Ext JS 4 Layouts Guide

David Feinberg

Pre-Release Content

All the content in this guide is based on a early pre-release version of Ext JS 4. The concepts and code examples presented in this guide may not be applicable to the final release.

Table of Contents

List of Figures

Ext JS Layout Guide

All the content in this guide is based on a early pre-release version of Ext JS 4. The concepts and code examples presented in this guide may not be applicable to the final release.

Overview

This guide provides a detailed walk-through of the layout system built-into Ext JS 4. Layouts control the size and position of user interface components within an Ext JS application. All the Container Layout classes included in the Ext JS library will be explored along with examples showing you how to use them. To make the most of this guide you should already have a copy of the Ext JS library, know how to set up a basic application, and have some familiarity using Containers and Panels.

The examples in this guide were written with a preview version of Ext JS 4 however all of the concepts apply to both Ext JS 3 and 4.

Box Model Layout vs Sencha Layouts

If you have a background developing web applications with hand coded HTML/CSS then you're already familiar with the traditional box model layout system. Within the box model DIV tags along with SECTION tags in HTML 5 become your primary means of containing items on an HTML page. Once the markup representing your application is in place CSS controls the layout and positioning of each component on the page. This is when the fun usually begins as you start wrestling with margins, padding, floats, as well as the inevitable set of CSS hacks you need to master in order to bring your interface to life on as many web browsers as possible.

Standard CSS layout features were born from traditional print layout techniques that worked great for text and images and but fall far short of helping developers layout out modern web applications.

Ext JS layouts have been specifically designed for JavaScript application developers using familiar patterns from desktop and mobile interface development. This approach abstracts the traditional box model allowing you to configure layouts directly from JavaScript with enough flexibility to manually adjust CSS parameters as needed.

What's a Sencha Layout?

Layouts control the size and position of the components within an Ext JS or Sencha Touch application. Each layout class determines the layout-related configuration properties you can set on individual interface components. Layout classes in Sencha Libraries fall into two class hierarchies *Component Layouts* and *Container Layouts*.

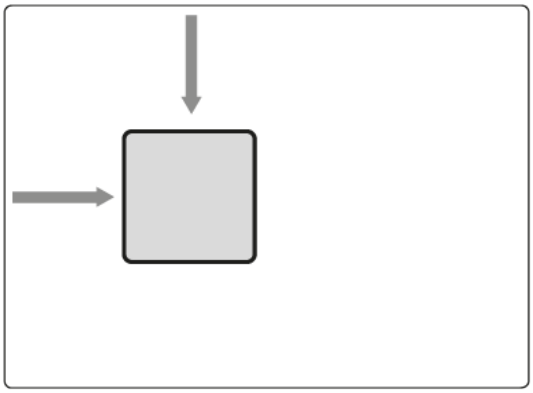
- A Component Layout controls the size and position of an individual Component, such as a Form Field or Button but has no effect on any of the child components that might be contained inside.
- A Container Layout manages how child components within a Container are arranged. A Container is a component designed to hold other user interface components.

Overview of Container Layouts

This section provides an overview of all the available Container Layouts built-into Ext JS.

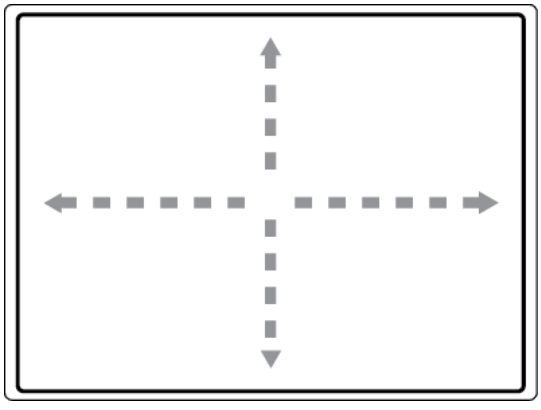
- Absolute
- Anchor
- Auto
- Border
- Card
- Column
- Fit
- HBox
- Table
- VBox

Absolute



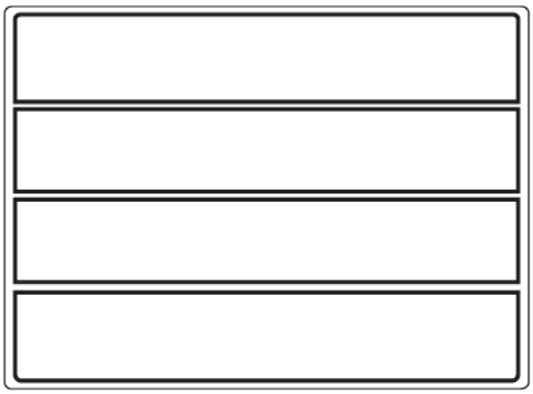
The AbsoluteLayout positions components inside Containers using relative x and y coordinates as configuration options.

Anchor



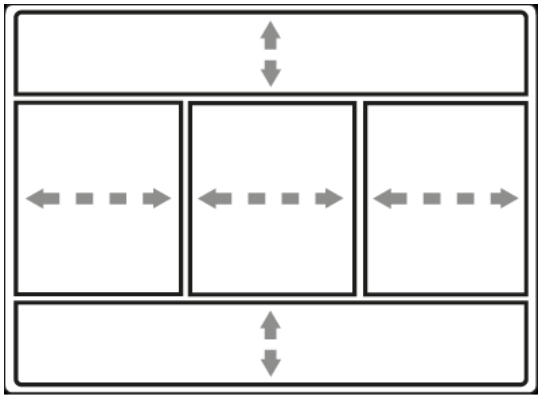
AnchorLayout sizes components relative to the dimensions of the containing component. If the Container is resized, all anchored items are automatically re-sized according to their anchor configuration options.

Auto



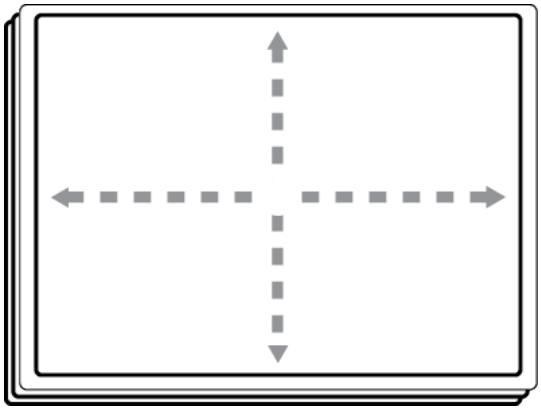
AutoLayout is the default layout manager used by Container components when a specific layout hasn't been configured. This layout works like the *normal flow* of an HTML document arranging components vertically one after another starting at the top of the Container.

Border



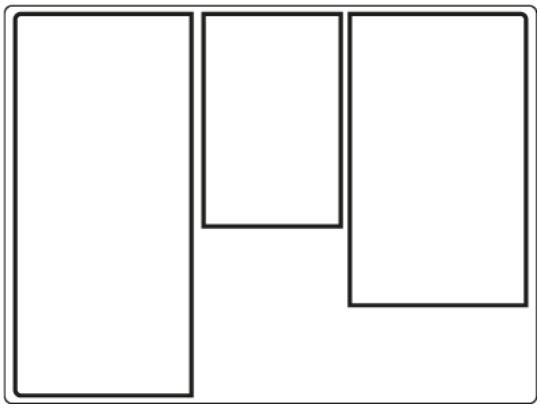
BorderLayout is a multi-pane, application-oriented interface layout that supports multiple nested panels, automatic splitter bars between regions and built-in support for expanding and collapsing regions.

Card



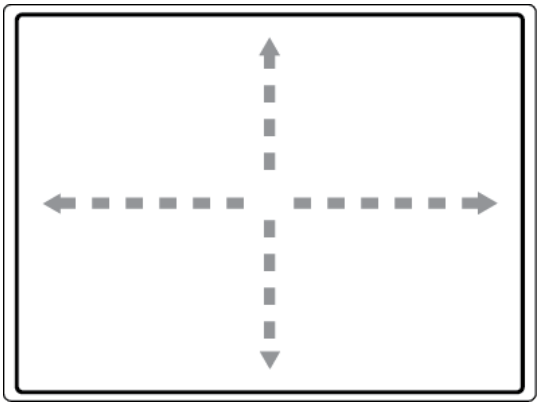
CardLayout manages multiple child components in a Container while only displaying one component at a time. This layout is commonly used for multi-step wizard style interfaces and carousels.

Column



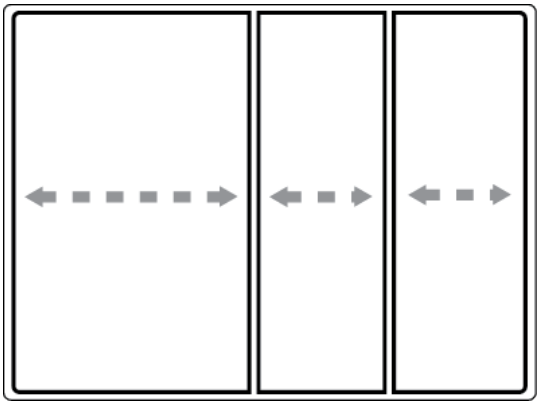
ColumnLayout is designed for creating structural layouts in a multi-column format where the width of each column can be specified as a percentage or fixed width, but the height is allowed to vary based on the content.

Fit



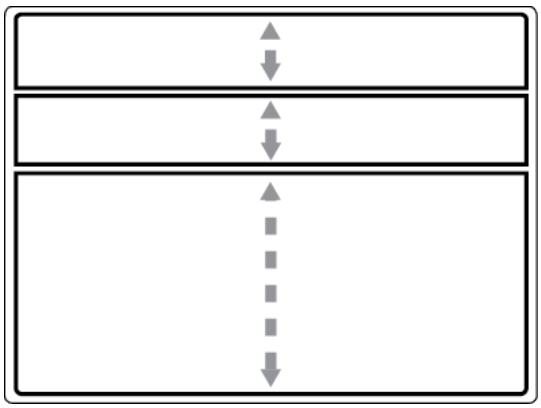
FitLayout is designed to manage a single child component so it always fills the available space in the Container. FitLayout will automatically resize the child component when the Container is resized.

HBox



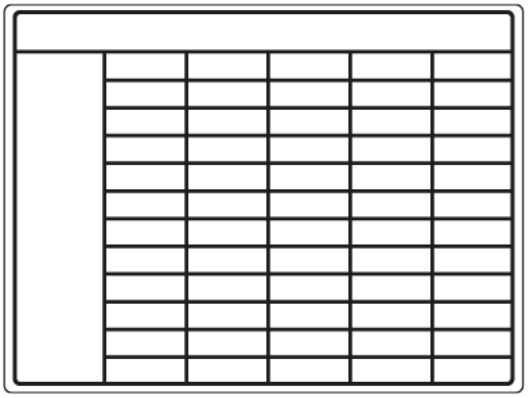
HBoxLayout arranges a Container's child components horizontally (left-to-right). Additional configuration options can be set to automatically manage the alignment and sizing of child components.

VBox



VBoxLayout arranges a Container's child components vertically (top-to-bottom). Additional configuration options can be set to automatically manage the alignment and sizing of child components.

Table



This layout allows you to easily render content into an HTML table. You can specify the number of columns in your table and use the built-in rowspan and colspan features to create complex table layouts.

Examples of Container Layouts

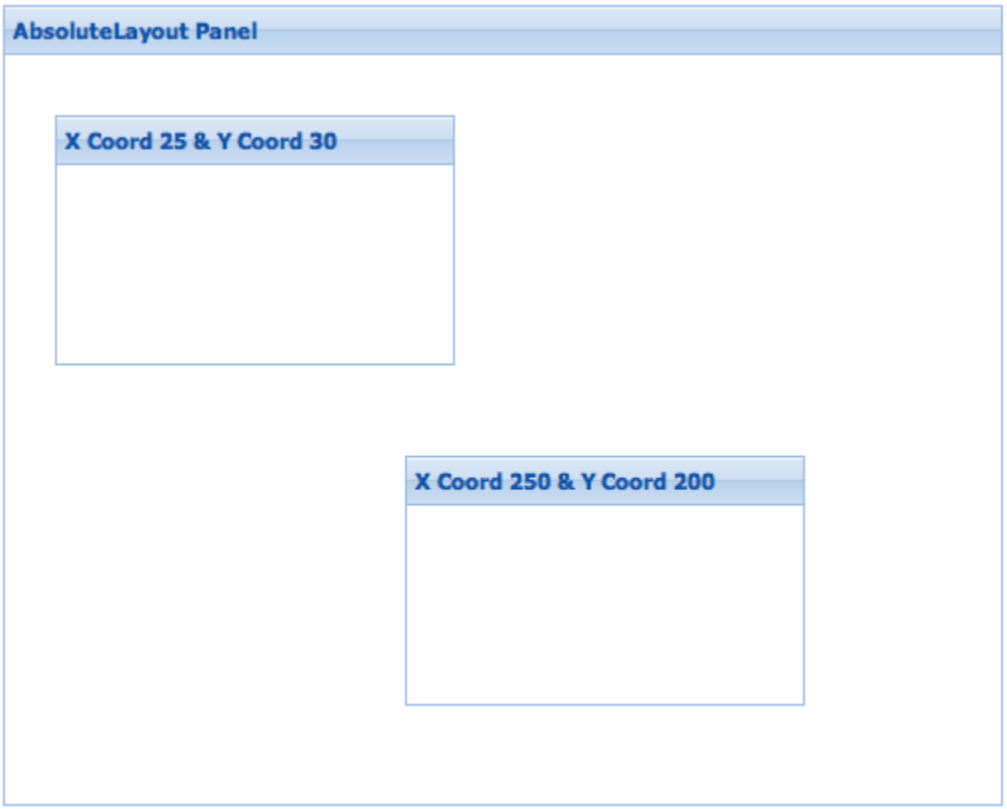
In this section we'll take a look at simple examples to demonstrate what each Container Layout looks like and go over the primary configuration options to help you start using them in your applications.

