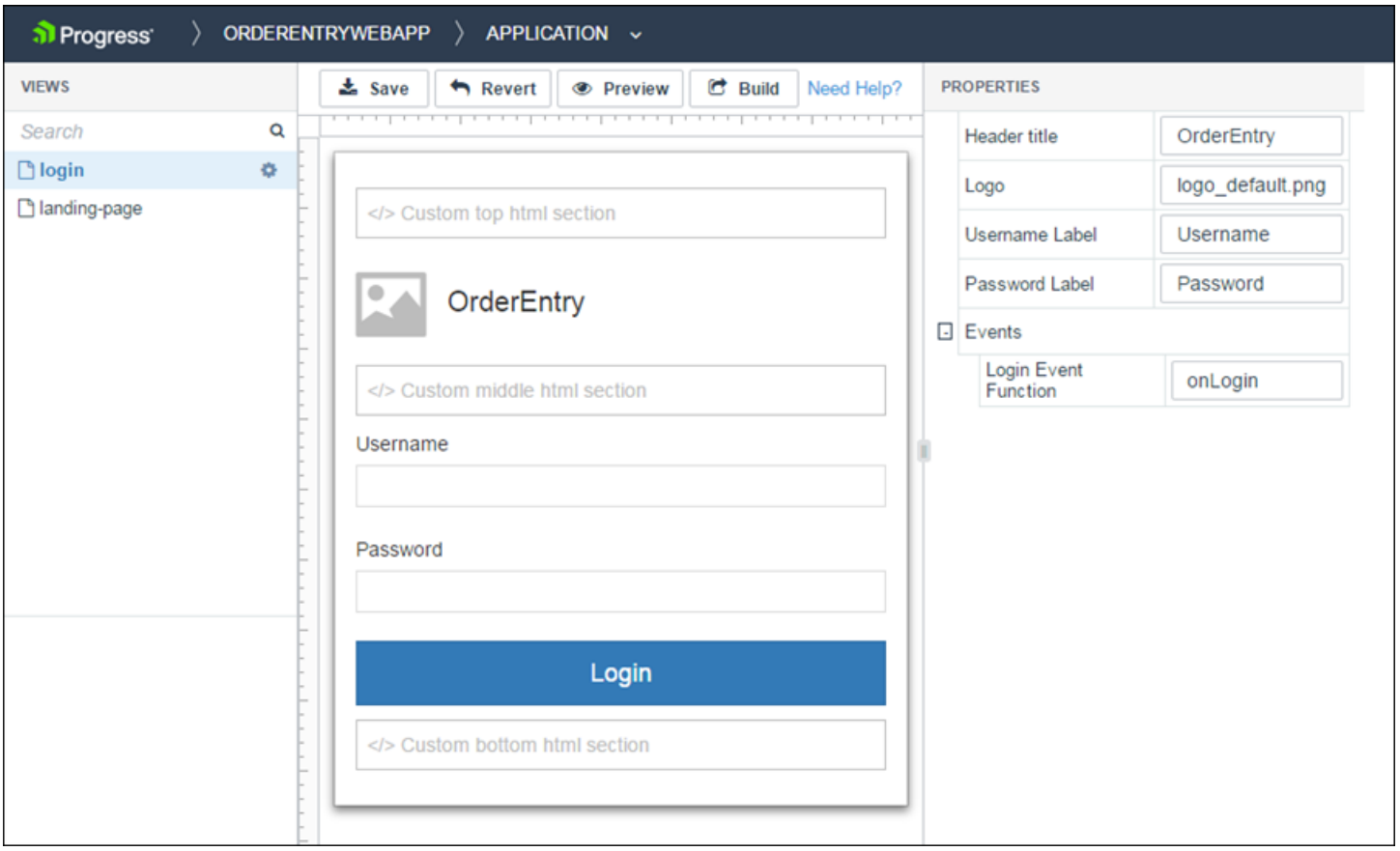
Figure 13: Login view running in app



After entering your credentials and clicking **Login**, the app authenticates the data provider bound to the view you have selected and provides access to the selected view only if authentication succeeds.

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 14: 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**.
  - **Preview** — Provides the option to either invoke the Kendo UI Generator to build and immediately preview a generated app in its currently saved state or to preview the most recent build (if one exists). The preview opens in a separate browser tab using live data from the mapped data sources.
  - **Build** — Invokes the Kendo UI Generator to build a generated app in its current state without running a preview.
  - — 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.

For more information on app builds, see App generation and deployment on page 68.

- **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 12).

  - **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 80 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:

- **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 76 and View-specific events on page 78 in this document.

Adding and editing a Data-Grid view

The Data-Grid view is a simple read-only grid that displays rows of records from the bound data source on the app views page.

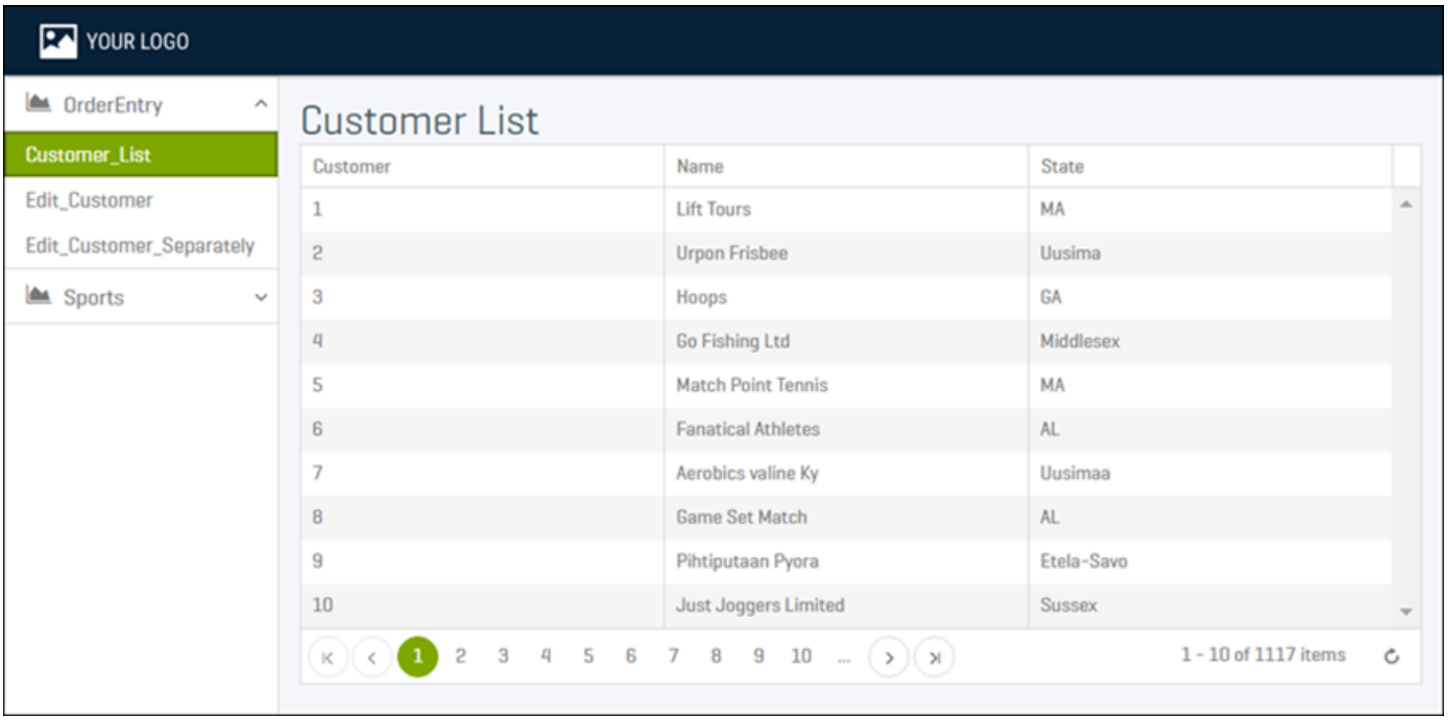
At run time, the grid initially displays by itself with no records selected. (Row selection in this grid has no built-in function except to highlight the selected row.) You can then navigate the rows of the grid a page at a time using a page selection control at the bottom of the grid.

At design time, you can 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 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 grid views with forms, with the addition of an appropriate built-in behavior for displaying data in a form when you select a row.

When you select a Data-Grid view in a module at run time, the app displays a page similar to the **Customer_List** view in this example:

Figure 15: Data-Grid view running in app

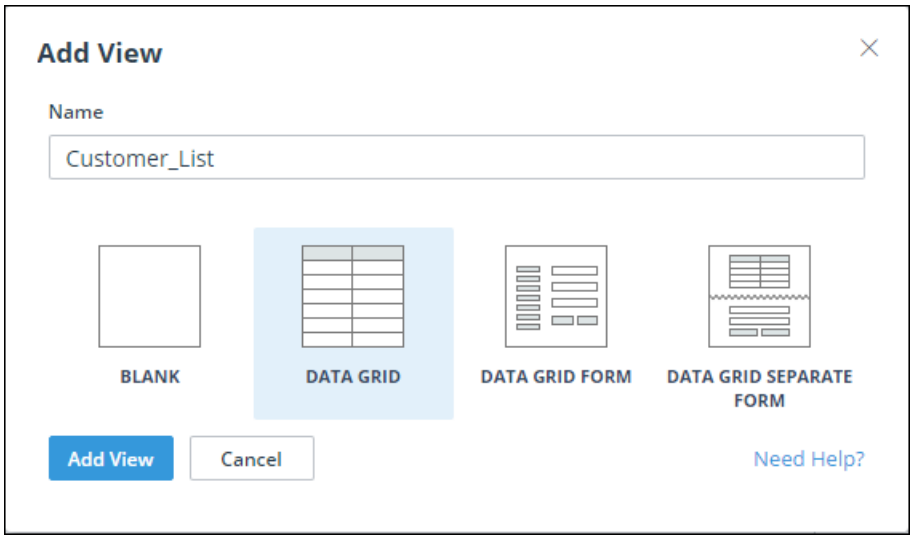


This view displays read-only rows of records from its bound data source in a tabular list, showing field values for selected columns of each record. The rows are displayed in pages with a design-time specified size. You can navigate through the pages of the grid view by selecting the first, previous, specific, next or last page control. 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, then click **Add View** at the top of the **VIEWS** pane (see Figure 17: Data-Grid view design page on page 37 for an example).

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

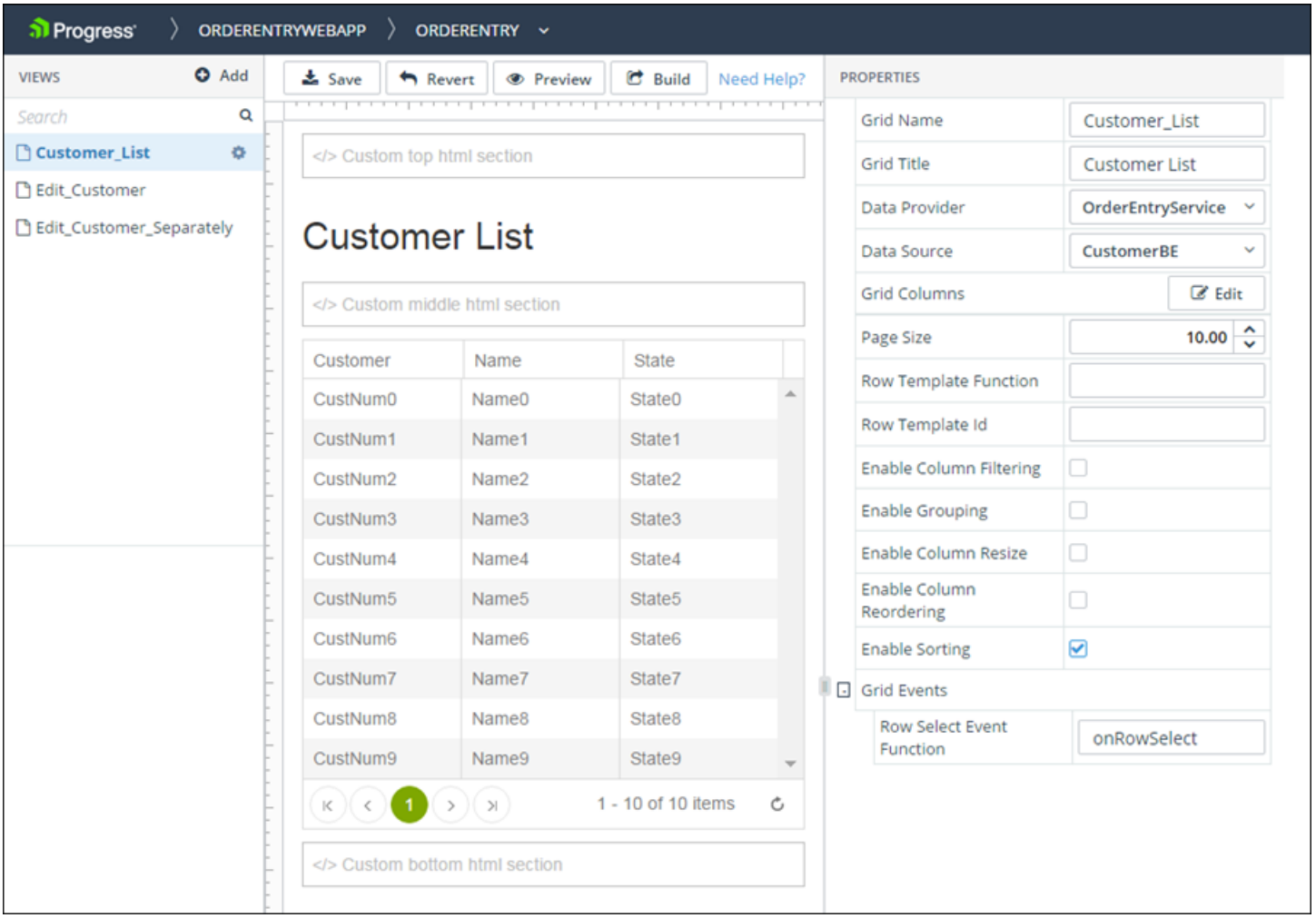
Figure 16: Add View dialog box creating a Data-Grid view



In this example, the dialog box has `Customer_List` entered as the value of the view **Name** and **DATA-GRID** selected as the view type. The view types listed in this dialog box (shown with a corresponding icon) identify the available view templates you can use to add a user-created view.

