Ext Designer for Ext JS 3

User's Guide

Table of Contents

Introduction

Sencha Ext Designer is a graphical user interface builder for Ext JS Web applications. The easy-to-use drag-and-drop Designer environment enables fast prototyping of application interface components, connecting interface components to data, and exporting well-formed, object-oriented code for each component.

Programmers and non-programmers alike can use Designer to collaborate on an application's design, which helps get projects started faster and enables faster iteration.

With Designer, you can, for example:

» Quickly and easily build complex forms.

» Change component layouts and swap control types with the click of a button.

» Focus on writing implementation code, rather than boilerplate user interface (UI) code.

Projects developed using Designer 1.2 can output code using either Ext JS version 3 or Ext JS version 4. This document covers projects that target Ext JS version 3.

This user guide is organized into the following chapters:

Chapter 1: Getting Started

A quick introduction to the basics of using Designer to build an interface, including an example exercise in which you build an application UI.

Chapter 2: Working With Layouts

How to set up and change basic container layouts you use to present content and data within an application.

Chapter 3: Designer Component Overview

An introduction to all the standard Ext JS components that can be selected and configured with Designer.

Chapter 4: Forms, Menus, and Trees in Designer

Using Designer to build common UI elements with Ext JS components.

Chapter 5: Component-Oriented Design

Advanced information about working with components in designer, including creating custom components and breaking an application into smaller parts that can be developed and maintained separately.

Chapter 6: Working with Data Stores

Setting up client-side data stores and displaying the data from them in an application.

Chapter 1: Getting Started with Designer

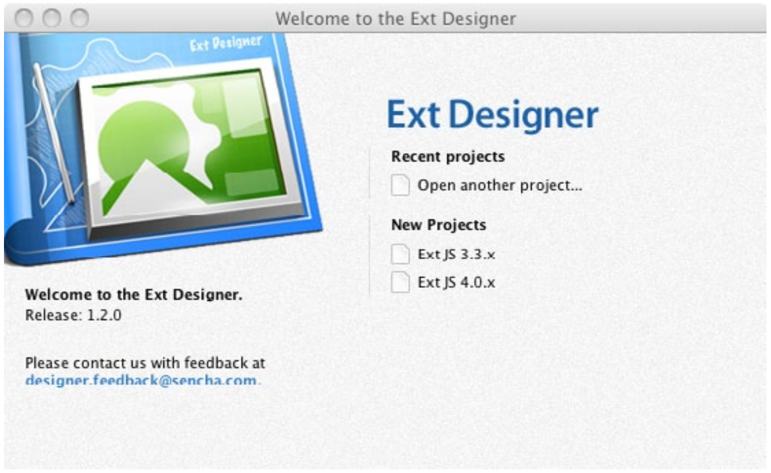
Designer can be used in conjunction with existing development environments and tools, but it's not a replacement for IDEs or text editors. The code generated by Designer can be edited with any common IDE or editor.

When using Designer, follow this basic workflow, iterating through the process as many times as needed to create a satisfactory UI:

1. Lay out UI components on the Designer canvas.
2. Configure the components.
3. Connect to data stores.
4. Export the project code targeting Ext JS 3.x framework. Designer generates two files, a base class (.base.js) and an implementation file (.js).
5. Implement event handling and custom methods in the generated .js file.

Navigating Designer

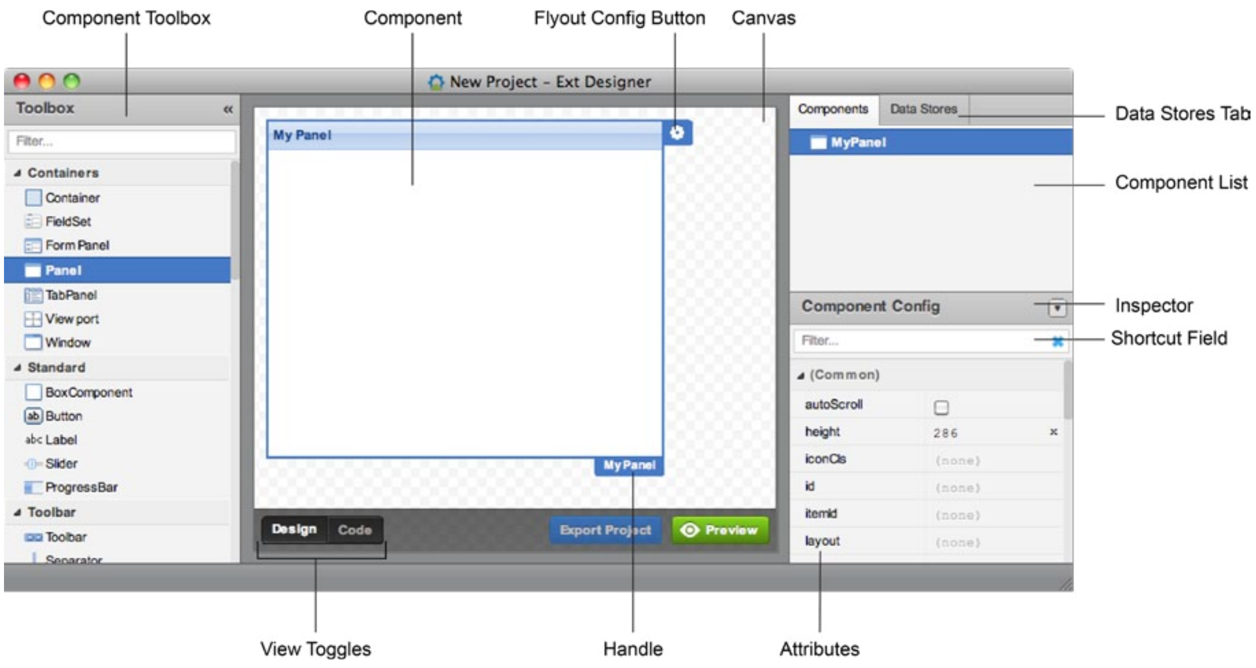
Launching Designer automatically displays a screen that asks if you want to open an existing project, or open a new project using either Ext JS 3.3.x or Ext JS 4.0.x, as shown here:



Welcome to the Ext Designer.
Release: 1.2.0

Please contact us with feedback at designer.feedback@sencha.com.

For the purposes of this guide, click Ext JS 3.3.x. Designer will open a blank canvas.



Here are the main elements of Designer:

Toolbox

Lists the components to build a UI. These correspond to standard Ext JS classes. For more information about each class, see the Ext JS 3 API Reference. You can drag and drop components from the Toolbox onto the canvas. The Toolbox title shows the version of Ext JS targeted by the project.

Canvas

A design space for assembling your UI. Add components, then resize and rearrange them and edit component titles and labels. (For more information see “Laying Out UI Components with Designer,” below.)

Components tab

Shows components added to the current project. From the Components tab, select, rearrange, duplicate, transform, and delete components in the canvas.

Component Config inspector

Use this to view and modify a selected component’s settings and work with Data Stores

Data Stores tab

Shows data sources added to a project. From here, add new JSON, Array, XML, and Direct data sources, add and remove a source’s data fields, and select, duplicate, or delete existing sources. View and modify a selected Data Store settings in the Component Config pane.

As you add components to the canvas, you can see them in a web browser by clicking the Preview button below the canvas. View the generated Javascript code by toggling between the design and code views. Save the code to an external file by clicking the Export button.

(Be sure to save your new project before exporting it.)

Shortcuts

Designer provides a number of navigation and configuration shortcuts, as follows:

» Double-click components in the Component Toolbox to add them to the canvas.

» Tab between in-line editable fields on the canvas.

» Locate particular attributes in the Component Config inspector by typing their name in the filter field.

» Set attribute values using Quickset: using the filter field in Component Config, typing the name of the attribute followed by a colon and the value you want to set. For example, *title: 'Car Listings'*. note: for strings you would wrap in single quotes.

Anatomy of a UI created with Designer

Designer enables flexible assembly of web page elements, easy reuse of components, and simplified maintenance of your UI. When laying out UI components with Designer, you drag a container such as a Window or FormPanel onto the canvas and add components to the container. By adding additional top-level containers to a project, you can lay out the different parts of the UI as separate entities. When Designer exports a project, each top-level container is represented by a class with the code for that class in a separate file.

Laying Out UI Components

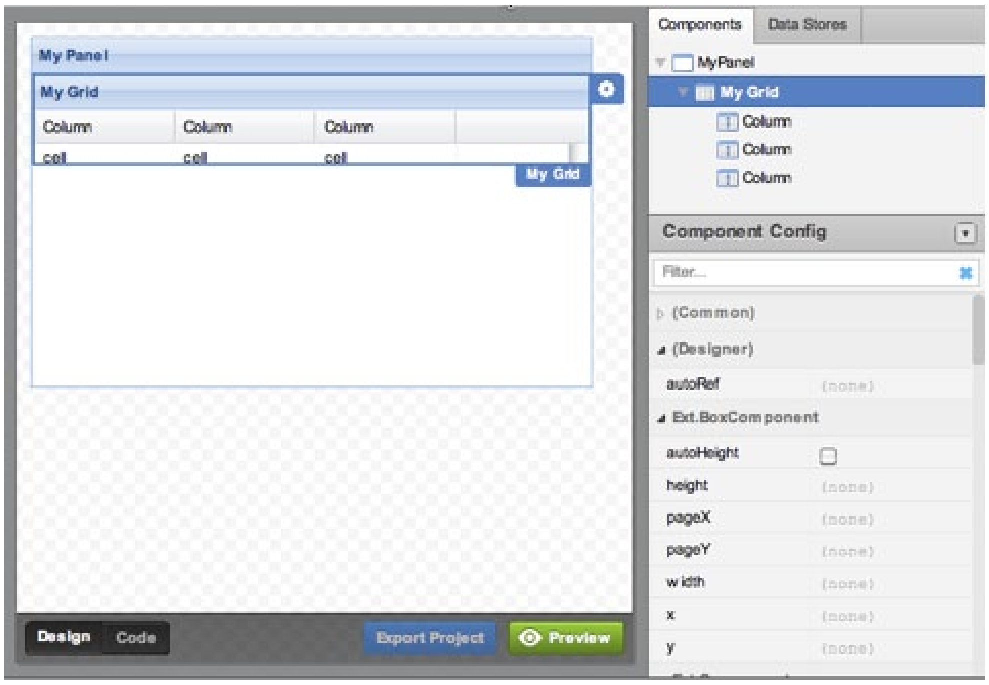
Designer leverages the powerful layout capabilities of Ext JS to simplify the creation of complex forms and make it easy to switch between alternate layout options.

Adding Components

To start assembling the application UI, drag components from the Toolbox onto the canvas. Designer ensures that components nest properly, and prevents the addition of incompatible components to a container. For example, Window or Viewport components can't be dropped into a Container, and Designer will display an icon showing this can't be done if you try to do this.

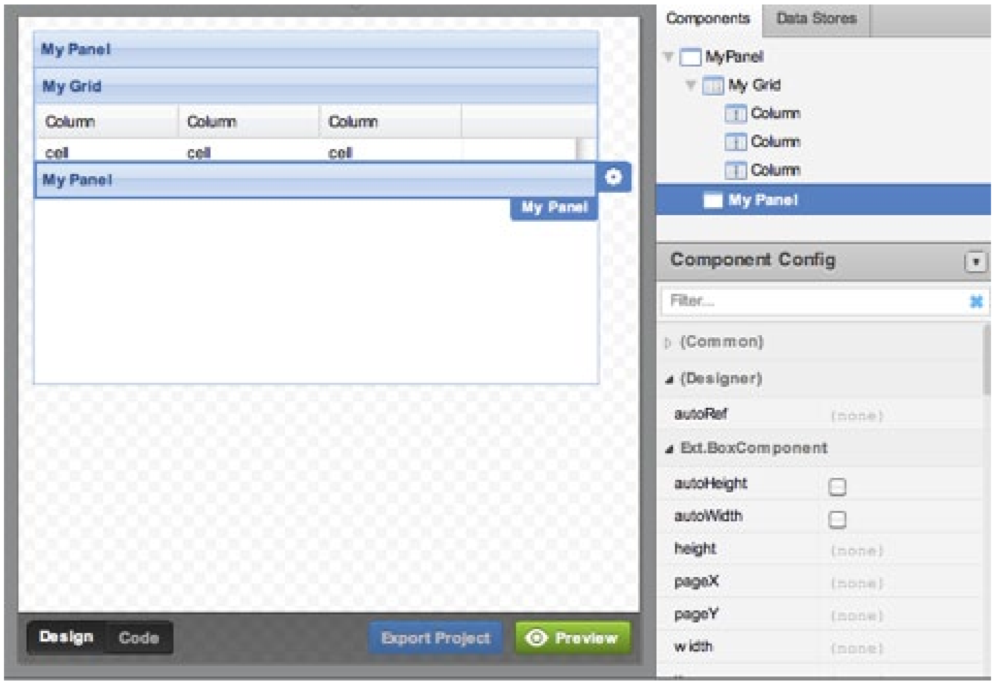
The following steps show how to assemble an application UI. (The examples are all drawn from the Designer demo application, Car Listings. You can see a screen capture of the building of the Car Listings UI using Designer; see Additional Resources at the end of this chapter.)

Drag a Panel container from the Toolbox onto the canvas. This is the top-level component for the application. Now scroll further down the Toolbox to the Grid item, and drag a Grid Panel into the Panel container. The result will look like the following:



This Grid Panel will later be used to display the available car listings and enable the user to select a listing to view.

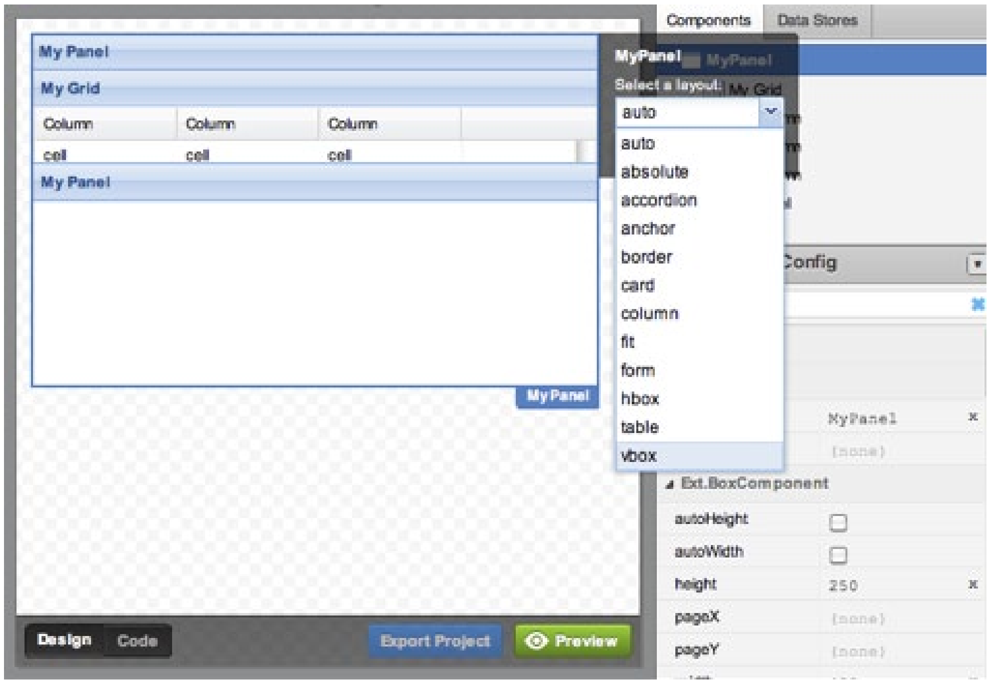
Scroll back up to the top of the Toolbox to Containers, and drag another Panel into the Panel container. This panel will later be used to display the car details for the listing selected in the Grid Panel. For now, you will see the following:



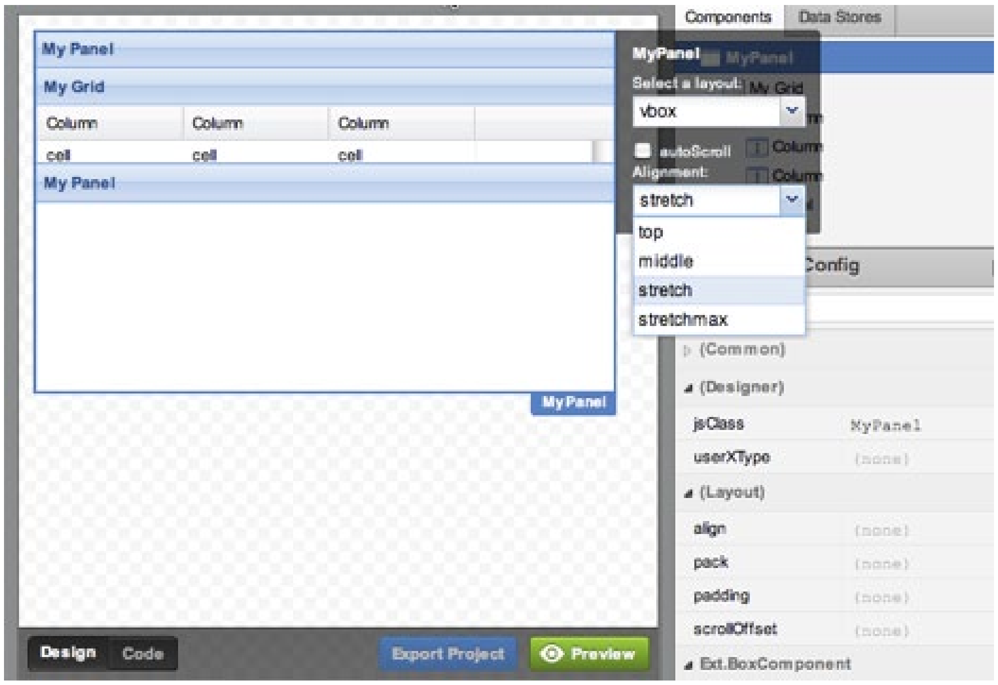
Positioning Components

The next step is to position the components just added to the application. By default, components are laid out using relative positioning. The best way to control the position of the elements on the canvas is to set the layout options on the containers and adjust the attributes that adjust the relative positions of each component.

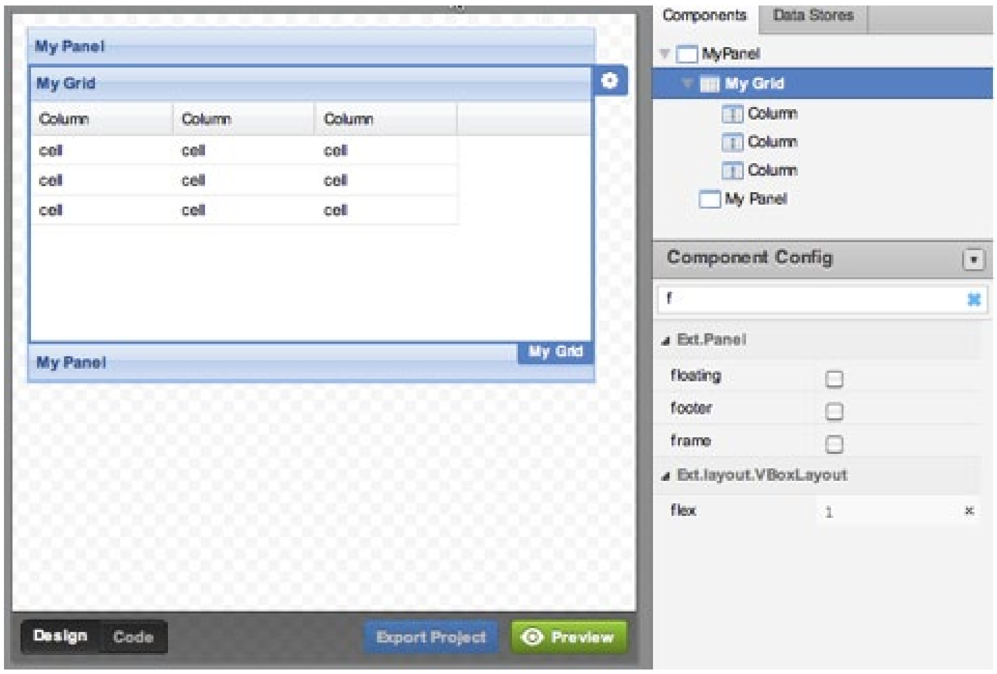
To start positioning the components in the example application, click the flyout config button (a gear-shaped button in the blue tab just to the right of the component name) on the top-level panel and set the layout to vbox, as shown here.



This will arrange the grid and sub-panel vertically. From this menu, the alignment and autoscroll attributes can be set. Set the alignment for the top-level panel to stretch, as shown below. This will cause the sub-components to stretch to fill the available space.



Next, position the Grid Panel that's part of the Container in the application. Select the Grid Panel (currently labelled 'My Grid') and set the flex attribute to 1 in the Component Config inspector, as shown here:



Tip: You can type the name or first few characters of an attribute in the text field at the top of the Component Config inspector to quickly navigate to a particular attribute.

The panel inherits the flex attribute from Ext.layout.VBoxLayout because the layout of the container is set to vbox. Setting the flex attribute of each of the components in the container to 1 will cause the components to take up the same amount of vertical space when the container is resized. (Similarly, if you wanted the sub-panel to take up two-thirds of the vertical space, you could set the flex value of the panel to 2 and the flex of the grid to 1.)

Do the same thing to the sub-panel that's been added to the top-level panel: Select the sub-panel (currently labelled 'My Panel, at the bottom of the top-level panel) and set the flex attribute to 1.

Although it's not recommended, you can choose the absolute layout option, where components are repositions by dragging them on the canvas. When working with Designer, however, it's preferable to rely on the Ext JS layout manager to control the relative positions of the components.

Layout Options

Setting the layout on a container controls how Ext JS lays out the components within that container. Switch between layout options by clicking a container's flyout config button and selecting a different layout.

Ext JS provides the following basic container layouts. Some support specific, commonly-used presentation models such as accordions and cards, while others provide more general-purpose models that can be used for a variety of applications. They're listed here; see

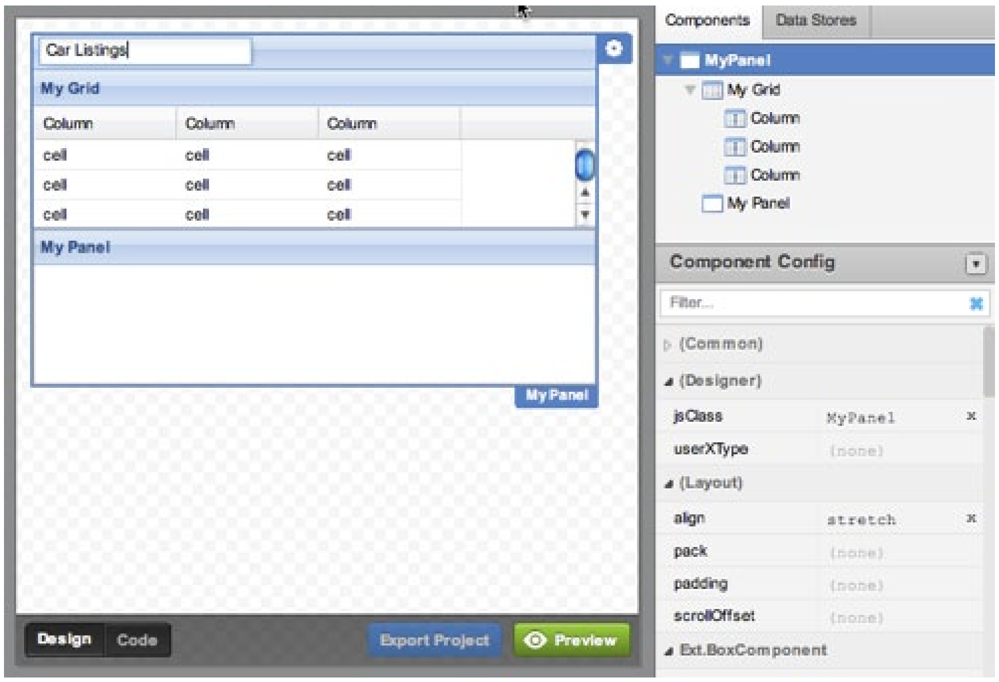
“Chapter 2: Working With Layouts in Designer” to learn how to select and configure layouts and see examples of them.

- » Auto
- » Absolute
- » Accordion
- » Anchor
- » Border
- » Card
- » Column
- » Fit
- » Form
- » Hbox
- » Table
- » Vbox

Configuring Components

Component attributes such as titles and labels can be edited directly in Designer. Just double-click the text you want to modify and type. The Component Config inspector enables configuration of all possible attributes for the selected component. Whenever you change the attribute from something other than the default value, Designer places an ‘x’ next to the attribute. This makes it easy to find edited attributes and revert to the default.

For the example application, start by setting the title and column heading attributes. Double-click the title of the top-level panel (‘My Panel’) and type `Car Listings`. This does the same as setting the title attribute in the Component Config inspector. You’ll see the following:

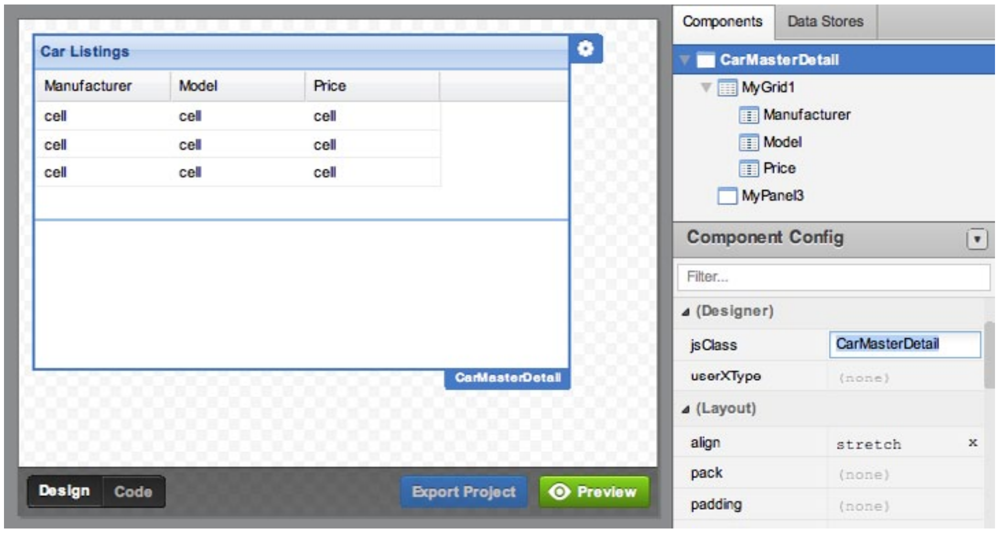


Next, double-click the text in the three column headings in the grid (‘Column’) one at a time. Type over each column head `Manufacturer`, `Model`, and `Price` (from left to right) to change their names.