AbsoluteLayout in Action

Ext.layout.AbsoluteLayout allows you to arrange components by setting x and y coordinates. These coordinates are relative to the top left corner of the containing component. Their positions will automatically be re-calculated as the containing component is resized or moved.

Absolutely positioned components can also use the *anchor* configuration option along with AnchorLayout values to automatically re-calculate their width and height based on the size of the containing component.

Figure 1. Two Inner Panels Absolutely Positioned in a Containing Panel




```
new Ext.panel.Panel({                          // 1
 width: 500,
 height: 400,
 title: "AbsoluteLayout Panel",
 layout: 'absolute',
 renderTo: document.body,
 items: [{
```

```
    xtype: 'panel',
    title: 'X Coord 25 & Y Coord 30',
    width: 200,
    height: 125,
    x: 25,                                  // 2
    y: 30
   },{
    xtype: 'panel',
    title: 'X Coord 250 & Y Coord 200',
    anchor: '40% 30%',                      // 3
    x: 250,
    y: 200
   }]
  });
```

1. A new *Ext.panel.Panel* is setup with the *layout* configuration option set to *absolute* (shorthand for specifying the *Ext.layout.AbsoluteLayout* class). Although not required for an AbsoluteLayout this example uses a fixed width and height to control the size of the main Panel.

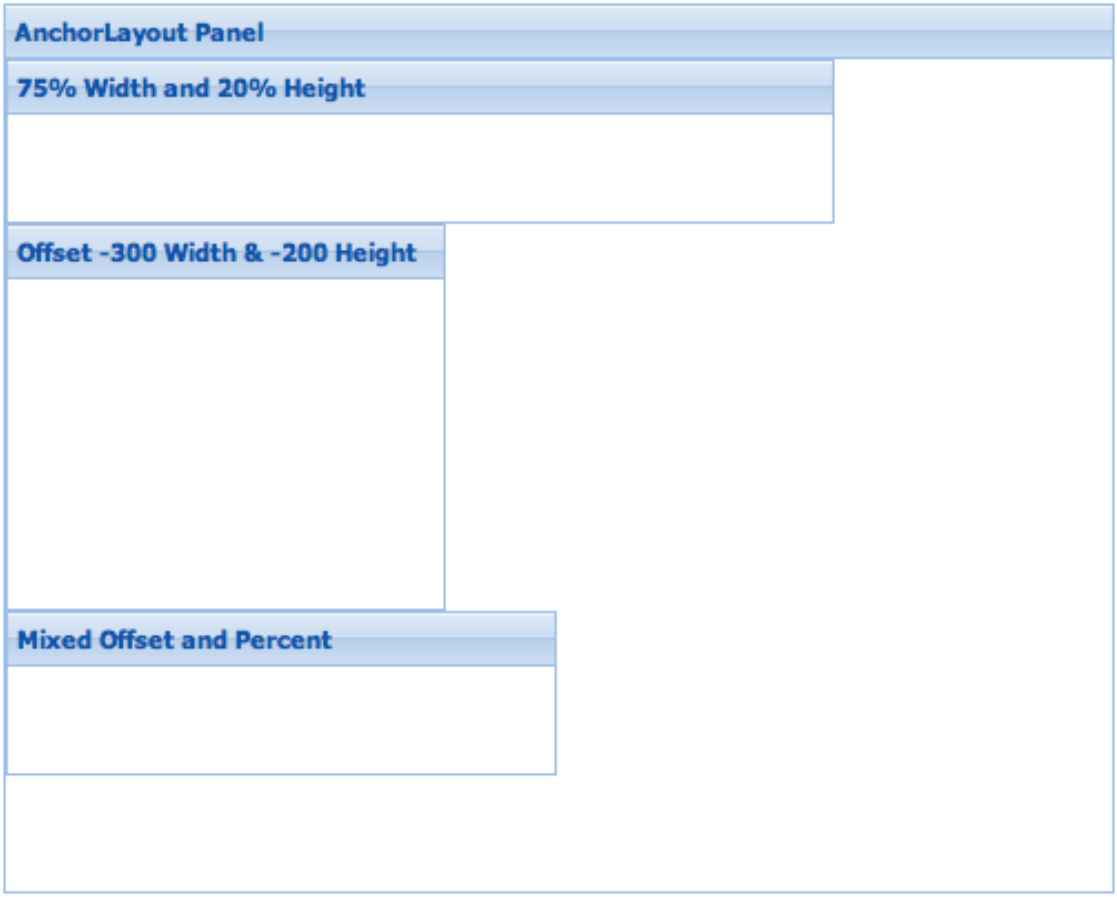
2. The first child Panel is positioned with a *x* coordinate of *25* and a *y* coordinate of *30*. With these settings the initial child Panel will remain 25 pixes to the right and 30 pixels down from the top left corner of the containing Panel.

3. The second child Panel uses the *anchor* configuration option to set its width to *40%* and height to *30%* of the containing Panel's dimensions. Its also absolutely positioned with its *x* coordinate set to *250* and *y* coordinate set to *200*. With this configuration the Panel will remain 250 pixels to the right and 200 pixels down from the top left corner of the containing Panel while always maintaining a relative width and height.

AnchorLayout in Action

Ext.layout.AnchorLayout allows you set the width and height of components relative to the size of their containing component. Anchor values can be specified as integers, percentages or a combination of both. When the containing component is resized all anchored components will be resized according to these values.

Figure 2. Three Inner Panels Anchored to a Containing Panel




```
new Ext.panel.Panel({                                  // 1
 width: 500,
 height: 400,
 title: "AnchorLayout Panel",
 layout: 'anchor',
 renderTo: document.body,
 items: [{
  xtype: 'panel',
  title: '75% Width and 20% Height',
  anchor: '75% 20%'                                    // 2
 },{
  xtype: 'panel',
  title: 'Offset -300 Width & -200 Height',
  anchor: '-300 -200'                                  // 3
```

```
  },{
   xtype: 'panel',
   title: 'Mixed Offset and Percent',
   anchor: '-250 20%'                                  // 4
  }]
});
```

1. A new *Ext.panel.Panel* is created with the *layout* configuration option set to *anchor* (shorthand for *Ext.layout.AnchorLayout* class). Although not required for AnchorLayout this example uses a fixed width and height to control the size of the main Panel.

2. The first child Panel is anchored to the containing Panel using two percentage values. The first value specifies a width of *75%* causing this child Panel to always take up *75%* of the parent Panel's width. The second value specifies a height of *20%* causing the child Panel's height to remain at 20% of the parent Panel's height. With these settings the initial width of the child Panel is 375 pixels (500 * 75%) and the initial height is 80 pixels (400 * 20%).

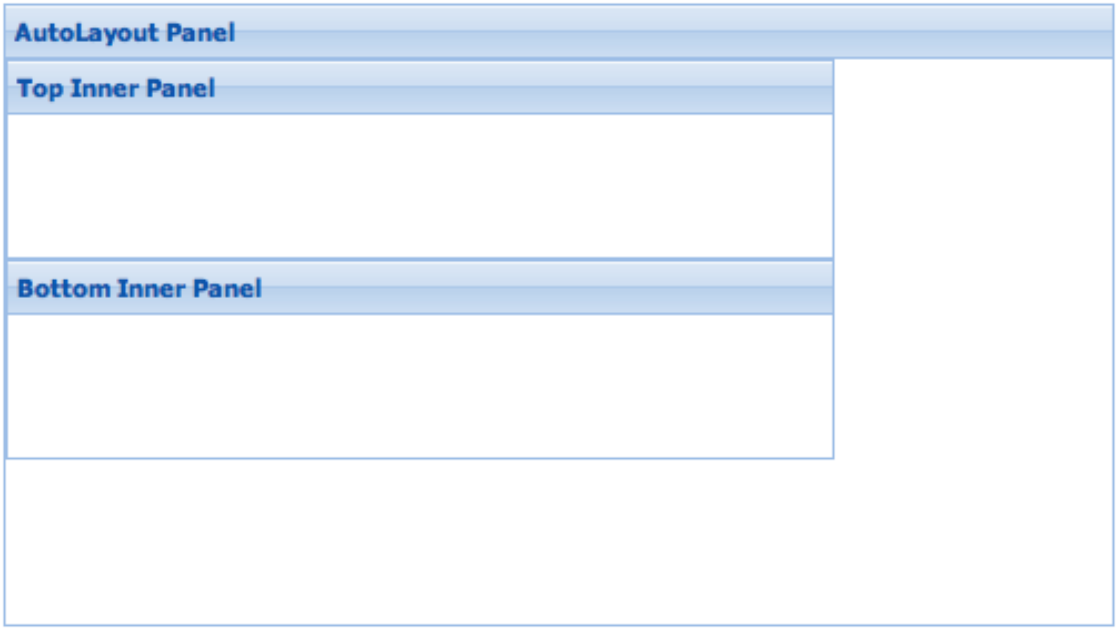
3. The second child Panel is anchored to the containing Panel using two offset values. The first value specifies a horizontal offset of *-300* causing this child Panel to always take up *300* pixels *less* than the parent Panel's total width. The second value specifies a vertical offset of *-200* causing the child Panel's height to take up 200 pixes less than the total height of the parent Panel. With these settings the initial width of this Panel is 200 pixels (500 - 300) and the initial height is 200 pixels (400 - 200).

4. The third child Panel is anchored to the containing Panel using an offset value of *-250* for the width and a percentage value of *20%* for the height. This sets the initial width to 250 pixels (500 - 250) and the initial height to 80 pixels (400 * 20%).

AutoLayout in Action

Ext.layout.AutoLayout is the default Container Layout that's applied to Containers and Panels when no layout is specified. The AutoLayout works just like the *normal flow* of a an HTML document. Components are laid out vertically one after another beginning at the top of the Container.

Figure 3. Two Inner Panels Inside a Containing Panel Using AutoLayout




```
new Ext.panel.Panel({                          // 1
 width: 500,
 height: 280,
 title: 'AutoLayout Panel',
 renderTo: document.body,
 items: [{                                     // 2
  xtype: 'panel',
  title: 'Top Inner Panel',
  width: '75%',
  height: 90
 },{
  xtype: 'panel',
  title: 'Bottom Inner Panel',
  width: '75%',
  height: 90
 }]
});
```

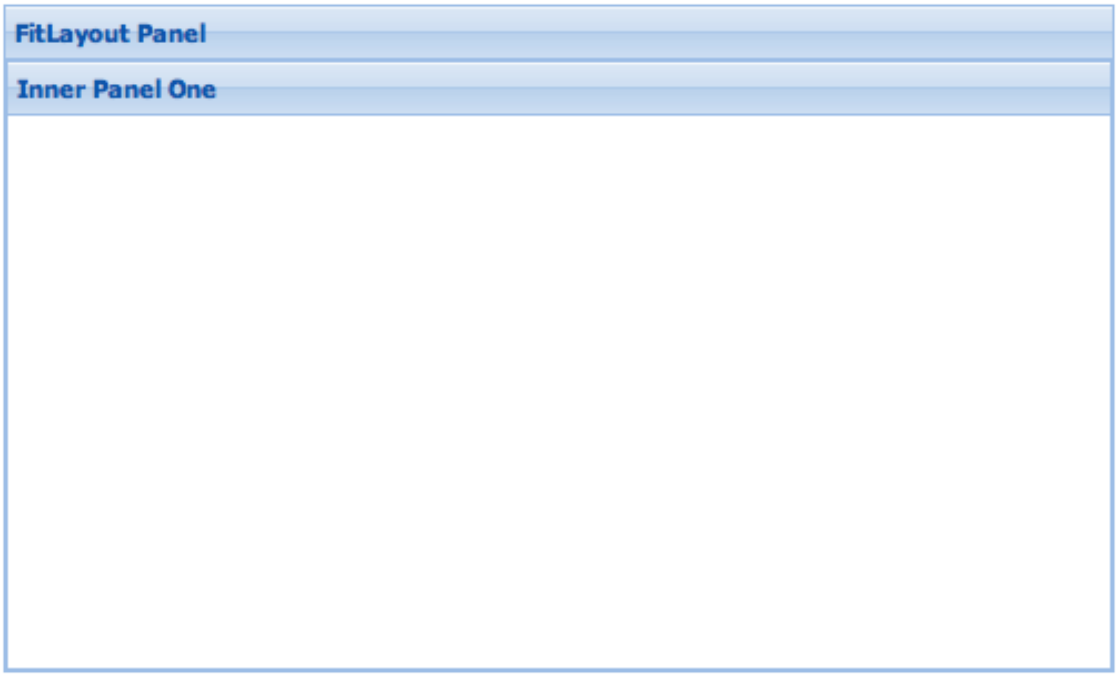
1. A new *Ext.panel.Panel* is setup without setting a specific *layout* configuration option so it defaults to *Ext.layout.container.Auto*.