After specifying the name 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 17: 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**.
  - **Preview** — Provides the option to either invoke the Kendo UI Generator to build and immediately preview a generated app in its currently saved state or to preview the most recent build (if one exists). The preview opens in a separate browser tab using live data from the mapped data sources.
  - **Build** — Invokes the Kendo UI Generator to build a generated app in its current state without running a preview.

-  — 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.

  For more information on app builds, see App generation and deployment on page 68.

- **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).
  - **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 80 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).

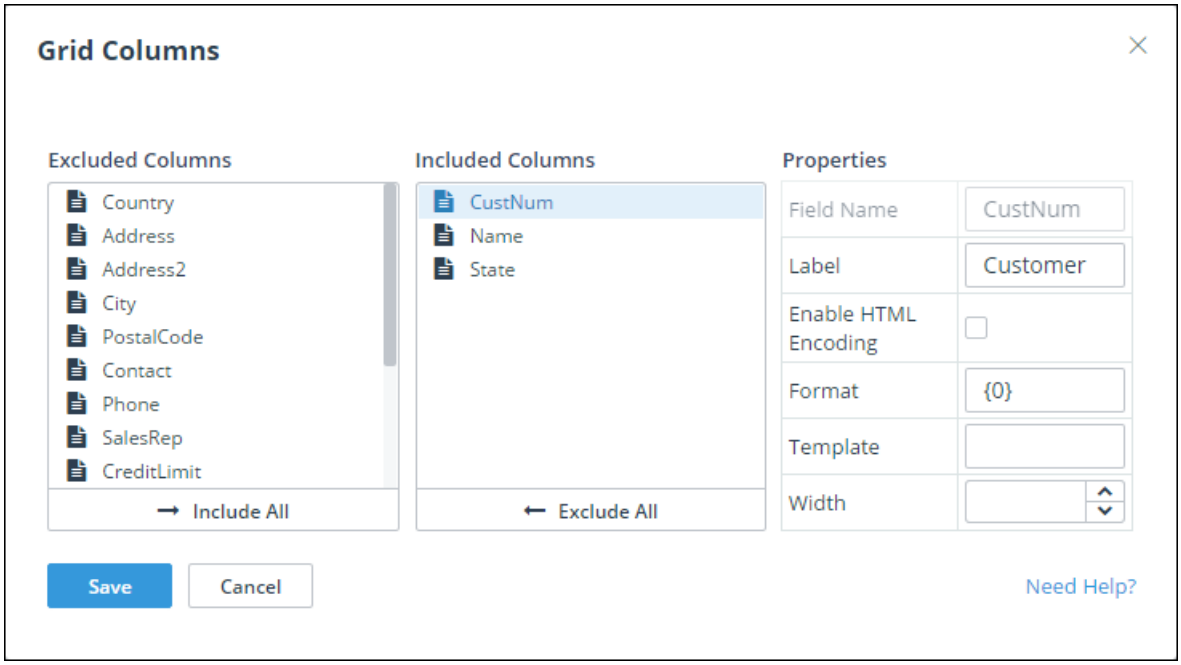
- **VIEW PROPERTIES pane (in panel on the right)** — Contains all the properties that you can set that change the design-time appearance and content of the view, as well as some settings that can affect view behavior, such as view-specific event handler settings.

  Additional properties and values of note include:

  - **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 19.
  - **Grid Columns** — Clicking **Edit** for this property opens a 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, as shown in this example for the three data source fields specified as **Included Columns**, `CustNum`, `Name`, and `State`:

Figure 18: Grid Columns dialog box



The specified **Properties** apply to each field that you select in the **Included Columns** list, as shown for the `CustNum` field. The **Template** property 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 83 in this document.

- **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 81 in this document.

- **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 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 22.

- **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.

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 76 and View-specific events on page 78 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. 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.

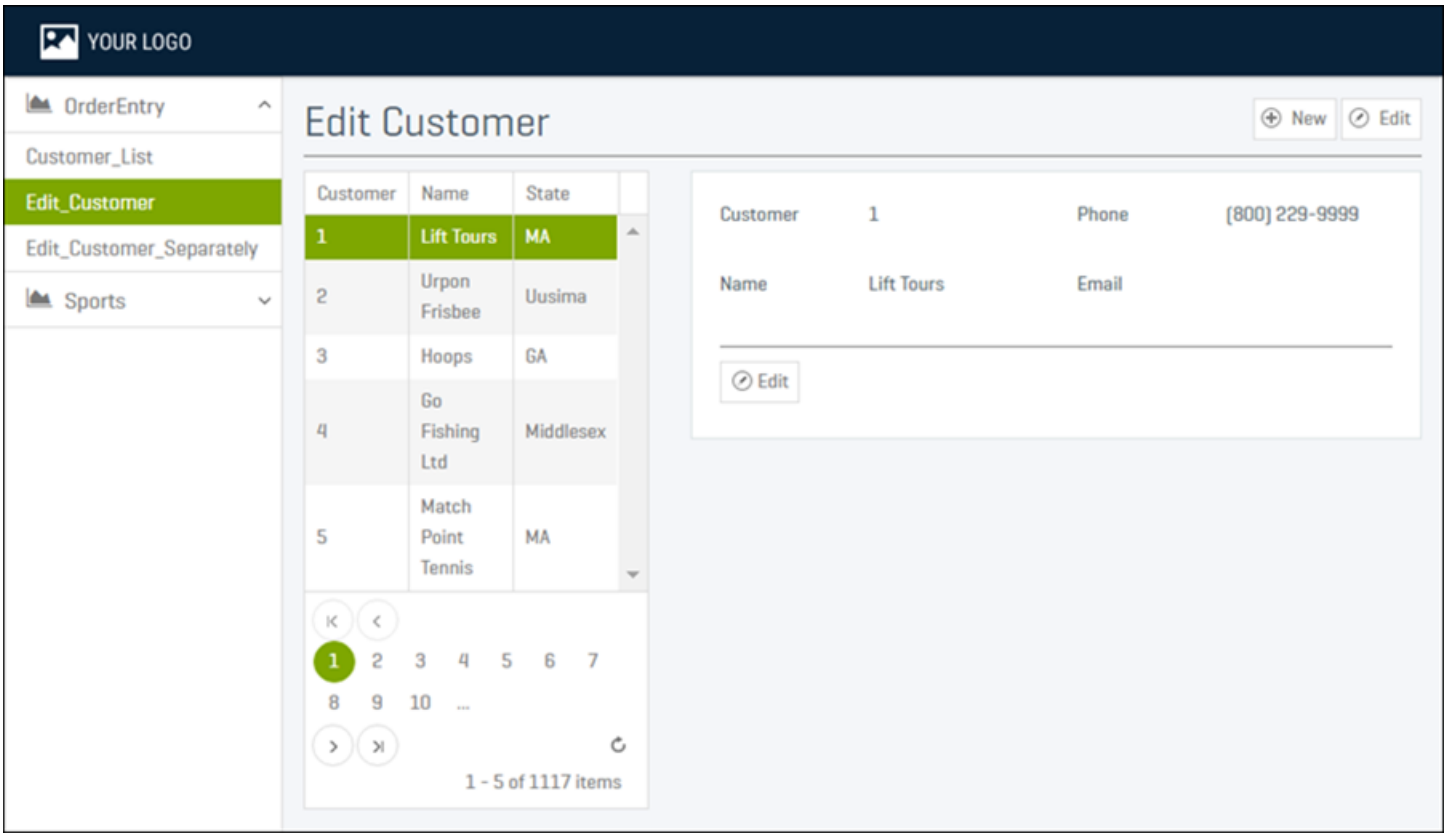
The same is true for a read-only-to-edit form, but you also have the option to edit the selected record or to add a new record to the bound data source. If you choose to edit the selected record, the view overlays the read-only form with a form that displays the record fields for editing according to 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 a Data-Grid-Form view in a module at run time, the app displays a page similar to the **Edit_Customer** view in this read-only-to-edit mode example:

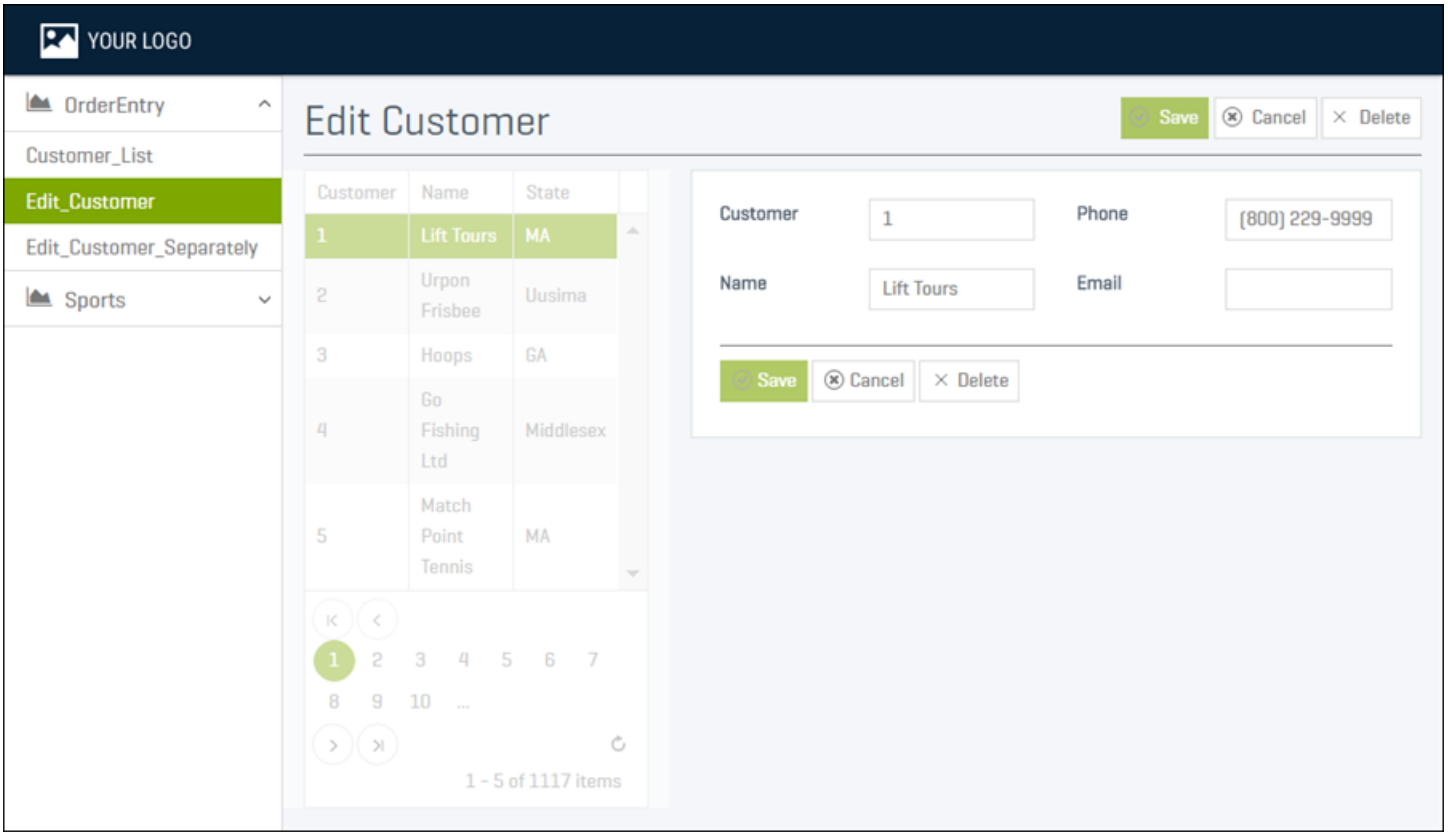
Figure 19: Data-Grid-Form view running in app with read-only form