Use the Component Config inspector in the lower right corner of the Designer window to set the rest of the component attributes in the example application.

First, remove the title bars from the grid panel and sub-panel within the top-level panel. Select each component and click the delete icon (x) to the right of the title attribute in the inspector. Now, the only title bar visible is the Car Listings title, as shown in the next image.

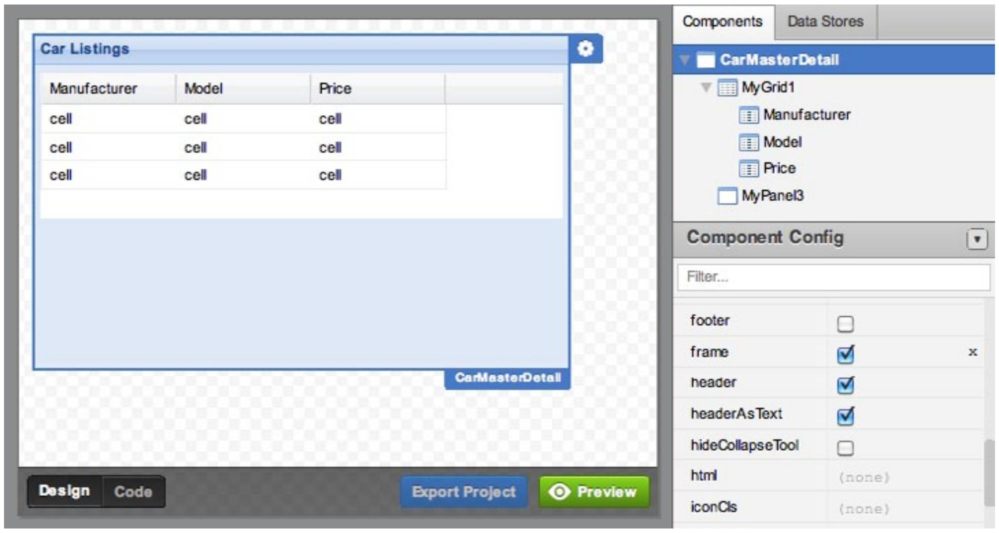
The component needs to have its own name in the code that will be generated for the example UI. To name the component in the code, select the top-level panel, which you just renamed 'Car Listings'. Double-click the text next to the jsClass attribute in the Component Config inspector ('MyPanel'). Type over it `CarMasterDetail`. The Designer window should now look like this:



jsClass	CarMasterDetail
userXType	(none)
align	stretch
pack	(none)
padding	(none)

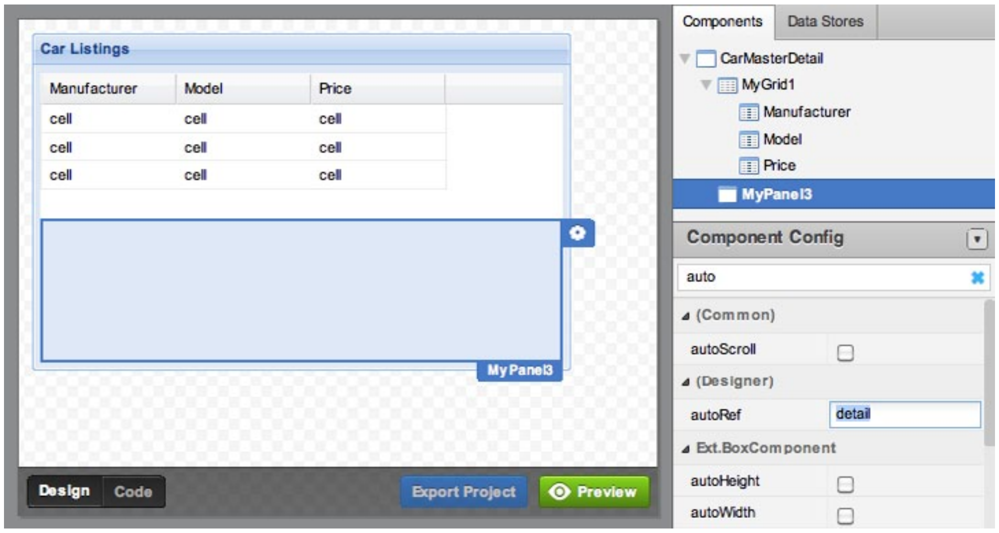
To see the code for the project, click the Code button below the canvas. You can toggle between the design and code views by clicking the Design and Code buttons.

Next, enable the frame attribute of the Car Listings panel. Instead of the plain 1px square borders, this renders the panel with additional styling, including rounded corners. Do this by scrolling further down the inspector and clicking the box next to the frame attribute. The box should now have a check mark in it, as shown below. Also, click the boxes next to the header and headerAsText attributes so that panel displays a header with its name.



footer	☐
frame	✓
header	✓
headerAsText	✓
hideCollapseTool	☐
html	(none)
iconCls	(none)

Now, configure auto references for the components so they can be directly referenced in the code regardless of how they are nested. First, select the Grid Panel and set the autoRef for the panel to 'grid.' Then, select the sub-panel and set its autoRef to 'detail', as shown here:



autoRef: detail

To add padding around the contents of the sub-panel, select the panel, type `p` to jump to the padding attribute, and type `10` in the box next to the name of the attribute. This sets padding to 10, the typical CSS padding attribute.

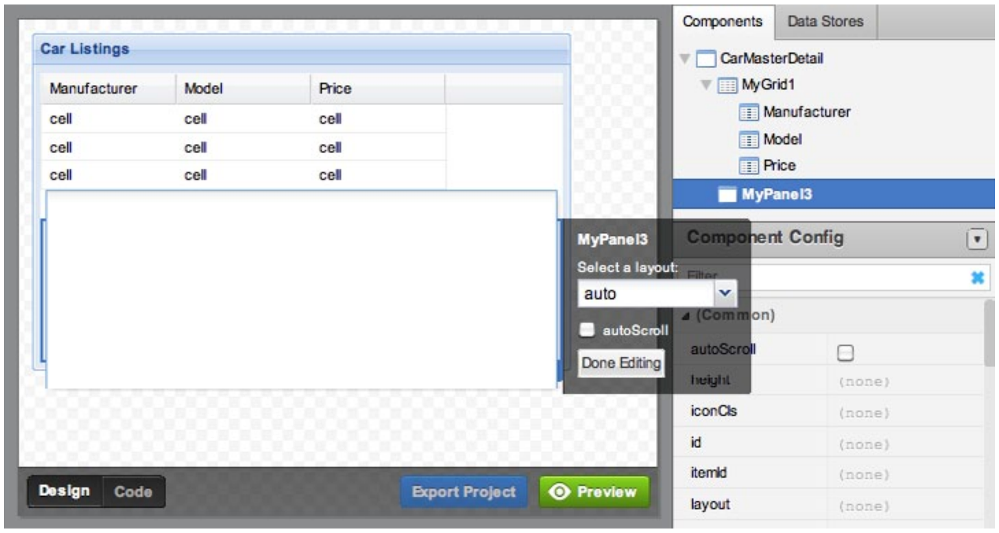
Using Templates

You can use templates to dynamically display information from a data store in a panel component. A template is an HTML fragment that can contain variables that reference fields in a data store. Templates also support auto-filling of arrays, conditional processing, math functions, and custom functions.

Variables are enclosed in curly braces. For example, {manufacturer} references the data field called ***manufacturer***. You can also specify formatting functions to control how the data is displayed. For example, {price:usMoney} uses the usMoney format to prepend a dollar sign and format the number as dollars and cents. See Ext.util.Format for the full range of available formatting functions.

The Car Listings example application uses a template to display the detail information for the selected listing. The image and wiki URL are pulled in from data fields in the cars.json data store. (See "Connecting to Data," below, for information about how to attach a data store.)

To configure the template in the example Car Listings UI, click the flyout config button on the sub-panel and then click Edit Template to add a template for the detail information. The body of the component becomes an editable text area, as shown here.



Enter the HTML mark-up for the template into the text area, as follows:

```
<img src="cars/{img}" style="float: right" />
Manufacturer: {manufacturer}<br/>
Model: <a href="{wiki}" target="_blank">{model}</a><br/>
Price: {price:usMoney}<br/>
```

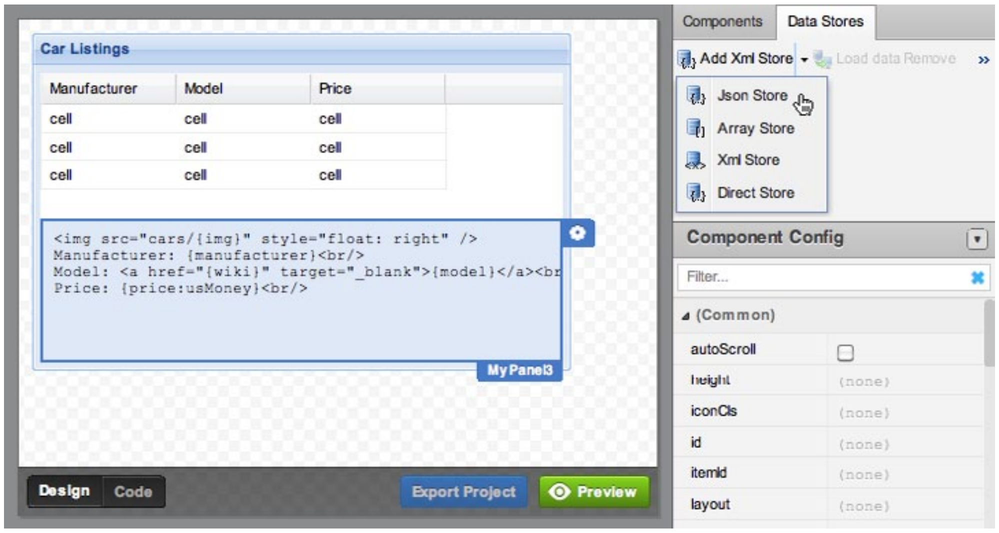
Click Done Editing to save the HTML to the example application.

Connecting to Data

You can attach Data Stores and bind them to the components in your UI from Designer.

The listing information displayed by the Car Listings application is read from a JSON data store called cars.json. To connect the data store and pull in manufacturer, model, price, wiki, and image data, start by adding a data store for the cars data.

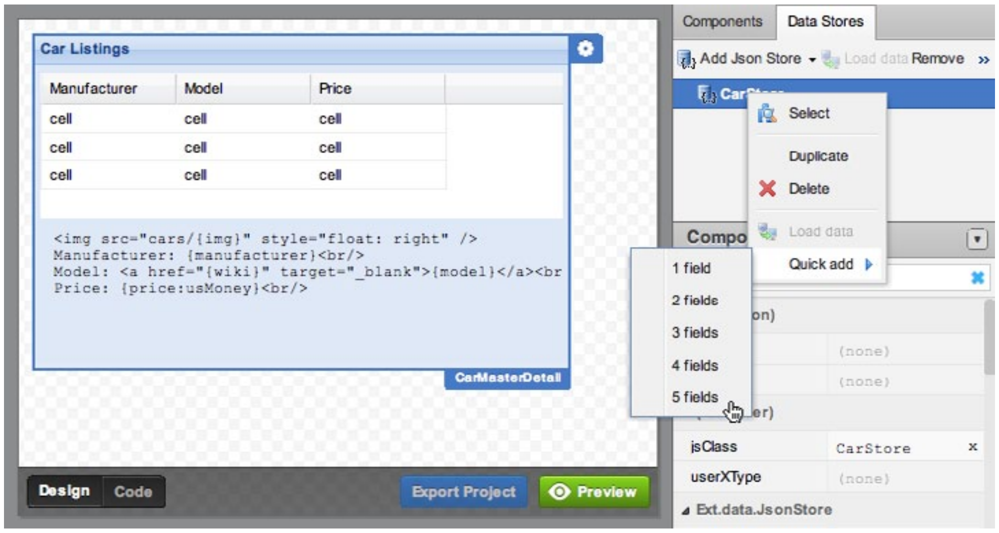
Select the Data Stores tab, then select Json Store from the Data Stores toolbar, as shown here:



```
<img src="cars/{img}" style="float: right" />
Manufacturer: {manufacturer}<br/>
Model: <a href="{wiki}" target="_blank">{model}</a><br
Price: {price:usMoney}<br/>
```

Select the newly-created store in the Data Stores tab ('CarStore'). Using the Component Config inspector, set the jsClass attribute to 'CarStore.' Set the storeId attribute to the same name. (The storeId is the name Designer displays in the list of available stores.)

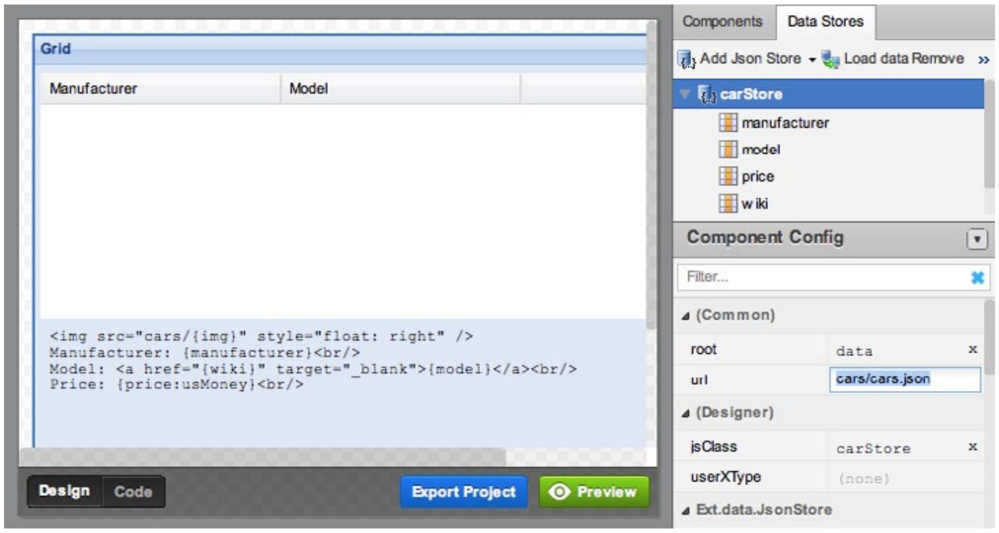
Right-click the Data Store and select Add Fields > 5 fields to add data fields to CarStore for each field defined in cars.json.



```
<img src="cars/{img}" style="float: right" />
Manufacturer: {manufacturer}<br/>
Model: <a href="{wiki}" target="_blank">{model}</a><br
Price: {price:usMoney}<br/>
```

jsClass	CarStore
userXType	(none)

Continue to configure the new data store ('CarStore') using the inspector by setting the url attribute to the relative path where the store will reside, that is `cars/cars.json`, as show below. This path is relative to the URL prefix specified in the Project Settings.



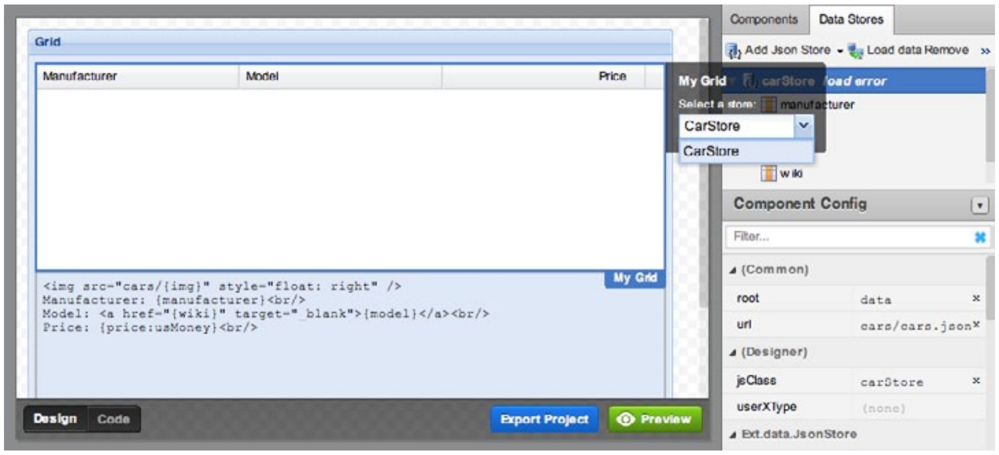
```
<img src="cars/{img}" style="float: right" />
Manufacturer: {manufacturer}<br/>
Model: <a href="{wiki}" target="_blank">{model}</a><br/>
Price: {price:usMoney}<br/>
```

(Common)	
root	data
url	cars/cars.json
(Designer)	
jsClass	carStore
userXType	(none)

Next, set the root attribute to `data` and enable the autoLoad attribute to configure CarStore to load automatically. (If you don't do this, you won't see any data when viewing the application index.html file.)

Now give each data field in CarStore a name. Select each of the five data fields one at a time in the Data Stores tab, then set the name attribute of each field in the inspector. From top to bottom, name them `manufacturer`, `model`, `price`, `wiki`, and `img`.

The next step is to bind the grid component to the store. Do this by clicking the flyout config button on the grid component and selecting CarStore, as shown here:

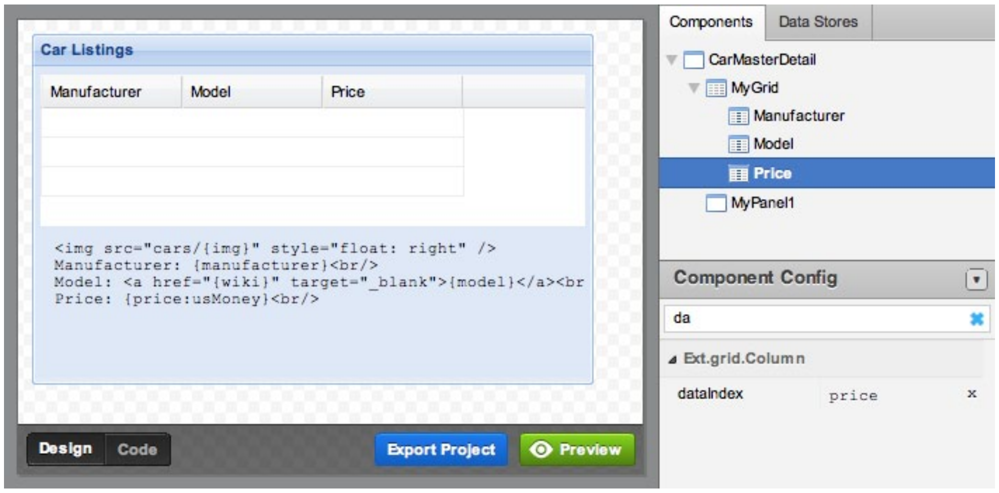


```
<img src="cars/{img}" style="float: right" />
Manufacturer: {manufacturer}<br/>
Model: <a href="{wiki}" target="_blank">{model}</a><br/>
Price: {price:usMoney}<br/>
```

Select a store: CarStore

(Common)
root: data
url: cars/cars.json
(Designer)
jsClass: carStore
userXType: (none)

Finally, link the grid columns to the appropriate data fields by selecting a column in the Components list and, in the inspector, setting the dataIndex attribute to the name of the data field, as shown just below. Data from the store is immediately displayed in the grid.



```
<img src="cars/{img}" style="float: right" />
Manufacturer: {manufacturer}<br/>
Model: <a href="{wiki}" target="_blank">{model}</a><br
Price: {price:usMoney}<br/>
```

Ext.grid.Column
dataIndex: price

Exporting a Project

Exporting a project generates the Javascript files for your application. When exporting a project, Designer creates the following two Javascript files for each top-level component:

» .base.js contains the base class that defines the component, for example, CarMasterDetail.base.js. Designer overwrites .base.js files every time it exports a project. *Do not modify these files directly.*

» .js is the implementation file. Designer generates the .js files only the first time it exports a project. Use this is the file when adding event handler code and custom methods. **Note**: if the jsClass is edited, Designer exports a new file with that class name and the original old_jsClass.js becomes obsolete.

Along with the Javascript files, Designer generates an xds_index.html file that loads the javascript and displays your app.

Attaching Event Handlers to UI Components

The files Designer generates can be imported to an external IDE or editor for customization or adding event handlers. Add event handlers by editing the .js files exported by Designer. Here's how to add an event handler to the Car Listings example application that displays the appropriate image and wiki information when a user selects a row in the grid.

In the file CarMasterDetail.js Designer created when it exported the Car Listings example application, create a selection model for the grid:

```
var sm = this.grid.getSelectionModel();
```

The default selection model for a grid is a RowSelectionModel. Whenever a row in the grid is selected, a rowselect event is fired. This event includes the SelectionModel, rowIndex and the Record that provides the data for the selected row.

Next, add an event handler to call a custom onRowSelect function when a row in the grid is selected:

```
sm.on('rowselect', this.onGridRowSelect, this);
```

Finally, implement onRowSelect to update the the row with the data from the data store:

```
onGridRowSelect: function(sm, rowIdx, r) {
      this.detail.update(r.data);
}
```

For more information about working with Ext JS grids, see the API documentation.

Additional Information

For more information about Designer and Ext JS:

- » Watch the Designer Demo, which shows how to build the Car Listings example application described here.
- » View the Designer webcast, which introduces Designer 1.2
- » For information about the latest Designer release and updates, see the Designer Changelog.
- » If you're new to Ext JS, see the component and container model information in the Ext JS Overview.
- » For the details about any Ext JS class or method, see the Ext JS API Reference.

Chapter 2: Working with Layouts

In Ext JS, layouts control the size and position of the components within an application. With Designer, Configuring a layout on each container lets you manage how that container's children are rendered. The container layout determines what size and position configuration options can be set on its child components.

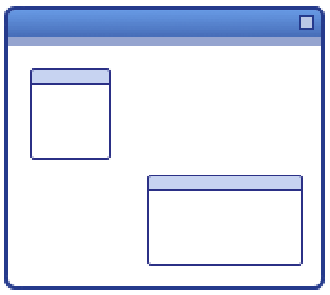
Basic Container Layouts

Ext JS provides a number of basic container layouts, which you can select and configure using Designer. Some support specific, commonly-used presentation models such as accordions and cards, while others provide more general-purpose models that can be used for a variety of applications.

Auto

Auto is the default layout. For general-purpose containers such as a Panel, using the auto layout means child components are rendered sequentially. It's important to keep in mind that some containers are automatically configured to use a layout other than the default auto. For example, FormPanel defaults to the form layout and TabPanel defaults to the card layout.

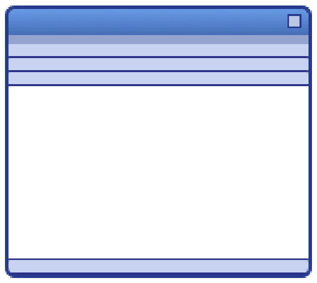
Absolute



Arranges components using explicit x/y positions relative to the container. This enables explicit repositioning and resizing of components within the container, providing precise control. Absolute-positioned components remain fixed even if their parent container is resized.

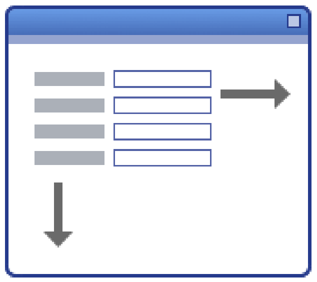
Designer displays a grid within a container that uses absolute layout. By default, components snap to the grid as they are repositioned. Clicking the container's flyout config button enables resizing or disabling the grid. The grid is only displayed as a layout guide in the Design view; it is not visible when the component is rendered.

Accordion



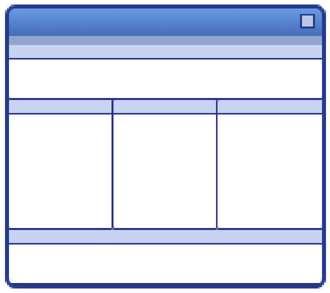
The accordion layout arranges panel components in a vertical stack where only one panel is expanded at a time. Only panels (including FormPanel and TabPanel) can be added to a container that uses the accordion layout.

Anchor



Arranges components relative to the sides of the container. Specify the width and height of child components as a percentage of the container or specify offsets from the right and bottom edges of the container. If the container is resized, the relative percentages or offsets are maintained.

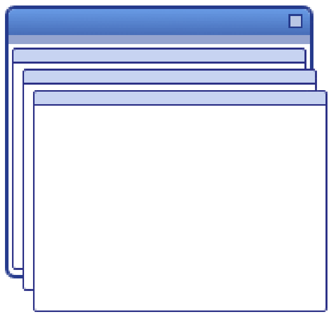
Border



Arranges panel components in a multi-pane layout according to one of five regions: North, South, East, West, or Center. A container that uses the border layout has to have a child assigned to the Center region. The center panel is automatically sized to fit the available space. Resize the North, South, East, and West panes on the canvas by clicking and dragging the right or bottom edge of the panel.

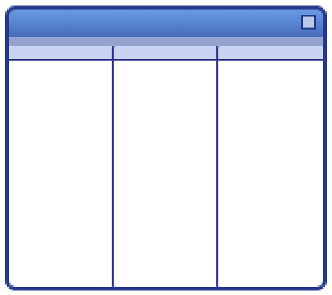
Make any of the panels in a border layout collapsible by enabling the collapsible attribute. When rendered, the child panels automatically resize when the container is resized.