2. The two inner Panels are automatically arranged without specifying any layout configuration options. The Panels are rendered vertically one after another inside the containing Panel.

FitLayout in Action

Ext.layout.container.Fit allows a single child component to fill the entire area of its containing component.

Figure 4. Inner Panel Inside a Containing Panel Using FitLayout




```
new Ext.panel.Panel({                    // 1
 width: 500,
 height: 300,
 title: "FitLayout Panel",
 layout: 'fit',
 renderTo: document.body,
 items: [{
  xtype: 'panel',                        // 2
  title: 'Inner Panel One'
 }]
});
```

1. A new *Ext.panel.Panel* is setup with its *layout* configuration option set to *fit* (shorthand for *Ext.layout.continer.Fit*. Although not required for a FitLayout this example uses a fixed width and height to control the size of the main Panel.

2. The inner Panel is automatically sized without setting any layout configuration options. It simply fills the available space of its containing Panel.

Note

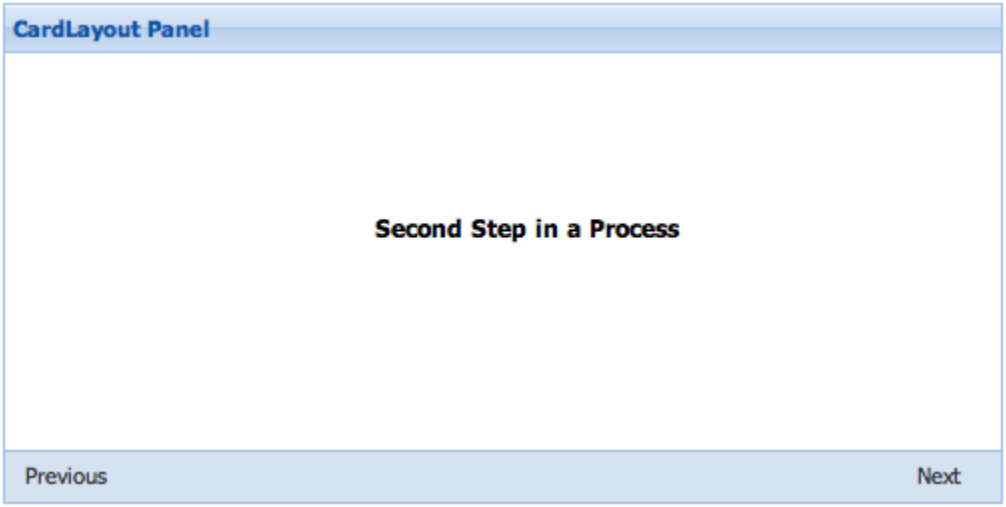
If FitLayout is set for a component that contains more than one item only the first item will be displayed.

CardLayout in Action

The CardLayout is designed support multi-step application workflows and wizard style interfaces where only one item from a collection of components is displayed at a time. The CardLayout includes several built-in methods to help you control the item currently being displayed (*setActiveItem*) and to navigate between items (*prev* and *next*).

The CardLayout provides as a foundation for building multi-step user experiences using custom controls to display and navigate child components.

Figure 5. Wizard Style Panels Using CardLayout




```
new Ext.panel.Panel({
 width: 500,
 height: 250,
 title: "CardLayout Panel",
 layout: 'card',                                    // 1
 renderTo: document.body,
 items: [{                                          // 2
  xtype: 'container',
  html: "<h1>Second Step in a Process</h1>",
  style: {marginTop: '80px', marginLeft: '185px'}
 },{
  xtype: 'container',
  html: "<h1>Second Step in a Process</h1>",
  style: {marginTop: '80px', marginLeft: '185px'}
 },{
  xtype: 'container',
  html: "<h1>Third Step in a Process</h1>",
  style: {marginTop: '80px', marginLeft: '185px'}
 }],
```

```
  dockedItems: [{
   xtype: 'toolbar',
   dock: 'bottom',
   items: [{                                                    // 3
     xtype: 'button',
     text: 'Previous',
     width: 60
    }, {
     xtype: 'tbfill'
    }, {
     xtype: 'button',
     text: 'Next',
     width: 60
   }]
  }]
 });
```

1. A new *Ext.panel.Panel* is setup with the *layout* configuration option set to *card* (shorthand for *Ext.layout.continer.Card*).

   Although not required for the CardLayout this example uses a fixed width and height to control the size of the main Panel.

2. Three simple Containers are setup inside the main Panel. Each Container displays a short message about each step within a multi-step process.

   By default the first Container will be displayed when the main Panel is first rendered.

3. A Toolbar is configured and docked to the bottom of the main Panel. The Toolbar contains two Buttons to demonstrate how you might provide interface controls to navigate through the items in the CardLayout. In a real application you would need to wire up these buttons to properly call *Ext.layout.container.Card.prev* and *Ext.layout.container.Card.next*.

Note

Buttons and other components can easily be wired to control CardLayout functionality however the Layout class itself doesn't provide built-in interface controls.

HBoxLayout in Action

Ext.layout.container.HBox allows you to arrange components horizontally (left-to-right) within a containing component. The HBoxLayout provides three main configuration options (align, flex and pack) to provide additional control over how components are arranged.

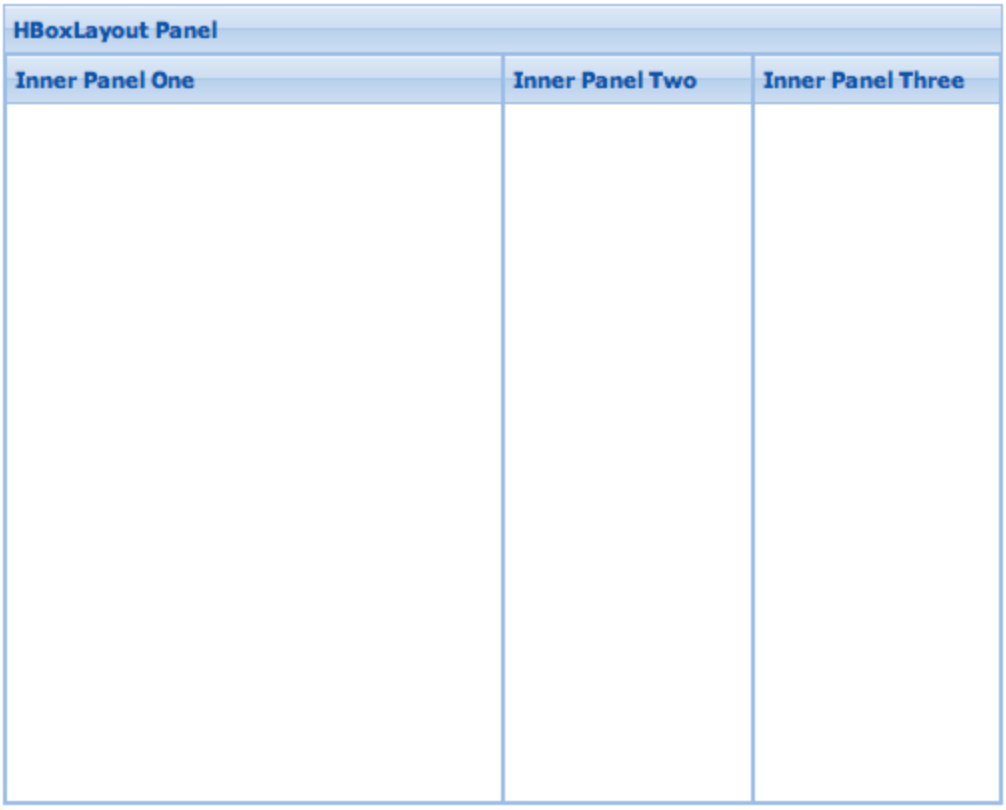
The *align* option determines how the child components are vertically aligned within the containing component. This option is set to *top* by default.

The *flex* option can be set on each child component to assign it a proportional value that determines how the width of the component will be *flexed* as the containing component is resized. The dynamic width is calculated by dividing the individual *flex* value set for that component by the sum of all the *flex* values set on neighboring components participating in the HBoxLayout.

The *pack* option determines how components will be horizontally *packed* together. This option is set to *start* by default causing child components to be packed together starting on the left side of the containing component.

You'll see how these three configuration options work together in the following example.

Figure 6. Three Panels Inside a Containing Panel Using HBoxLayout




```
new Ext.panel.Panel({
 width: 500,
 height: 400,
 title: "HBoxLayout Panel",
```

```
 layout: {                          // 1
  type: 'hbox',
  align: 'stretch'
 },
 renderTo: document.body,
 items: [{                          // 2
  xtype: 'panel',
  title: 'Inner Panel One',
  flex: 2
 },{
  xtype: 'panel',
  title: 'Inner Panel Two',
  flex: 1
 },{
  xtype: 'panel',
  title: 'Inner Panel Three',
  flex: 1
 }]
});
```

1. A new *Ext.panel.Panel* is setup. Here a *layout* configuration object is used to configure the layout options together in one place. The layout *type* configuration option is set to *hbox* which is shorthand for *Ext.layout.continer.HBox*. The layout *align* option is set to *stretch* allowing the child components to stretch vertically to fill the available space in the containing component.

   Although not required for an HBoxLayout this example uses a fixed width and height to control the size of the containing Panel.

2. The three inner Panels are automatically sized according to the *align* configuration set on the containing Panel and the *flex* configuration option set on each of the inner Panels.

   The first inner Panel *flex* value is set to *2* and the second and third Panel *flex* values are set to *1*. As a result of this configuration the width of the first inner Panel will be set to 50% of the containing Panel's width calculated as 2 / (2 + 1 + 1) = .50 and the other two Panels will each take up 25% of the containing Panel's width calculated as 1 / (2 + 1 + 1) = .25.

VBoxLayout in Action

Ext.layout.container.VBox allows you to arrange components vertically (top-to-bottom) within a containing component. The VBoxLayout provides three main configuration options (align, flex and pack) to provide additional control over how components are arranged.

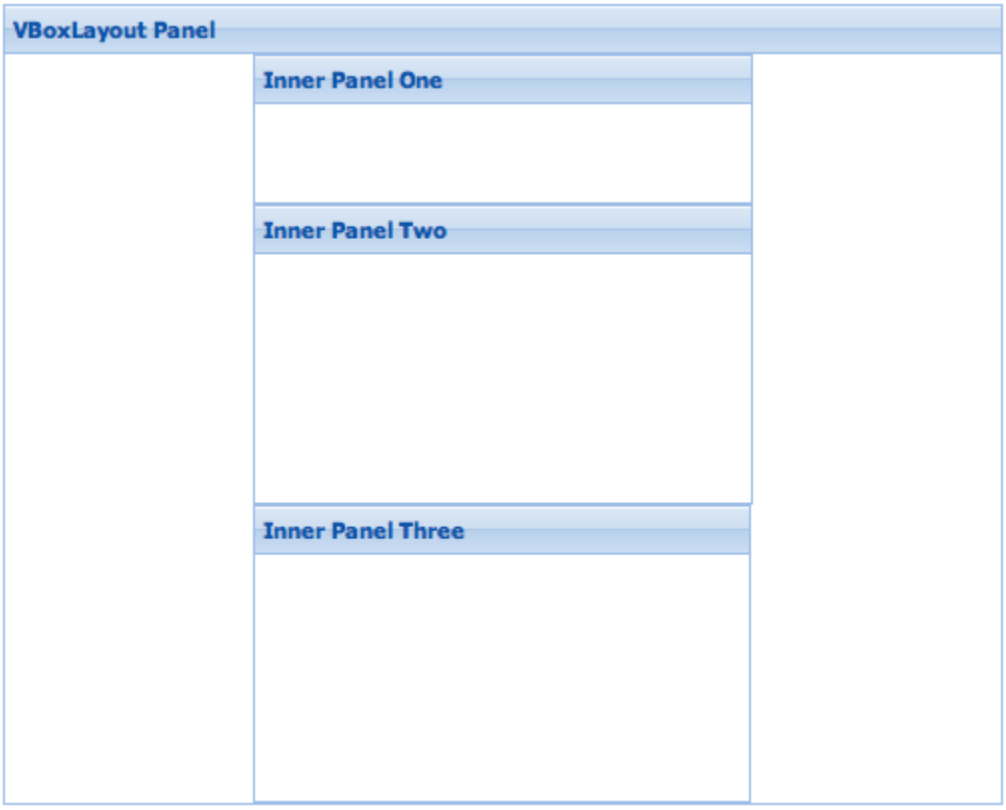
The *align* option determines how the child components are horizontally aligned within the containing component. This option is set to *left* by default.