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 35) in a split screen with the read-only form containing four fields. 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

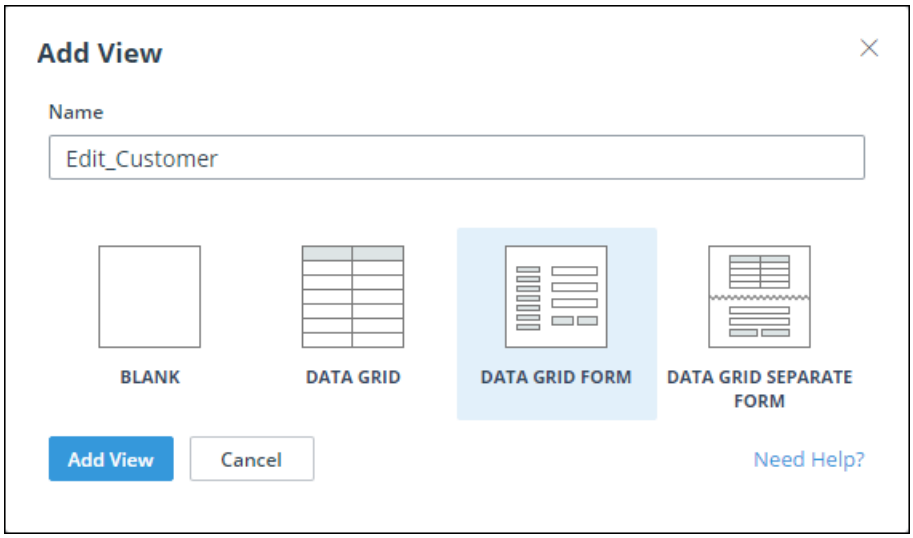


Each field in the editable form is displayed according to the editor type that has been defined for it in the bound data source (see Adding and editing a data source on page 22). From here, the edited record can be saved (by clicking **Save**), deleted (by clicking **Delete**), or the edit canceled (by 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, then click **Add View** at the top of the **VIEWS** pane (see Figure 22: Data-Grid-Form view design page on page 44 for an example).

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

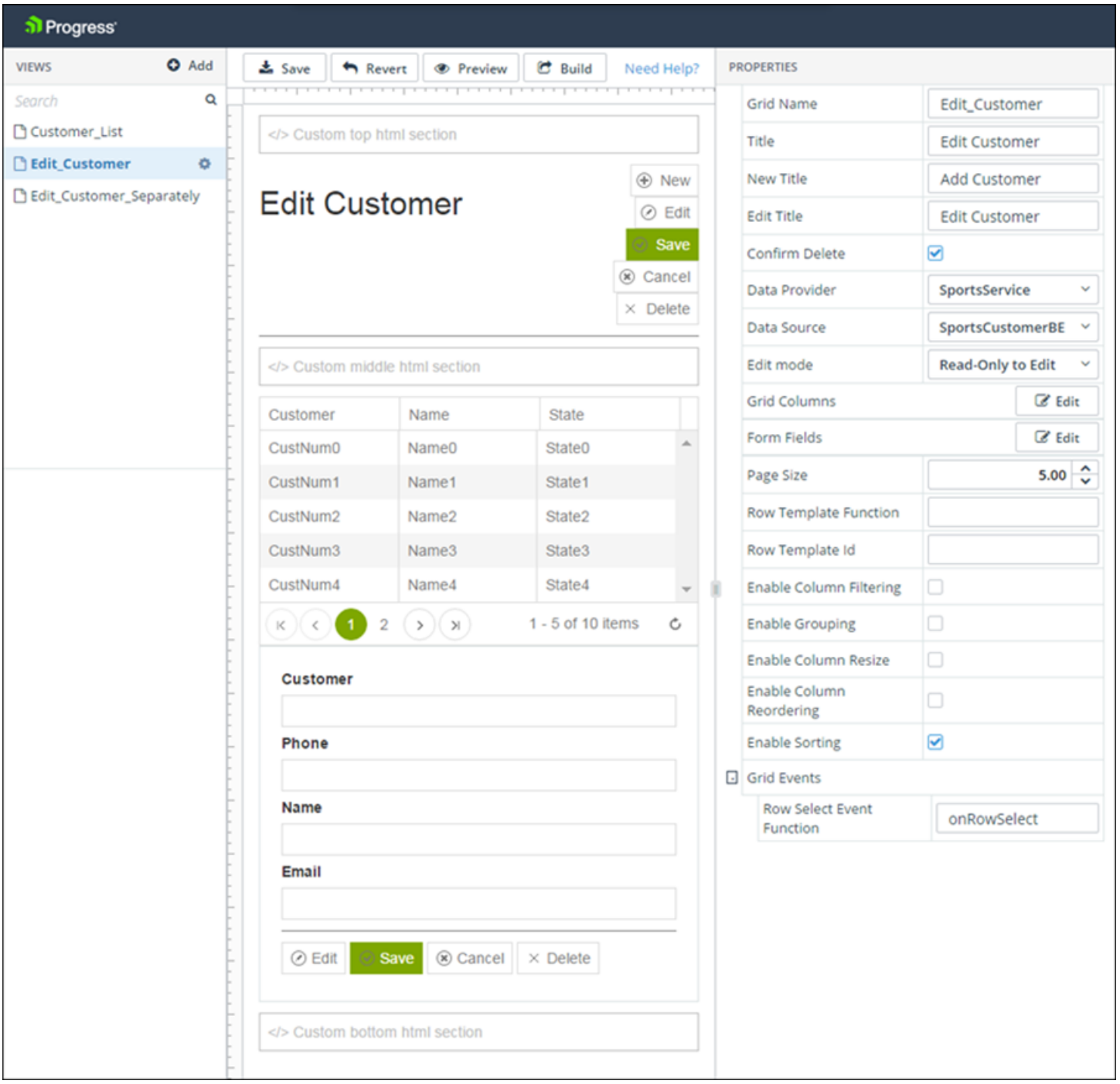
Figure 21: Add View dialog box creating a Data-Grid-Form view



In this example, the dialog box has `Edit_Customer` entered as the value of the view **Name** and **DATA-GRID-FORM** selected as the view type. The view types listed in this dialog box (shown with a corresponding icon) identify the available view templates you can use to add a user-created view.

After specifying the name 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**.
- **Preview** — Provides the option to either invoke the Kendo UI Generator to build and immediately preview a generated app in its currently saved state or to preview the most recent build (if one exists). The preview opens in a separate browser tab using live data from the mapped data sources.
- **Build** — Invokes the Kendo UI Generator to build a generated app in its current state without running a preview.
-  — 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.

For more information on app builds, see App generation and deployment on page 68.

- **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 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).

    Note: To fit on the document page with a readable size, the example view design panel is horizontally compressed so that the form appears below the grid with the form buttons above the grid and below the form instead of above and below the form. Also, the fields on the compressed form appear out of order compared to their usual arrangement. Typically, at full size, the form in the view design page is to the right of the grid, as in the run-time view example shown previously.

  - **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 80 in this document.

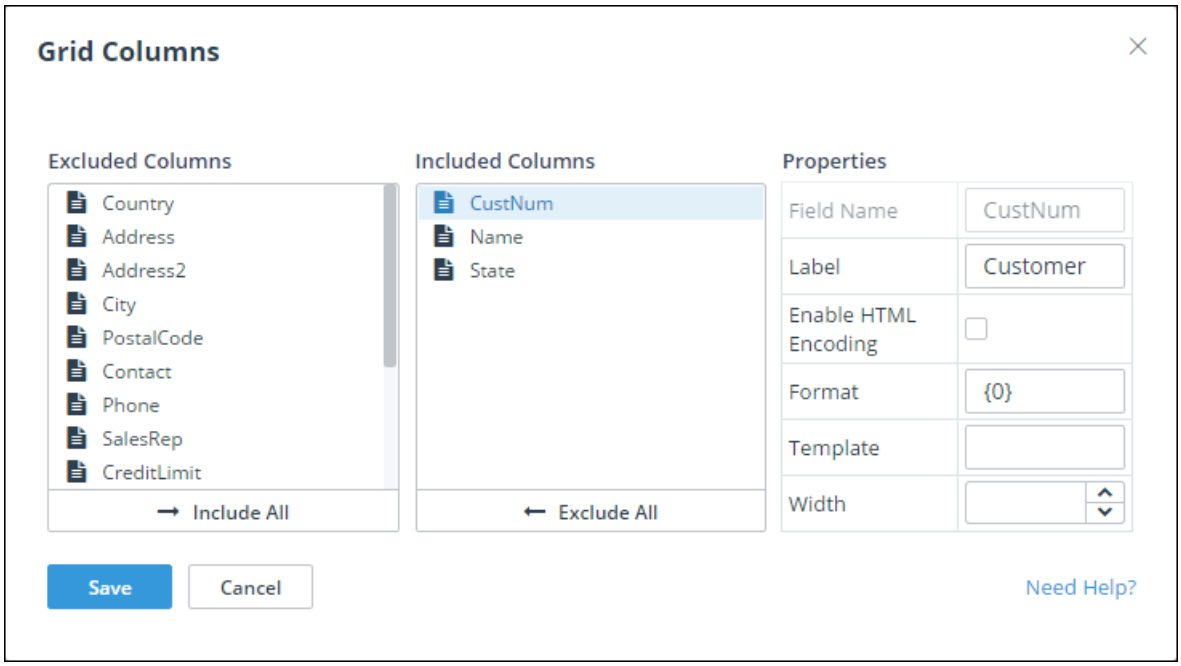
The design simulation in a view design panel changes as you modify some of the view properties that affect its appearance, and even as you click around in the simulated view design panel, depending on these property settings. For example, the **Edit_Customer** view title is set to `Edit Customer` using 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).

- **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 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 19.
  - **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 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, as shown in this example for the three data source fields specified as **Included Columns**, `CustNum`, `Name`, and `State`:

    Figure 23: Grid Columns dialog box

    The specified **Properties** apply to each field that you select in the **Included Columns** list, as shown for the `CustNum` field. The **Template** property 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 83 in this document.

  - **Form Fields** — Clicking **Edit** for this property opens a dialog box that allows you to specify what data source fields appear as fields on a form, in this example, `CustNum`, `Name`, `Phone`, and `EmailAddress`.

This dialog box is similar to the **Grid Columns** dialog box with fewer **Properties** that affect how each field that you select in the **Included Fields** list is displayed in the form (**Label Text** and **Format** only).

- **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 81 in this document.

- **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 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 22.

- **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.

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 76 and View-specific events on page 78 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. 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. 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.

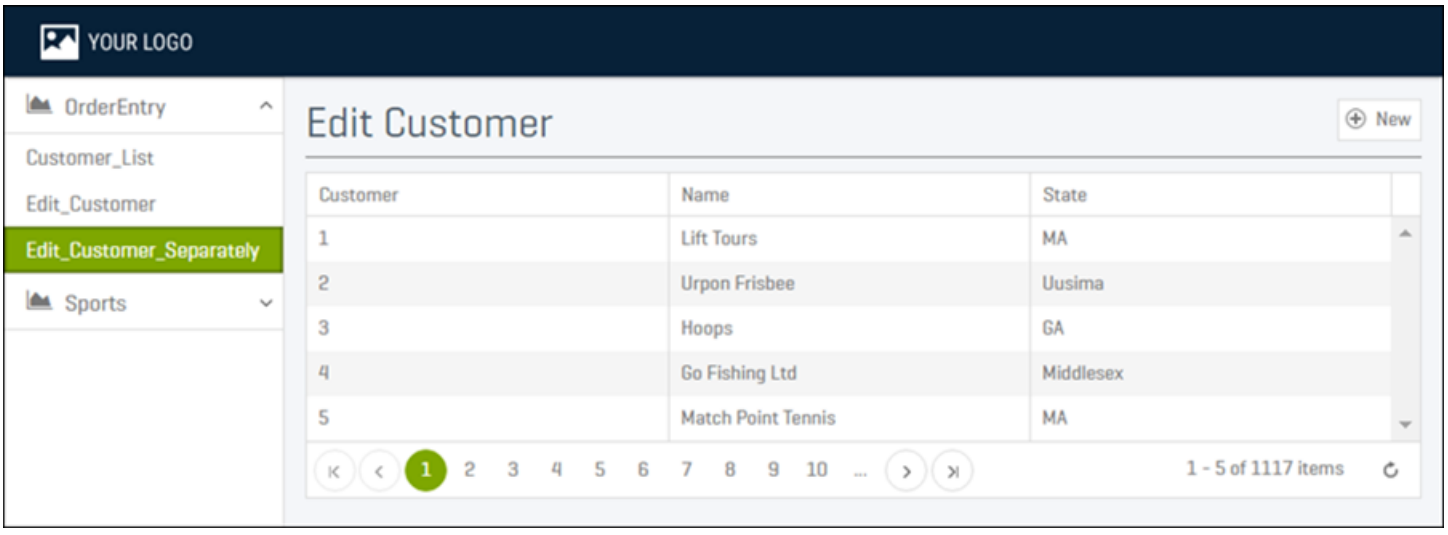
If you select the grid option to add a new record, the view overlays the grid with a similarly editable form that displays the fields for a new record with initial values that you can change before adding the record to the data source.