Card



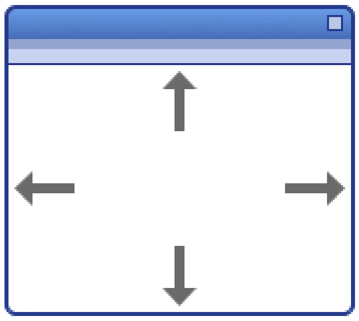
Used to enable stepping through a series of components one at a time by arranging child components so that only one can be visible at a time. Specify the component to be made visible by setting the setActiveItem property. This behavior is typically attached to a UI navigation element, such as Previous and Next buttons in the footer of the container. It's commonly used to create wizards.

Column



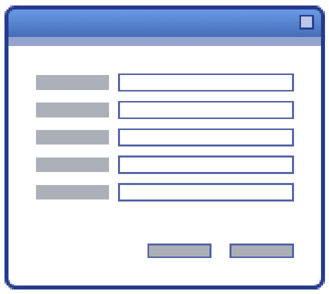
Arranges components in a multicolumn layout. The width of each column can be specified either as a percentage (column width) or an absolute pixel width (width). The column height varies based on the contents. Enable autoscroll it application data requires viewing column contents that exceed the container height.

Fit



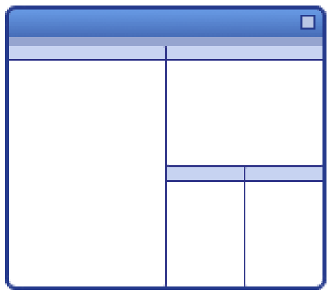
Expands a single child component to fill the available space. For example, use this to create a dialog box that contains a single TabPanel. If the container is a type of panel component, you can also add a Toolbar to it.

Form



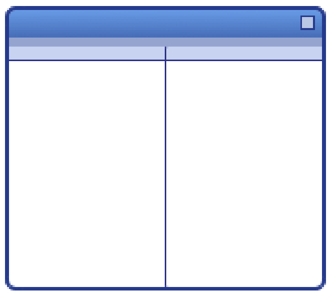
Arranges a collection of labeled form fields. A FormPanel uses the form layout by default.

Table



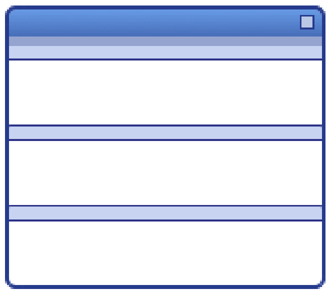
Arranges components in an HTML table. The number of columns in the table need to be specified. Developer enables creation of complex layouts by specifying the rowspan and colspan attributes on the child components.

Hbox



Arranges the child components horizontally. Setting the alignment of the container to stretch causes the child components to fill the available vertical space. Setting the flex attribute of the child components controls the proportion of the horizontal space each component fills.

Vbox



Arranges the child components vertically. Setting the alignment of the container to stretch causes the child components to fill the available horizontal space. Setting the flex attribute of the child components controls the proportion of the vertical space each component fills.

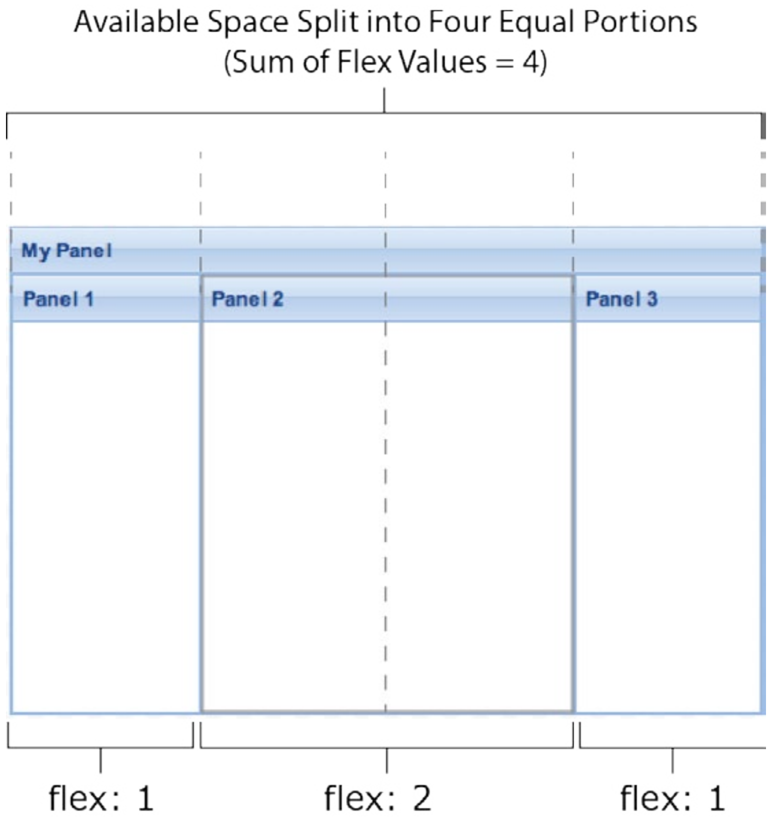
Nested Layouts

When Designer nests containers, the layout configuration for the parent container manages the layout of whatever child components (including other containers) it contains. The children will have the same layouts as the parent that contains them. The layout doesn't affect the contents of any child containers, only the containers themselves.

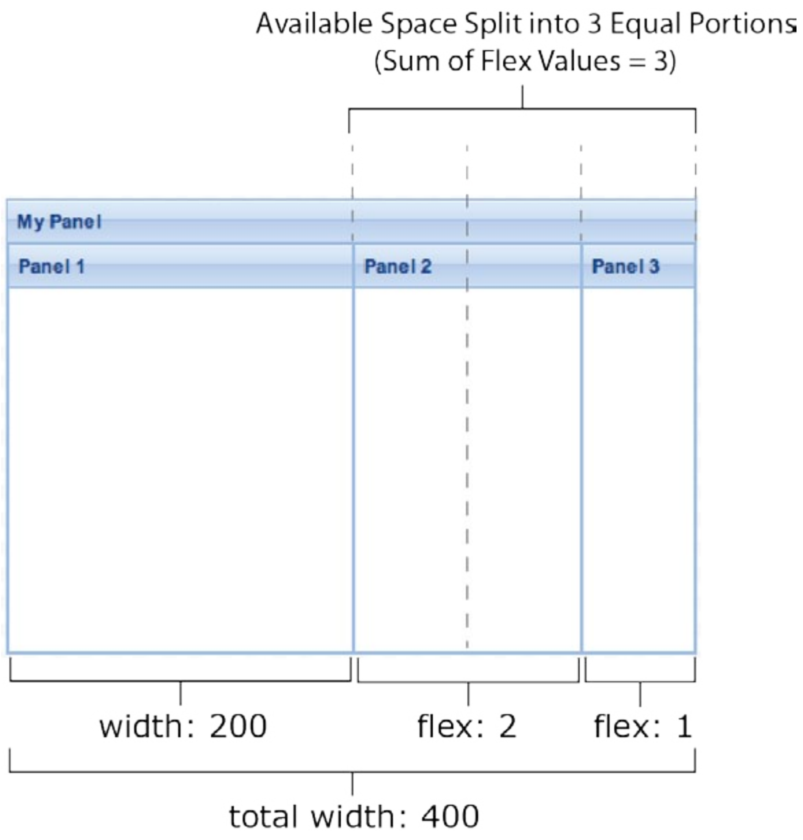
Flexible Box Layouts

The hbox and vbox layouts enable child components to be resized to fit the available space in a container using the component flex attribute. The flex attribute is a numerical value that represents the proportion of the available space that will be allotted to a component. You can set the flex attribute to any floating-point value, including whole numbers and fractions.

For example, consider a component with three subpanels in which flex is set to '1' for Panel 1 and Panel 3, and flex is set to '2' for Panel 2. The available space is divided into four equal portions (the sum of the flex values), and Panel 1 and Panel 3 each get one portion while Panel 2 gets two, as shown here.



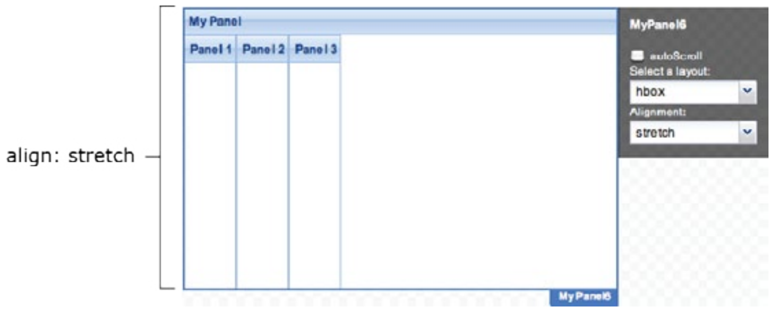
If you set an absolute width or height for some components and a flex value for others, the absolute sizes are subtracted from the total available space and the remaining space is allotted to the flexed components. For example, if the container is 400 pixels wide and the width of Panel 1 is set to 200 pixels, the panels with flex attributes set share the remaining 200 pixels. If Panel 2 has a flex of 2 and Panel 3 has a flex of 1, Panel 2 will get two-thirds of the space and Panel 3 will get one-third of the space. See below.



If neither an absolute size or a flex value are specified for a component, the framework checks to see if the size is defined in the application's CSS. If no size is specified in the CSS, the framework assigns the minimum necessary space to the item.

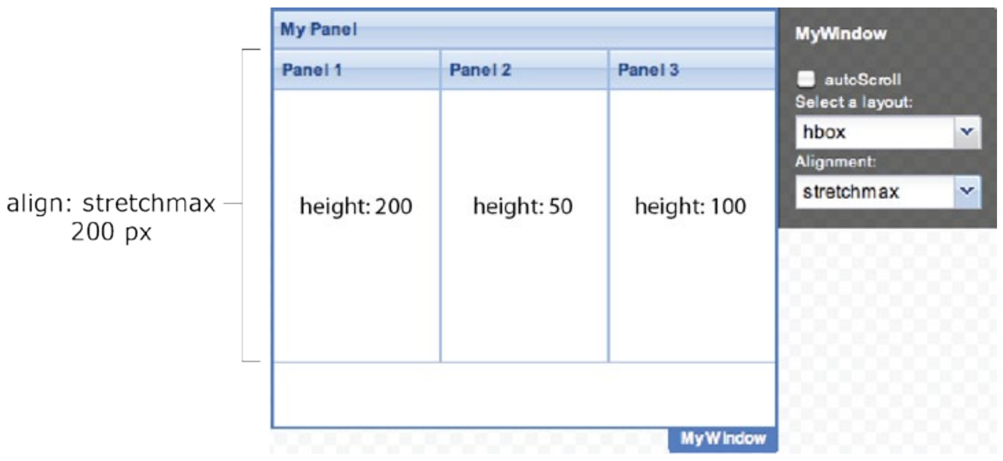
Stretching Components to Fit

If 'stretch' is specified as the alignment option for a container that uses the hbox or vbox layout, its sub-components are automatically stretched to horizontally or vertically fit the size of the container. When hbox is used, the sub-components are stretched vertically. With vbox, the sub-components are stretched horizontally. For example, when 'stretch' is set on a panel that uses hbox, each of the sub-panels is automatically stretched to fill the available vertical space.



Select a layout: hbox
Alignment: stretch

The stretchmax option works just like stretch, except it stretches sub-components to the size of the tallest or widest component, rather than the size of the container.



Configuring the Layout for a Container

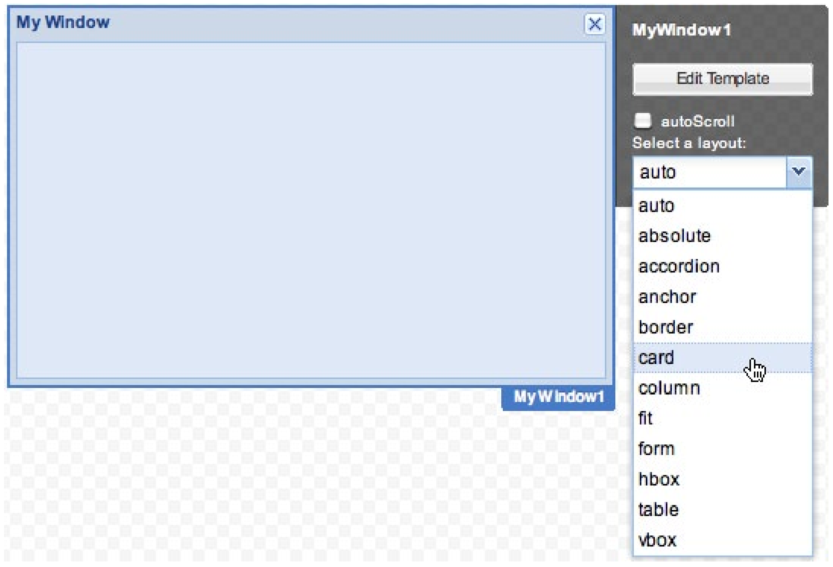
The Designer UI provides two ways to set the layout for a container, the container's flyout config butto and the Component Config inspector. Use either way to configure layouts

Using CardLayout to Create a Wizard

If a component uses the card layout, its children are only visible one at a time, making it an ideal option for creating a wizard. The following provides a detailed example of working with layouts in Designer, showing how to create a three-step registration wizard using the card layout.

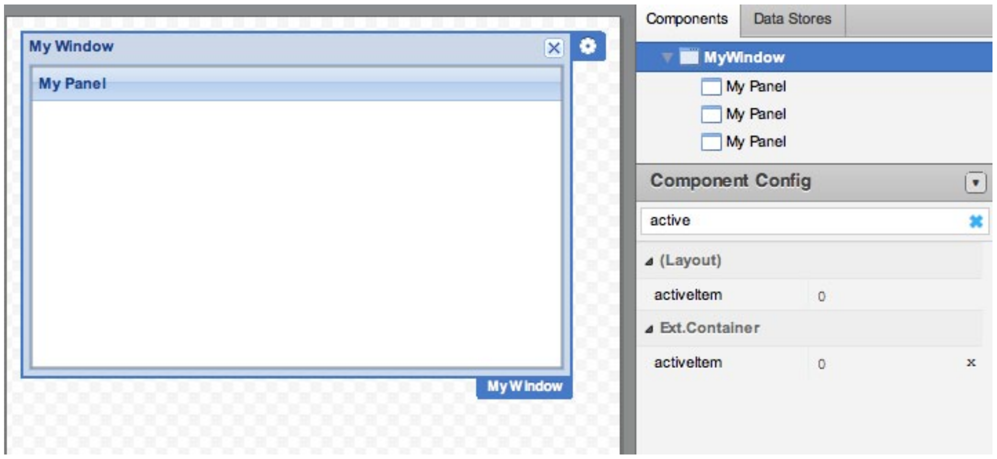
The basic approach is to add subpanels to a Window that uses the card layout and configure a navigation toolbar to step through the panels. Window components are specialized types of panels that can be resized and dragged. Then, you implement a handler that calls the setActiveItem function to display the appropriate panel when the user clicks a navigation button within the Window.

Start by dragging a Window from the Toolbox onto the Designer canvas. A Window can only be added as a top-level component; it cannot be added as a child of an existing component. Click the Window's flyout config button and select card from the layout menu to apply the card layout to the Window, as shown below. Also, name the wizard by double-clicking the Window title on the canvas to edit it. (Another way to edit the Window title is to set the title attribute in the Component Config inspector.)

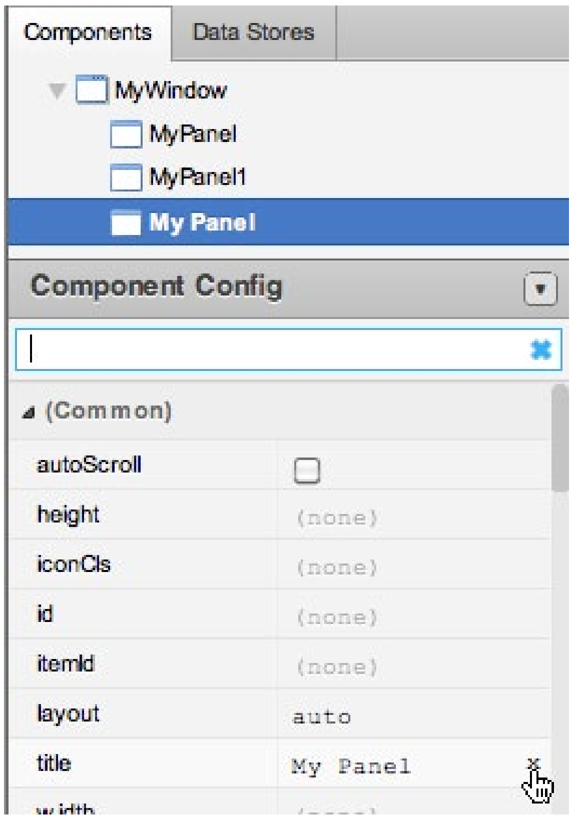


Next, drag a Panel component onto the Window; this subpanel will be used to create the first step in the wizard. Panels in a CardLayout are numbered in the order they are added to the container, starting with item 0. By default, item 0 is set as the active item. To change the active item within Designer, select the Window and set the activeItem attribute to the panel you want to make active.

Add two more panels to the Window for the second and third steps of the Wizard, as shown just below. Either drag them onto the title bar of the Window on the canvas or onto the Window in the Components tab.

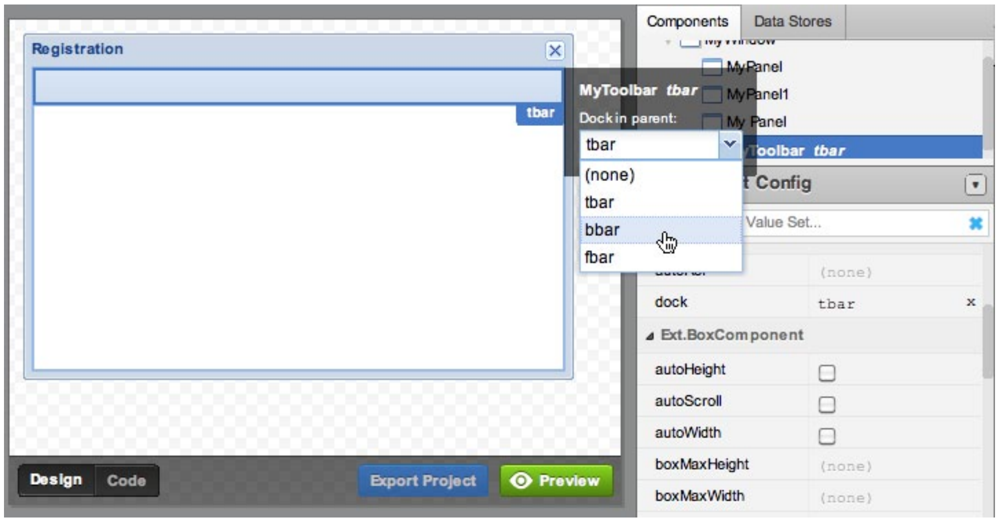


As subpanels are added, hide their title bars by selecting each subpanel in the Components tab and clicking the delete icon next to its title attribute in the Component Config inspector, like this:



autoScroll	☐
height	(none)
iconCls	(none)
id	(none)
itemId	(none)
layout	auto
title	My Panel

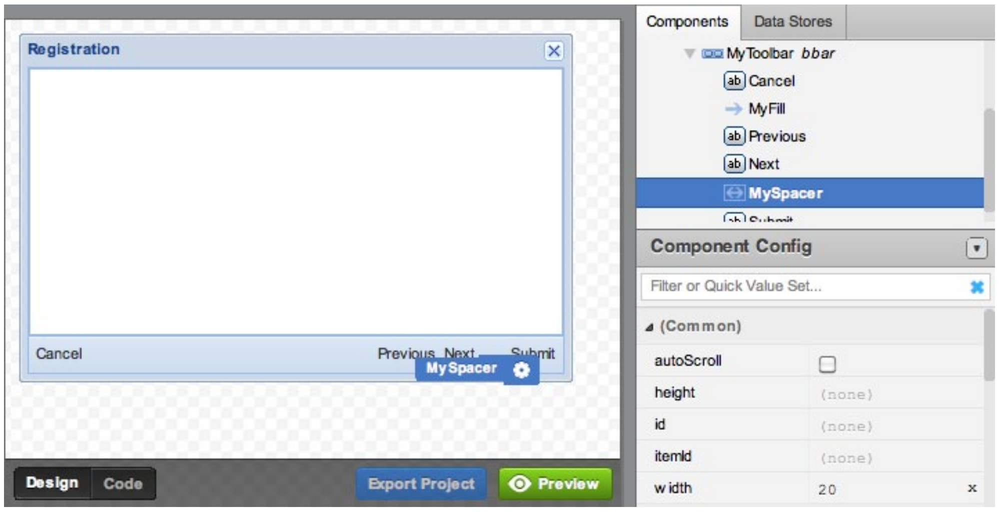
The wizard need navigation buttons to move from one step to the next. Do this by dragging a Toolbar from the Toolbox to the top-level Window and dock it at the bottom of the Window (bbar), as shown here:



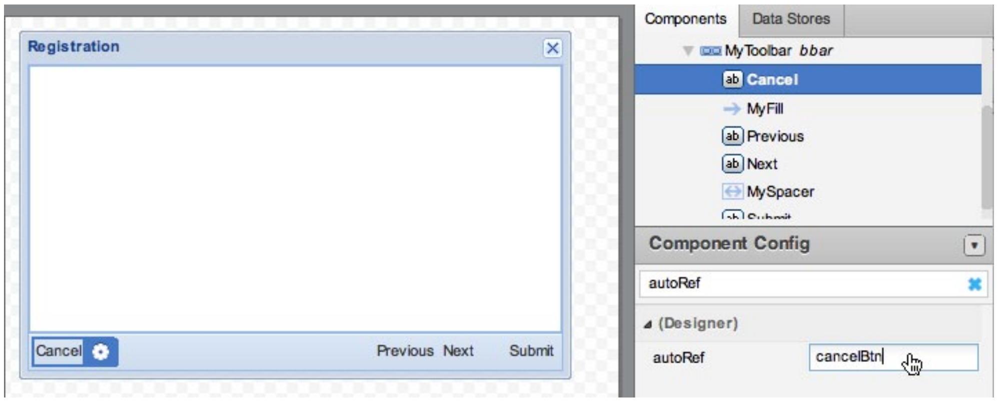
dock	tbar
autoHeight	☐
autoScroll	☐
autoWidth	☐
boxMaxHeight	(none)
boxMaxWidth	(none)

Then, add four buttons to the Toolbar and name the buttons Cancel, Previous, Next, and Submit. Double-click the first button label on the canvas and type over to name them, and use tab to move to the next button in the Toolbar until you've named each button.

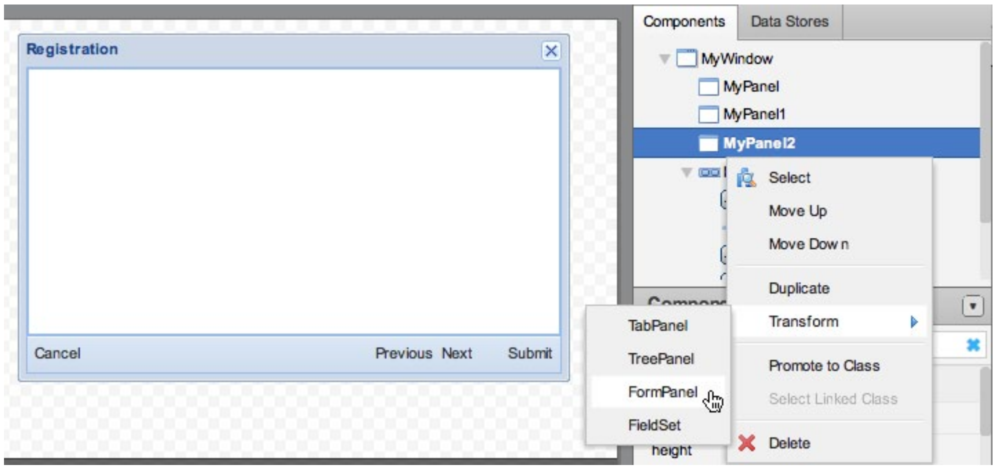
The buttons need a little more work to make them more usable by both the user and the developer. First, align the buttons by adding a Fill between the Cancel and Previous button and a Spacer of width 20 between the Next and Submit buttons.



Then, set the autoRef attribute for the buttons so they can be easily referenced in the navigation handler. For example, set the attributes to autoRef: cancelBtn, autoRef: prevBtn, autoRef: nextBtn, and autoRef: submitBtn.



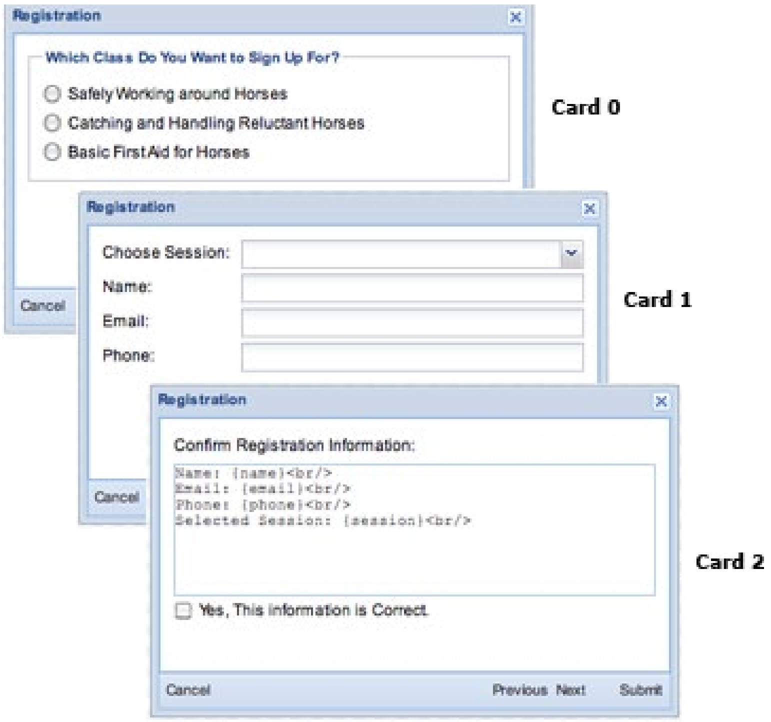
Now we're ready to add the content to each card that will be used in the wizard. However, since the wizard needs to gather user input, each card should be a FormPanel rather than a Panel. To change the Panels into FormPanels, right-click each one and choose Transform > FormPanel.