The *flex* option can be set on each child component to assign it a proportional value that determines how the height of the component will be *flexed* as the containing component is resized. The dynamic height is calculated by dividing the individual *flex* value set for that component by the sum of all the *flex* values set on neighboring components participating in the VBoxLayout.

The *pack* option determines how components will be horizontally *packed* together. This option is set to *start* by default causing child components to be packed together starting on the left side of the containing component.

You'll see how these three configuration options work together in the following example.

Figure 7. Three Panels Inside a Containing Panel Using VBoxLayout




```
new Ext.panel.Panel({
 width: 500,
 height: 400,
 title: "VBoxLayout Panel",
```

```
 layout: {                          // 1
  type: 'vbox',
  align: 'center'
 },
 renderTo: document.body,
 items: [{                          // 2
  xtype: 'panel',
  title: 'Inner Panel One',
  width: 250,
  flex: 2
 },{
  xtype: 'panel',
  title: 'Inner Panel Two',
  width: 250,
  flex: 4
 },{
  xtype: 'panel',
  title: 'Inner Panel Three',
  width: '50%',
  flex: 4
 }]
});
```

1. A new *Ext.panel.Panel* is setup. Here a *layout* configuration object is used to configure the layout options together in one place. The layout *type* configuration option is set to *vbox* (shorthand for *Ext.layout.continer.VBox*). The layout *align* option is set to *center* causing the child components to be centered horizontally inside the containing component.

   Although not required for an VBoxLayout this example uses a fixed width and height to control the size of the containing Panel.

2. The width of each of the inner Panels is explicitly set using the *width* configuration option. This can either be an integer like the first two Panels or a percentage as shown for the third Panel.

   The height of each inner Panel is automatically determined by the *flex* configuration option set on each Panel.

   The first inner Panel *flex* value is set to *2* and the second and third Panel *flex* values are set to *4*. As a result of this configuration the height of the first inner Panel will be set to 20% of the containing Panel's height calculated as 2 / (2 + 4 + 4) = .20 and the other two Panels will each take up 40% of the containing Panel's height calculated as 4 / (2 + 4 + 4) = .40.

ColumnLayout in Action

Ext.layout.container.Column extends the Ext.layout.container.HBox layout allowing you to create multi-ColumnLayouts with dynamic column widths and heights.

The width of each Column can be specified in pixels, proportions or a combination of both. To specify a value in pixels you can use the *width* configuration option and for proportions the *flex* option.

When using *flex*, the proportional width of each component is calculated by dividing the individual *flex* value set for that component by the sum of all *flex* values set for components participating in the ColumnLayout.

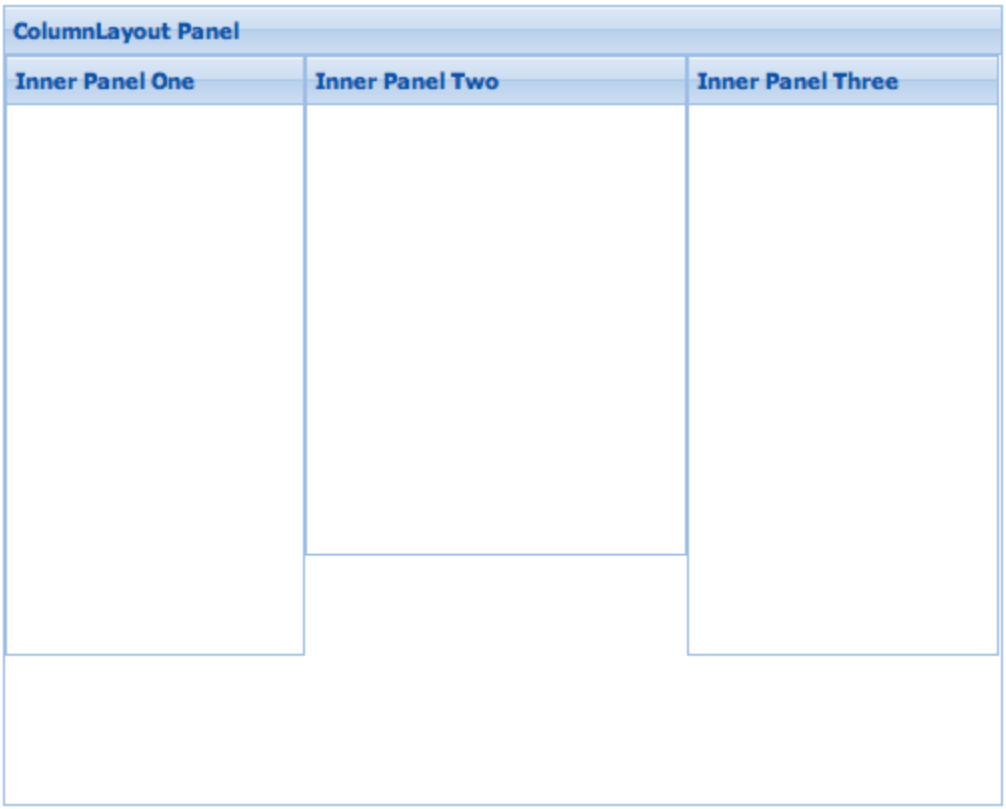
Note

When specifying *flex* proportions remember to use either integers or decimals.

Note

When combining pixel widths and proportional widths the pixel width values will be subtracted from the overall width of the container before the proportional values are calculated.

Figure 8. Three Panels Inside a Containing Panel Using ColumnLayout




```
new Ext.panel.Panel({
```

```
    width: 500,
    height: 400,
    title: "ColumnLayout Panel",
    layout: 'column',                        // 1
    renderTo: document.body,
    items: [{
     xtype: 'panel',
     title: 'Inner Panel One',
     width: 150,                             // 2
     height: 300
    },{
     xtype: 'panel',
     title: 'Inner Panel Two',
     flex: .60,                              // 3
     height: 250
    },{
     xtype: 'panel',
     title: 'Inner Panel Three',
     flex: .40,
     height: 300
    }]
   });
```

1. A new *Ext.panel.Panel* is setup with the *layout* configuration option set to *column* (shorthand for *Ext.layout.continer.Column*).

   Although not required for the ColumnLayout this example uses a fixed width and height to control the size of the containing Panel.

2. Using the *width* and *height* configuration options the width of the first inner Panel is set to 150 pixels and the height is set to 300 pixels.

3. The second and third inner Panels use the *flex* configuration option to set their widths as a percentage of the available width (350 pixels) after 150 pixels (first Panel's width) has been subtracted from the containing Panel's width of 500 pixels.

   With this configuration the second inner Panel will take up 210 pixels = .60 * (500 - 150) and the third inner Panel will take up 140 pixels = .40 * (500 - 150).

Note

The flex values behave like percentages in this example because we've made sure they all up to 1 (.60 + .40).

Note

The *height* for both panels are have been set to pixel values. If height values weren't specified they would be calculated by their contents.

TableLayout in Action

Ext.layout.container.Table allows you to programmatically render content into a HTML Table with configuration options that mirror regular HTML Table markup.

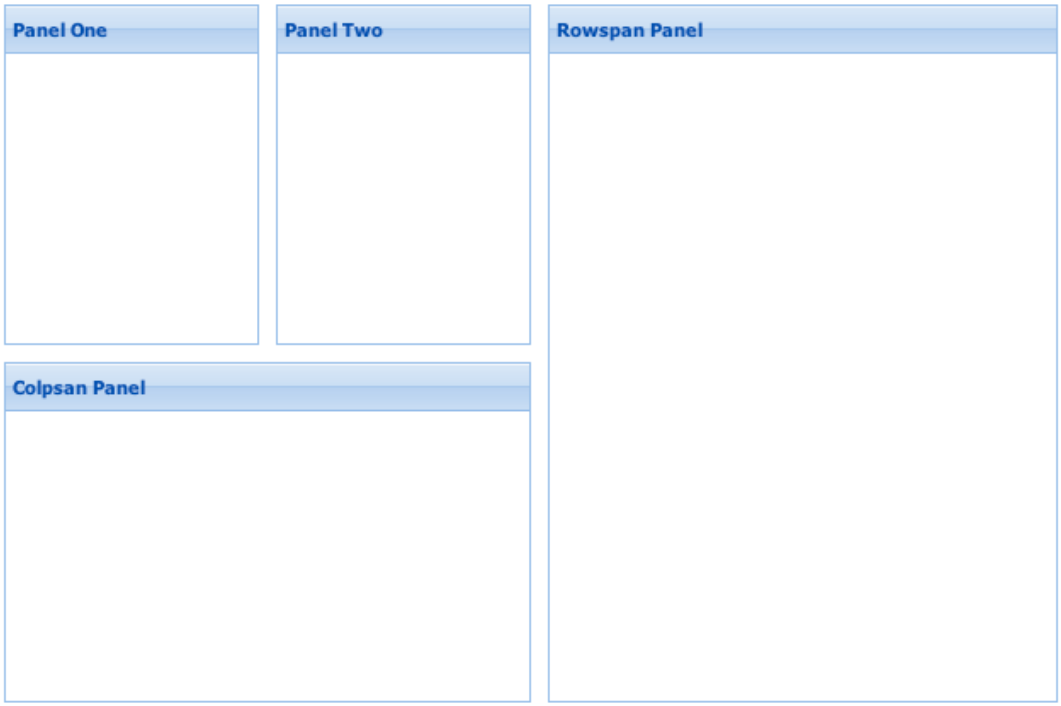
All child components participating in a TableLayout can optionally specify two span attributes through the configuration options *rowspan* and *colspan*. These work exactly like their HTML counterparts to help you arrange table items.

Instead of explicitly creating and nesting rows and columns as you would in HTML, you simply specify the total column count using the *columns* configuration option on the containing component and then start adding panels in their natural order from left to right, top to bottom. The layout will automatically figure out how to arrange the components based on the column count, rowspan and colspan settings.

Note

Just like with HTML Tables, your rowspans and colspans must add up correctly in your overall layout or you'll end up with missing and/or extra cells!

Figure 9. Panels Inside a Container Using TableLayout




```
new Ext.Container({
 id: 'table-panel',
 layout: {                          // 1
  type: 'table',
  columns: 3
 },
 renderTo: document.body,
 items: [{                          // 2
```

```
    xtype: 'panel',
    title: 'Panel One',
    width: 150,
    height: 200
   },{
    xtype: 'panel',
    title: 'Panel Two',
    width: 150,
    height: 200
   },{
    xtype: 'panel',
    title: 'Rowspan Panel',
    width: 300,
    height: 410,
    rowspan: 2                          // 3
   },{
    xtype: 'panel',
    title: 'Colpsan Panel',
    width: 310,
    height: 200,
    colspan: 2                        //  4
   }]
  });
```

```
#table-panel td {
padding: 5px;                 // 5
}
```

1. A new *Ext.panel.Panel* is setup. Here a *layout* configuration object is used to configure the layout options together in one place. The layout *type* configuration option is set to *table* (shorthand for *Ext.layout.continer.Table*). The *columns* configuration option determines the number of columns that are created in the table. Here *3* columns will hold all of the child components.

2. The first two Panels are setup with 150 pixel widths and 200 pixel heights. These two Panels are rendered into the first two columns of the first table row.

3. The third Panel uses the *rowspan* configuration option to span the first two rows of the table. The *height* for this Panel is set to *410* so it fits neatly next to the first two Panels after accounting for CSS padding.

4. The fourth Panel uses the *colspan* configuration option to span the first two columns of the table. Similar to the third Panel the *width* of this Panel is set to 310 so it fits neatly under the first two Panels after accounting for CSS padding.

5. To help give this example a clean look 5 pixels of padding is added to each table cell using CSS.

BorderLayout in Action

BorderLayout is a multi-pane, application-oriented UI layout that supports multiple nested panels, splitter bars between regions and built-in support for expanding and collapsing regions. BorderLayout makes it easy to layout desktop style application interfaces using Ext JS.