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 **Edit_Customer_Separately** view in this read-only-to-edit mode example:

Figure 24: Data-Grid-Separate-Form view running in app

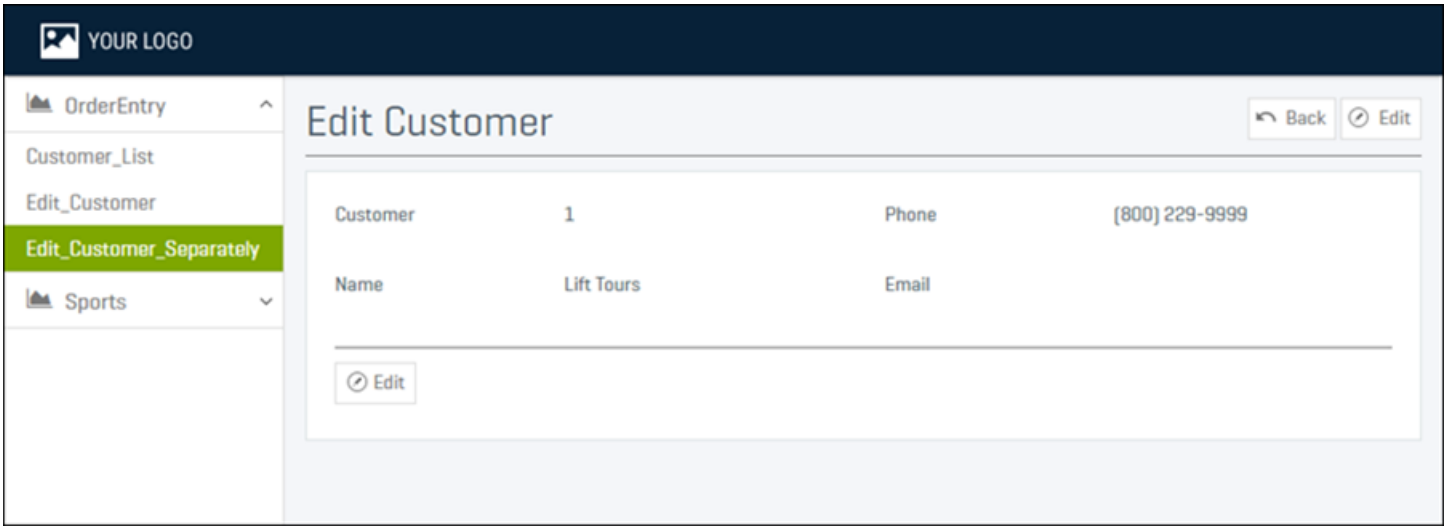


The view opens with a read-only grid similar to the previous **Customer_List** Data-Grid view example (see Adding and editing a Data-Grid view on page 35).

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:

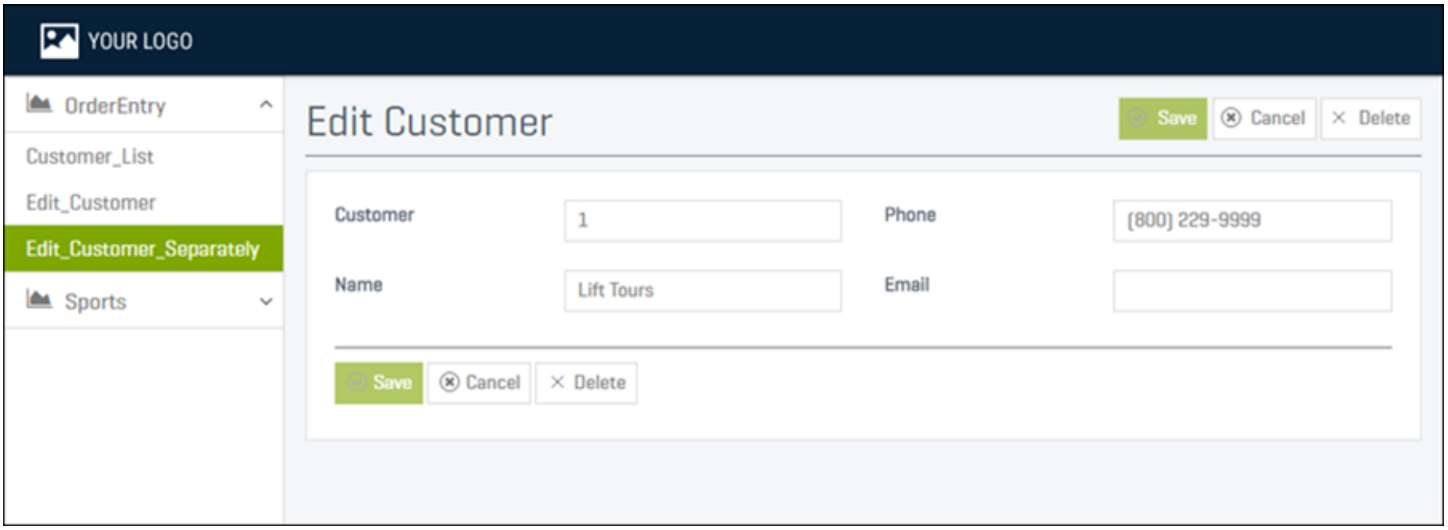
Note: Clicking **New** (above the grid) overlays the grid with an editable form for adding a new record to the bound data source, which is not shown.

Figure 25: 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 26: Data-Grid-Separate-Form view running in app with editable form overlaying the read-only form

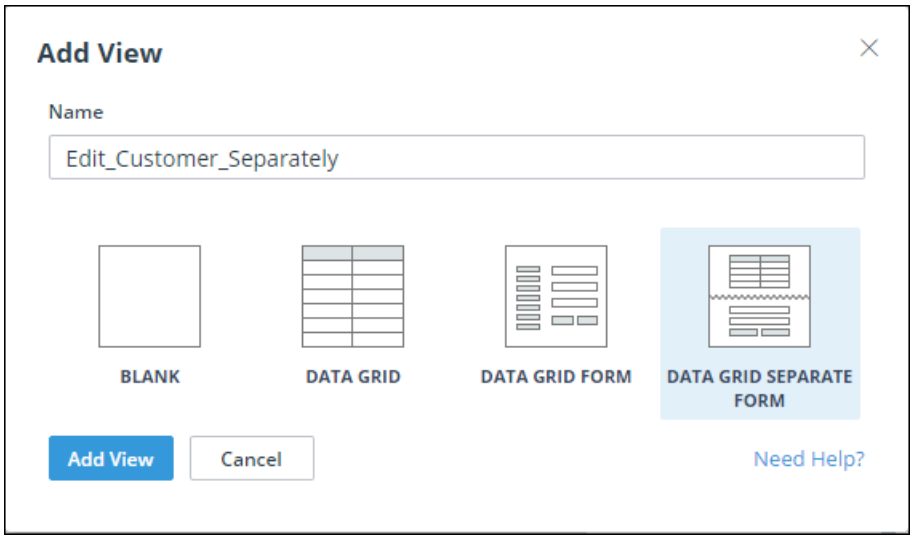


Each field in the editable form is displayed according to the editor type that has been defined for it in the bound data source (see Adding and editing a data source on page 22). From here, the edited record can be saved (by clicking **Save**), deleted (by clicking **Delete**), or the edit canceled (by clicking **Cancel**), all of which return to the read-only form displaying fields from an appropriate record.

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 return to the read-only grid with an appropriate row highlighted.

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

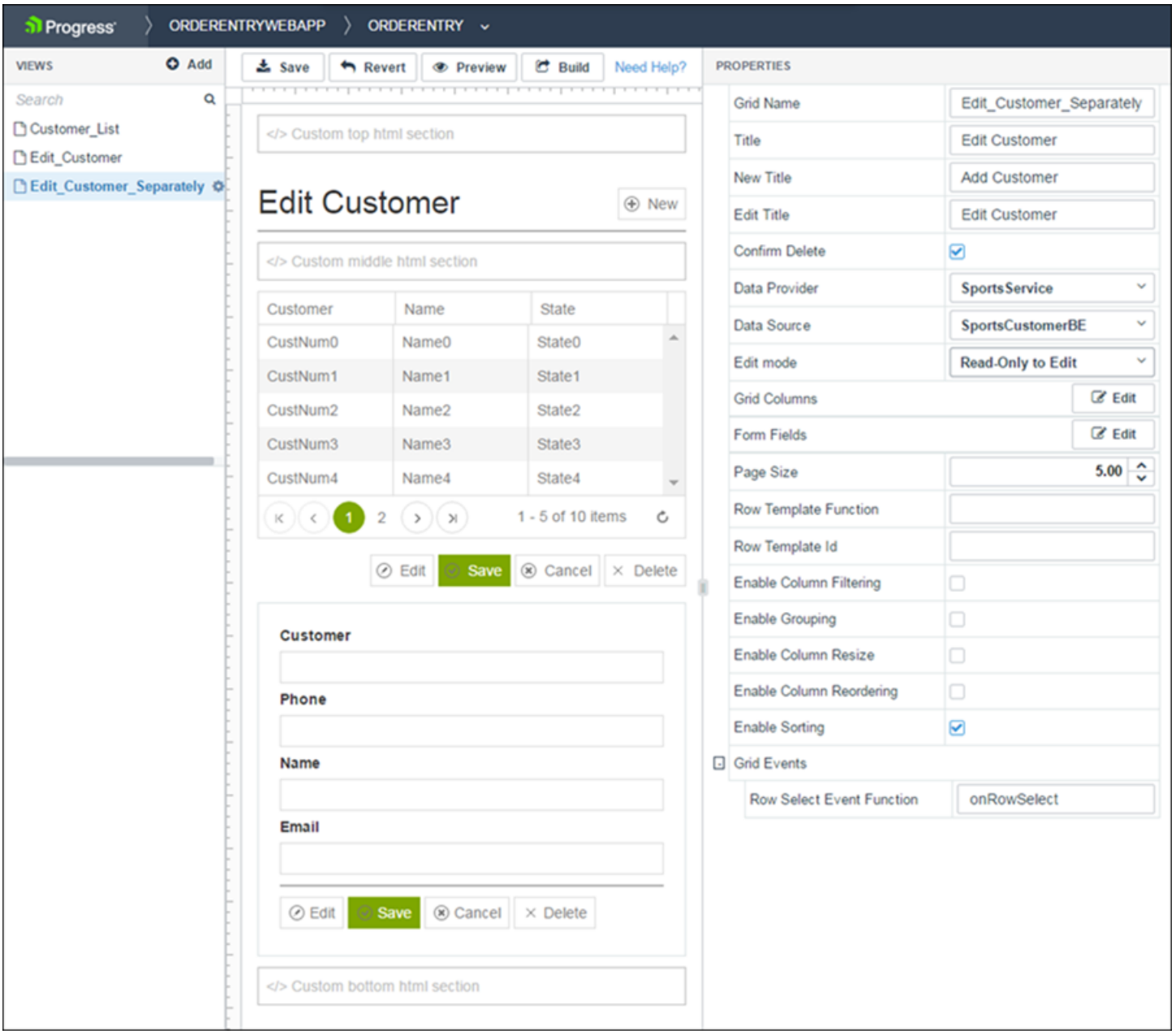
Figure 27: Add View dialog box creating a Data-Grid-Separate-Form view



In this example, the dialog box has `Edit_Customer_Separately` entered as the value of the view **Name** and **DATA-GRID-SEPARATE-FORM** selected as the view type. The view types listed in this dialog box (shown with a corresponding icon) identify the available view templates you can use to add a user-created view.

After specifying the name 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 28: 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**.

- **Preview** — Provides the option to either invoke the Kendo UI Generator to build and immediately preview a generated app in its currently saved state or to preview the most recent build (if one exists). The preview opens in a separate browser tab using live data from the mapped data sources.
- **Build** — Invokes the Kendo UI Generator to build a generated app in its current state without running a preview.
-  — 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.

For more information on app builds, see App generation and deployment on page 68.

- **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).

    Note: To fit on the document page with a readable size, the example view design panel is horizontally compressed so that the form fields appear out of order compared to their usual arrangement.

  - **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 80 in this document.