For this example, we built a registration wizard for a series of horsemanship clinics with the three cards shown below. By default, card 0 is the active card. To add form fields to the card 1 and card 2, select the Window and set its activeItem attribute to the panel you want to work on.

For this example, we built a registration wizard for a series of horsemanship clinics with three cards, shown below. By default, card 0 is the active card. To add form fields to card 1 and card 2, select the Window and set its activeItem attribute to the panel you want to work on.

For more information about creating forms in Designer, see 'Building Forms' in Chapter 3: Working with Components.



Registration

Which Class Do You Want to Sign Up For?

- Safely Working around Horses
- Catching and Handling Reluctant Horses
- Basic First Aid for Horses

Cancel

Registration

Choose Session:
Name:
Email:
Phone:

Cancel

Registration

Confirm Registration Information:

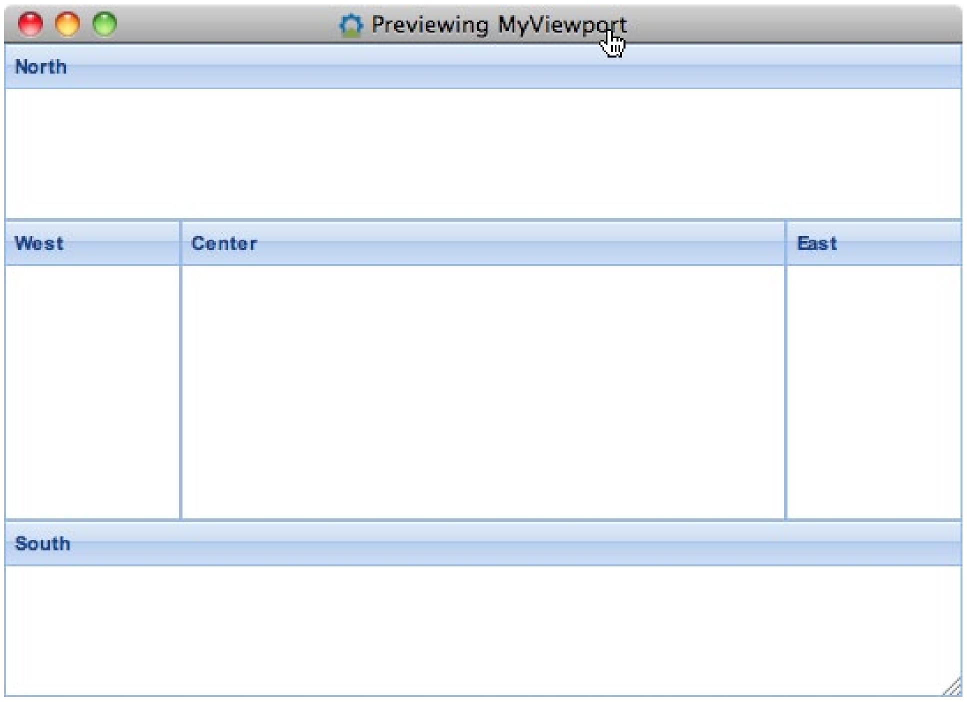
```
Name: {name}<br/>
Email: {email}<br/>
Phone: {phone}<br/>
Selected Session: {session}<br/>
```

☐ Yes, This information is Correct.

Cancel Previous Next Submit

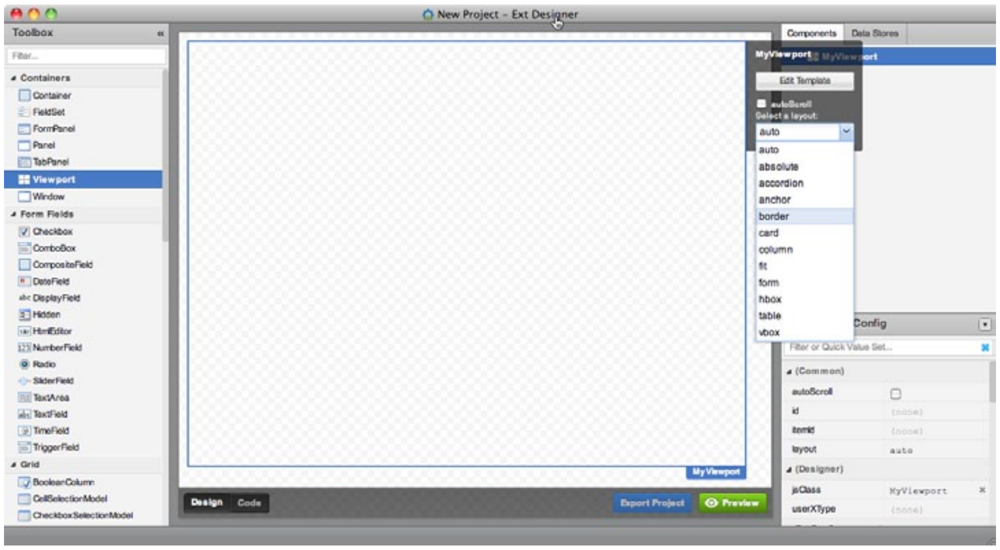
Using Border Layout for a Viewport

Use the Viewport container for applications that need the entire content area in a browser window (that is, the entire browser viewport). Viewport usually uses the border layout to arrange a collection of subpanels according to the regions North, South, East, West, or Center, as shown below. With the border layout, there must be a panel assigned to the Center region, which is automatically sized to fit the available space.



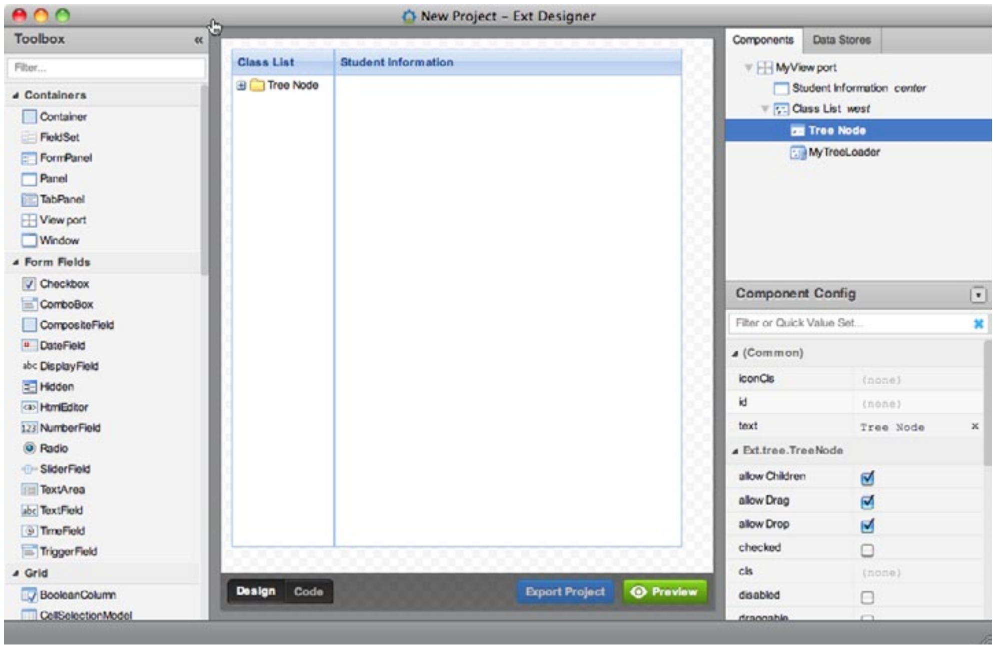
Following is another example UI that uses a Viewport with the border layout, in this case a viewer students would use to register for classes.

Start building the viewer by dragging a Viewport from the Toolbar onto the Designer Canvas. A Viewport can only be added as a top-level component; it cannot be added as a child of an existing component. Select the border layout by clicking the Viewport flyout config button and selecting border from the layout menu, like this:



Next, drag a Panel into the Viewport. Because this is the only component currently in the layout, it is automatically assigned to the Center region. This Panel will display information about people who have signed up for one of our classes, so name the Panel Student Information.

Add a TreePanel to the Viewport by selecting the Viewport and double-clicking TreePanel in the Toolbox to add it as a child of the Viewport. Alternately, drag the TreePanel onto the Viewport in the Components tab. The TreePanel will automatically be assigned to the West region. Students will use the tree to navigate through the classes they can take, so name it Class List, as shown here:



(Common)	
iconCls	(none)
id	(none)
text	Tree Node
Ext.tree.TreeNode	
allow Children	✓
allow Drag	✓
allow Drop	✓
checked	☐
cls	(none)
disabled	☐

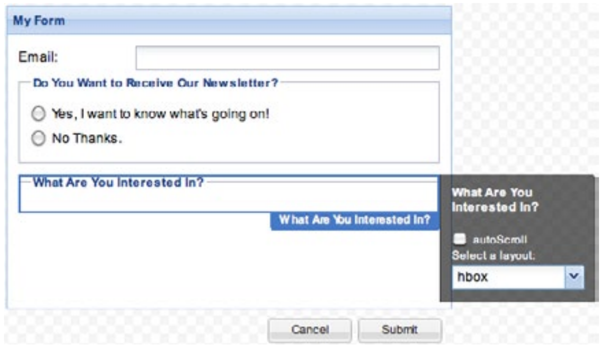
Note that it is possible to change the region that a subcomponent is assigned to. To do so, set its region attribute in the Component Config inspector.

The next step would be to configure the Class List tree and the Student Information Panel to display content about classes. A template could be used to display data for individual students in the Student Information Panel.

Using hbox Layout to Create Multiple Columns

The hbox layout enables horizontal arrangement of subcomponents, while vbox lays out subcomponents vertically. These general-purpose layouts provide a lot of control over how components are positioned without having to resort to using absolute positioning.

Take as an example, arranging several related checkboxes in multiple columns to conserve space. To do this, start by adding a FieldSet container to your FormPanel parent for the checkboxes and setting the layout of the FieldSet to hbox, like this:

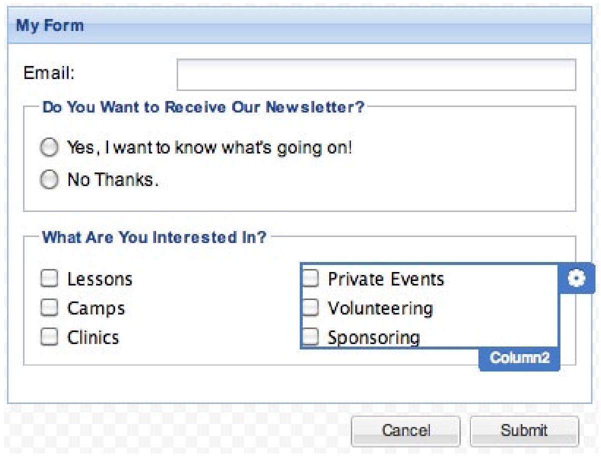

My Form
Email:
Do You Want to Receive Our Newsletter?
Yes, I want to know what's going on!
No Thanks.
What Are You Interested In?
What Are You Interested In?
What Are You Interested In?
autoScroll
Cancel
Submit

Select a layout: hbox

Next, add a Container component to the FieldSet for each column. For each component, set flex to 1 and set the height to accommodate all the checkboxes that will be added. For example, 60 pixels will accommodate three rows of checkboxes.

It's easiest to select the column containers from the Component list rather than from the canvas. (When they are first added to the FieldSet, they are only 2 pixels tall.)

Finally, add checkboxes to each column container and set their boxLabel attributes. To specify margins around the checkboxes as shown just below, change the layout of the column containers from auto to vbox, and then set the margin attribute for each individual checkbox.


My Form
Email:
Do You Want to Receive Our Newsletter?
Yes, I want to know what's going on!
No Thanks.
What Are You Interested In?
Lessons
Camps
Clinics
Private Events
Volunteering
Sponsoring
Column2
Cancel
Submit

Chapter 3: Component Overview

Designer supports all of the standard Ext JS UI components. This chapter provides an overview of the standard components available through the Designer Toolbox and how to work with them, including the following types of components:

» Containers
» Form Fields
» Grid
» Menu
» Standard
» Toolbar
» Tree
» Views

For additional information about individual components, see the Ext JS API Documentation. Also, keep in mind that custom components created in Ext JS can be saved and accessed through the Toolbox, as well as exported from Designer and saved to your development system for later importing back into other Designer projects.

Adding Components to a UI

Building an application UI starts with dragging a container to the Designer canvas, selecting from a variety of common UI containers provided by Ext JS, including the most basic container—called simply Container—as well Window, Panel, and Viewport; see the next section for an overview of all of them.

The next steps are to add display and control components to the container and arrange the components with the container's layout options. For more about layouts, see Chapter 2 in this guide and check out the Ext JS Layout Browser. Dragging additional containers within the first container adds them as children; dragging them to an empty portion of the canvas adds them as new top-level containers.

When nesting containers, make sure not to add redundant containers to the hierarchy. For example, if you want to display a TreePanel and a GridPanel in a Viewport that uses Border-Layout, you can add them directly to the Viewport and set their region attributes to control their positions. There's no need to add left and center Panel components to the Viewport first, and then add the TreePanel and GridPanel to those Panels.

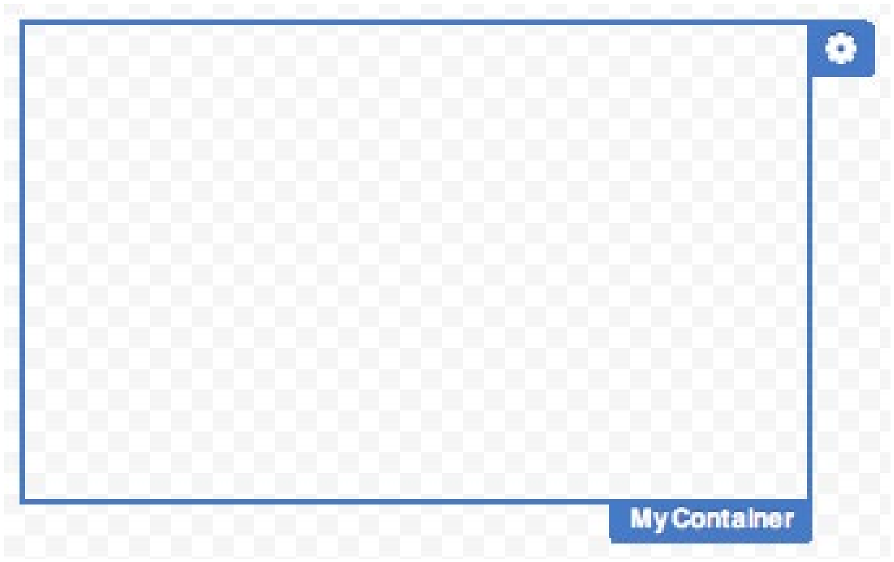
Designer prevents the addition of invalid components to a parent container. For example, Viewports and Windows can only be used as top-level components and cannot be nested within other containers. If you try to do that, Designer displays a small icon letting you know it can't be done.

When Designer exports a project, it automatically generates a separate class file for each top-level component. Nested components can be exported as separate classes by using the Promote to Class option. This enables Designer to generate several smaller, easier to maintain implementation files for a complex interface rather than a single, large, monolithic code file. It also makes it easier to reuse custom components.

Containers Overview

Ext JS provides a variety of standard container types that can be added to a UI and configured using Designer to meet most development needs. Let's take a look at all of them.

Container



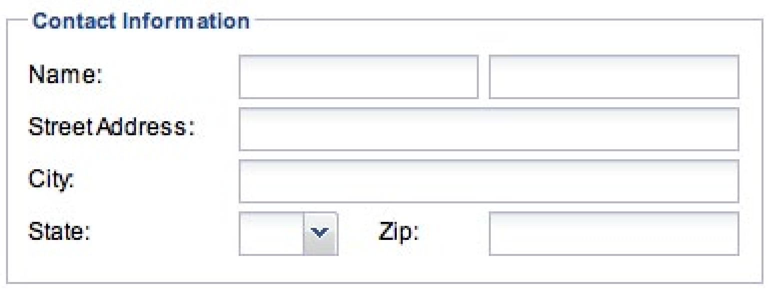
Container is the simplest component that can contain other components. All other container types are extensions of the Container class.

A Container is simply a logical container. Unlike a Panel or Window, it doesn't have any default visual characteristics.

Although typically used less than more specialized containers, Container provides a lightweight option for cases in which you don't need (or want) the added functionality. For example, using Container is the preferred way to create a multicolumn layout within a form.

The default layout for Container is ContainerLayout, which renders nested components as-is. With the default layout for Container, nested components will not be resized when Container is resized.

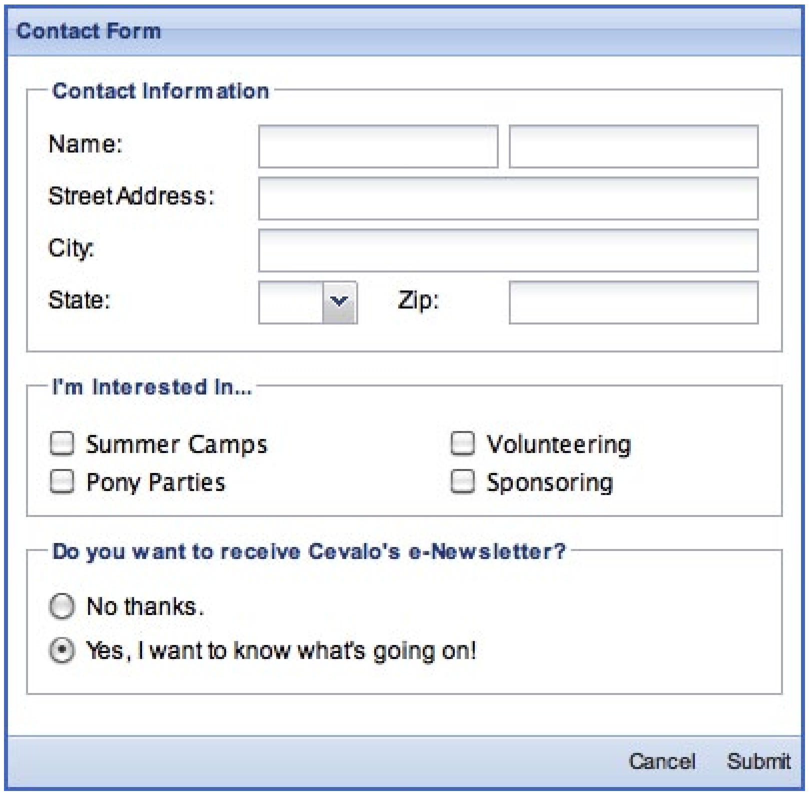
FieldSet



FieldSet is used to group related fields within a FormPanel. Specifying its title attribute displays the text for the title as a header within the FieldSet's frame.

Typically, FieldSet contains Form Fields, but a FieldSet can also contain nested containers. For example, nested container components can be used within a FieldSet to create a multicolumn layout. Note that the nested container with the added fields must also use FormLayout, the layout used by FormPanel.

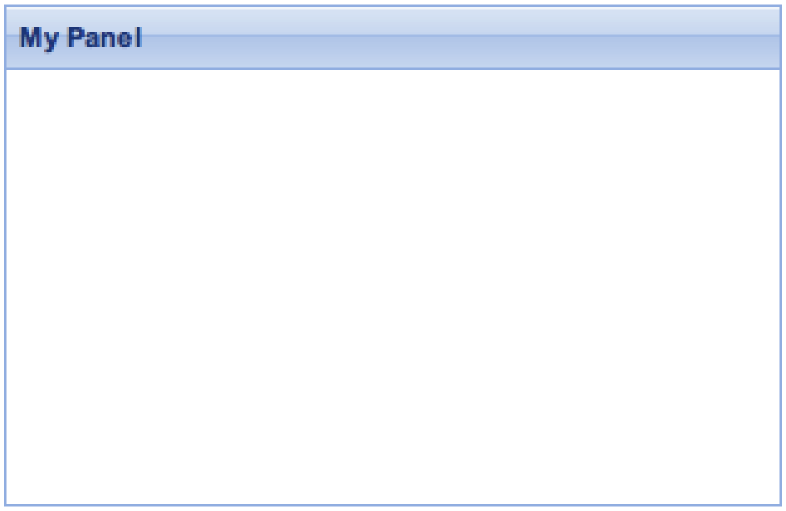
FormPanel



FormPanel is a specialized container that uses FormLayout to render a collection of fields and labels. In addition to the various Form Fields, containers such as Container and FieldSet can be added to a FormPanel. For example, nested Containers might be used to build a multicolumn form. Any nested containers with the added fields must also use FormLayout.

Internally, a FormPanel uses a BasicForm to handle file uploads, data validation, and submission.

Panel

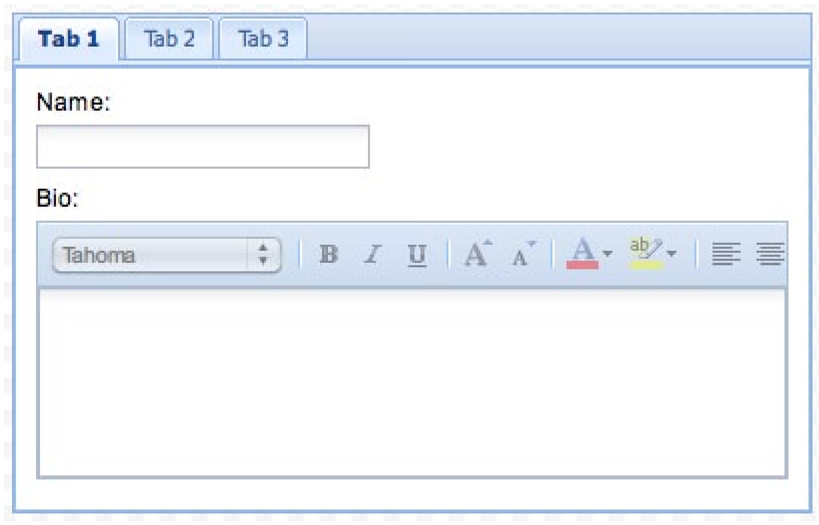


Panel is the basic building block for user interfaces providing robust application functionality. A Panel can be added to any type of container and can contain separate header, footer, and body sections.

In addition to the generic Panel container, Ext JS provides a number of specialized types of panels, including FormPanel, TabPanel, GridPanel, and TreePanel. The Window and FieldSet containers are also extensions of Panel.

By default, a Panel uses ContainerLayout, which simply renders nested components in the order they are specified in the Panel class. Choose an appropriate layout to control the position and sizing of the nested components.

TabPanel



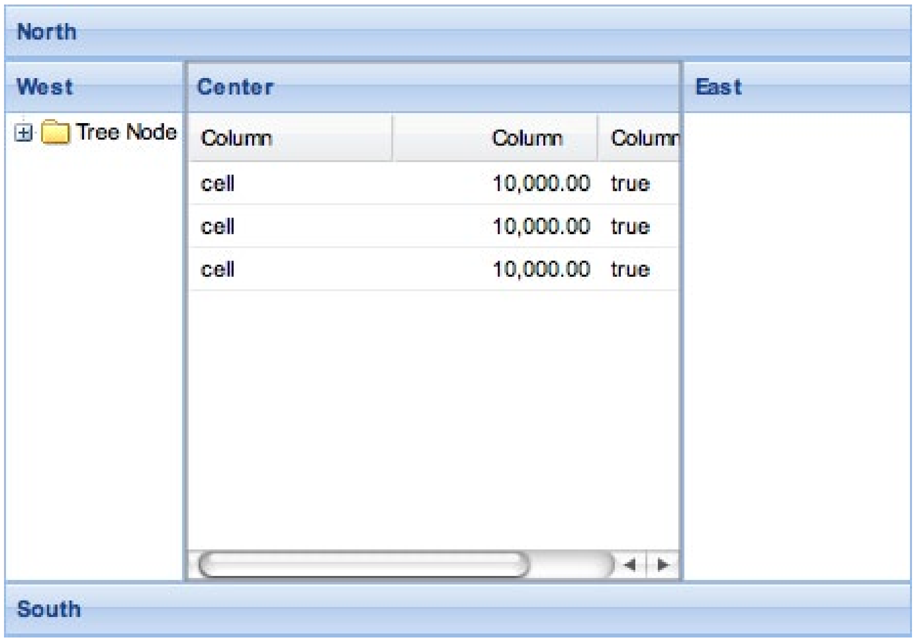
TabPanel is a specialized type of Panel that uses CardLayout to display a collection of nested components as separate tabs.

A TabPanel's title attribute is not displayed

TabPanel uses the header and footer space for the tab selector buttons. If an application needs to display a header, wrap the TabPanel in a Panel container that uses FitLayout.

Designer adds a TabPanel with three tab components to the canvas by default. Additional subcomponents can be added to each tab, and tabs can be added by dragging components onto the TabPanel.

Viewport



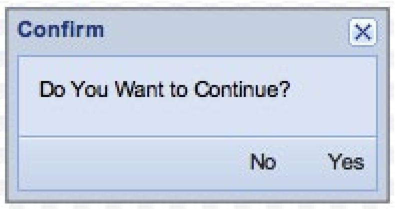
Column	Column	Column
cell	10,000.00	true
cell	10,000.00	true
cell	10,000.00	true

Viewport is used to represent the entire viewable application area in a browser window. A Viewport automatically sizes itself to the size of the browser viewport.