BorderLayouts consist of up to five main regions *north*, *south*, *west*, *east* and *center*. Each *region* is typically a Container or Panel that will hold all the components for that section of your interface.

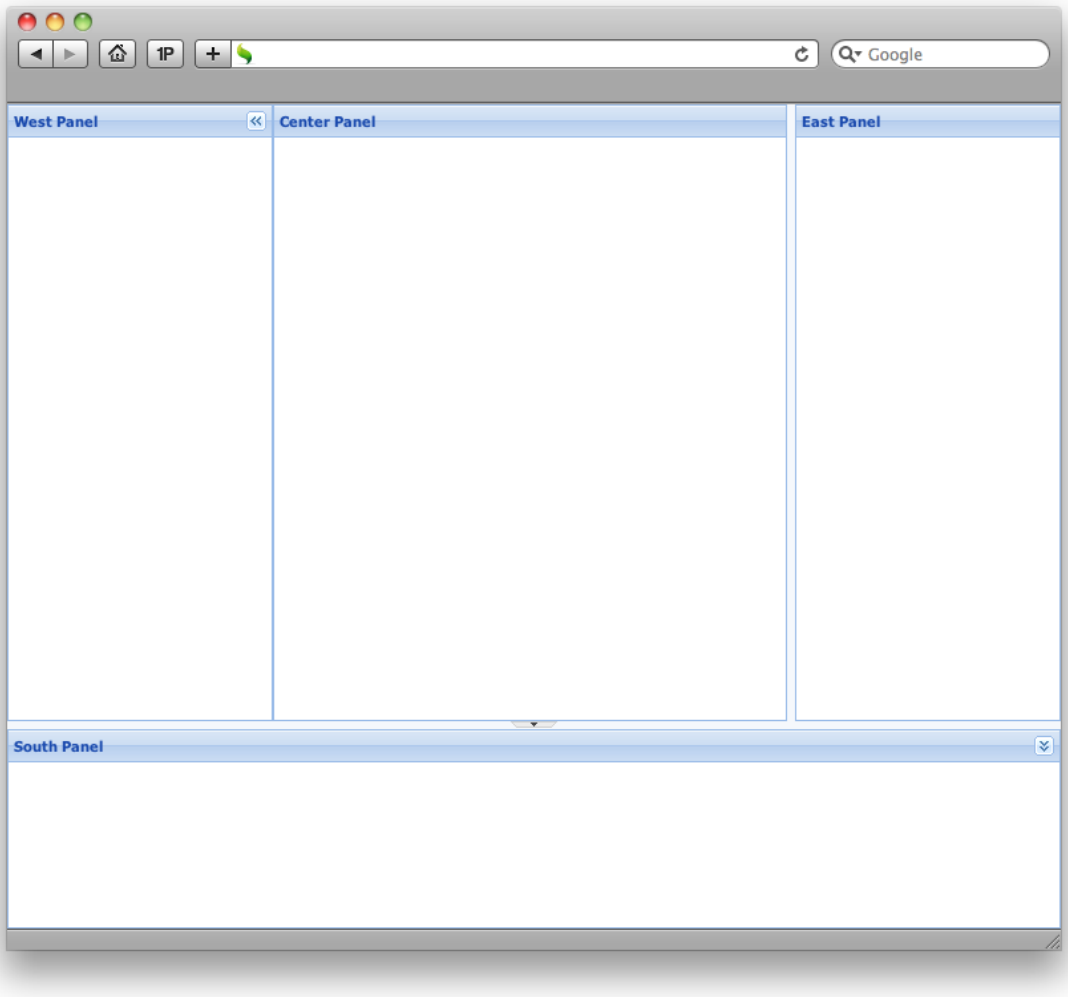
BorderLayout Requirements and Notes

- For a *region* to be *collapsible* it needs to be setup as a *Ext.panel.Panel*.
- BorderLayout requires a child component with the *region* configuration option set to *center* in order to display correctly. The *center region* will always resize to fill the remaining space not used by other regions in the layout.
- The *width* of child items configured as *west* or *east* regions can either be set explicitly using the *width* config option or by setting a *flex* value.
- The *height* of child items configured as *north* or *south* regions can either be set explicitly using the *height* config option or by setting a *flex* value to determine height.
- Regions of a BorderLayout are fixed at render time and can't be added or removed post-render. The best practice for changing interface components when using the BorderLayout is to add and remove child components from the *regions* in your layout. For example if you have a Panel configured for the *east* region you can add and remove child components of that Panel. What you can't do is destroy the east *region* Panel and try to add another one back after the BorderLayout has rendered.

```
new Ext.Viewport({
 title: 'BorderLayout Demo',
 layout: 'border',                 // 1
 items: [{
  xtype: 'panel',
  title: 'West Panel',
  region: 'west',                  // 2
  collapsible: true,
  width: 200
 },{
  xtype: 'panel',
  title: 'Center Panel',
  region: 'center'                // 3
 },{
  xtype: 'panel',
  title: 'East Panel',
  region: 'east',                 // 4
  split: true,
  width: 200,
  minSize: 100,
  maxSize: 400
```

```
 },{
  xtype: 'panel',
  title: 'South Panel',
  region: 'south',                    // 5
  split: true,
  collapsible: true,
  collapseMode: 'mini',
  height: 150
 }]
});
```

Figure 10. Viewport Using the BorderLayout



1. A new *Ext.container.Viewport* is setup with the *layout* configuration option set to *border* (shorthand for *Ext.layout.continer.Border*).

2. The first Panel is setup as the *west region*. It's made collapsible by setting the *collapsible* configuration option to *true*. The *width* configuration option is set to *200* pixels. Setting an explicit width here has the effect of locking the Panel's width to this value.

3. The second Panel is setup as the *center region*. This is required for the BorderLayout to render correctly. Notice how a *width* hasn't been defined for this region as its always resizes to fill the remaining space not taken up by the other regions in the *BorderLayout*.

4. The third Panel is setup as the *east region*. By setting the *split* config option to *true* this Panel will render with a splitter bar that can be dragged to adjust the width of the Panel. The initial *width* is set to *200* pixels and the *minSize* and *maxSize* configuration options allow you to control the minimum and maximum width boundaries a user can set when they resize the Panel by dragging the splitter bar to change it's width.

5. The fourth Panel is setup as the *south region*. The *split* is set to *true* allowing users to resize the Panel's height by dragging the splitter bar. It's made collapsible by setting the *collapsible* configuration option to *true*. Setting *collapseMode* to *mini* will completely hide the Panel including the title area when the collapse icon is clicked. In mini mode the user will only see a small arrow in the middle of the splitter bar that allows them to drag the Panel back into view.

Layout Tips

Default Layout

Ext.layout.container.Auto is the default layout that will be used for a component if a layout isn't specifically set. This can be a common source of unexpected results if you start adding components like GridPanels and TreePanels to a Container using *Auto Container Layout* as none of your child components will be re-sized or adjusted when the parent Container is re-sized.

When to Call doLayout

Starting with Ext JS 4 calling *doLayout* is no longer required when adding components to a Container that has already been rendered. The new layout engine handles this for you and automatically renders new items into a Container when you call it's *add* method.

If you need to add multiple items to a component and you don't want them to be seen until all your calls have been completed the best practice is to create an array of components and then pass the array into the *add* method once all your items are ready.

To manually achieve the same result you can also set *suspendLayout* to *true* on the Container before adding components. This will prevent the Layout from being updated when new components are added. Then when you're ready to display the newly added components remember to set *suspendLayout* to *false* and then call *doLayout* to recalculate the Layout.

Scrolling and Dynamic Container Sizing

When you start creating advanced interface layouts with Ext JS getting scrolling and dynamic Container sizing to work precisely the way you want can be a common source of confusion. The next series of examples explore how different Ext JS layouts effect scrolling behavior.

Scrolling AutoLayout - Containers and Panels

By default Containers and Panels won't automatically display scrollbars if the components they hold extend beyond their width or height. In some cases you can change this behavior simply by setting the *autoScroll* configuration option to *true*. The next few examples explore where scrollbars appear with different combinations of a nested Containers and Panels.

Figure 11. AutoLayout - Scrolling Nested Panels with a Title Bar


```
new Ext.panel.Panel({
 title: 'Outer Panel',
 renderTo: document.body,
 width: 400,
 height: 300,
 layout: 'auto',
 autoScroll: true,
 items: [{
  xtype: 'panel',
```

```
  title: 'Inner Panel',
  height: 400
 }]
});
```

A new *Ext.panel.Panel* is setup with a *width* of *400* and *height* of *300*. The *layout* is set to *Ext.layout.container.Auto*.

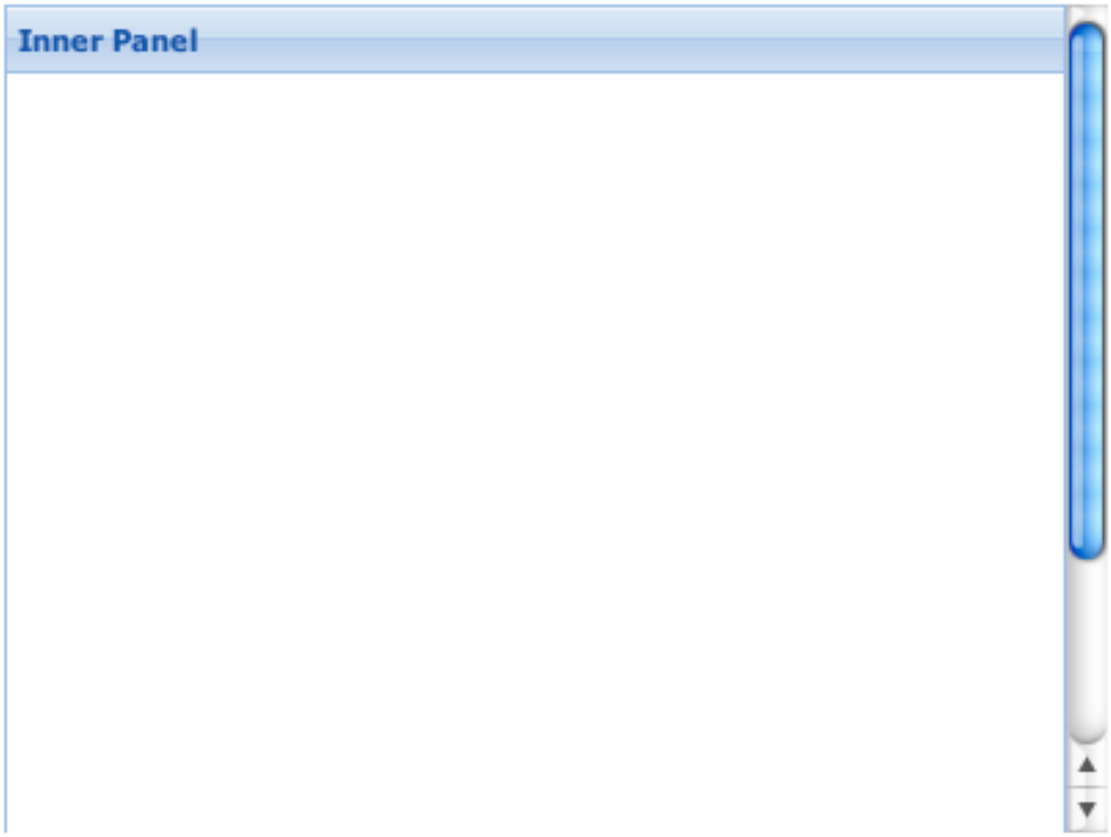
The *autoScroll* config option has been set to *true*. This will cause scroll bars to display whenever items contained inside the Panel exceed its defined width or height.

The inner Panel is setup with a *height* of *400* so it exceeds the height set of the outer Panel causing the vertical scrollbars to display.

In this example you'll notice the scrollbars appear just under the Panel title bar. This follows the same pattern as scrollbars in a desktop user interface.

The next example demonstrates a different approach to providing scrollbars using a Panel nested inside a Container.

Figure 12. AutoLayout - Scrolling a Panel Inside a Container




```
new Ext.Container({                // 1
 renderTo: document.body,
 width: 400,
 height: 300,
```

```
  layout: 'auto',
  autoScroll: true,
  items: [{
   xtype: 'panel',
   title: 'Inner Panel',
   height: 400
  }]
 });
```

This example uses an Ext.Container instead of a Panel. When this renders you'll notice the scrollbars appear along the entire side of the Container. In this example the bottom part of the Container looks likes its missing. This happens because the Container doesn't have a border configured and the Panel's bottom border is out of view until you scroll down.

You could improve the outcome of this example by setting the *border* configuration option on the Container.