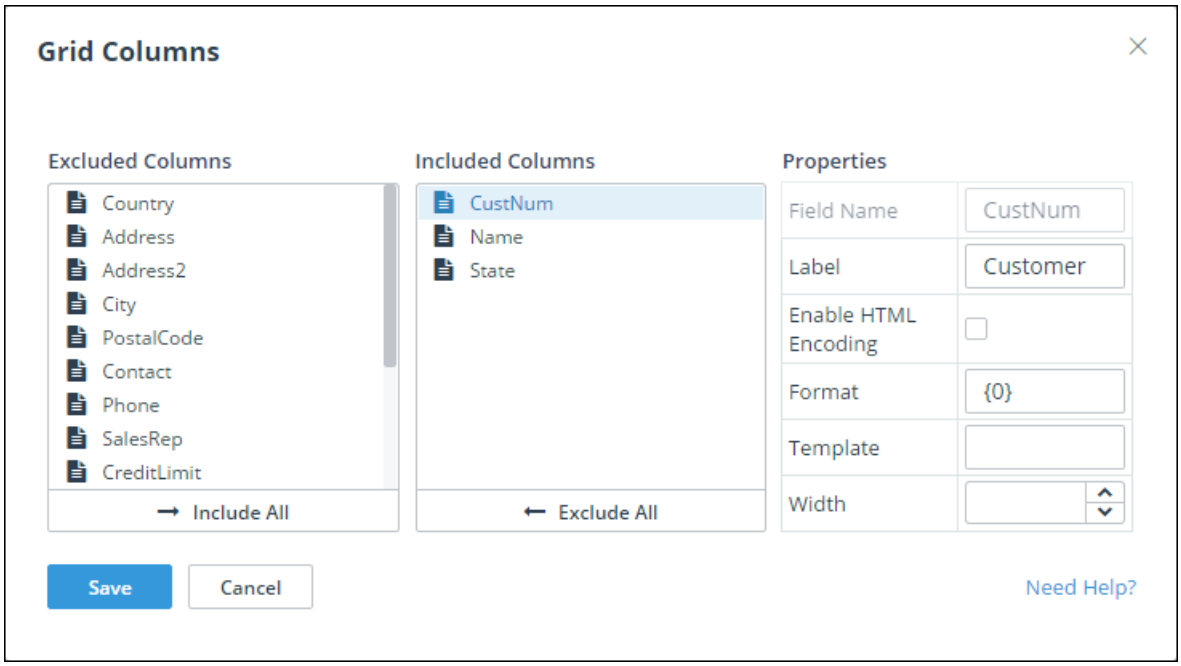
  The design simulation in a view design panel changes as you modify some of the view properties that affect its appearance, and even as you click around in the simulated view design panel, depending on these property settings. For example, the **Edit_Customer_Separately** view title is set to `Edit Customer` 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).

- **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 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 19.
  - **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 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, as shown in this example for the three data source fields specified as **Included Columns**, `CustNum`, `Name`, and `State`:

    Figure 29: Grid Columns dialog box

    The specified **Properties** apply to each field that you select in the **Included Columns** list, as shown for the `CustNum` field. The **Template** property 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 83 in this document.

  - **Form Fields** — Clicking **Edit** for this property opens a dialog box that allows you to specify what data source fields appear as fields on a form, in this example, `CustNum`, `Name`, `Phone`, and `EmailAddress`. This dialog box is similar to the **Grid Columns** dialog box with fewer **Properties** that affect how each field that you select in the **Included Fields** list is displayed in the form (**Label Text** and **Format** only).
  - **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 81 in this document.

- **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 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 22.

- **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.

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 76 and View-specific events on page 78 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 (rows and columns), data management (e.g., Grid), editor (e.g., Text Box), scheduling (e.g., Calendar), navigation (e.g., Toolbar), and custom HTML components. 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.

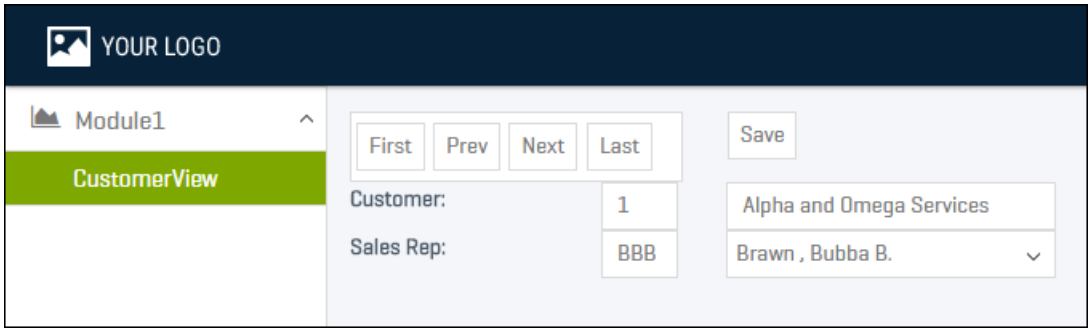
To define a view instance you have created, you drag-and-drop these two types of components onto the view design panel: initially, the *layout* components (rows and columns), then the *content* components (such as editable grid and field input components), which you can drop into layout columns. You 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 form of data navigation.

Also, unlike the Data-Grid* views, the Blank view allows you to specify multiple data sources 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 you must modify depending on the components and events.

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

Figure 30: Blank view running in app with toolbar and form



The Toolbar component is configured with four buttons for navigating the records of a `Customer` data source. This data source is in turn related to a `Salesrep` data source by a foreign key on the `SalesRep` field in both data sources. (**Note:** You must maintain such relationships in the event function code you write.)

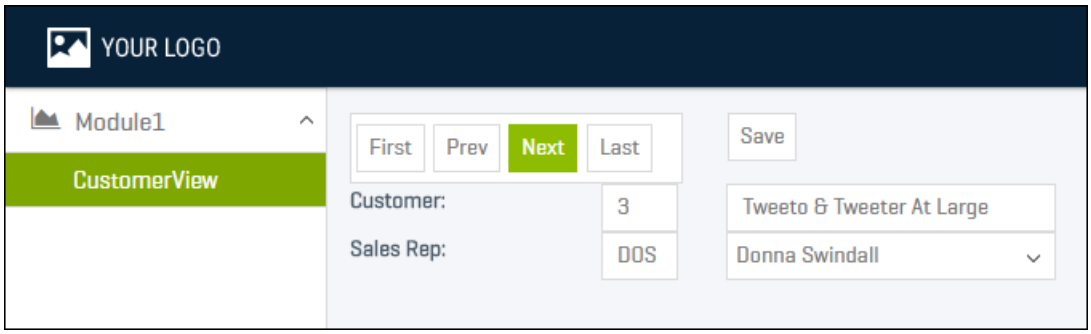
The **Customer:** and **Sales Rep:** labels are hard coded and displayed through Label components. The customer number, **1**, is a `CustNum` field value displayed from the `Customer` data source using a Disabled Text Box component, which displays data in a read-only mode. **Alpha and Omega Services** is the name of the customer uniquely identified by the customer number and is a `Name` field value displayed from the `Customer` 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, **BBB**, is the value displayed from the `SalesRep` field of the `Salesrep` data source using a Disabled Text Box. **Brawn, Bubba B.** is the name of the sales rep uniquely identified by the sales rep code and is displayed from the `RepName` field of the `Salesrep` data source using a Drop Down List component, which allows any other sales rep to be selected from the `Salesrep` data source to update its association with the current `Customer` record.

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

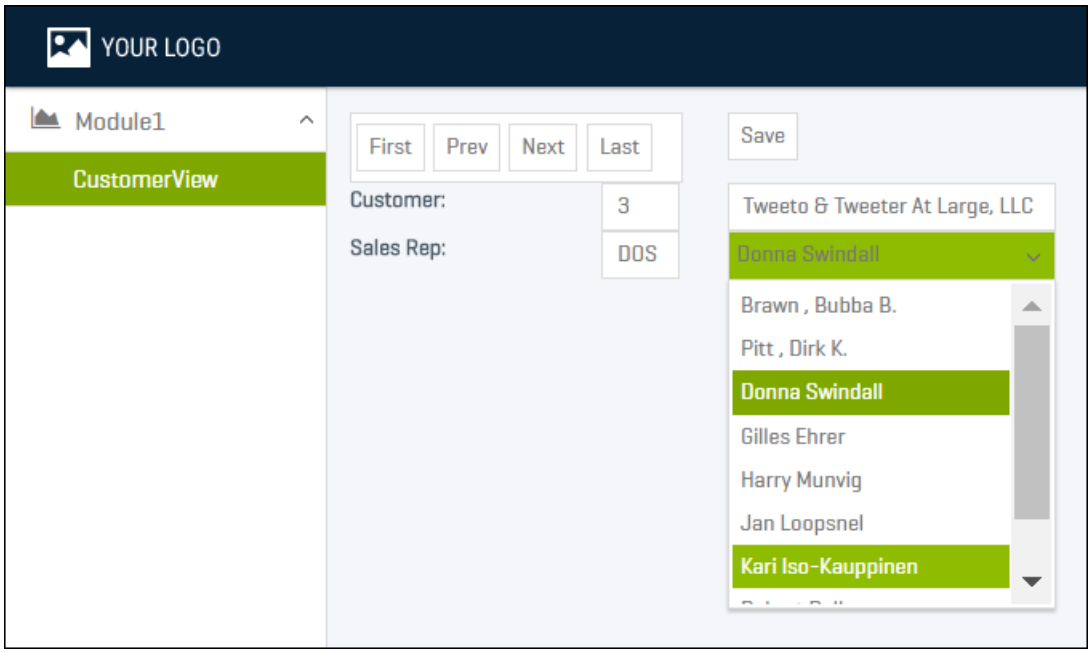
For example, clicking **Next** in the toolbar a couple of times displays field values from the following `Customer` and associated `Salesrep` records:

Figure 31: Blank view running in app clicking toolbar item



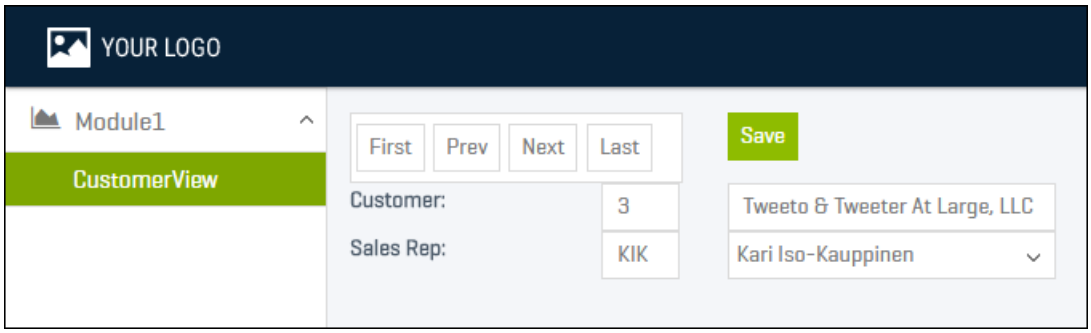
You can change the customer name (to **Tweeto & Tweeter At Large, LLC**) and the associated sales rep (to **Kari Iso-Kauppinen**), as shown:

Figure 32: Blank view running in app with change to form fields



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

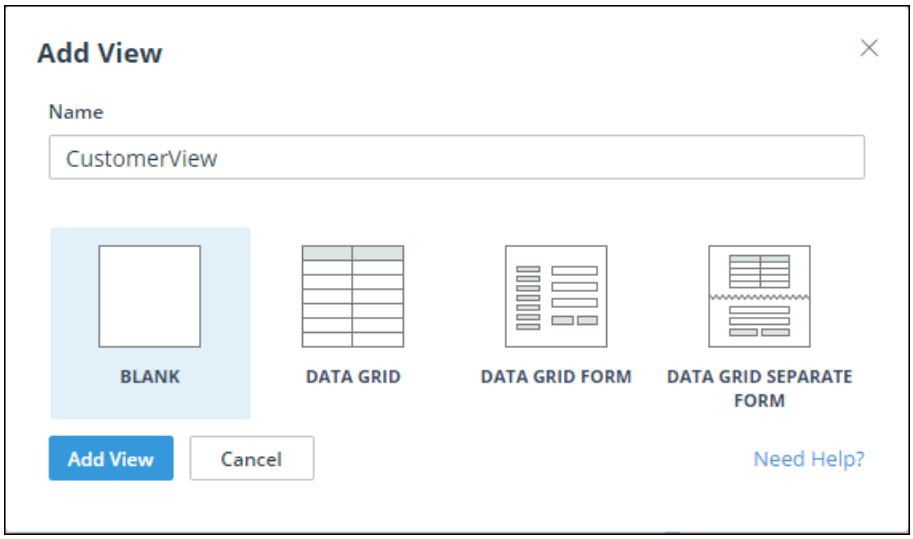
Figure 33: Blank view running in app with changes saved



To add a Blank view to a user-created module, edit the module, which opens a view design page in the module, then click **Add View** at the top of the **VIEWS** pane (see Figure 35: Blank view design page on page 58 for an example).