Each page can have only one Viewport, but it can contain nested panels that each has their own layouts and nested components. A Viewport is not scrollable—if scrolling is required, use scrollable child components within the Viewport.

Typically, ViewPort uses the BorderLayout layout with panels positioned within the Viewport by setting their region attributes to North, South, East, West, or Center. If no region is specified for a component, it defaults to the Center region.

Window



Do You Want to Continue?

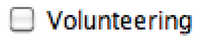
Window is a specialized type of Panel that is resizable and draggable. Windows can also be maximized to fill the viewport, minimized, and restored to their previous size. Unlike an ordinary Panel, Windows float and can be closed.

Windows are commonly used to present dialogs and errors.

Form Components Overview

Next, let's look at Form Field options within Ext JS that can be added to an application and configured with Designer. To build a form, add Form Field components to a FormPanel. Use FieldSet to group related fields with a FieldSet. To create multicolumn forms, add nested Containers for the columns. For more information about designing forms, see Building Forms, below.

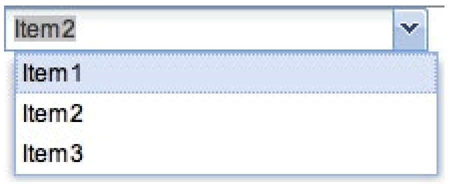
Checkbox



Checkbox represents a single checkbox field. Specify the label for a Checkbox by setting the boxLabel attribute. Create a checkbox group by adding multiple checkbox components to a Container.

Because of problems with RadioGroup and CheckboxGroup in ExtJS 3.x, these components are not included in the Designer Toolbox. See Adding a Group of Radio Buttons or Checkboxes for more information about how to use containers to build radio and checkbox groups with Designer.

ComboBox

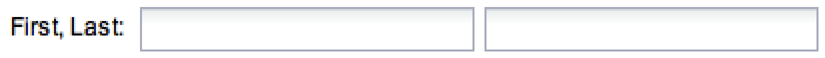


ComboBox enables users to select from a list of items.

To configure the items for a ComboBox, connect it to a data store. For more information, see Populating a ComboBox.

The height of a ComboBox is always set automatically, and its width can only be changed if it isn't used in an anchor, form, or fit layout and also is not contained within an EditorGrid Column.

CompositeField

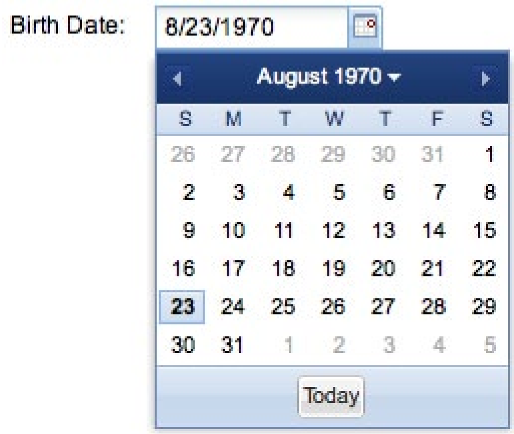


CompositeField enables a UI to easily display multiple fields on the same row of a form. A common use of CompositeField is for multipart name fields. However, a CompositeField can contain any type of Form Field, not just TextFields.

If no fieldLabel is specified on a CompositeField, the label defaults to a list of its children's fieldLabel attributes. (Like all fields, a CompositeField's label is automatically rendered when it's placed in a FormPanel container.)

In Designer, a CompositeField contains one TextField component by default when it's dragged to the canvas. Add fields by dragging them into the CompositeField or duplicating existing fields.

DateField



DateField provides a date-picker for ExtJS applications. It also provides automatic data validation for dates that the user enters manually.

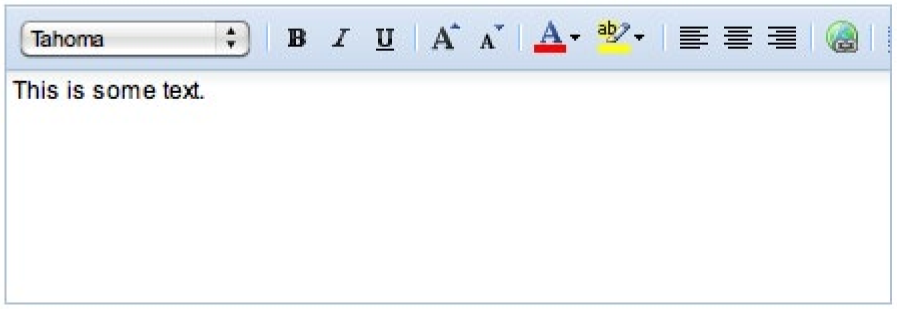
DisplayField

DisplayField renders view-only text that is not validated or submitted with a form. In application code, call setValue on the DisplayField to set the display text.

Hidden

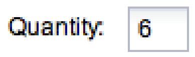
Hidden is a field that is not displayed within a form, but can be used to pass data when the form is submitted. In Designer, dragging a Hidden field onto the canvas does not result in any visual representation of the field, but it is listed on the Components tab.

HtmlEditor



HtmlEditor is a lightweight WYSIWYG HTML editor used within forms to enable users to submit styled text. Tooltips are defined for the editor toolbar; to enable them, initialize QuickTips.

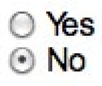
NumberField



NumberField is a specialized text field that automatically performs numeric data validation and only allows the user to enter numeric values.

Designer lets you set a maximum value for the field as well as other attributes on the field to control whether or not it will accept decimal or negative values. If decimal values are permitted, the precision and separator character can also be set.

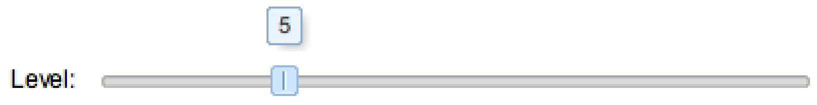
Radio



Radio represents a single radio button. Set the boxLabel attribute to specify the button label, and add multiple Radio components to a Container to create a radio button group.

To restrict the user to selecting a single radio button within a group, set the same name attribute for each button.

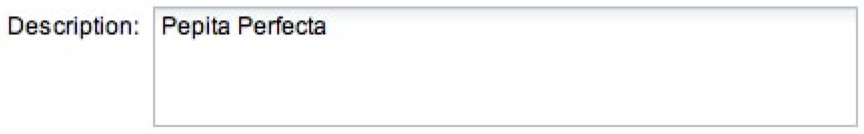
SliderField



SliderField enables a form field to use a slider control, providing an alternative to using NumberField for entry of numeric data.

In Designer, by default useTips is enabled with a SliderField to show the selected value, that the minValue is 0 and maxValue is 100, and the increment is 1.

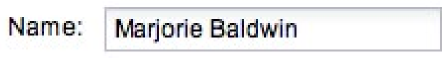
TextArea



TextArea is a specialized TextField that supports multiline text entry for gathering larger amounts of user input.

To let users enter styled text, use the HtmlEditor component instead.

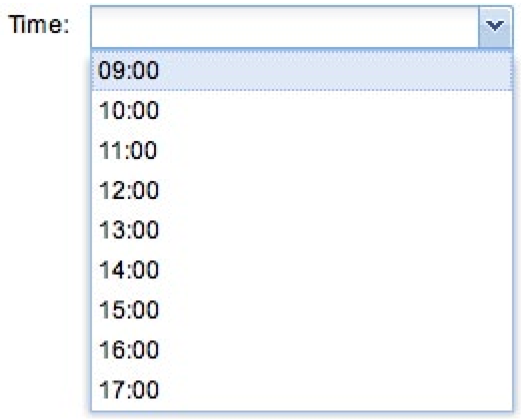
TextField



TextField is a basic text entry field.

In addition to providing a commonly used form element itself, TextField is used as a building block for a number of specialized field types, including NumberField, TextArea, TriggerField, and ComboBox. TextField provides built-in support for data validation. For information about customizing validation behavior, see the ExtJS API Documentation.

TimeField

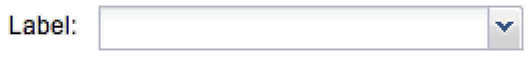


TimeField is a specialized ComboBox for selecting a time. Configure the time range by setting the minValue and maxValue. By default, the list displays times in 15-minute intervals. Configure the interval by setting the increment attribute.

TimeField supports time and date formats specified in the parsing and formatting syntax defined in Date.js. If the input doesn't match the expected format, TimeField and automatically tries to parse it using alternate formats.

By default, the format is set to g:i A, which displays times using a 12-hour clock, for example, 3:15 PM. To use a 24-hour clock (where 3:15 PM becomes 15:15), set the format attribute to H:i

TriggerField



TriggerField wraps a TextField to add a clickable trigger button, like that used for a ComboBox.

By default a TriggerField looks like a ComboBox, but the trigger does not have a default action.

Provide a custom action to Trigger Field by overriding onTriggerclick, or extend TriggerField to implement a custom reusable component. ComboBox extends TriggerField to provide standard combo box behavior. TimeField is also a specialized TriggerField.

Grids Components Overview

To display data from a Store in an interactive table component, use a GridPanel component. GridPanel has built-in support for resizing and sorting columns, as well as dragging and dropping columns. It also supports horizontal scrolling and single and multiple selections. EditorGridPanel adds support for inline editing.

Some applications need full control over how the data is formatted and laid out without the column-management features provided by GridPanel. In these circumstances a ListView is probably a better option than Grid.

Here is an overview of the Ext JS components used to build grids and how to work with them from Designer.

BooleanColumn

A BooleanColumn is a specialized Column for displaying Boolean data in a Grid.

CellSelectionModel

CellSelectionModel is a specialized SelectionModel that can be used to select individual cells within a GridPanel instead of selecting entire rows.

CheckboxSelectionModel

CheckboxSelectionModel is a specialized RowSelectionModel that adds a column of checkboxes to a GridPanel that can be used to select or deselect rows. This is commonly used to enable actions such as move or delete on a selected group of items.

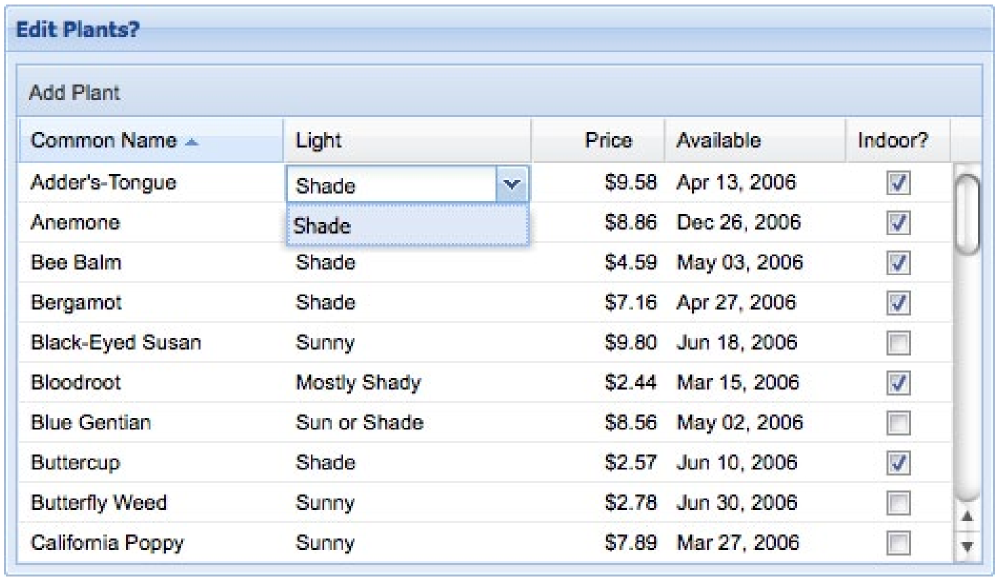
Column

A Column controls how the data in a GridPanel column will be rendered. Column can be used directly to display textual data, and ExtJS provides specialized Column types for displaying Boolean data, numeric data, dates, and templated data.

DateColumn

A DateColumn is a specialized Column for displaying dates in a Grid. Specify the Format attribute to control how the date is formatted. DateColumn supports the date formats specified in the parsing and formatting syntax defined in Date.js.

EditorGridPanel

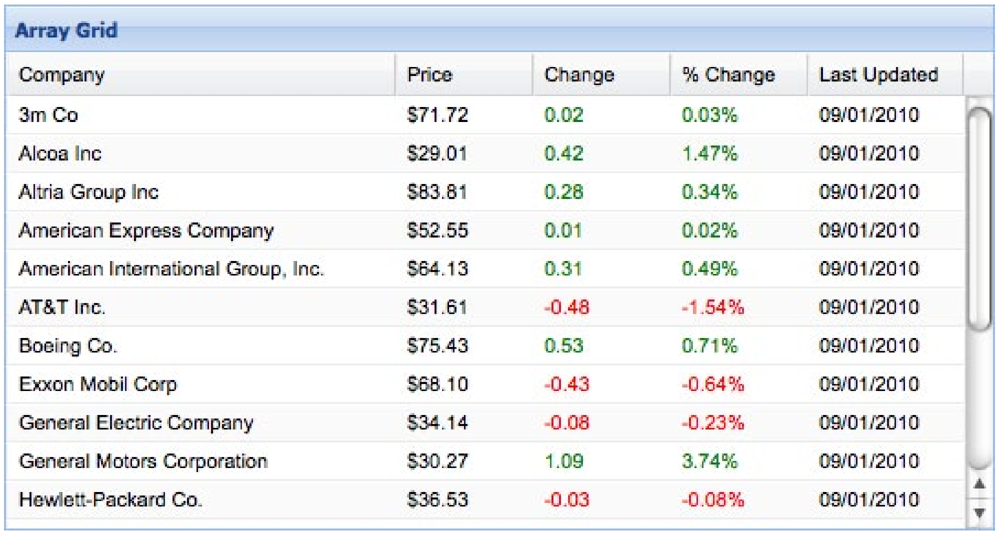

Edit Plants?

Add Plant

Common Name ▲	Light	Price	Available	Indoor?
Adder's-Tongue	Shade	$9.58	Apr 13, 2006	✓
Anemone	Shade	$8.86	Dec 26, 2006	✓
Bee Balm	Shade	$4.59	May 03, 2006	✓
Bergamot	Shade	$7.16	Apr 27, 2006	✓
Black-Eyed Susan	Sunny	$9.80	Jun 18, 2006	☐
Bloodroot	Mostly Shady	$2.44	Mar 15, 2006	✓
Blue Gentian	Sun or Shade	$8.56	May 02, 2006	☐
Buttercup	Shade	$2.57	Jun 10, 2006	✓
Butterfly Weed	Sunny	$2.78	Jun 30, 2006	☐
California Poppy	Sunny	$7.89	Mar 27, 2006	☐

An EditorGridPanel is a specialized GridPanel that enables cells to be edited. For a Column in a GridPanel to be editable, it needs to be configured with an appropriate editor, which is simply the form field that will be used to edit the data.

GridPanel



Array Grid

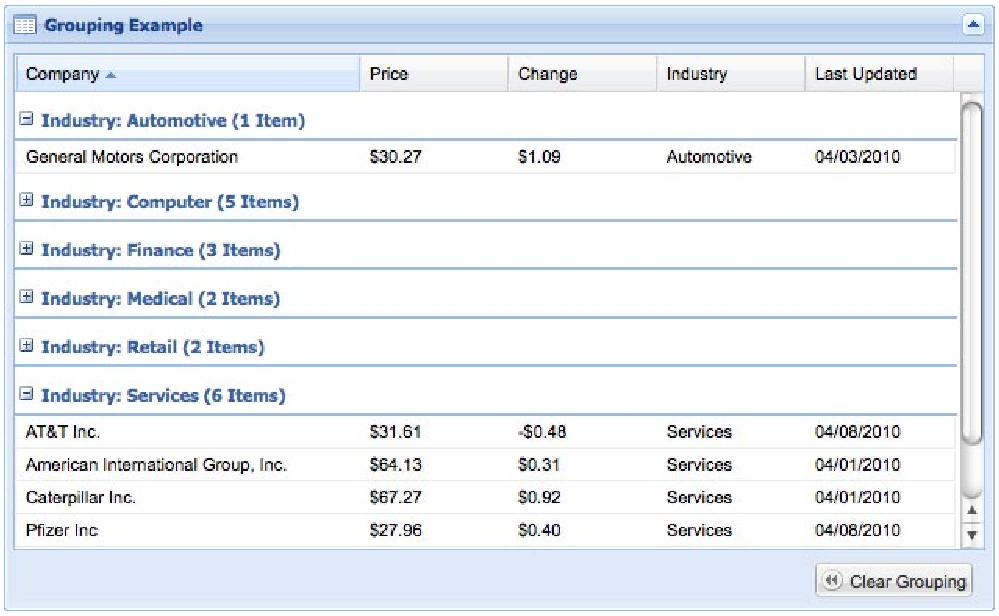
Company	Price	Change	% Change	Last Updated
3m Co	$71.72	0.02	0.03%	09/01/2010
Alcoa Inc	$29.01	0.42	1.47%	09/01/2010
Altria Group Inc	$83.81	0.28	0.34%	09/01/2010
American Express Company	$52.55	0.01	0.02%	09/01/2010
American International Group, Inc.	$64.13	0.31	0.49%	09/01/2010
AT&T Inc.	$31.61	-0.48	-1.54%	09/01/2010
Boeing Co.	$75.43	0.53	0.71%	09/01/2010
Exxon Mobil Corp	$68.10	-0.43	-0.64%	09/01/2010
General Electric Company	$34.14	-0.08	-0.23%	09/01/2010
General Motors Corporation	$30.27	1.09	3.74%	09/01/2010
Hewlett-Packard Co.	$36.53	-0.03	-0.08%	09/01/2010

A GridPanel displays tabular data in rows and columns. The data displayed by a GridPanel is read from a Store. A GridPanel's ColumnModel controls how the data is rendered into the columns of the grid, and Column components encapsulate the configuration information needed to render individual columns.

GridView

A GridView encapsulates the user interface for a GridPanel. Adding a GridView to a Grid-Panel enables access to the grid's user interface elements for creation of special display effects.

GroupingView



Grouping Example

Company	Price	Change	Industry	Last Updated
⊟ Industry: Automotive (1 Item)				
General Motors Corporation	$30.27	$1.09	Automotive	04/03/2010
⊞ Industry: Computer (5 Items)				
⊞ Industry: Finance (3 Items)				
⊞ Industry: Medical (2 Items)				
⊞ Industry: Retail (2 Items)				
⊟ Industry: Services (6 Items)				
AT&T Inc.	$31.61	-$0.48	Services	04/08/2010
American International Group, Inc.	$64.13	$0.31	Services	04/01/2010
Caterpillar Inc.	$67.27	$0.92	Services	04/01/2010
Pfizer Inc	$27.96	$0.40	Services	04/08/2010

A GroupingView is a specialized GridView that adds the ability to group related rows in the grid and expand and collapse the groups. For example, for a grid that displays customer names and addresses, group them by city or state.

To enable grouping, the GridPanel must be configured to use a GroupingStore, which is a specialized store that specifies the grouping field and direction in addition to the data itself.

NumberColumn

A NumberColumn is a specialized Column for displaying numeric values in a Grid. Specify the Format attribute to control how the number is formatted. The formatting string is defined according to Ext.util.Format.number.

RowSelectionModel

RowSelectionModel is the default SelectionModel used for a GridPanel. It supports multiple selections and keyboard selection and navigation. Disable multiple selections by enabling the singleSelect attribute. Disable the moveEditorOnEnter attribute to prevent Enter and Shift-Enter from moving the Editor between rows. See also CellSelectionModel and Check-boxSelectionModel.

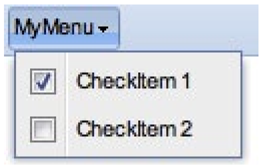
TemplateColumn

A TemplateColumn is a specialized Column that processes a record's data using the specified template to generate the value to display in the Grid. The template is set in the tpl attribute.

Menu Components Overview

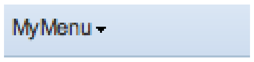
Ext JS provides a set of Menu components that are used to build menus from Designer. To build a menu bar, add a Button for each of menu to a Toolbar and then configure a Menu component for each button. For more information about building menus, see Building Menus. Following is an overview of Ext JS Menu components.

CheckItem



A CheckItem is a specialized MenuItem that includes a checkbox or radio button for toggling a MenuItem on and off. If the group attribute is set, all items with the same group name are treated as a single-select radio button group.

Menu

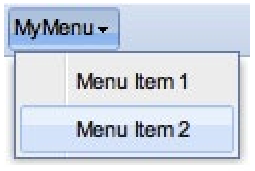


A Menu is a container for a collection of menu items.

To create a menu, add a Menu to a Button component and then add CheckItem, Menu-Item, Separator, and TextItem components to the Menu.

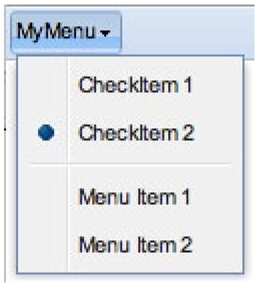
A Menu can contain any type of component although it typically includes only standard menu item components.

Menu Item



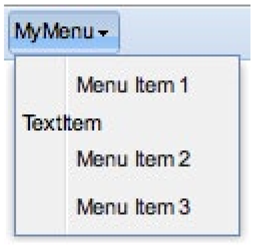
A Menu Item is a standard selectable Menu option.

Separator



A Separator is a divider that can be added to a Menu. A Separator is typically used to separate logical groups of Menu Items.

TextItem



A TextItem is a non-selectable text string that can be added to a Menu. A TextItem is typically used as a header for a group of Menu Items.

Standard Components

Standard components provide the basic building blocks for your UI. The following introduces each of the Ext JS standard components.

BoxComponent

BoxComponent is the basis for all components that need to be sized using a width and height and managed within a layout. It's not usually used directly—instead it's more typical to use a Container or one of the other specialized BoxComponents such as a Button or a Field.

Button

A Button can be an icon, text, or both. You can set a Button's icon attribute to specify an image to use for the button. To display both an icon and text, set the iconCls attribute to specify the CSS class for the label to x-btn-text-icon.

Buttons are also used to build menus—a menu is just a Button component that has a nested Menu component. To create a menu bar, add a Button for each menu to a Toolbar, and then configure a Menu component for each button.

In Designer, configure a menu by clicking a Button's flyout configuration button, or manually drag a Menu component onto the button.

CycleButton

A CycleButton is a specialized SplitButton that contains a group of CheckItems. A Cycle-Button automatically cycles through the items on click—when the user clicks an item, the Button text is replaced with the text of the clicked item.

Label

A Label contains the text that identifies a Component in a form. Typically Label is not used directly. Labels are automatically created for Components and can be set through the field-Label attribute. (For Checkbox fields, set boxLabel to specify the label text that is displayed beside the checkbox.) A Component's label is only displayed when it is rendered by a container that uses FormLayout.

MultiSlider

A MultiSlider is a slider control that supports multiple thumbs. A MultiSlider can be added to any Container and placed either horizontally or vertically. To create a slider with multiple thumbs, specify an array for the values attribute rather than specifying a single value. To use a slider within a Form, use a SliderField.

ProgressBar

A ProgressBar displays the progress of a task. ProgressBar supports two modes: manual and automatic. Manual mode enables explicit updating of a task's progress. When it's important to show progress throughout an operation that has predictable milestones, use manual mode. Automatic mode enables the progress bar to be displayed for a fixed amount of time or to simply be run until it's cleared. To display progress for a timed or asynchronous task, use automatic mode.

Spacer

A Spacer is a BoxComponent used to add a specific amount of white space to a layout.

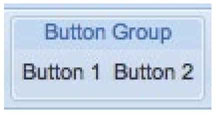
SplitButton

A SplitButton is a specialized Button that has a built-in dropdown arrow that can fire an event separately from the button's click event. CycleButton extends SplitButton to enable the user to cycle through a set of menu items. ComboBox uses a SplitButton to implement standard combo-box behavior.

Toolbar

To create a toolbar that displays a collection of buttons, menus, or other controls, you add a Toolbar component and then add the control components to the Toolbar. Toolbars are typically added to a Panel container, and can be docked at the top (tbar), bottom (bbar), or footer (fbar). Here are the various Toolbar components provided by Ext JS, including Toobar Items--literally items used to enhance Toolbar appearance.

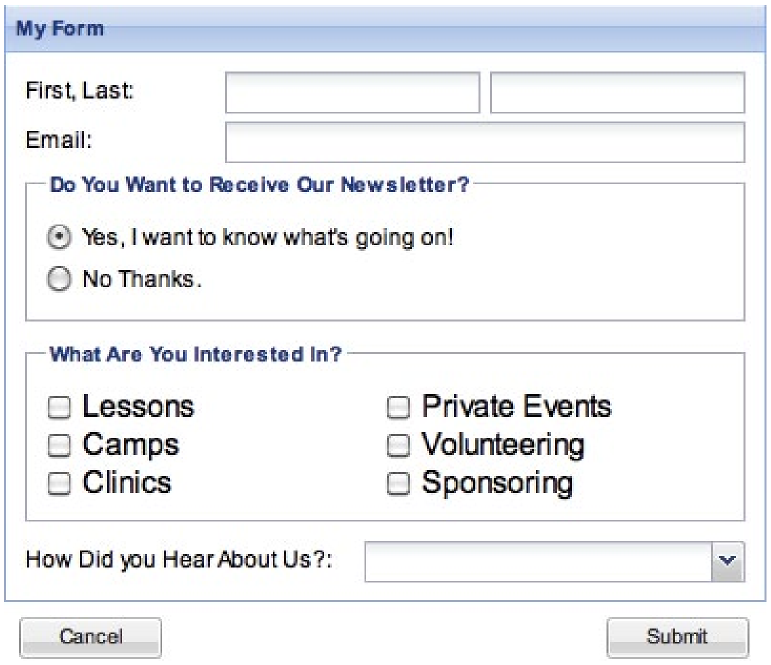
ButtonGroup



A ButtonGroup is a specialized Panel container for a collection of buttons. Conceptually, a ButtonGroup is similar to a menu in that it contains a collection of related items the user can choose from.