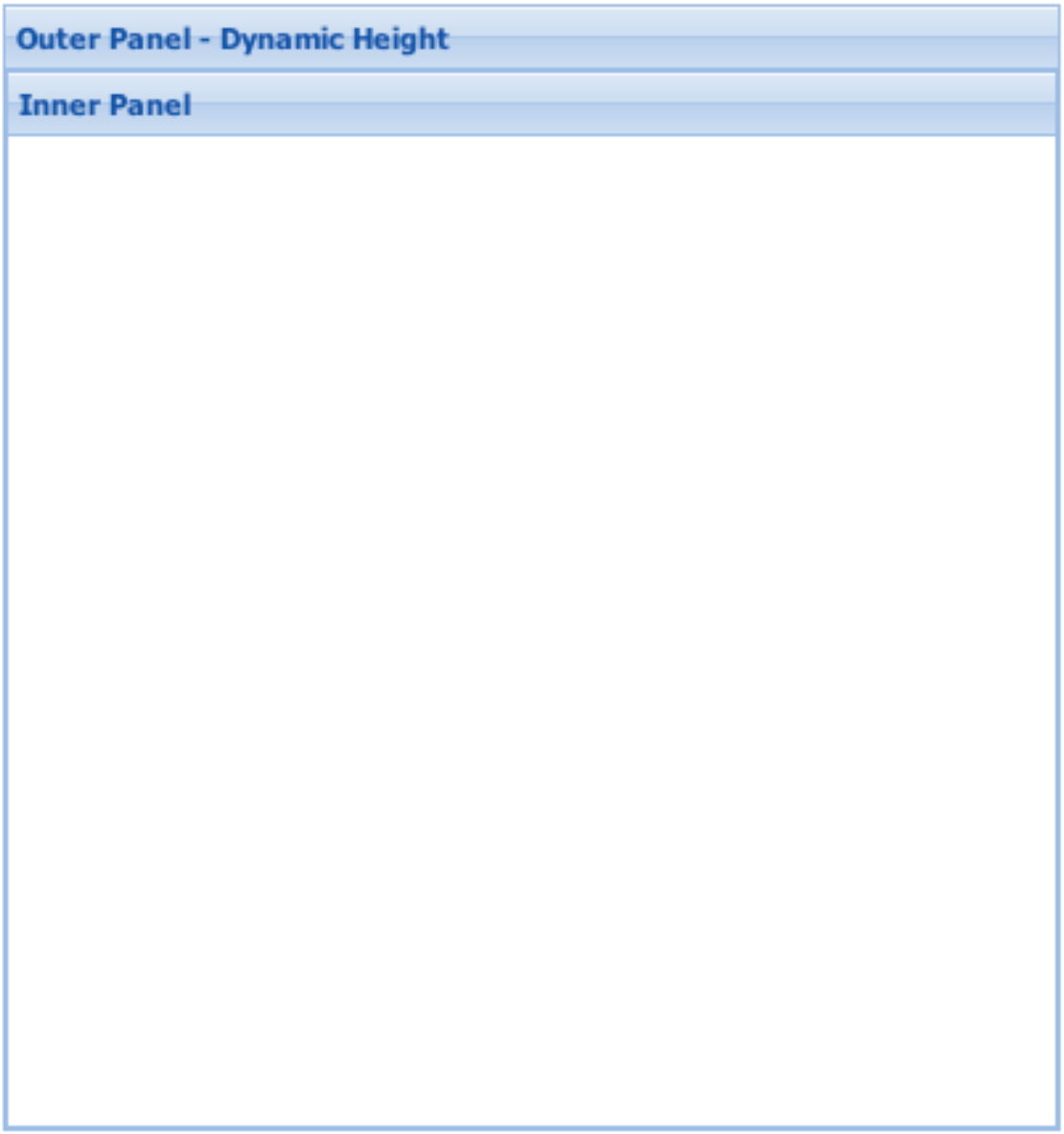
Figure 13. AutoLayout - Scrolling Nested Panels Without a Title Bar


```
new Ext.panel.Panel({
 renderTo: document.body,
 width: 400,
 height: 300,
 layout: 'auto',
 autoScroll: true,
```

```
 items: [{
  xtype: 'panel',
  title: 'Inner Panel',
  height: 400
 }]
});
```

This example is just like the first one but doesn't set a *title* value for the outer Panel. As a result you can see the full border around the outer Panel and also have scrollbars extend the full height of the outer Panel.

Figure 14. AutoLayout - Dynamic Container Height




```
new Ext.Panel({
```

```
 title: 'Outer Panel - Dynamic Height',
 renderTo: document.body,
 width: 400,
 layout: 'auto',
 autoScroll: true,
 items: [{
  xtype: 'panel',
  title: 'Inner Panel',
  height: 400
 }]
});
```

This example removes the *height* option from the Outer Panel configuration object. By not specifying a height the Panel's height will automatically be sized to fit its contents (in this case the Inner Panel).

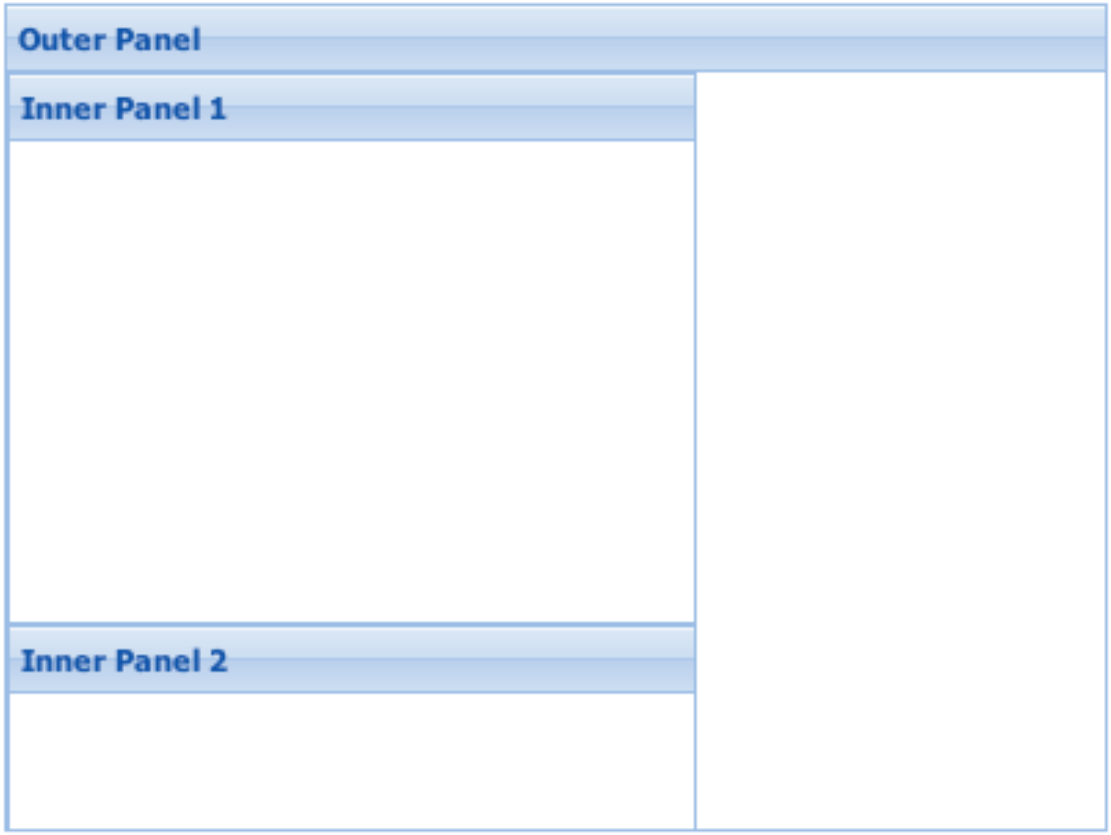
Note

The *width* configuration option is still left in place. If a width hasn't been defined for a Container or any of it's parent Containers then the width of the item will become the width of the browser at render time.

Understanding how the *autoScroll* configuration option works in the context of nested Panels and the AutoLayout is a great starting point for learning how different Container Layouts handle scrolling. The next section explores how scrolling works with VBox and HBoxLayouts.

Scrolling VBox and HBoxLayouts

Figure 15. VBox - Inner Panels with Defined Heights

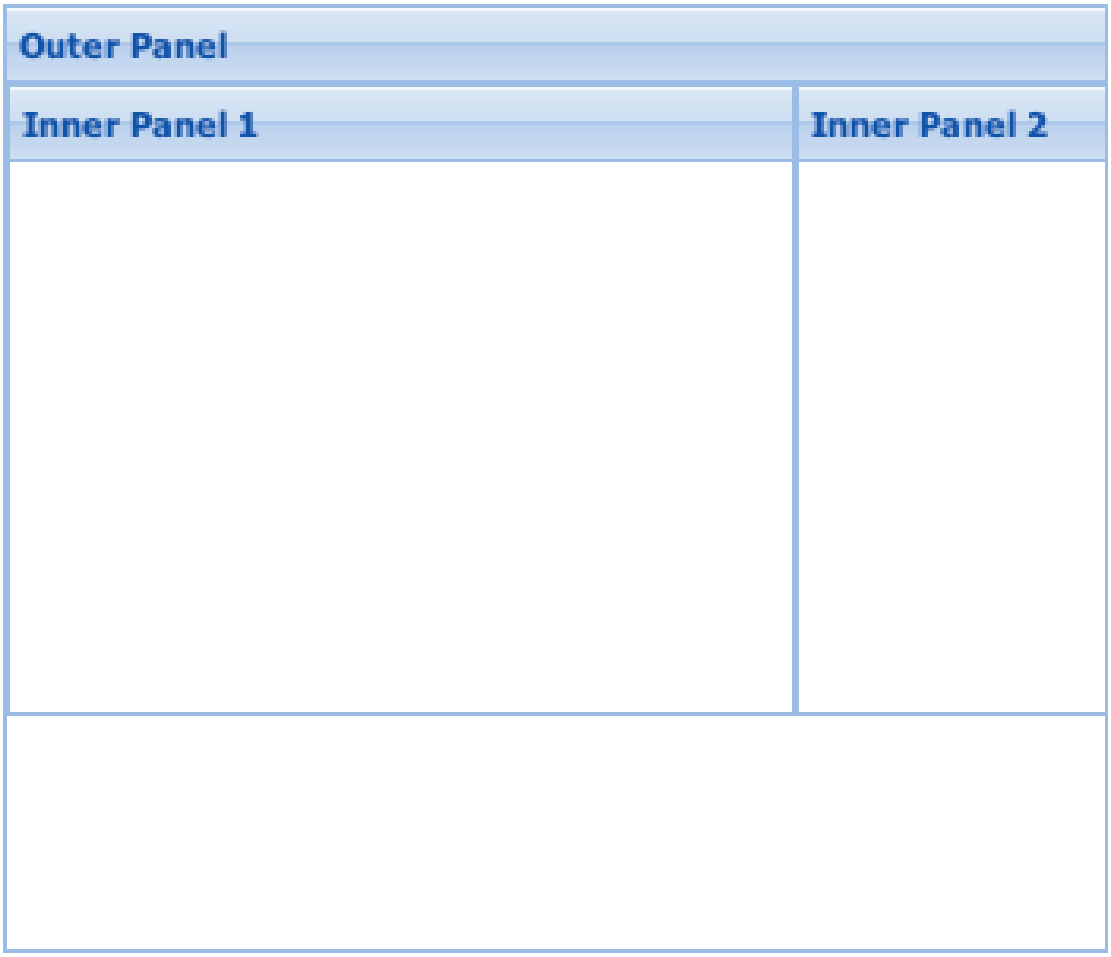



```
new Ext.panel.Panel({
 title: 'Outer Panel',
 renderTo: document.body,
 width: 400,
 height: 300,
 layout: {
  type:'vbox'
 },
 autoScroll: true,
 items: [{
  xtype: 'panel',x
  title: 'Inner Panel 1',
  width: 250,
  height: 200
 },{
  xtype: 'panel',
  title: 'Inner Panel 2',
  width: 250,
  height: 200
 }]
});
```

Looking at the code you'll notice heights have been set for all three Panels and the main Panel uses the *VBoxLayout*.

The height of the two inner Panels adds up to *400 pixels* while the height of the outer Panel is only *300 pixels*. Even though the *autoScroll* configuration option is set to *true* vertical scrollbars won't be displayed, causing the inner Panels to be clipped at 300 pixels.