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

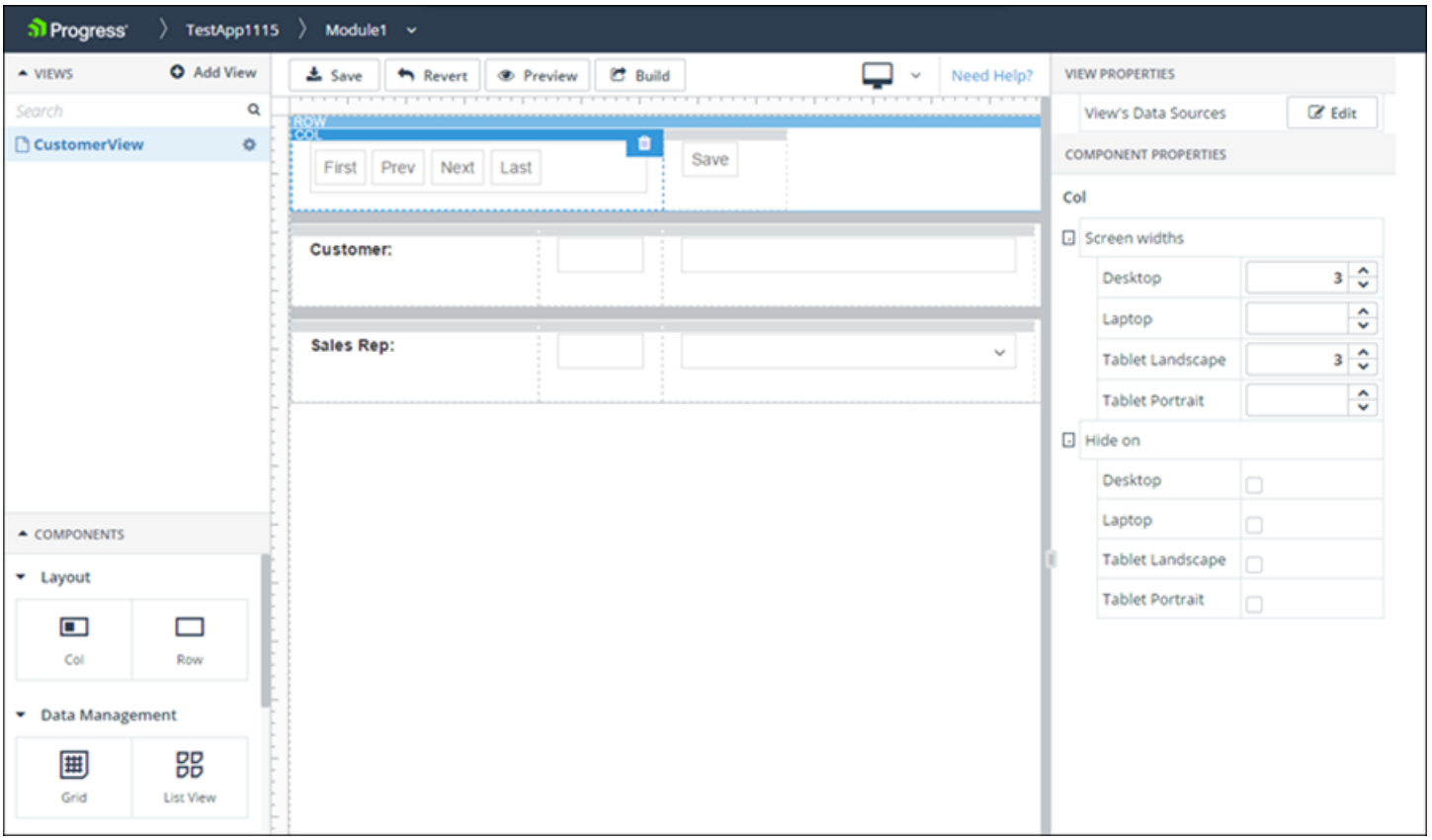
Figure 34: Add View dialog box creating a Blank view



In this example, the dialog box has `CustomerView` entered as the value of the view **Name** and **BLANK** (the default) selected as the view type. The view types listed in this dialog box (shown with a corresponding icon) identify the available view templates you can use to add a user-created view.

After specifying the name and view type, click **Add View** to create the specified view and display its view design page for editing, as shown for the **CustomerView** Blank view in this example:

Figure 35: 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**.
  - **Preview** — Provides the option to either invoke the Kendo UI Generator to build and immediately preview a generated app in its currently saved state or to preview the most recent build (if one exists). The preview opens in a separate browser tab using live data from the mapped data sources.
  - **Build** — Invokes the Kendo UI Generator to build a generated app in its current state without running a preview.
  -  — 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.

For more information on app builds, see App generation and deployment on page 68.

- **VIEWS pane (in panel on the left)** — Lists the views in the current module, including the example **CustomerView** 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.
  - **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**).
  - **Custom** — Content components that support customizable content (for example, **Custom Html**, which allows you to enter a single `<div>` element).

  This section provides more information on working with layout and content components based on this **CustomerView** 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. 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. In this way, you can have layout columns that contain rows, each of which contain their own columns, and each 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.

  A layout column represents one or more horizontal *slots* of screen space, whose size depends generally on the device and orientation. You 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 **CustomerView** 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 `Customer` data source navigation in the first column and a separate **Button** (with the text **Save**) in the second column.

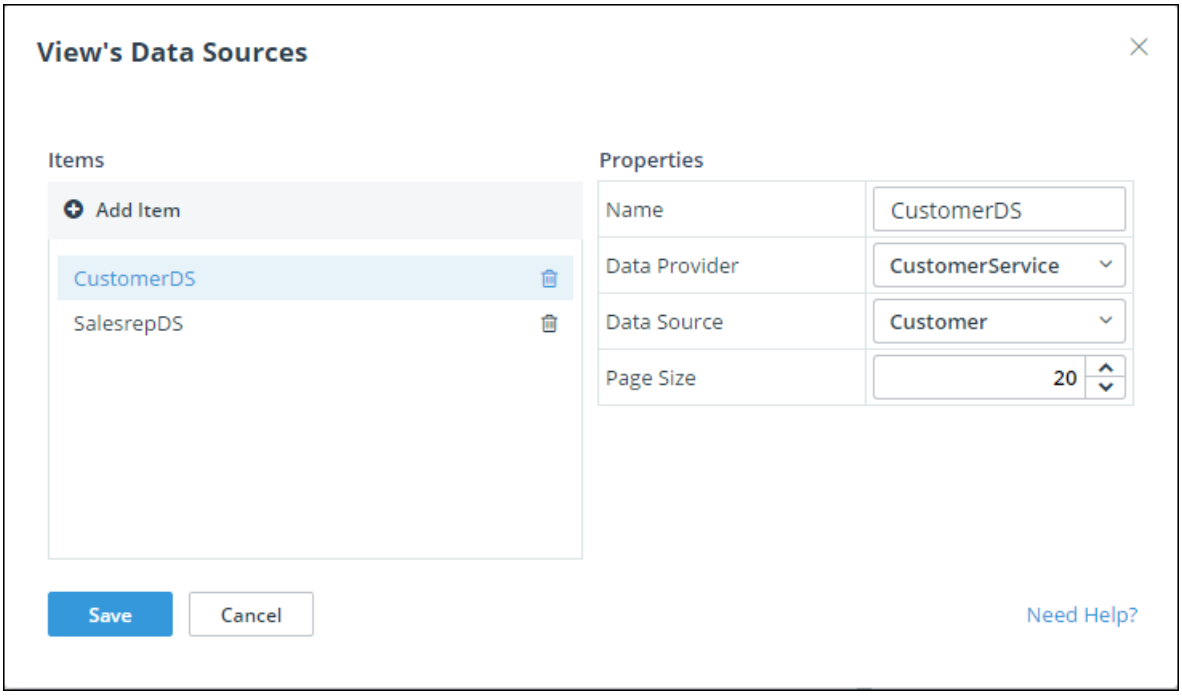
- **In the second row** — A **Label** with the text **Customer:** in the first column, a **Disabled Text Box** in the second column for displaying the read-only value of the `CustNum` field from the current record in the `Customer` data source, and a **Text Box** in the third column for displaying and editing the value of the Name field in the current record of the `Customer` 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 `Customer` data source (and foreign key to the `Salesrep` data source), and a **Drop Down List** in the third column for displaying the `RepName` field from the corresponding record of the `Salesrep` data source and for selecting a new `Salesrep` record (based on its `RepName` field value) in order to change the sales rep associated the current `Customer` record by changing its `SalesRep` field value to the value in the selected `Salesrep` 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 80 in this document.

- **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 **CustomerView** example, as follows:

Figure 36: 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 19). (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**, the names of which 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 **CustomerView** 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 76 in this document. An example of custom coding for the general `onShow` event function appears in the sample Table 4: view-factory.js file for CustomerView on page 65, 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 **CustomerView** view, for example (see Figure 35: Blank view design page on page 58), the first **Col** component is selected in the first row, with its two groups of properties shown as follows:

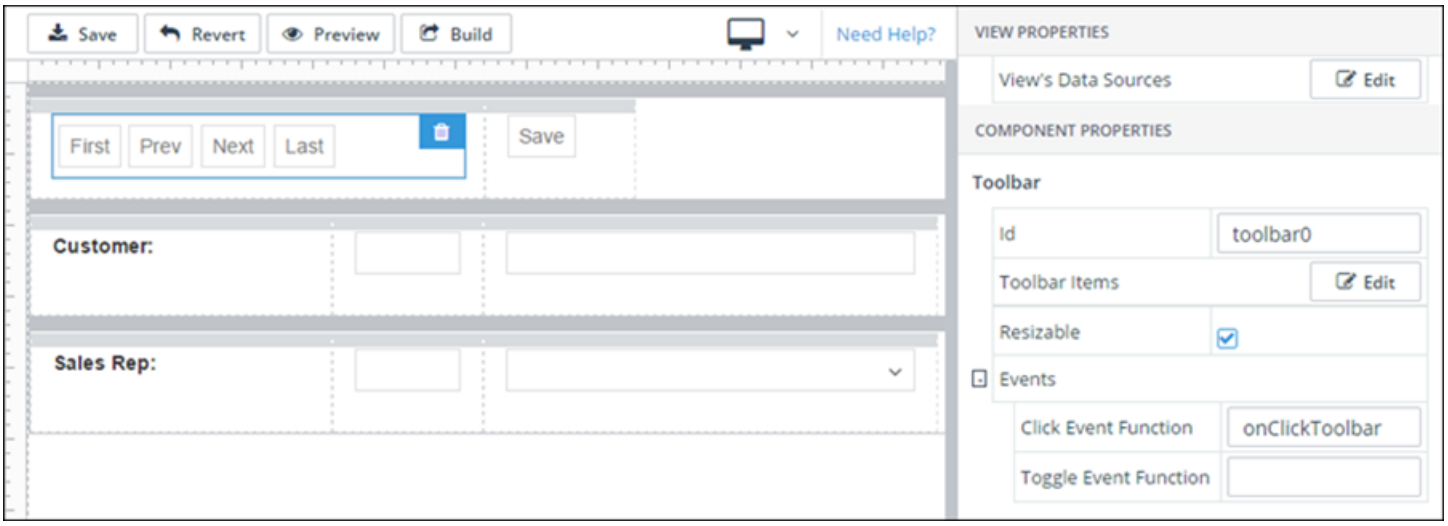
  - **Screen widths** — Properties that you can set to specify the number of slots of horizontal screen space for a given device and orientation. A setting for the **Tablet Landscape** property is required, because this value is used by default for any other device and orientation properties you do not 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 **CustomerView** 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 **CustomerView** view design panel showing the Toolbar selected with its property settings displayed:

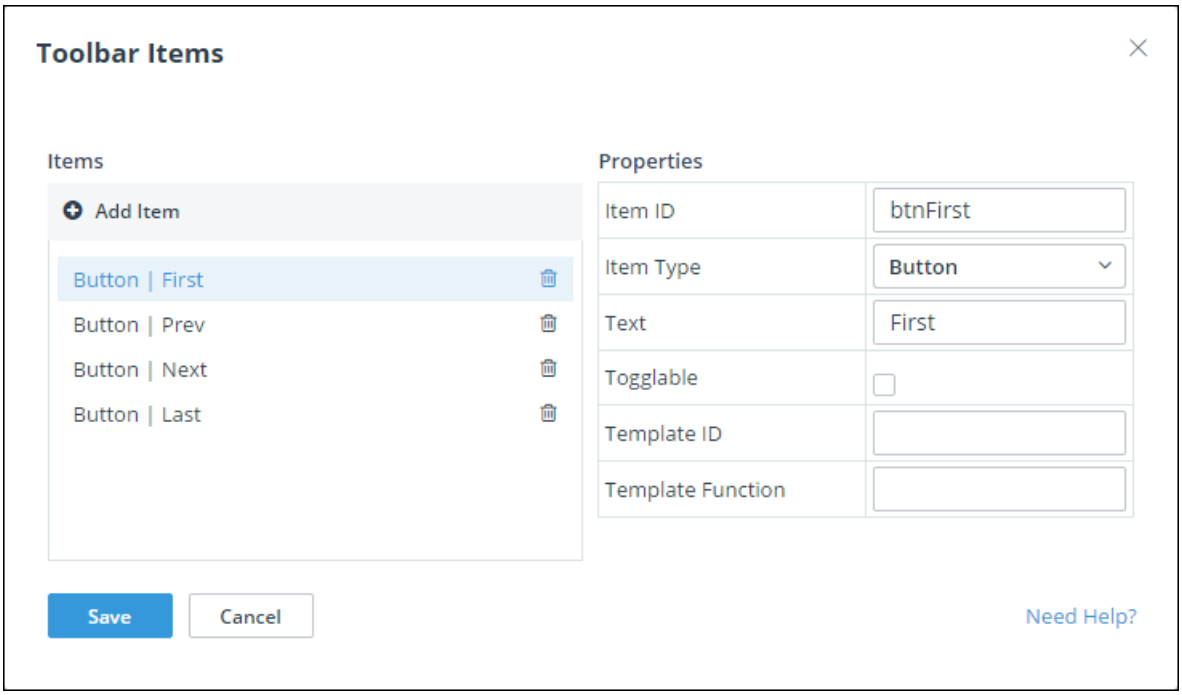
Figure 37: Editing Blank view navigation using a toolbar



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 38: Adding and editing Blank view toolbar items



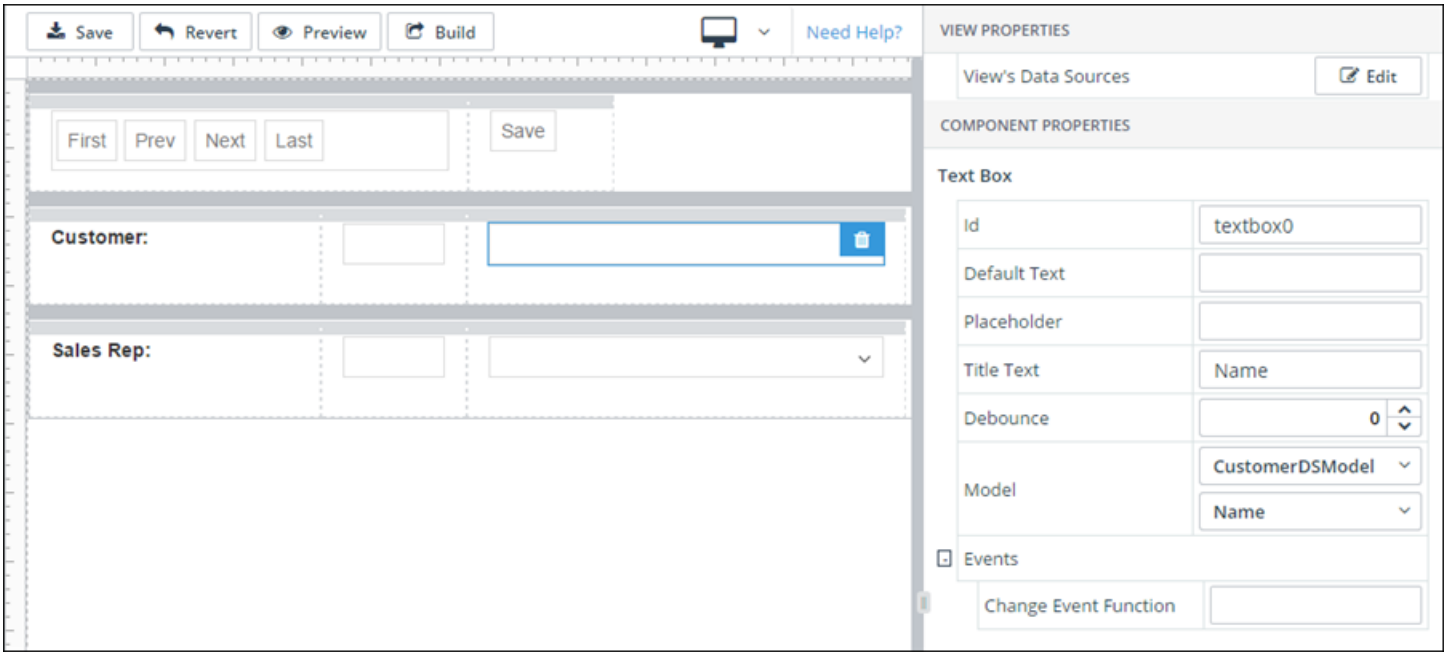
The settings for this Toolbar instance include properties to add or edit the four buttons, which are displayed with the label text **First**, **Prev**, **Next**, and **Last**. You can add an item by clicking **+ Add Item** or delete an item by clicking its trash icon, use the **Properties** that appear for each item to complete its definition, then click **Save** to save the latest definitions. Basically, you set the **Properties** for each item depending on the **Item Type** you have selected for it. For example, the selected button has its **Item ID** with the value to access in code set to `btnFirst`, its **Item Type** set to `Button` (with `Separator` or `Template` as the other options), and its **Text** property set to `First`.

- **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.

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 **CustomerView** view design panel showing the selected Text Box with its property settings displayed:

Figure 39: Editing Blank view data binding for a component using a model



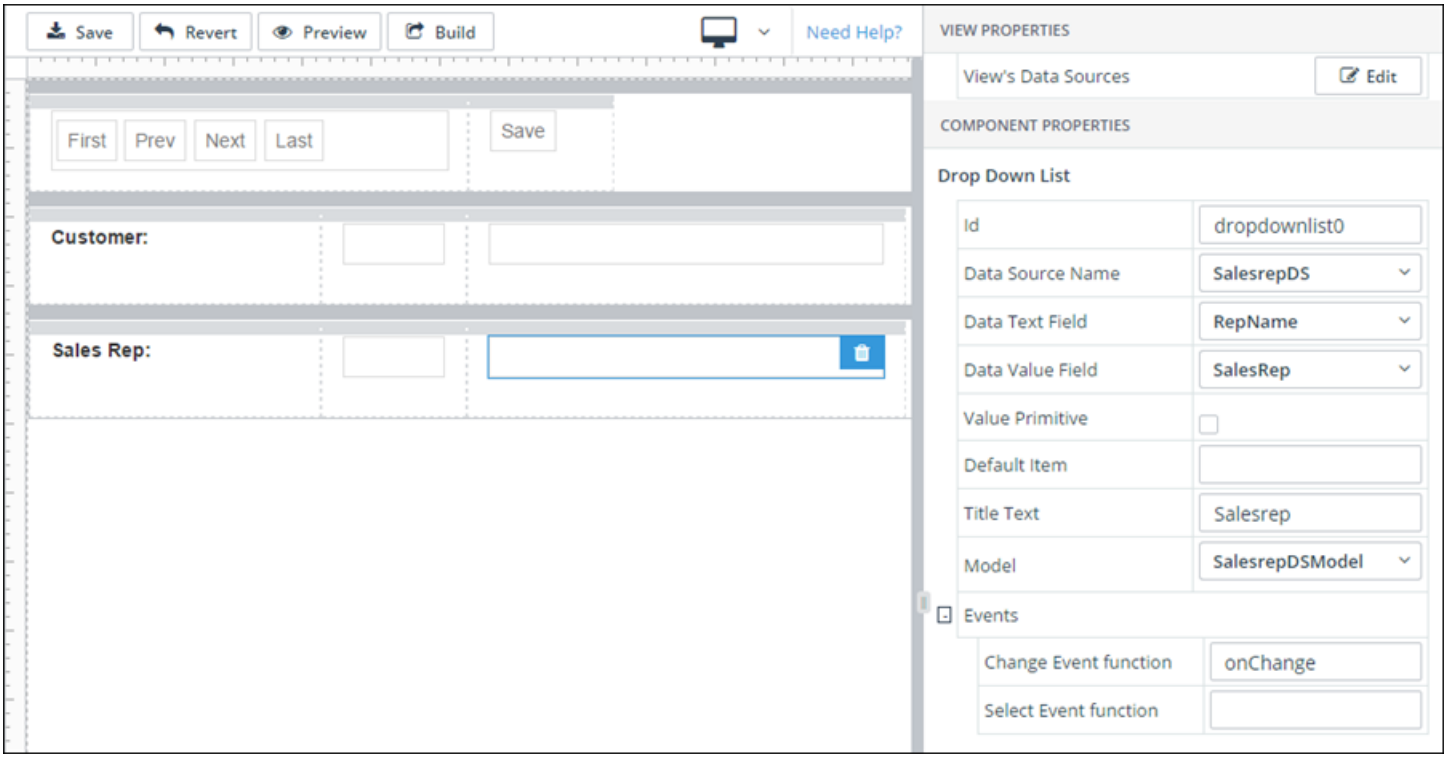
These settings include:

- **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 **CustomerView** 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 **CustomerView** view design panel showing the selected Drop Down List with its property settings displayed:

Figure 40: 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.
- **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 `view-factory.js` file where you can code the event functions for this view, with property settings from the **CustomerView** example shown in bold:

Table 4: view-factory.js file for CustomerView

```
/* global angular */
(function(angular) {

    angular
        .module('viewFactories')
        .factory('module1CustomerView', ['$q', 'dsService', function($q, dsService) {

            function Module1CustomerView() {
                this.scope = null;
            }

            Module1CustomerView.prototype = {
                /*  The resolve method could return arbitrary data,
                    which will be available in the "viewShowHandler"
                    and "viewHideHandler" handler as the customData argument */
                onInit: function($stateParams) {
                    return $q(function(resolve, reject) {
                        resolve({});
                    });
                },
                /* "customData" is the data return by the viewInitHandler handler*/
                onShow: function($scope, customData) {
                    this.scope = $scope;
                    var that = this;
                    this.customerDS = $scope._$ds.CustomerDS;
                    this.salesrepDS = $scope._$ds.SalesrepDS;
                    this.customerModel = $scope._$viewModels.CustomerDSModel;
                    this.salesrepModel = $scope._$viewModels.SalesrepDSModel;

                    this.customerIdx = 0;

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

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

                    });
                },
                /* "customData" is the data return by the viewInitHandler handler*/
                onHide: function(customData) {

                },

                onClickToolbar: function(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;
            }
        },
        displayCustomer: function(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();
        },
        displaySalesrep: function(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: function(e) {
```

```
                    this.customerModel.SalesRep = angular.element("#dropdownlist0"
                        ).data("kendoDropDownList").dataItem().SalesRep;
                },
                onSaveClick: function(e) {
                    var dataItem = this.customerDS.get(this.customerModel.id);

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

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

            return new Module1CustomerView();
        }]);

})(angular);
```

For more information on `view-factory.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 71.

The Kendo UI Builder also supports a `dsService` API that is referenced but not used in the sample `view-factory.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`. These properties live in `$scope._$['ds']` and `$scope._$['viewModels']` of the view, where `ds` is the name of its corresponding data source.

If you set the **Model** property of a data bound component, that model is 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 `view-factory.js` file in the generated application. The `view-factory.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 `view-factory.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) {
}
```

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

Table 6: dsService API examples

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

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

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

createClick: function () {
    this.scope._$viewModels['dsModel']
        = dsService.create(this.scope._$ds['ds'], this.scope._$viewModels['dsModel']);
},

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

You can also find a link to sample web apps posted on the associated Progress Community page that are built using the Kendo UI Builder and use this `dsService` API.

App generation and deployment

When you build or build and preview an app in the Kendo UI Designer, the Kendo UI Generator generates depoyable 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 71).

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

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

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

You can then use a Web UI project in Progress Developer Studio for OpenEdge that you have created with the same name as your web app, and that shares the same `build-output` folder 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.

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:

- Customize the view templates
- Custom semantic types
- Custom UI editor types
- 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:\OpenEdge\WRK\11.6\workspace\OrderEntryWebApp` for the web app named `OrderEntryWebApp`.
- *KUIBInstalDir* — 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
- Customize the view templates
- Custom semantic types
- Custom UI editor types
- 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 assets on page 73
- HTML code on page 73
- JavaScript code on page 73

Custom assets

Every Web app has a folder for static assets to be included with the app. by default these are populated with the default fonts, images, and styles. 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\src\assets\fonts
webAppDir\src\assets\images
webAppDir\src\assets\styles
```

HTML code

Every web app has a folder for custom HTML sections that is automatically populated with corresponding files for each view created in your web app. These custom section files are created for each view as follows:

```
webAppDir\src\html\ModuleName-ViewName\customSection.html
```

Where:

ModuleName

Is the name of the module in which the view is created.

ViewName

Is the name of the view where the custom HTML section appears.

customSection

Is `topSection`, `middleSection` or `bottomSection`, as defined for each view type.