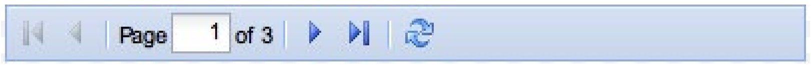
Any type of button can be added to a ButtonGroup—Button, CycleButton, or SplitButton. However, unlike a Menu, a ButtonGroup can't include other control components.

Fill


My Form
First, Last:
Email:
Do You Want to Receive Our Newsletter?
Yes, I want to know what's going on!
No Thanks.
What Are You Interested In?
Lessons
Camps
Clinics
Private Events
Volunteering
Sponsoring
How Did you Hear About Us?:
Cancel
Submit

A Fill is a specialized Spacer that can be added to a Toolbar when some components need to be left-aligned and the remainder to be right-aligned. When a Fill is inserted between Toolbar components, all components following the Fill will be right-aligned. For example, inserting a Fill component between a Cancel Button and a Submit button to a Toolbar that has buttonAlign set to left left-aligns the Cancel Button and right-aligns the Submit Button, as shown above.

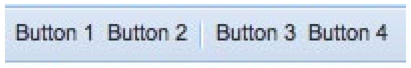
PagingToolbar



A PagingToolbar is a specialized Toolbar that provides controls for paging through large data sets.

PagingToolbar provides automatic paging control by loading blocks of data from a source into a data Store. Set the pageSize attribute to specify how many items to be displayed at a time. PagingToolbar is designed to be used with a GridPanel.

Separator

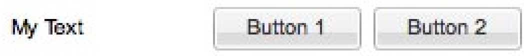


A Separator is a Toolbar Item that places a divider between components in a Toolbar. A Separator is typically used to separate logical groups of buttons in the Toolbar. ButtonGroup can also be used to group buttons.

Spacer

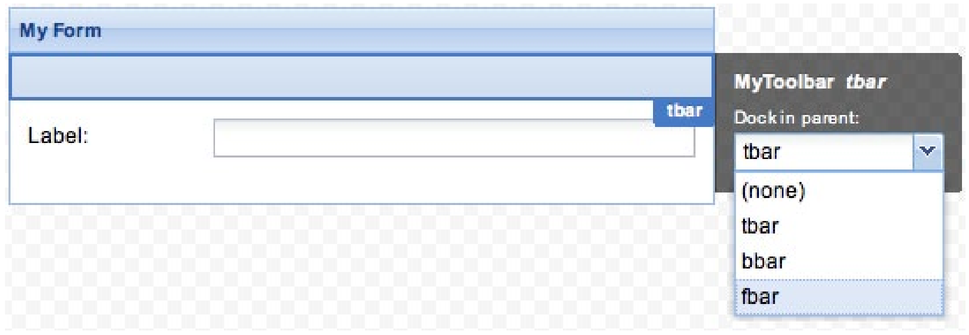
A Spacer is a Toolbar Item used to insert an arbitrary amount of space between components in a Toolbar. To control the amount of space, set the Spacer's width attribute. To left-align some Toolbar components and right-align the rest, use Fill instead of adding a Spacer.

Text



Text is a Toolbar Item used to add a non-selectable text string to a Toolbar as a label or heading.

Toolbar



Toolbar is a container for control components such as buttons and menus, and can be docked at the top (tbar), bottom (bbar), or footer (fbar) of a Panel.

Typical uses for a Toolbar are to display a menu bar at the top of a Panel, or to display buttons at the bottom of a dialog Window or FormPanel.

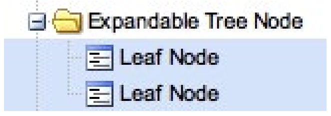
Tree Components

Trees display hierarchical data in a list that can be expanded and collapsed. The basic Ext JS tree building block is the TreePanel container. A TreePanel contains a root node and any number of child nodes. You can build a static tree, or use a TreeLoader to load data into the tree asynchronously. Here are the Tree components provided by Ext JS.

AsyncTreeNode

AsyncTreeNode is a specialized TreeNode that supports asynchronous loading using a Tree-Loader. When a node is expanded, data is loaded from the URL specified in the TreeLoader.

MultiSelectionModel



MultiSelectionModel enables multiple selections within a TreePanel. By default, the selection model for a TreePanel only supports single selections.

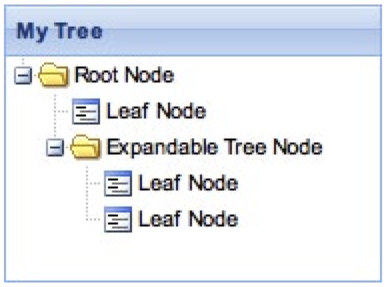
TreeLoader

A TreeLoader enables child nodes to be loaded when the parent is expanded. The child nodes are retrieved from a URL that returns an array of node definition objects.

TreeNode

A TreeNode is an item in a TreePanel component. For a container node, enable the expandable attribute. For a leaf node, enable the leaf attribute.

TreePanel



A TreePanel is a specialized Panel component for displaying hierarchical data in a collapsible list. At a minimum, a TreePanel has a TreeNode that's designated as the root. Additional TreeNodes can be added to the root node.

TreePanel can use a TreeLoader to dynamically load node data. When a node is expanded, TreeLoader automatically retrieves an array of node definitions from the specified URL.

Views Components

Views display dynamic data from a Store; Ext JS provides both a DataView or ListView component for building views. Both components provide complete control over the formatting and layout of the data through an XTemplate. Here are the Ext JS View-related components:

DataView

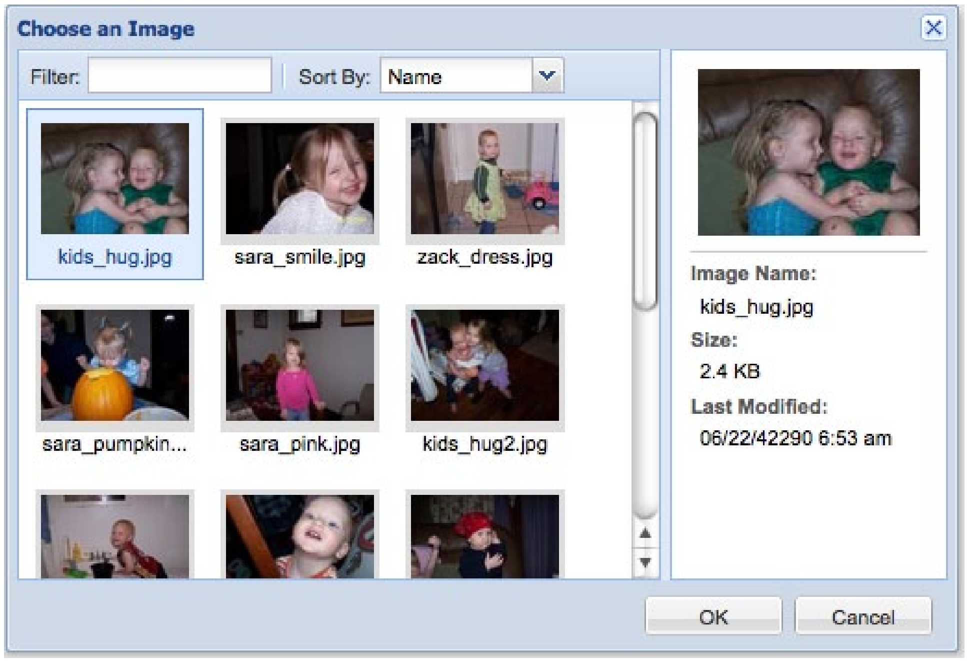
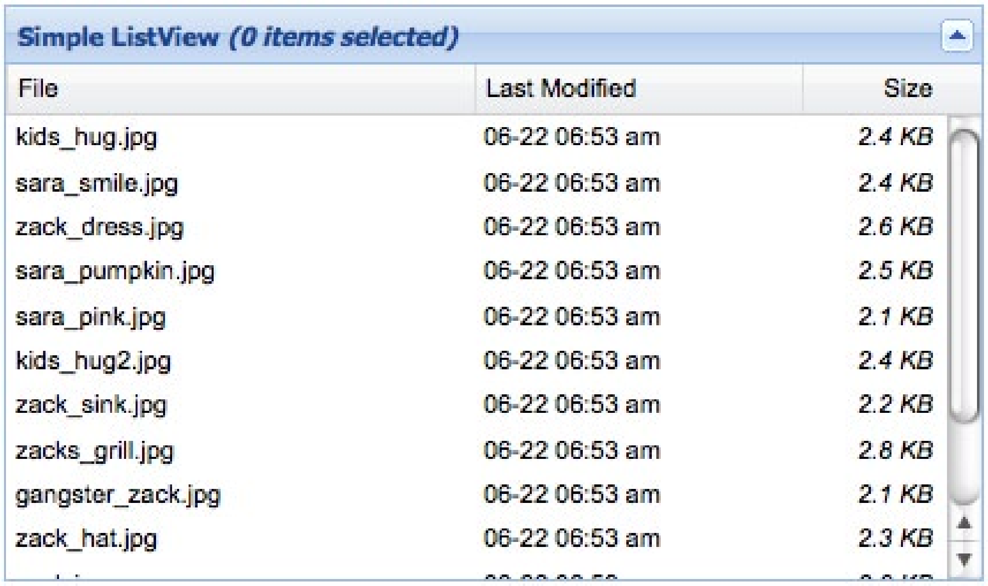


Image Name:
kids_hug.jpg
Size:
2.4 KB
Last Modified:
06/22/42290 6:53 am

A DataView displays data from a Store using an XTemplate to format and lay out the data. When data in the attached store changes, the DataView automatically updates. DataView provides built-in support for common actions such as click and mouseover and supports both single and multiple selections.

Because a DataView is a BoxComponent, it can be managed by its parent's layout. Note that a DataView isn't a Panel component, however. A Toolbar can't be directly attached to a DataView; instead it needs to be wrapped in a Panel and dock it on the Panel.

ListView



Simple ListView *(0 items selected)*

File	Last Modified	Size
kids_hug.jpg	06-22 06:53 am	2.4 KB
sara_smile.jpg	06-22 06:53 am	2.4 KB
zack_dress.jpg	06-22 06:53 am	2.6 KB
sara_pumpkin.jpg	06-22 06:53 am	2.5 KB
sara_pink.jpg	06-22 06:53 am	2.1 KB
kids_hug2.jpg	06-22 06:53 am	2.4 KB
zack_sink.jpg	06-22 06:53 am	2.2 KB
zacks_grill.jpg	06-22 06:53 am	2.8 KB
gangster_zack.jpg	06-22 06:53 am	2.1 KB
zack_hat.jpg	06-22 06:53 am	2.3 KB

A ListView is a specialized DataView that displays data in a tabular format, much like a Grid-Panel.

ListView provides full control over the formatting and layout of the list through an XTemplate and provides built-in support for resizing columns and selecting data. However, it does not implement horizontal scrolling and provides no support for drag and drop or inline editing.

ListView BooleanColumn

A BooleanColumn is a specialized List Column for displaying Boolean data in a ListView.

ListView Column

A Column controls how the data in a ListView column will be rendered. Column can be used directly to display textual data, and ExtJS provides specialized Column types for displaying Boolean data, numeric data, and dates.

ListView DateColumn

A DateColumn is a specialized Column for displaying dates in a Grid. Specify the Format attribute to control how the date is formatted. DateColumn supports the date formats specified in the parsing and formatting syntax defined in Date.js.

ListView NumberColumn

A NumberColumn is a specialized Column for displaying numeric values in a ListView. Specify the Format attribute to control how the number is formatted. The formatting string is defined according to Ext.util.Format.number.

Chapter 4: Forms, Menus, and Trees

One of the main purposes of Designer is to make it easy to build complex UI elements with the components introduced in Chapter 3. This chapter will show how to use Designer to build common UI elements with Ext JS components. It covers how to build forms, menus and how to populate trees with data. Designer simplifies these tasks by enabling immediate visual feedback of the changes made to the UI.

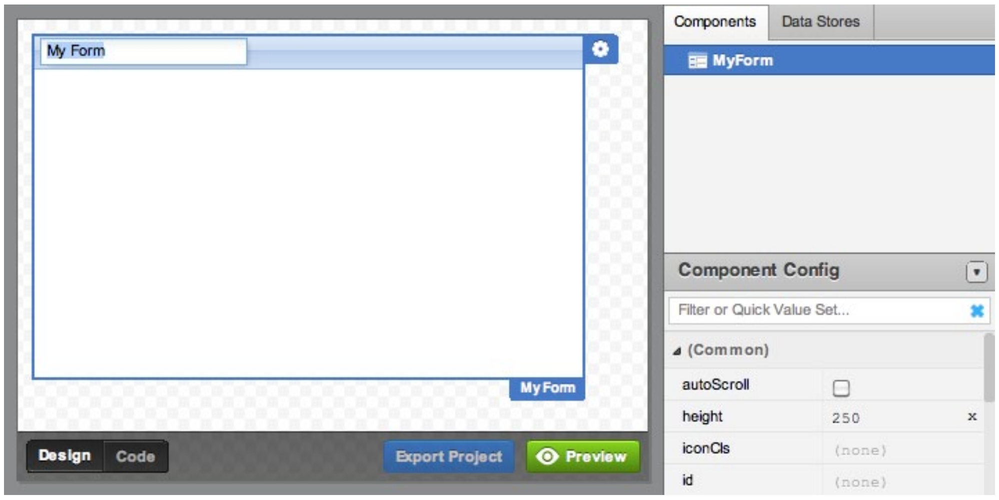
Building Forms

This section shows how to build a simple form in Designer and attach an event handler for form submission. It also shows how to add radio buttons and checkbox groups, arrange multiple fields in a row, create multicolumn forms, and populate a ComboBox using a local store.

Building a Simple Form

To create a form, start with a FormPanel container. A FormPanel automatically displays the labels of any Form Fields that are added to it. A form's submit and cancel buttons are added to a Toolbar that is typically docked in the footer of the FormPanel. Once a form has been laid out in Designer, the project is exported and the generated code can be edited to attach event handlers for the submit and cancel buttons.

To get started, double-click FormPanel in the Toolbox to add a new top-level container to the canvas. Edit the form title by double clicking the default title ('My Form') and resize the panel by dragging its frame. The new project should like the following:

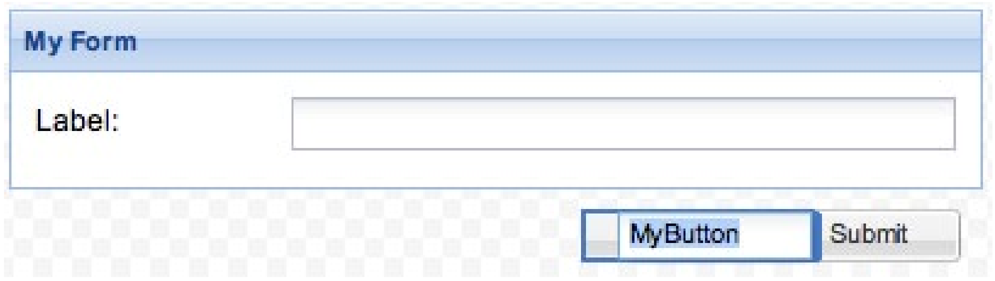


autoScroll	☐
height	250
iconCls	(none)
id	(none)

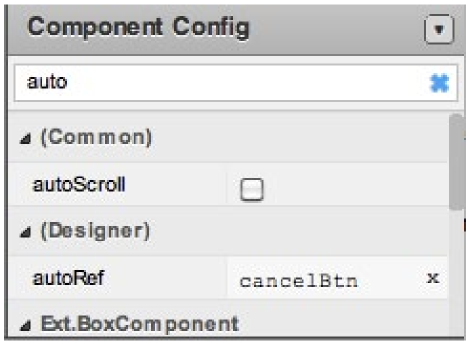
Next, add fields to the FormPanel. For example, double-click TextField to add a text entry field. Edit the field label by double-clicking the existing label.

Now drag a Toolbar into the FormPanel for the form's submit and cancel buttons. Click the flyout config button on the Toolbar and select fbar to dock the toolbar in the FormPanel footer. This will display the buttons below the form.

Drag buttons into the Toolbar for the Submit and Cancel buttons, and edit their labels by double-clicking the default label ('MyButton'), like this:



Now set the autoRef attribute for each button. For example, autoRef: submitBtn and autoRef: cancelBtn, as shown just below. This makes it easy to reference the buttons to attach event handlers when editing the project code after it's been exported.



autoScroll	☐
autoRef	cancelBtn

The next steps are to add events to the project by editing the code generated by Designer. To do this, first save and export the project. Look for the generated .js file for your form; it will have a name like MyForm.js. After the initComponent call, specify the functions to invoke when users click the Submit and Cancel buttons. The code should look something like the following:

```
MyForm = Ext.extend(MyFormUi, {
      initComponent: function() {
      MyForm.superclass.initComponent.call(this);
      this.submitBtn.on('click', this.onSubmitBtnClick, this);
      this.cancelBtn.on('click', this.onCancelBtnClick, this);
      },
});
```

Now, add the implementations for your submit and cancel handler functions to your form class with code that looks like this:

```
MyForm = Ext.extend(MyFormUi, {
 initComponent: function() {
      MyForm.superclass.initComponent.call(this);
      this.submitBtn.on('click', this.onSubmitBtnClick, this);
       this.cancelBtn.on('click', this.onCancelBtnClick, this);
 },
      onSubmitBtnClick: function() {
      // your implementation here!
      }
      onCancelBtnClick: function() {
      // your implementation here!
 }
});
```

Changing the Width of Form Components

Form fields placed in FormPanel are automatically configured with an anchor of 100%. To change the width of a field, clear the anchor attribute and set the width attribute (in pixels).

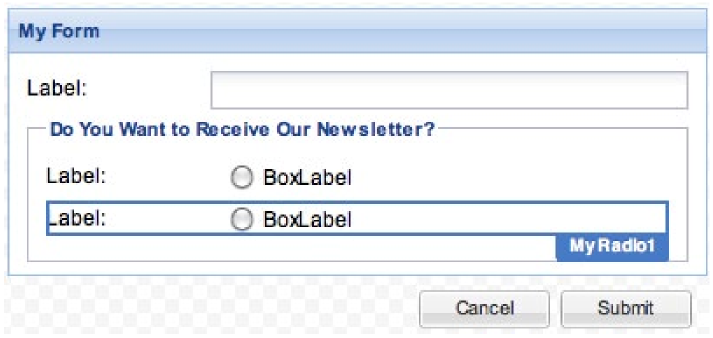
Adding a password field

Forms often provide at least basic security with a password. To create a password field, add a text field to the form and set the field's inputType attribute to password in the Component Config inspector.

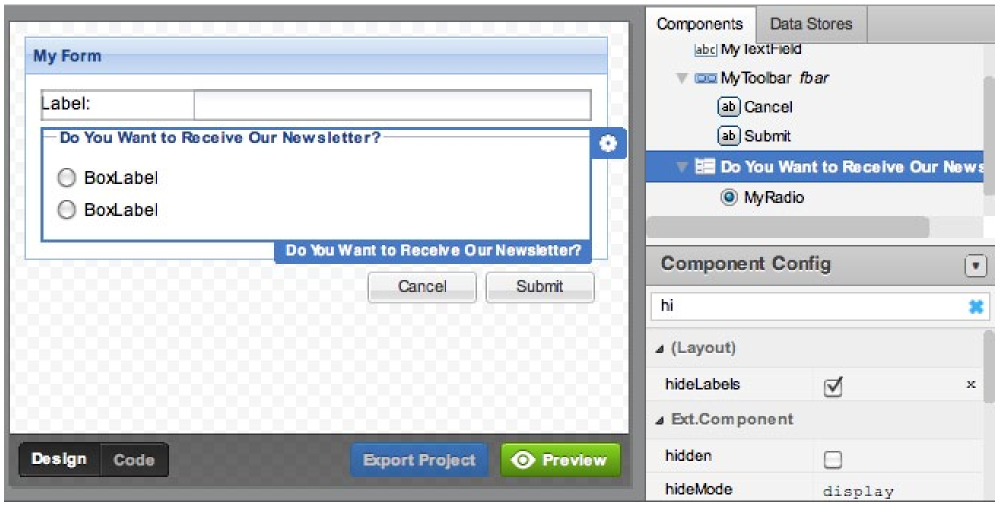
Adding a Group of Radio Buttons or Checkboxes

In Designer, you create a group of radio buttons or checkboxes by adding them to any type of Ext JS container. In this case, use a FieldSet component. To restrict the user to selecting only one of the buttons in a radio button group, set the same name attribute on each of the buttons.

To add radio buttons for a Yes/No selection to the example form, start by dragging a Field-Set container into the FormPanel. Then, drag two Radio Fields into the FieldSet, like this:



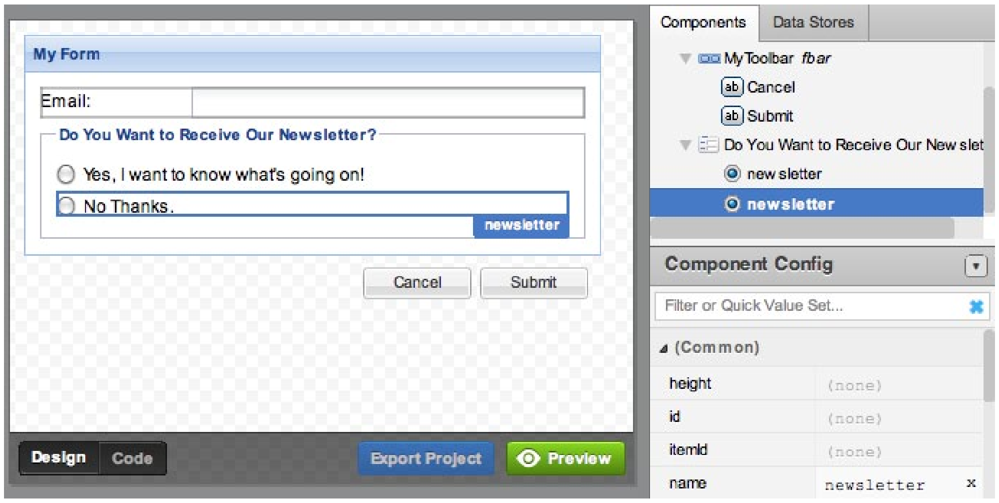
Since the buttons are contained within a FieldSet, the fieldLabels aren't necessary. To hide them, select the FieldSet and enable hideLabels.



hi

(Layout)
hideLabels ☑
Ext.Component
hidden ☐
hideMode display

Next, double click the default label for each Radio ('BoxLabel') to set the text for the Yes and No options. Set the name attribute of each Radio to the same name, for example newsletter, as shown below. This will prevent both buttons from being selected at the same time. To specify one of the buttons as the default, enable its checked attribute.



(Common)
height (none)
id (none)
itemId (none)
name newsletter

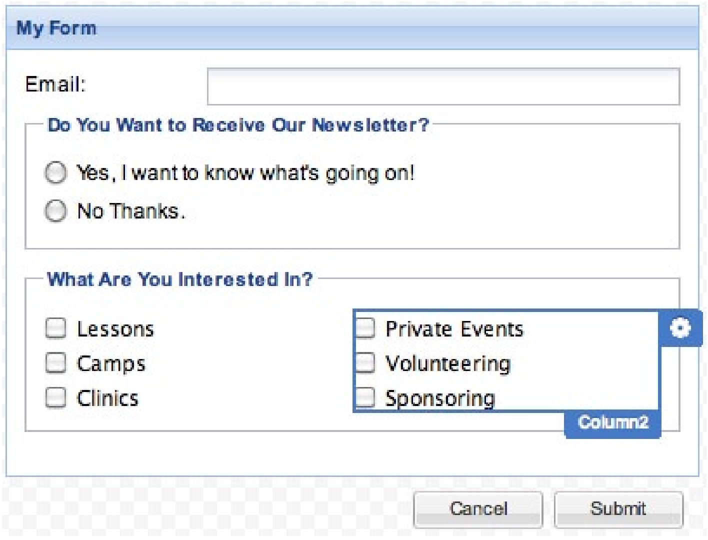
Arranging Fields in Multiple Columns

For a form with a large number of fields, it's possible to arrange them in more than one column to minimize scrolling. To do this, use nested Containers within the FormPanel. Either a particular section or the entire form can be laid out in multiple columns, depending on the relationship between the form fields.

If you have several related checkboxes, for example, they could be arranged in multiple columns to conserve space. To do this, add a FieldSet to the FormPanel for the checkboxes and set the layout of the FieldSet to hbox.

Next add a Container component to the FieldSet for each column and configure the following settings for both. It's easiest to select the column containers from the Component list rather than from the canvas:

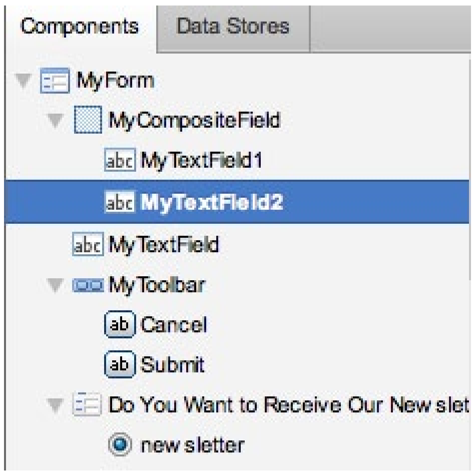
Set layout to vbox and flex to 1and the height to accommodate all needed checkboxes. For example, 60 pixels will accommodate three rows of checkboxes. Now add checkboxes to each column container and set their boxLabel attributes. The form should now look something like the following:


My Form
Email:
Do You Want to Receive Our Newsletter?
Yes, I want to know what's going on!
No Thanks.
What Are You Interested In?
Lessons
Camps
Clinics
Private Events
Volunteering
Sponsoring
Column2
Cancel
Submit

Aligning Fields Horizontally

In some cases a group of related fields in a form need to be aligned on one line, instead of in multiple columns, say for a multipart name field. To do this, use the CompositeField component. Start by adding a CompositeField to the FormPanel. By default, a CompositeField contains a single TextField.