Figure 16. HBox - Inner Panels with Defined Widths




```
new Ext.panel.Panel({
 title: 'Outer Panel',
 renderTo: document.body,
 width: 350,
 height: 300,
 layout: {
  type:'hbox'
 },
 autoScroll: true,
 items: [{
  xtype: 'panel',
```

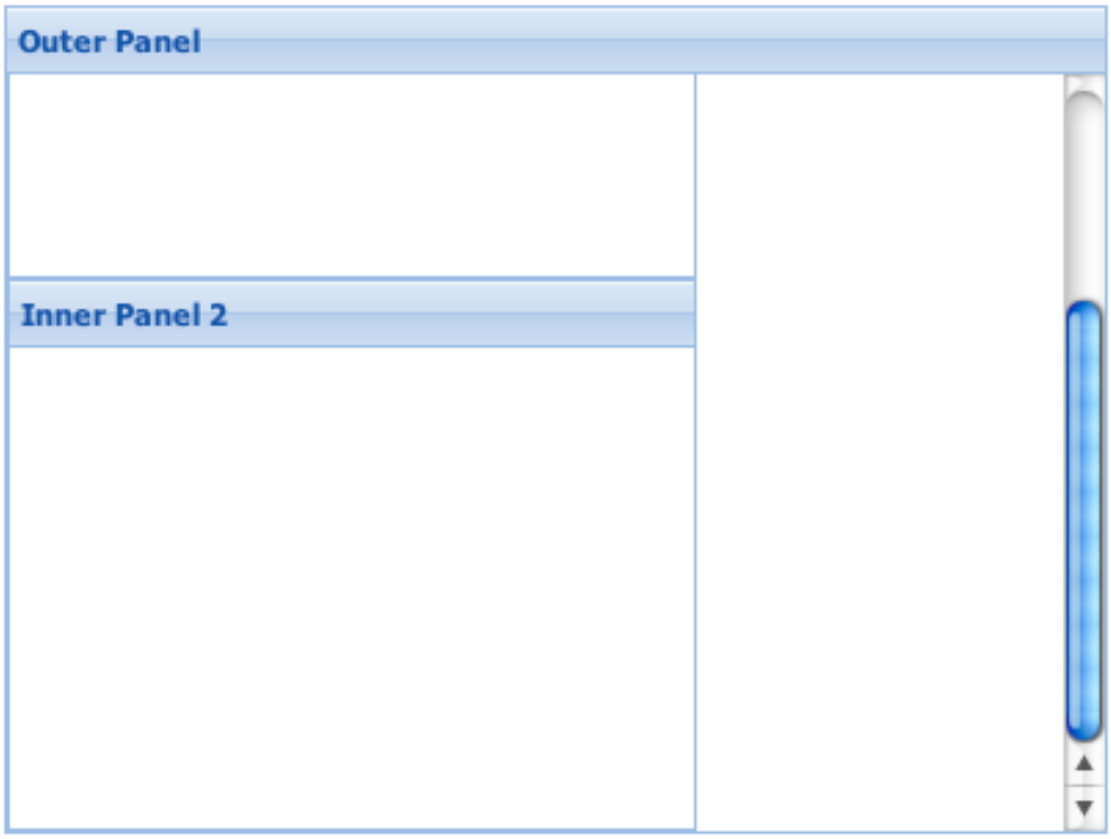
```
   title: 'Inner Panel 1',
   width: 250,
   height: 200
  },{
   xtype: 'panel',
   title: 'Inner Panel 2',
   width: 250,
   height: 200
  }]
});
```

Similar to the *VBox* example the combined widths of the two inner Panels in this *HBoxLayout* add up to *500 pixels* while the *width* of the Container is only *300* pixels.

The clipping happens in both of these scenarios because the standard layout pattern for *Ext.layout.container.VBox* is to automatically manage the heights of components in the Layout while *Ext.layout.container.HBox* automatically manages the widths of components. VBox and HBox use the *flex* values to correctly size each child component.

By setting specific height and width values for the child components we deviated from the default layout pattern and didn't get the results we were anticipating. To get the desired effect of automatically displaying scrollbars we need to set an additional configuration option which we'll demonstrate in the next two examples.

Figure 17. VBox - Scrolling Inner Panels With Defined Heights



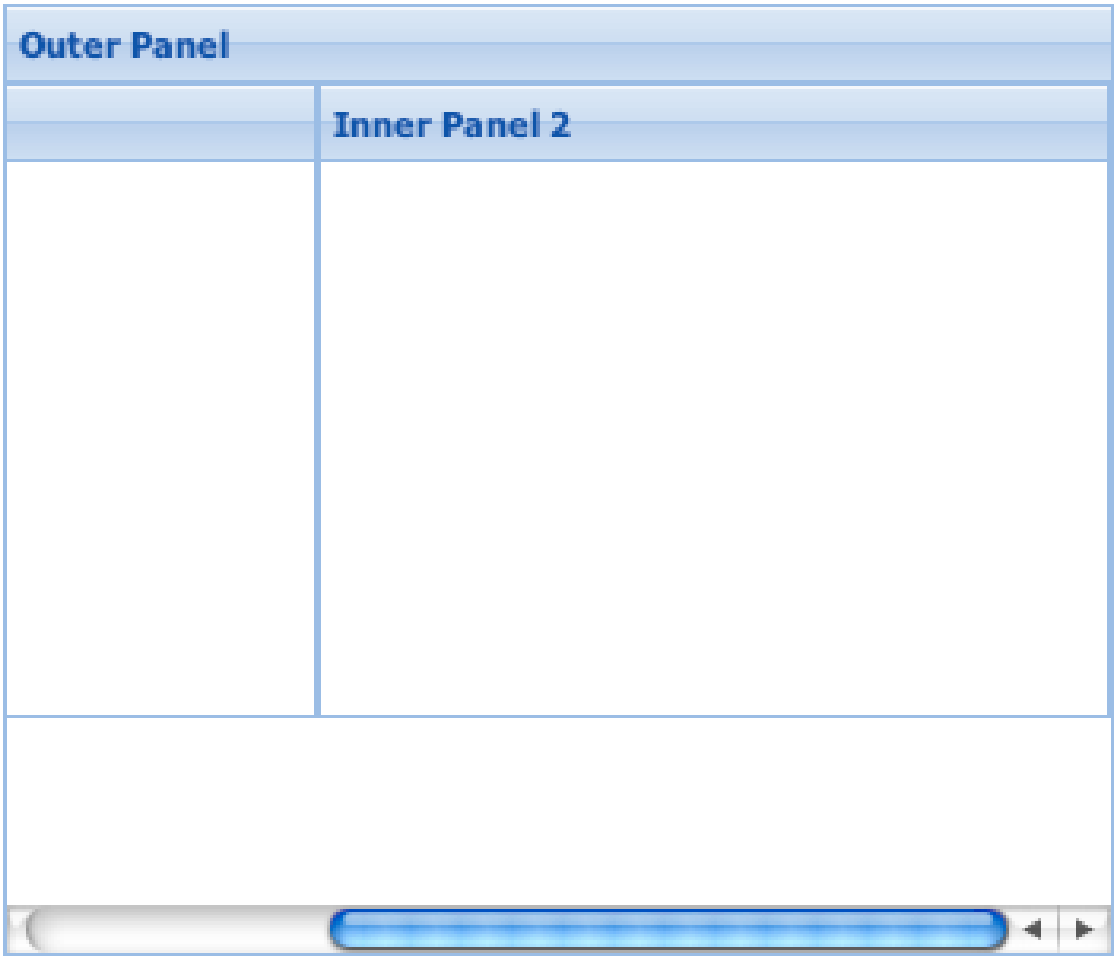
```
new Ext.panel.Panel({
 title: 'Outer Panel',
 renderTo: document.body,
 width: 400,
 height: 300,
 layout: {
  type:'vbox',
  autoSize: true       // provides vertical scrollbars
 },
 autoScroll: true,
 items: [{
  xtype: 'panel',x
  title: 'Inner Panel 1',
  width: 250,
  height: 200
 },{
  xtype: 'panel',
  title: 'Inner Panel 2',
  width: 250,
  height: 200
 }]
});
```

Setting the *autoSize* layout configuration option to *true* will automatically display vertical scrollbars when the defined heights of child components exceed the height of the Containing component.

Note

Unlike *autoScroll* which is a **component configuration** option *autoSize* is a **layout configuration** option and needs to be placed inside your layout configuration object to function correctly.

Figure 18. HBox - Scrolling Inner Panels With Defined Widths



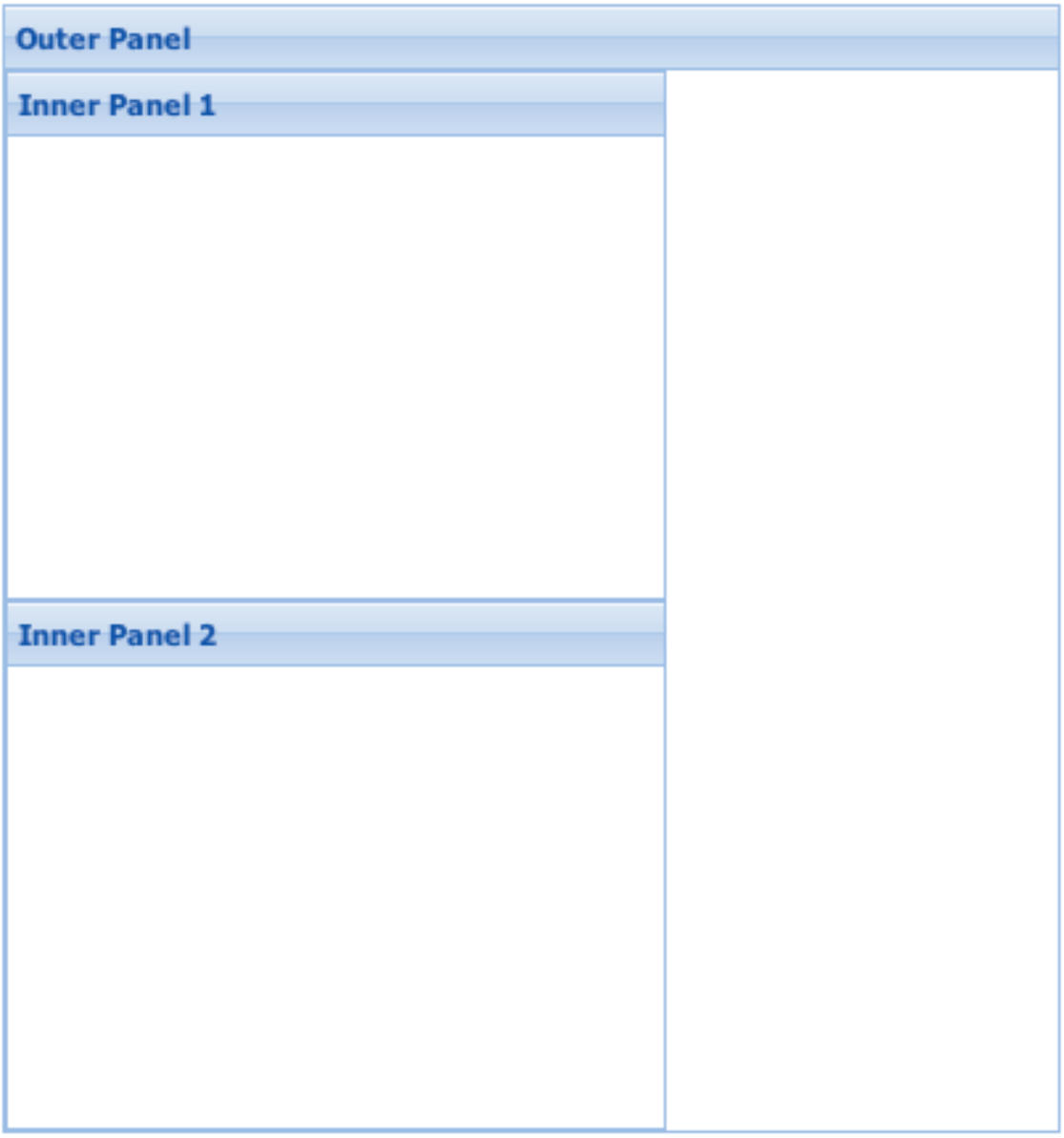

```
new Ext.panel.Panel({
 title: 'Outer Panel',
 renderTo: document.body,
 width: 350,
 height: 300,
 layout: {
  type:'hbox',
  autoSize: true       // provides horizontal scrollbars
 },
 autoScroll: true,
 items: [{
  xtype: 'panel',
  title: 'Inner Panel 1',
  width: 250,
  height: 200
 },{
  xtype: 'panel',
  title: 'Inner Panel 2',
  width: 250,
```

```
  height: 200
 }]
});
```

Setting the *autoSize* configuration option to *true* on the HBox will automatically display horizontal scrollbars when the defined widths of child components exceed the width of the Containing component.

Figure 19. VBox - Dynamic Container Height

With a quick config change we can setup the outer Panel to automatically determine its height based on its contents.