JavaScript code

Every web app has a folder for custom event handlers that is automatically populated for each view created in your web app. These custom event handlers are all created in the following JavaScript file for every view:

```
webAppDir\src\scripts\ModuleName-ViewName\view-factory.js
```

Where:

ModuleName

Is the name of the module in which the view is created.

ViewName

Is the name of a view where the JavaScript file is created.

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\src\assets\images
```

Customize the view templates

View templates can be customized for your company.

There are several files that make up the template and all of these files must be edited as a group.

Caution: View templates must be edited with extreme care. Always make a backup copy before editing any of these files.

You can find the files for each built-in view template at the following locations:

- Data-Grid view:

```
KUIBInstalDir\schemas\components\data-grid.json
KUIBInstalDir\generator-starnova\generators\app\resources\components\data-grid\*
```

- Data-Grid-Form view:

```
KUIBInstalDir\schemas\components\data-grid-form.json
KUIBInstalDir\generator-starnova\generators\app\resources\components\data-grid-form\*
```

- Data-Grid-Separate-Form view:

```
KUIBInstalDir\schemas\components\data-grid-separate-form.json
KUIBInstalDir\generator-starnova\generators\app\resources\components\data-grid-separate-form\*
```

- Landing-Page view:

```
KUIBInstalDir\schemas\system-components\landing-page.json
KUIBInstalDir\generator-starnova\generators\app\resources\components\landing-page\*
```

- Login view:

```
KUIBInstalDir\schemas\system-components\login.json
KUIBInstalDir\generator-starnova\generators\app\resources\components\login\*
```

Custom semantic types

You can create your own custom semantic types in OpenEdge and associate them with the editor types that are available in the Kendo UI Builder.

For details, see the following topics:

- OpenEdge Data Object Services on page 75
- Kendo UI Builder on page 75

OpenEdge Data Object Services

Semantic types can be added to temp-table schema using source code annotations. A set of built-in semantic type annotations are supported by Progress Developer Studio for OpenEdge and the definitions for them can be found in the following file:

```
OpenEdgeInstalDir\oeide\eclipse\plugins\com.openedge.pdt.text_OEversion\annotation\AnnotationConfig.json
```

Where:

OpenEdgeInstalDir

Is your OpenEdge installation directory, for example, `C:\Progress\OpenEdge`.

OEversion

Is your OpenEdge version number, for example, `11.6.3.00`.

You can add your own semantic type annotations to this file.

Caution: Always make a backup copy of this file before editing it.

Kendo UI Builder

For your new semantic types defined for OpenEdge, you need to map one or more UI editor types to the semantic type. You can do this in the following file:

Caution: Always make a backup copy of this file before editing it.

```
KUIBInstalDir\semantic-types-map.json
```

Custom UI editor types

Editor types define a UI component and general properties for that type of UI component, independent of an actual rendering, such as using Kendo UI. The actual rendering of the component is done by a component template.

You can add your own editor types, or edit the built-in editor types, by creating new or editing existing instances of these files:

Caution: Always make a backup copy of any existing file before editing it.

```
KUIBInstalDir\schemas\components\editorType.json
```

Where *editorType* is the name of the editor type you want to create or edit in the Kendo UI Builder.

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 must be defined in the `view-factory.js` file for the view. This file can be found for each view at the following location, and initially contains empty function definitions for these events with default names:

```
webAppDir\src\scripts\ModuleName-ViewName\view-factory.js
```

For more information on this file and its location, see JavaScript code on page 73.

The following table describes the supported general view events and the default names of the event functions that run for them when they fire.

Table 7: Default general view event functions

View event	Default function	Description
`hide`	`onHide`	This event fires when the view is hidden. You can find more information at http://docs.telerik.com/kendo-ui/api/javascript/view#events-hide.

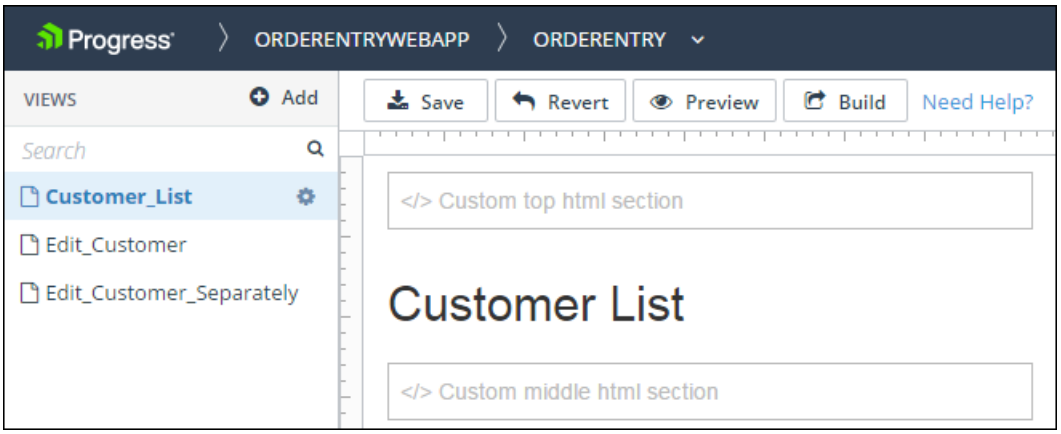
View event	Default function	Description
`init`	`onInit`	This event fires when the view is initialized. You can find more information at http://docs.telerik.com/kendo-ui/api/javascript/view#events-init.
`show`	`onShow`	This event fires when the view is made visible. You can find more information at http://docs.telerik.com/kendo-ui/api/javascript/view#events-show.

You can change the default names of these functions in the `view-factory.js` 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 `view-factory.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 general view events, these function names are specified in the Edit View dialog box, which you can access in the Designer as follows:

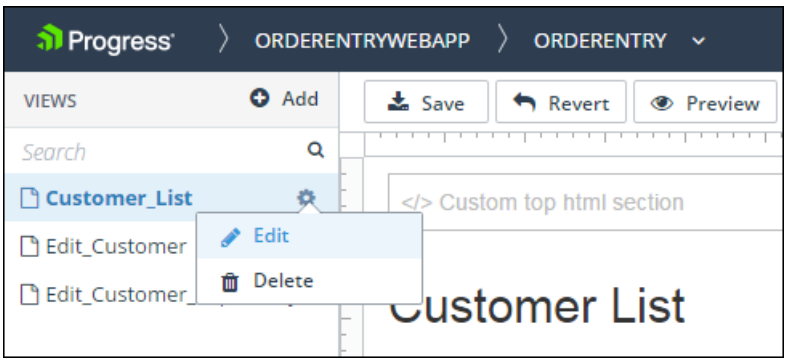
1. Edit the view by first editing its module in the app design page (see Creating and designing an app on page 14). This displays the view design page for the first view in the module and lists all its remaining views (for a Data-Grid view, see Adding and editing a Data-Grid view on page 35).
2. Select 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 41: Editing general view events



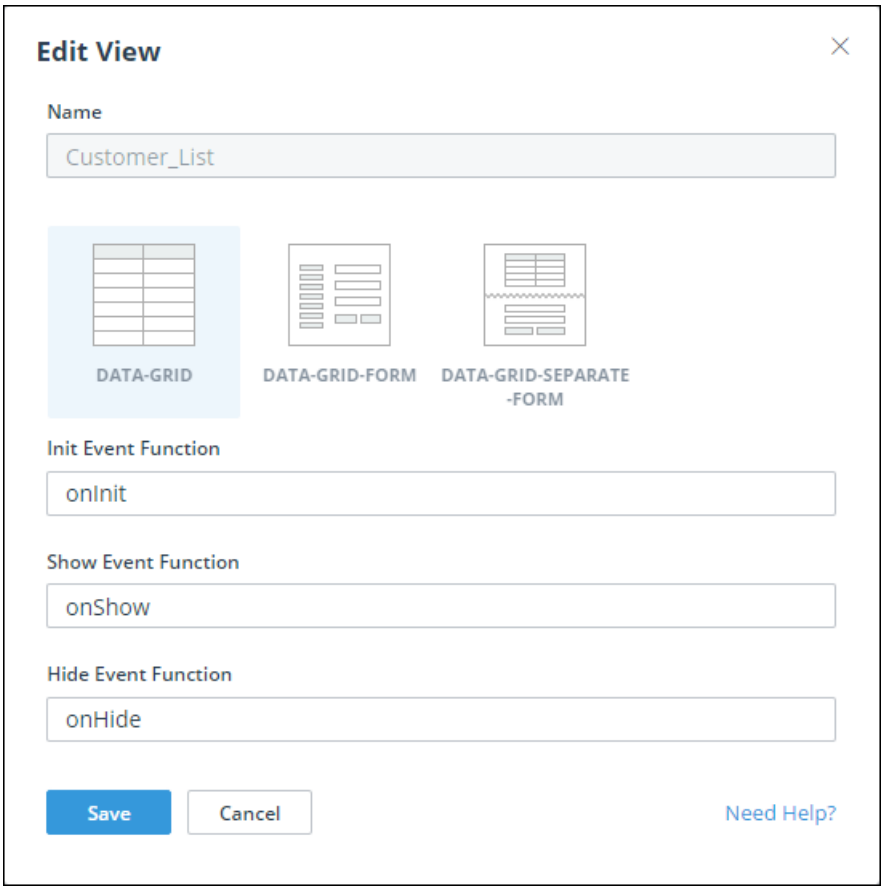
3. Click the gear control to display a drop-down menu, then click **Edit**:

   Figure 42: 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 43: 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 `view-factory.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\src\scripts\ModuleName-ViewName\view-factory.js
```

For more information on this file and its location, see JavaScript code on page 73.

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

Table 8: Default view-specific event functions

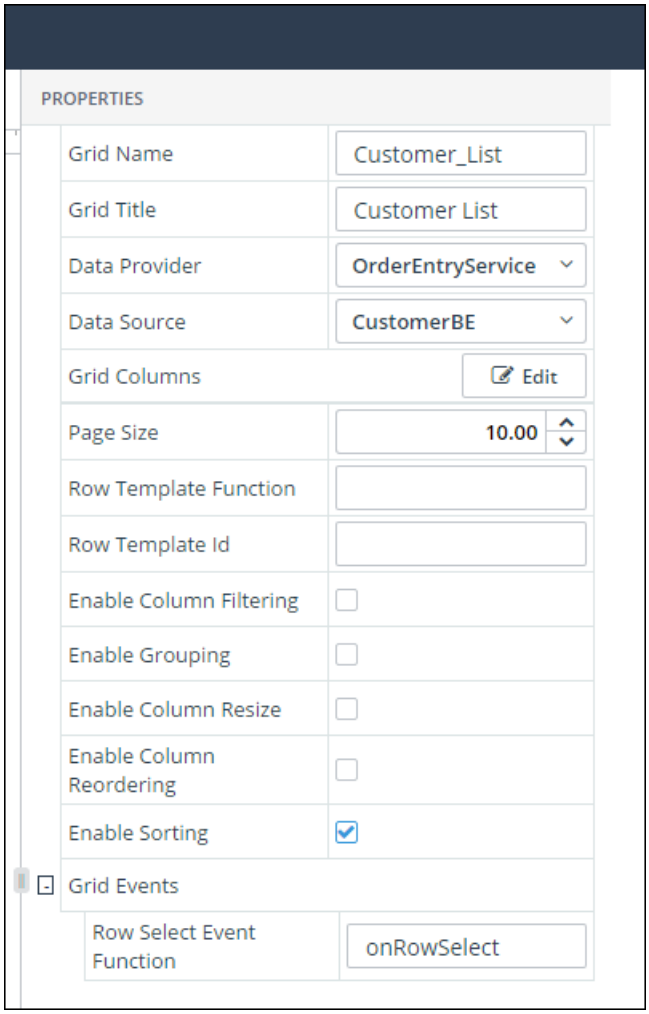
View	Event	Default function	Description
Login	`Login`	`onLogin`	This event fires when the user clicks **Login** on the login view.
Data-Grid Data-Grid-Form Data-Grid-Separate-Form	`Row Select`	`onRowSelect`	This event fires when the selected row changes in the grid.

You can change the default names of these functions in the `view-factory.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 `view-factory.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 14. For information on editing views, see Modules and views on page 30.

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 35):

Figure 44: 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\src\html\ModuleName-ViewName\customSection.html
```

For more information on the names and locations of these custom section files, see HTML code on page 73.

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

Table 9: Custom sections and corresponding section files

Custom section	Corresponding file
Top	`topSection.html`
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 81 and Column templates on page 83.

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 defined in an HTML file.
- **Row Template Function** — Specifies the name of a function in the grid view's `view-factory.js` file.

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

For details, see the following topics:

- Row template format on page 81
- Row template ID on page 82
- Row template function on page 82

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

Row template ID

The **Row Template ID** property is 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 80).

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** property allows 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 `view-factory.js` file for a given grid view found at the following location:

```
webAppDir\src\scripts\ModuleName-ViewName
```

For more information, see JavaScript code on page 73.

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 `view-factory.js` file:

```
rowTemplate: function(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 81).

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 30.

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.