Add a second TextField to the CompositeField, either by dragging it onto the CompositeField component itself or onto the CompositeField item in the Components list. The following shows the second method:

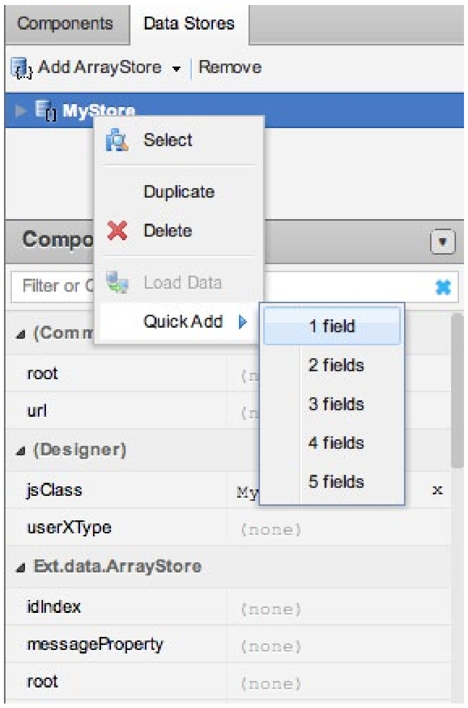


Finally, set the fieldLabel attribute on each contained field and remove the fieldLabel from the CompositeField itself. The CompositeField label will now be a list of the labels for the fields contained in the multipart field.

Populating a ComboBox

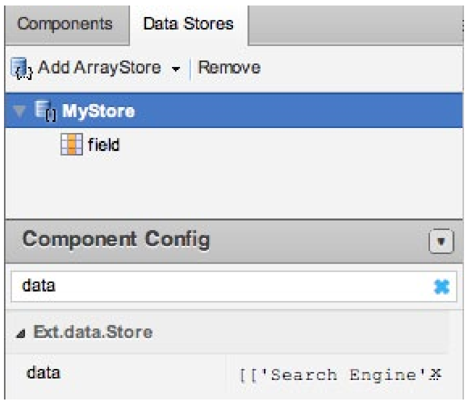
The items listed in a ComboBox are defined in a data Store. The easiest way to set up a Store for a ComboBox is to create a local Array Store, as follows:

Choose Add ArrayStore from the Data Stores tab, then right-click the new Store and choose Quick Add > 1 field, as shown here:



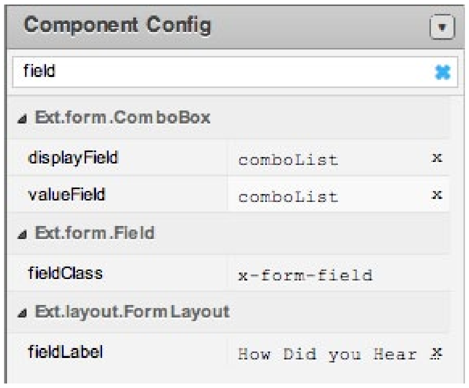
Give the field a name, for example, comboList. Next, specify the list of items to show in the ComboBox as an array of arrays in the Store's data attribute, for example:

```
[['Search Engine'], ['Online Ad'], ['Facebook']].
```



```
data
Ext.data.Store
data        [['Search Engine'
```

To use the local array Store to populate a ComboBox, select the Store using the ComboBox flyout config button. Then configure the ComboBox to specify which field(s) in the Store to use as the displayField and valueField, as shown here:



```
field
Ext.form.ComboBox
displayField    comboList
valueField      comboList
Ext.form.Field
fieldClass      x-form-field
Ext.layout.FormLayout
fieldLabel      How Did you Hear
```

Finally, set the triggerAction attribute to all to display all of the available items in the Store in the drop-down list. Set the ComboBox mode attribute to local.

Building Menus

Any Button can be turned into a menu by adding a Menu component to it and then adding menu items to the Menu component. As with any other Designer project, once the component is complete, it's exported so that event handlers can be added to the generated code.

To create a typical menu bar, start by dragging a Toolbar onto the canvas, then drag buttons onto the Toolbar for each menu. Start with just two buttons, although more can be added if needed. Next, edit the button labels to set the name of each menu, for example, MyApp and Tools.

Now add a Menu component and two menu items to each button. To do this, click the button's flyout config button and select Add to add a Menu component. This also automatically adds a Menu Item to the Menu. Click Add again to add a second item to the menu. The same thing can be accomplished by dragging a Menu onto a button; similarly items like Menu Item, CheckItem, Separator, and TextItem can be dragged onto the component.

Next, give each menu item a name to display in the menu. For example, set the items in the MyApp menu to About and Preferences, and the items in the Tools menu to Import and Export. Then, to make it easy to reference the items in event handlers, set the autoRef attribute of each menu item. For example, set the autoRef for the Export menu item to exportItm.

As with other components, the next step is to save, export, and add events. In the generated .js file created when Designer exports the project, after the initComponent call specify the functions to invoke when the user clicks menu items. The code for this should look like the following:

```
MyToolbar1Ui = Ext.extend(Ext.Toolbar, {
 width: 400,
 initComponent: function() {
      this.items = [
      {
      xtype: 'button',
      text: 'MyApp',
      menu: {
            xtype: 'menu',
            items: [{
                  xtype: 'menuitem',
                  text: 'About'
                  },{
                  xtype: 'menuitem',
                  text: 'Preferences'
                  }]
            }
      },{
      xtype: 'button',
      text: 'Tools',
      menu: {
            xtype: 'menu',
            items: [{
                  xtype: 'menuitem',
                  text: 'Import'
                  },{
                  xtype: 'menuitem',
                  text: 'Export',
                  ref: '../../exportItm'
                  }]
            }
      }];
      MyToolbar1Ui.superclass.initComponent.call(this);
      this.aboutItm.on('click', this.onAboutItmClick, this);
      this.prefItm.on('click', this.onPrefItmClick, this);
      this.importItm.on('click', this.onImportItmClick, this);
      this.exportItm.on('click', this.onExportItmClick, this);
      }
});
```

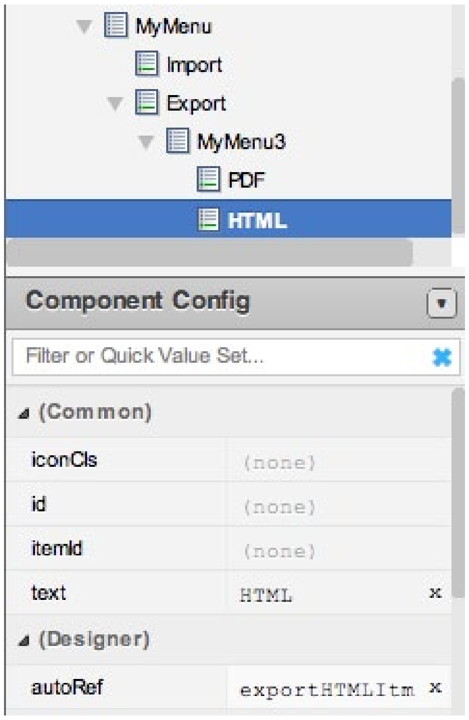
Finally, add the implementations for the handler functions to the project's Toolbar class, as follows:

```
MyToolbar1Ui = Ext.extend(Ext.Toolbar, {
      width: 400,
      initComponent: function() {
      .
      .
      .
      MyToolbar1Ui.superclass.initComponent.call(this);
      this.aboutItm.on('click', this.onAboutItmClick, this);
      this.prefItm.on('click', this.onPrefItmClick, this);
      this.importItm.on('click', this.onImportItmClick, this);
      this.exportItm.on('click', this.onExportItmClick, this);
      }
      onAboutItmClick: function() {
      // your implementation here!
      }
      onPrefItmClick: function() {
      // your implementation here!
      }
      onImportItmClick: function() {
      // your implementation here!
      }
      onImportItmClick: function() {
      // your implementation here!
}});
```

Creating Submenus

To create submenus in Designer, add a Menu component to an existing menu item and then add the submenu items. Attaching an event handler to a submenu item is just like attaching a handler to any other menu item.

For example, to add a submenu to the Export menu item in the Menu Bar example, drag a Menu component onto the Export menu item, either on the canvas or in the Components tab. Next add a Menu Item for each submenu item; for this example add one for PDF and one for HTML. Set the text and autoRef attribute for each submenu item, for example, text: HTML and autoRef: exportHTMLItm, like this:



MyMenu
Import
Export
MyMenu3
PDF
HTML

Component Config

Filter or Quick Value Set...

(Common)

iconCls	(none)
id	(none)
itemId	(none)
text	HTML

(Designer)

autoRef	exportHTMLItm

Populating Trees

When adding a TreePanel to the Canvas, Designer automatically adds a root node and a TreeLoader. To use the TreeLoader to populate the tree, start by setting the URL attribute of the TreeLoader to point to the location from which to retrieve the node definitions. This location is specified relative to the URL Prefix configured in the Project Settings. To view or change the URL prefix, choose Project Settings from the Designer Edit Menu.

Next define the nodes. When a node is expanded, a server request is sent and the child nodes are loaded from the URL specified in the TreeLoader. The response must be a JavaScript Array definition whose elements are node definition objects, like the following:

```
[{
      id: 1,
      text: ‘A leaf Node’,
      leaf: true
 },{
      id: 2,
      text: ‘A folder Node’,
      children: [{
            id: 3,
            text: ‘A child Node’,
            leaf: true
      }]
}]
```

Chapter 5: Component-Oriented Design

Designer makes it easy to use the standard Ext JS component building blocks to assemble a UI. Simply drag a container such as a Viewport or Window onto the canvas and start adding components to it. That's a simple way to start building a basic UI.

An application of any complexity, however, needs to be organized into smaller pieces. This lets the development team employ a more efficient strategy in which the pieces of the application can be designed, implemented, and maintained separately and more easily reused. This chapter focuses on how to undertake such a component-oriented design approach using Designer.

Designer provides two mechanisms for facilitating component-oriented design:

» Components can specifically be added to a project as top-level components. For example, top-level containers can be added to a project for each page or dialog.

» Any child component can be made into a top-level component with the Promote to Class feature. This makes it easy to refactor and reuse components as the application design comes together.

A Designer project can contain any number of top-level components. As we've seen, when Designer exports a project, it generates separate class files for each top-level component.. These include the subclasses defined in the generated .base.js files, which are overwritten every time a project is exported.

In the corresponding .js files, the preconfigured classes are extended so event handlers, additional configurations and custom methods can be implemented with them. The .js file for a top-level component is created the ***first*** time it's exported, but it's not overwritten on subsequent exports. If the .js class is edited, Designer will export a new file with that class name and the previously generated .js file will become obsolete.

Organizing a Designer project as a collection of top-level components can make it easier to continue to develop the project with Designer. It can take a longer time to render large, deeply nested views on the canvas. Building main application views using other top-level components makes it possible to work on those components individually. That way, the whole UI doesn't have to be re-rendered when each piece has been changed. By using linked instances within main application views, it's still easy to view all the pieces in context.

Let's look at the ways to create top-level components using Designer as well as techniques for reusing top-level components.

Adding Top-Level Components

The most basic way to create a top-level component is simply to add one to your project. As we've already seen, Designer provides several ways to do this:

» Without any components selected, double-click a component in the Toolbox.

» Drag a component from the Toolbox to any empty area of the canvas.

» Select New Component from the Component menu.

Remember that Window and Viewport containers can *only* be added as top-level components.

To change the top-level component displayed on the canvas, simply select the component in the Components tab to display it.

Promoting a Component to a Class

Any item in the Components list can be promoted to a class. For example, let's look at Menu components.

Designer currently can only attach Menus to Buttons or other Menu Items - a Menu cannot be added by itself as an individual, top-level component. Instead, the Designer Promote to Class feature needs to be used to convert a Menu to a top-level component.

To do this, create a Button and attach a Menu to it. Then, right-click the Menu in the Components tab and choose Promote to Class. The Menu becomes a top-level component and the Button component contains a link to that class. When the project is exported, Designer generates separate .js and .base.js files for the promoted Menu component.

This technique also works for other components that are designed to be attached to other components, such as Grid columns.

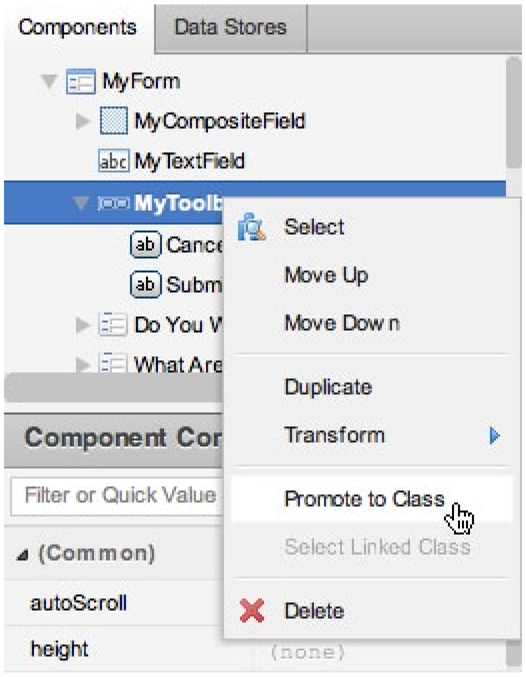
Promoting Child Components

The Promote to Class option can also be used to convert any child component in a Designer project to a top-level component. When a child component is promoted, its place in the component tree is taken by a linked instance of the promoted class. Any changes made directly on the linked instance will override the attributes of the top-level component.

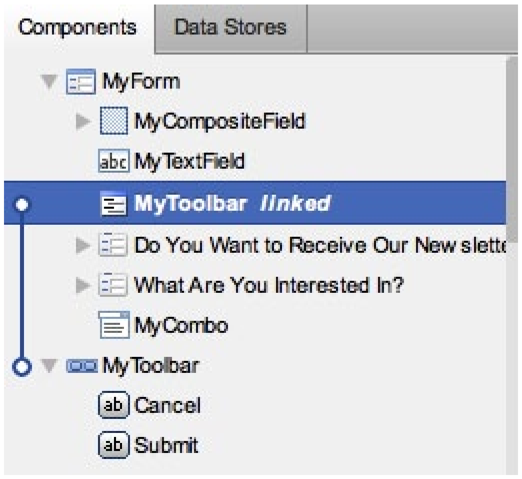
Changes that should be inherited by all linked instances should be made on the top-level component. For information about how to create additional linked instances of a top-level component, see Reusing a Top-Level Component, below.)

Dragging a child component to an empty area on the canvas also makes it a top-level component. However, this *moves* the component to the top level *without* creating a linked instance.

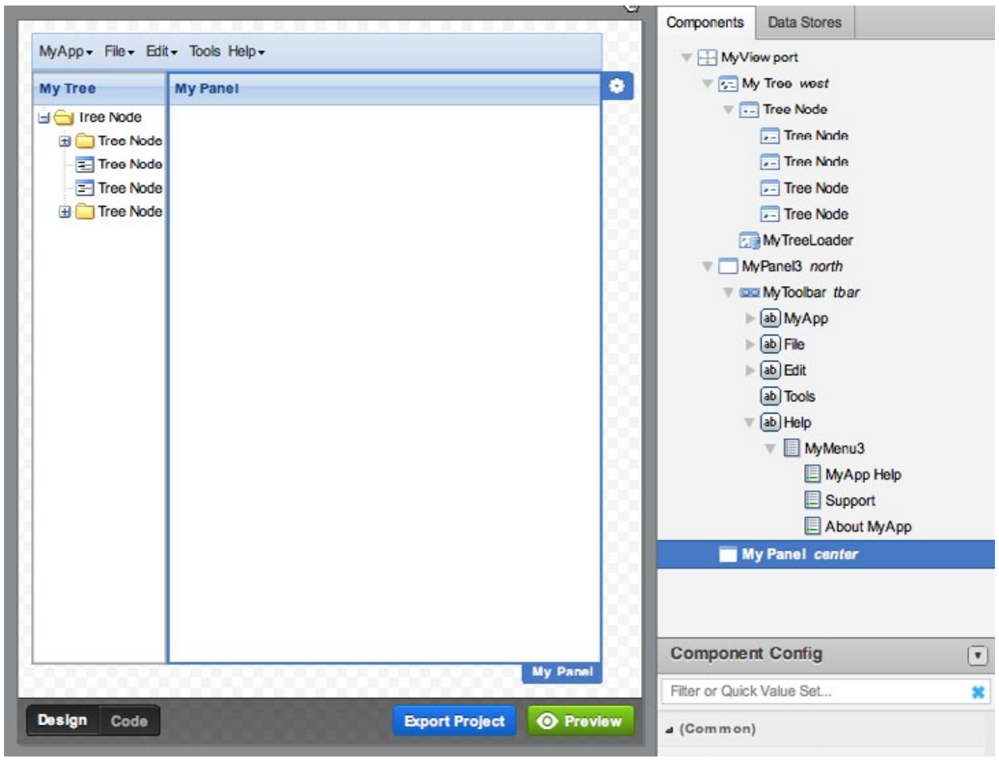
To promote a child component, right-click on the component in the Components tab and select Promote to Class, as shown here:



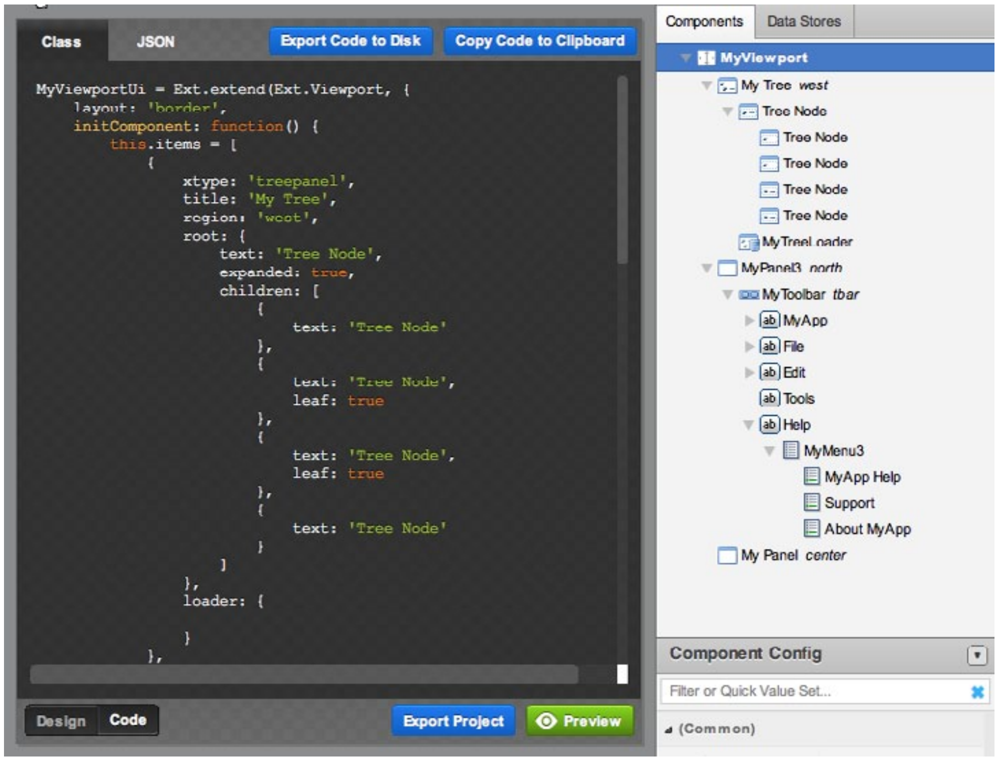
The promoted component and its linked instance are shown in the Components tab, like this:



To take a specific example, a Viewport could be used for a main application page at the start of the project and all the main pieces for the UI could be dragged into the Viewport. In the Designer canvas, the results would look like this:



Switch to the code view for the component by clicking the Code button. It shows that Designer would generate a single class for the Viewport and all of its subcomponents, as illustrated here:

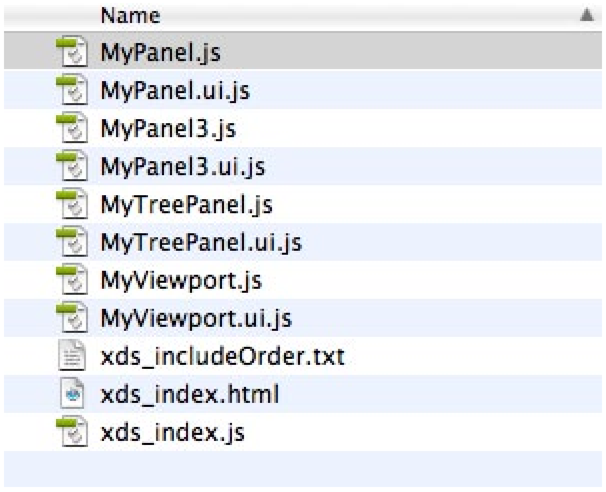


```
MyViewportUi = Ext.extend(Ext.Viewport, {
    layout: 'border',
    initComponent: function() {
        this.items = [
            {
                xtype: 'treepanel',
                title: 'My Tree',
                region: 'west',
                root: {
                    text: 'Tree Node',
                    expanded: true,
                    children: [
                        {
                            text: 'Tree Node'
                        },
                        {
                            text: 'Tree Node',
                            leaf: true
                        },
                        {
                            text: 'Tree Node',
                            leaf: true
                        },
                        {
                            text: 'Tree Node'
                        }
                    ]
                },
                loader: {

                }
            },
```

To have Designer generate separate class files for each of the main components in the Viewport, use the Promote to Class feature in the same way described in the previous section.

Now each of the panels within the Viewport can be developed separately, and when Designer exports the project separate class files are generated for each panel. The directory for the exported project should look like this:

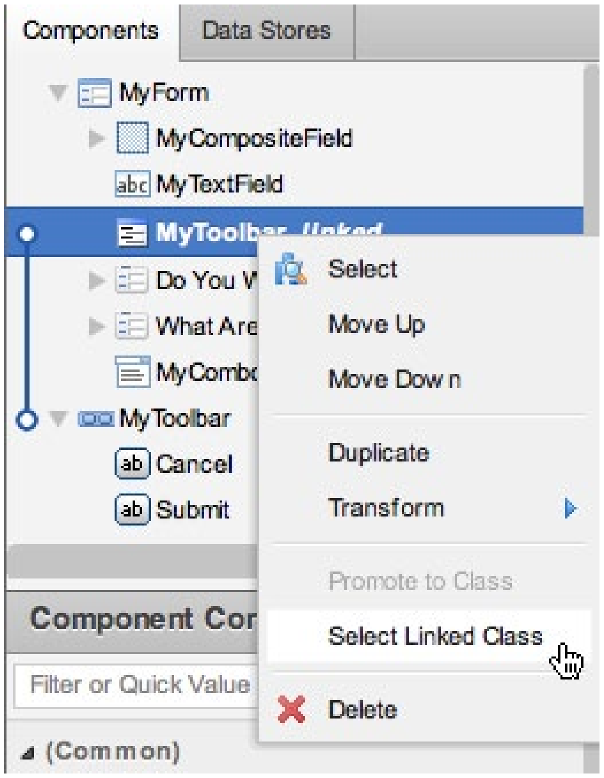


Name
MyPanel.js
MyPanel.ui.js
MyPanel3.js
MyPanel3.ui.js
MyTreePanel.js
MyTreePanel.ui.js
MyViewport.js
MyViewport.ui.js
xds_includeOrder.txt
xds_index.html
xds_index.js

Selecting a Linked Instance's Class

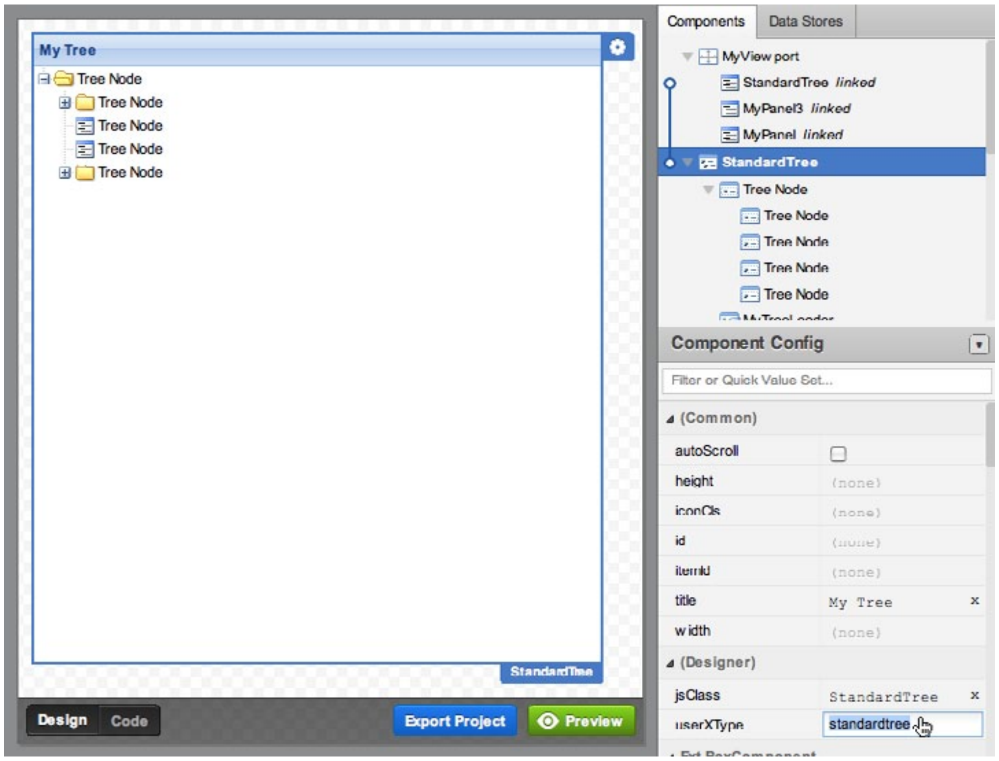
You can select a linked instance to modify the attributes of the instance. To make changes to all instances of a top-level component, select the top-level component.

To select the class associated with a linked instance, right-click the linked instance and choose Select Linked Class, as shown below. Another way to do this is to double-click the linked instance on the canvas or in the Components tab.



Setting a Top-Level Component's xtype and Class Name

When promoting a child component, Designer automatically generates an xtype and class name for the new class. The generated values can be changed and the linked instances will be automatically updated to use the new settings. To change the generated values, select the top-level component and set the user Type and jsClass attributes in the Component Config inspector, as shown here:



(Common)	
autoScroll	☐
height	(none)
iconCls	(none)
id	(none)
itemId	(none)
title	My Tree
width	(none)
(Designer)	
jsClass	StandardTree
userXType	standardtree

Reusing a Top-Level Component

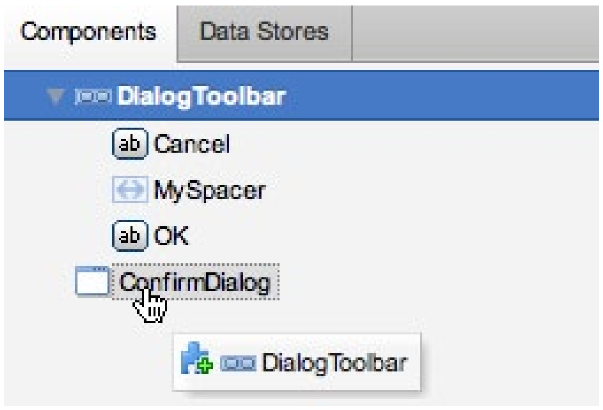
When Designer promotes a component to a class, a linked instance is automatically created where the component used to reside. Designer also makes it easy to reuse components within your project by providing a way to explicitly create a linked instance of *any* top-level component. Whenever a top-level component is dragged into a container, Designer provides three options:

» Moving it to the new location.

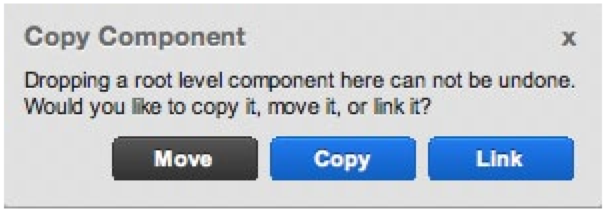
» Creating a copy of it in the new location.

» Creating a linked instance in the new location.

For example, let's look at a dialog that needs to use a standard OK/Cancel toolbar. In the Components tab, drag the toolbar onto the dialog component, like this:

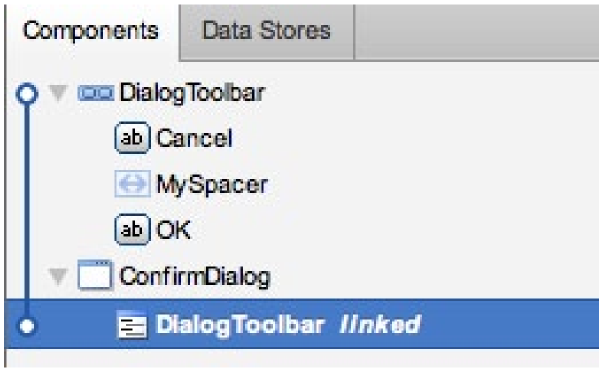


Designer will ask whether to move, copy, or link the component with the following dialog. Click Link.


Copy Component

Dropping a root level component here can not be undone. Would you like to copy it, move it, or link it?

The Components tab identifies the new child component as a linked instance:



Linked instances of components can also be copied. To do so, right-click on the component and choose Duplicate.

Chapter 6: Working With Data Stores

Data Stores provide a client-side data cache for UI components. An Ext JS Data Store retrieves data from a source such as an XML file and makes the data available for display within a UI component such as a GridPanel.

To do this, a Store uses a DataReader to read structured data from a source such as an XML file or JSON packet and creates an array of Record objects that can be accessed by the UI components. The read requests are handled by a DataProxy that knows how to access the source and pass the data to the DataReader.

The general process for setting up a Data Store include the specifying the data format and where it's located, mapping fields in the data source to the fields that will be made available to UI component, and configuring the UI components to use the fields that display the data

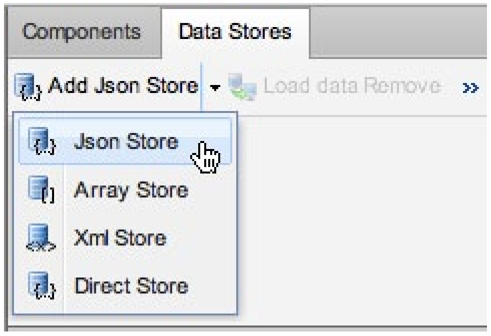
This chapter provides specific techniques for working with Ext JS Data Stores in Designer.

Using Data Stores

To display data in a UI component using a Data Store, first select the type of Store that matches the format of the source data. Next, specify the location where data will read from in the Store's URL attribute and add fields to the Data Store and map them to the source data. Then load the data into the Data Store and configure the UI component to use specific fields from the Data Store. Let's take a closer look at each of these steps.

Choosing a Store Type

Designer provides choice between several types of Data Stores. Each type defines the kind of DataReader and DataProxy that will be used to retrieve and parse data from the source.



Choose between the following:

» Json Store—Retrieves data from a JSON packet using a JsonReader and HttpProxy.

» Array Store—Retrieves data from a local array using an ArrayReader and Memory-Proxy.

» XML Store—Retrieves data from an XML file using XmlReader and HttpProxy.

» Direct Store—Retrieves data from a server-side provider using a JsonReader and DirectProxy.

To add a Store in Designer, select the Data Stores tab and choose the type of store, as shown here:

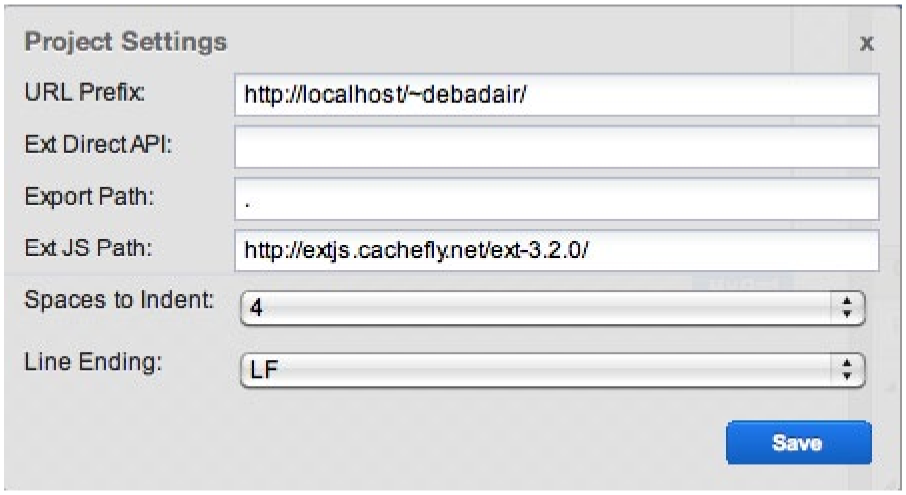
Cross-Domain Requests

An HttpProxy can only retrieve data from within the same domain. This means that a Json Store or XML Store can't be created to get data from a remote source. Creating cross-domain requests requires use of a ScriptTagProxy.

Specifying the Location of the Source Data

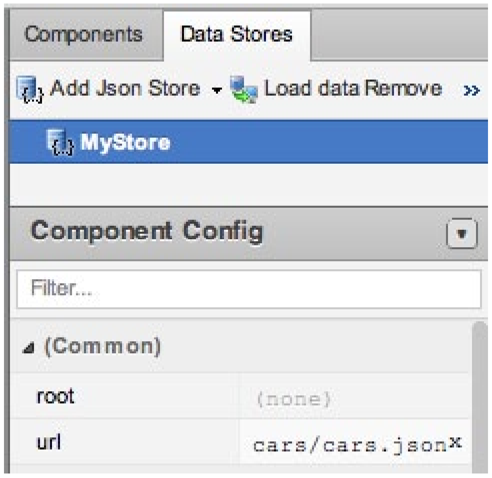
When Designer creates a store, the location of source data needs to be specified. The location specified in a Store's URL attribute is relative to the URL prefix that's specified in the project's preferences.

To set the URL prefix for a project, select Project Settings from the Edit menu. Then, enter the URL prefix that should be prepended to the URL attributes specified for individual components, like this:



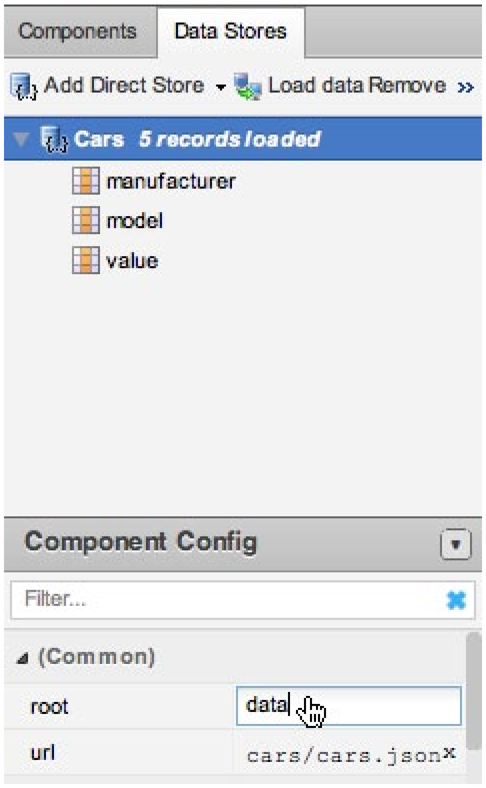
URL Prefix:	http://localhost/~debadair/
Ext Direct API:	
Export Path:	.
Ext JS Path:	http://extjs.cachefly.net/ext-3.2.0/
Spaces to Indent:	4
Line Ending:	LF

Next, specify the location of the source data for a Store by selecting the Store in the Data Stores tab and setting the Store's URL attribute to point to the source data, as shown here:



root	(none)
url	cars/cars.json

For most Store types, the root attribute needs to be set to tell the reader the name of the property from which to read the data, like this:

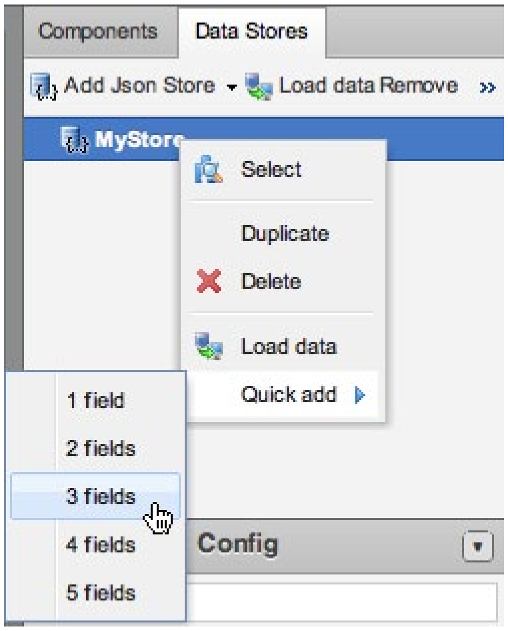


root: data

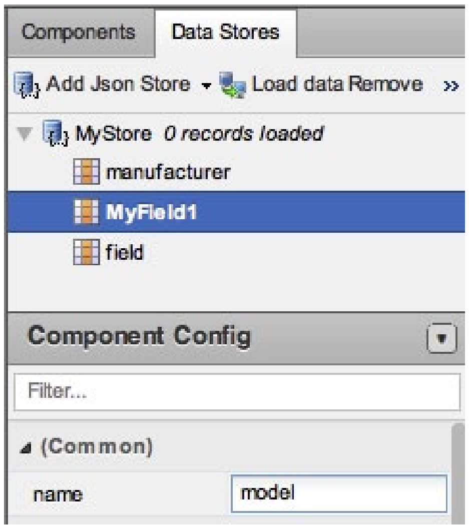
url: cars/cars.json

Mapping Data Fields

The next step is to add a field to the Store for each element that needs to load from the source. Right-click the store in the Data Stores tab and select Add Fields and the number to add to the store, as follows:



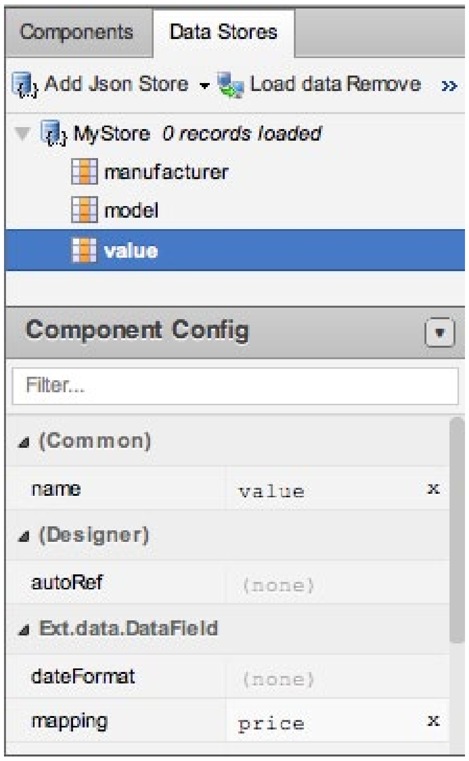
Then, give each field a name by setting its name attribute, as shown here:



name: model

By default, each field in the data source is mapped to a field of the same name in the source data. However, a field can be mapped to any arbitrary source field by specifying the mapping attribute in the field configuration. For example, you could drop underscores, change the capitalization from the way it appears in the source data, or map to a field with a completely different name.

To map a field to a source field of a different name, select the field in the Data Stores tab and set the field's mapping attribute to identify the source data to map to the field, as shown here:



name: value
autoRef: (none)
dateFormat: (none)
mapping: price

The format for a data read from the source field can also be controlled through Designer. For example, to display the contents of a date field in a specific format, specify the dateFormat attribute in the field configuration. This is a PHP-style date formatting string, for more information about this, see Date.

Similarly, the SortType attribute can be set to control how the field is treated when sorting. Do this by specifying one of the predefined SortType functions or by defining and using a custom sort function.

Loading Data into a Store

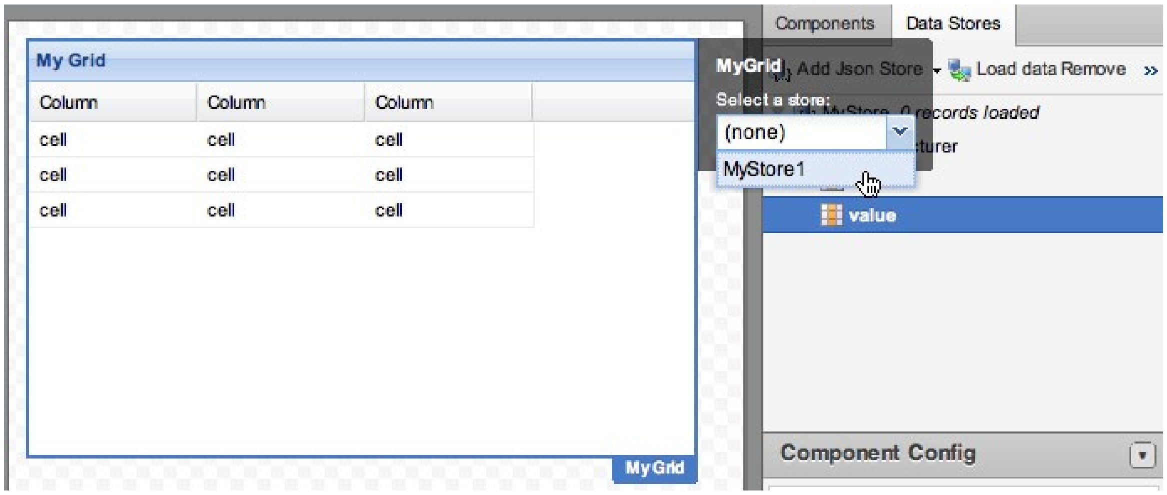
To automatically load data into a Store, enable its autoLoad attribute. This causes the store's load method to be called automatically when it's created.

If the data cannot be read from the source, an error message is displayed that contains the location where Designer expected to find the source data. If autoLoad is not enabled, an application needs to explicitly call load on the store to load data.

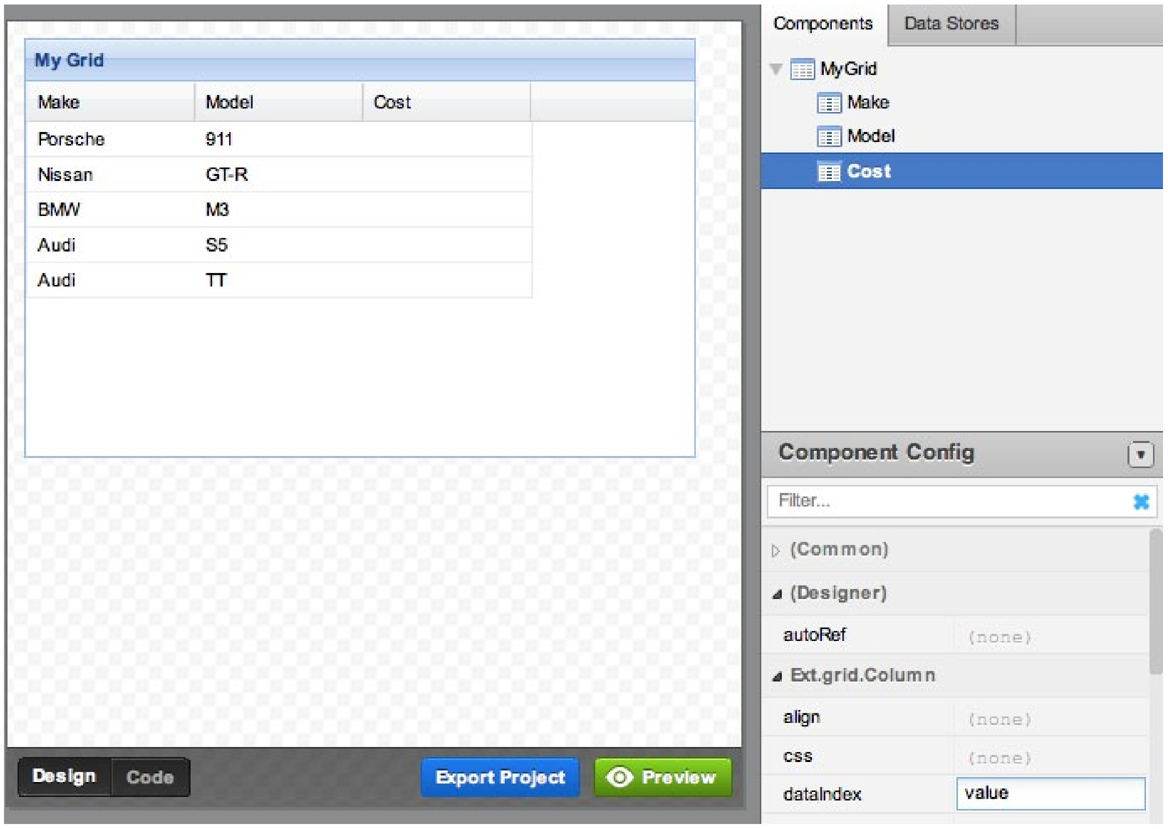
Binding a Store to a UI Component

Once a Data Store has been set up, binding it to a UI component to display the data is easy using the same techniques introduced earlier in this guide. Here's how to do it.

Click the flyout config button on the component and select the Data Store you want to use from the list for that component, in the example shown here, 'MyStore1':



Next, configure the component to use the data from the store. This varies depending on the type of component. For example, for a DataGrid set the dataIndex of each column to the field to be displayed, as shown below in an example from the Car Listings example UI built in Chapter 1. Note that the data is displayed immediately when dataIndex is set.


My Grid

Make	Model	Cost
Porsche	911	
Nissan	GT-R	
BMW	M3	
Audi	S5	
Audi	TT	

autoRef	(none)
Ext.grid.Column	
align	(none)
css	(none)
dataIndex	value

For a ComboBox, specify the itemId and name attributes to correspond to the appropriate fields in the Data Store.

The data should display immediately in the UI component. If it doesn't, do the following:

» Make sure the store can be loaded. The most common problem is incorrectly specifying the path to the store.

» Check the Data Store configuration. Have you defined the fields you're trying to display? If needed, is the root specified correctly?

» Check the component configuration. Have you correctly specified which fields you want to display?

Data Store Examples

The following examples show how to create a Data Store in Designer and connect it to various types of source data.

Using a Json Store

Follow these steps to use a Json Store.

Create the Json file that contains the data to be loaded into the UI. For this example, create a file called customers.json that contains the following data:

```
{
 customers: [
 {name: "Bradley Banks", age: 36, zip: "10573"},
 {name: “Sarah Carlson”, age: 59, zip: “48075”},
 {name: “Annmarie Wright”, age: 53, zip: “48226”}
 ]
}
```

Save the customers.json file on the host specified by the project’s URL Prefix. For example, if the URL prefix is set to http://localhost, make the file available at http://localhost/data/customers.json.

In Designer, go to the Data Stores tab and select Add Json Store. Set the root attribute of the Store to “customers”. This matches the name of the array specified in the Json file that contains the data you want to load.

Set the Store’s idProperty attribute to “name”. Set the Store’s URL attribute to the location of the source file on the host specified by the project’s URL Prefix. Since the Json store has been saved in the data directory on localhost, set the URL attribute to “data/customers.json”.

Right-click the Store component and select Add Fields > 3 fields—one for each of the name:value pairs to access from the elements in the customers array. Set the name of the first field to “name”. Set the name of the second field to “age” and set it’s type to “int”. Setting the type field enables the field to be sorted correctly. Set the name of the third field to “zipcode”. Since this is different from the name used to reference this value in the Json file, the field’s mapping attribute also needs to be set, in this case to “zip”.

With the Store selected, click Load data**.** When the data is successfully loaded from the source, a status message indicating the number of fields loaded is displayed in the Data Stores tab. If the data cannot be loaded, an error message displays that includes the URL from which Designer attempted to load the data.

Now, bind the Json Store to a UI component using the process just discussed and use the fields in the Store to dynamically load data into the component.

Using an Array Store

Here’s how to use an Array Store in Designer.

Create a file that contains the array data to load in the UI. For this example, create a Json file called contacts.json that contains the following data:

```
[
 [“Ace Supplies”, “Emma Knauer”, “555-3529”],
 [“Best Goods”, “Joseph Kahn”, “555-8797”],
 [“First Choice”, “Matthew Willbanks”, “555-4954”]
]
```

Save the contacts.json file on the host specified by the project’s URL Prefix. For example, if the URL prefix is set to “http://localhost”, make the file available at http://localhost/data/contacts.json.

In Designer, go to the Data Stores tab and select Add Array Store. Set the Store’s idIndex property to “0”. This indicates that the first element in each row array (the contact name) should be used as the index.

Set the Store’s URL attribute to the location of the source file on the host specified by the project’s URL Prefix. Since the Json file has been saved in the data directory on localhost, set the URL attribute to “data/contacts.json”.

Right-click the Store component and select Add Fields > 3 fields—one for each element the application needs to reach from the row arrays in the source file. Name the three fields “name”, “contact”, and “phone”.

With the Store selected, click Load data. When the data is successfully loaded from the source, a status message indicating the number of fields loaded is displayed in the Data Stores tab. If the data cannot be loaded, an error message displays that includes the URL from which Designer attempted to load the data.

Finally, bind the new Array Store to a UI component and use the fields in the Store to dynamically load data into the component.

Using an XML Store

Here’s another example, this one showing how to use an XML store.

Create an XML file that contains the data to load into the UI. For this example, create a file called products.xml that contains the following data:

```
<?xml version=”1.0” encoding=”UTF-8”?>
      <Products xmlns=”http://example.org”>
            <Product>
      <Name>Widget</Name>
      <Price>11.95</Price>
      <ImageData>
      <Url>widget.png</Url>
      <Width>300</Width>
      <Height>400</Height>
      </ImageData>
</Product>
<Product>
      <Name>Sprocket</Name>
      <Price>5.95</Price>
      <ImageData>
      <Url>sc.png</Url>
      <Width>300</Width>
      <Height>400</Height>
      </ImageData>
</Product>
<Product>
      <Name>Gadget</Name>
      <Price>19.95</Price>
      <ImageData>
      <Url>widget.png</Url>
      <Width>300</Width>
      <Height>400</Height>
      </ImageData>
</Product>
      </Products>
```

Save the products.xml file on the host specified by the project's URL Prefix. For example, if the URL prefix is set to "http://localhost", make the file available at http://localhost/data/products.xml.

In Designer, go to the Data Stores tab and select Add XmlStore. Set the Store's URL attribute to the location of the source file on the host specified by the project's URL Prefix. Since the XML file has been saved to the data directory on localhost, set the URL attribute to data/products.xml.

Set the record attribute to the name of the XML element that contains the data to load, in this case Product.

Right-click the Store and select Add Fields > 3 fields—one for each of the subelements to access for each Product. Set the name of the first field to "name" and its mapping attribute to "Name". The mapping is case sensitive and must match the element name.

Set the name attribute of the second field to "price", its mapping attribute to "Price", and its type to "float". Set the name attribute of the third field to "imageUrl" and the mapping attribute to "ImageData > Url". Note that this uses a DomQuery selector to access the Url subelement of ImageData.

Make sure the Store is selected and click Load data. When the data is successfully loaded from the source, a status message indicating the number of fields loaded is displayed in the Data Stores tab. If the data cannot be loaded, an error message displays that includes the URL from which Designer attempted to load the data.

Finally, bind the new Store to a UI component and use the fields in the Store to dynamically load data into the component.

We hope this has helped you learn how to use Sencha Ext Designer.

For additional resources and materials, please visit www.sencha.com/learn/