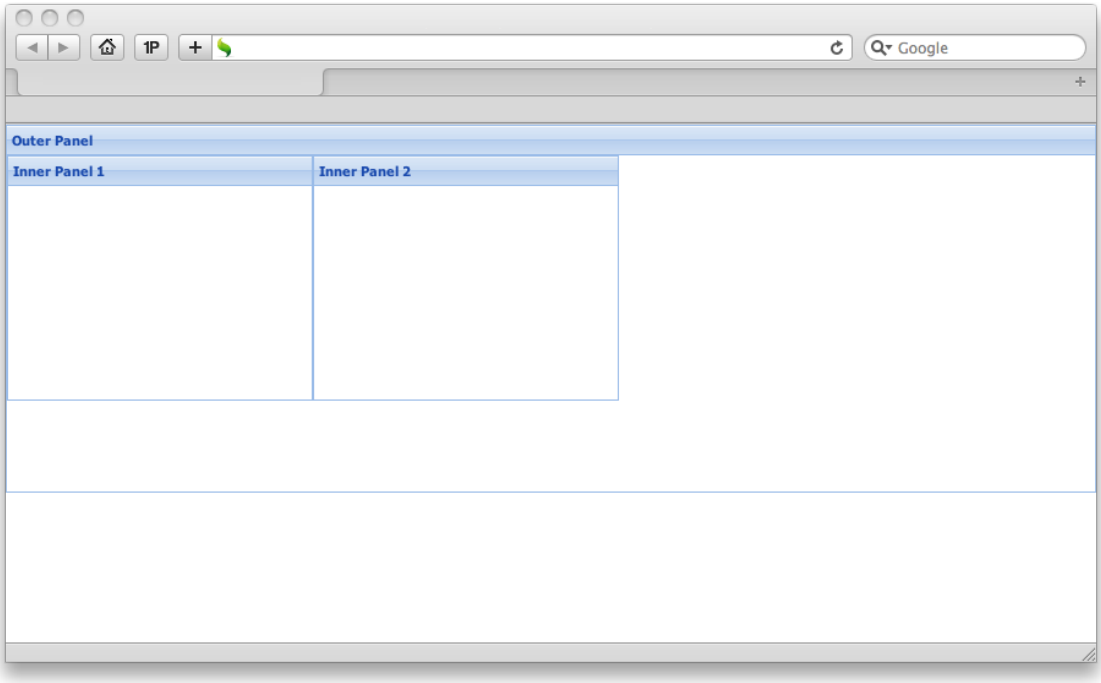



```
new Ext.panel.Panel({
```

```
 title: 'Outer Panel',
 renderTo: document.body,
 width: 400,
 layout: {
  type:'vbox',
  autoSize: true
 },
 autoScroll: true,
 items: [{
  xtype: 'panel',x
  title: 'Inner Panel 1',
  width: 250,
  height: 200
 },{
  xtype: 'panel',
  title: 'Inner Panel 2',
  width: 250,
  height: 200
 }]
});
```

Removing the *height* config option and keeping the *autoSize* option set to *true* will cause the Containing Panel to automatically adjust it's height to contain all its child components without the need for scrollbars.

Figure 20. HBox - Dynamic Container Width




```
new Ext.panel.Panel({
 title: 'Outer Panel',
 renderTo: document.body,
```

```
 height: 300,
 layout: {
  type:'hbox'
 },
 items: [{
  xtype: 'panel',
  title: 'Inner Panel 1',
  width: 250,
  height: 200
 },{
  xtype: 'panel',
  title: 'Inner Panel 2',
  width: 250,
  height: 200
 }]
});
```

Removing the *width* config option will cause the Panel to stretch to fit it's outer most constraint (the browser window in this example). The happens regardless of whether or not the *autoSize* config option is set. Unlike the VBoxLayout which expanded to fit the total heights of its child components HBoxLayouts won't follow that pattern and require some kind of containing element to determine their final width.

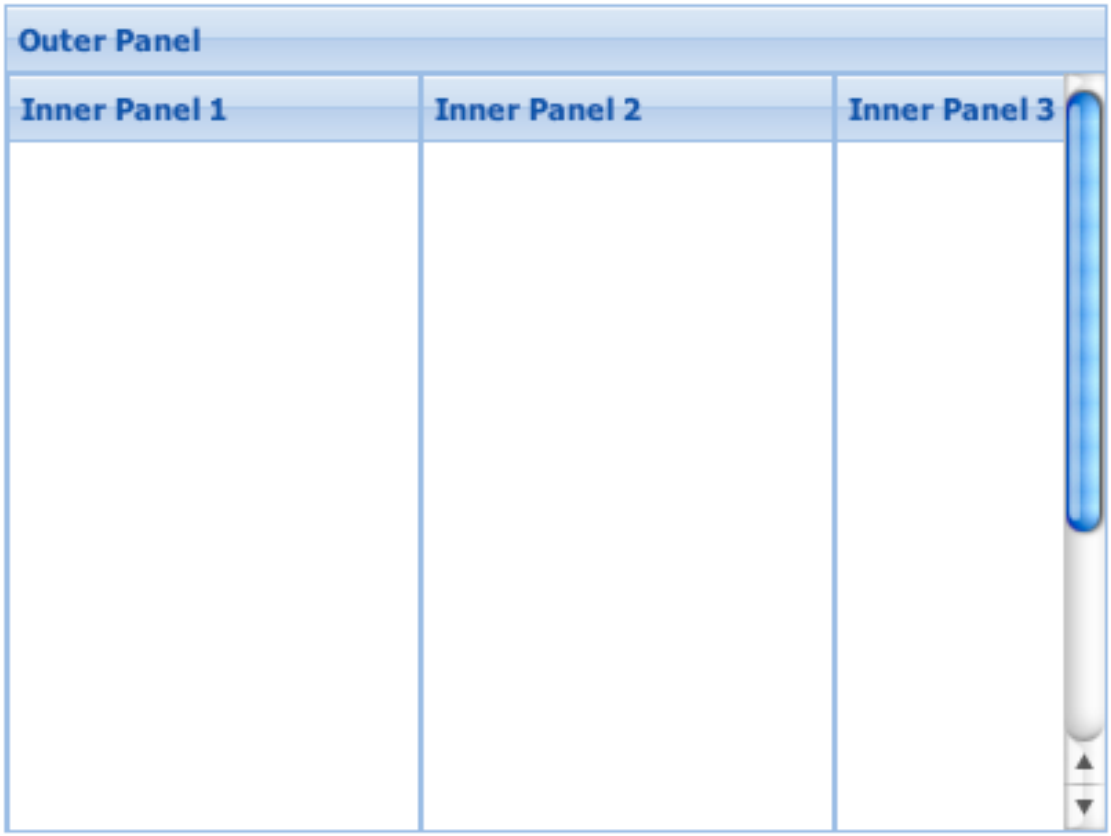
Note

In the example above when the browser window is resized the HBox will continue to keep the same width (calculated when it was rendered).

Scrolling ColumnLayout

Even though the *ColumnLayout* extends the *HBoxLayout* the way it handles scrolling is slightly different than HBox.

Figure 21. ColumnLayout With autoScroll



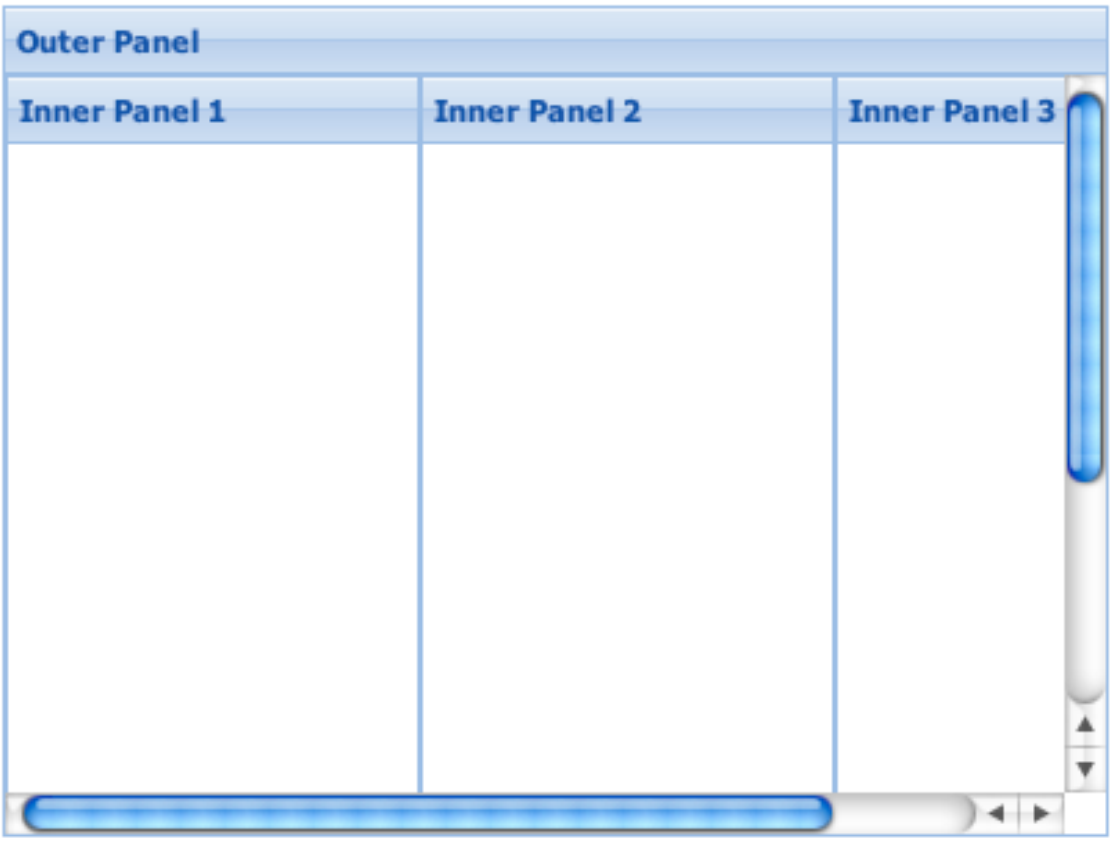

```
new Ext.panel.Panel({
 title: 'Outer Panel',
 renderTo: document.body,
 width: 400,
 height: 300,
 layout: {
  type: 'column'
 },
 autoScroll: true,
 items: [{
  xtype: 'panel',
  title: 'Inner Panel 1',
  width: 150,
  height: 400
 },{
  xtype: 'panel',
  title: 'Inner Panel 2',
  width: 150,
```

```
  height: 400
 },{
  xtype: 'panel',
  title: 'Inner Panel 3',
  width: 150,
  height: 400
 }]
});
```

Looking at the code you'll notice widths and heights have been set for all three inner Panels and the main Panel is using the *ColumnLayout*.

The widths of the three inner Panels add up to *450 pixels* while the width of the outer Panel is only *400 pixels*. The heights of the three inner Panels are all set to *400 pixels* while the outer Panel height is only *300 pixels*. Setting the *autoScroll* configuration option to *true* is the only thing needed to display a vertical scrollbar along the outer Panel. You might have noticed setting *autoScroll* alone doesn't provide a horizontal scrollbar.

Figure 22. ColumnLayout With autoScroll And autoSize




```
new Ext.panel.Panel({
 title: 'Outer Panel',
 renderTo: document.body,
 width: 400,
 height: 300,
```

```
 layout: {
  type: 'column',
  autoSize: true,
 },
 autoScroll: true,
 items: [{
  xtype: 'panel',
  title: 'Inner Panel 1',
  width: 150,
  height: 400
 },{
  xtype: 'panel',
  title: 'Inner Panel 2',
  width: 150,
  height: 400
 },{
  xtype: 'panel',
  title: 'Inner Panel 3',
  width: 150,
  height: 400
 }]
});
```

Setting the *autoSize* layout configuration option for the ColumnLayout in addition to setting the *autoScroll* component configuration option will display both vertical and horizontal scrollbars along the outer Panel.

The Unscrollables - Fit, Card and BorderLayouts

FitLayout, *CardLayout* and *BorderLayout* are all designed to automatically size themselves to fill the available space within their Containing Component. As a result these three layouts don't include support for displaying vertical or horizontal scrollbars.

If you find yourself in a situation where you need one of these layouts to provide scrollbars that's a good indication you're using the wrong Container Layout for the job. Consider using the *AutoLayout* or one of the Box Layouts (*HBox* or *VBox*) along with the appropriate configuration options.

Although BorderLayout doesn't provide scrollbars spanning the length of the Container, its interior regions are able to display both horizontal and vertical scrollbars. The next section shows an example of how scrolling works within these regions.

Scrolling Applications - Viewport

As developers we tend not to think about how our applications will handle scrolling until something doesn't work the way we were hoping. This section demonstrates different approaches to application layout with a focus on understanding how these choices impact the scrolling behavior of your application.

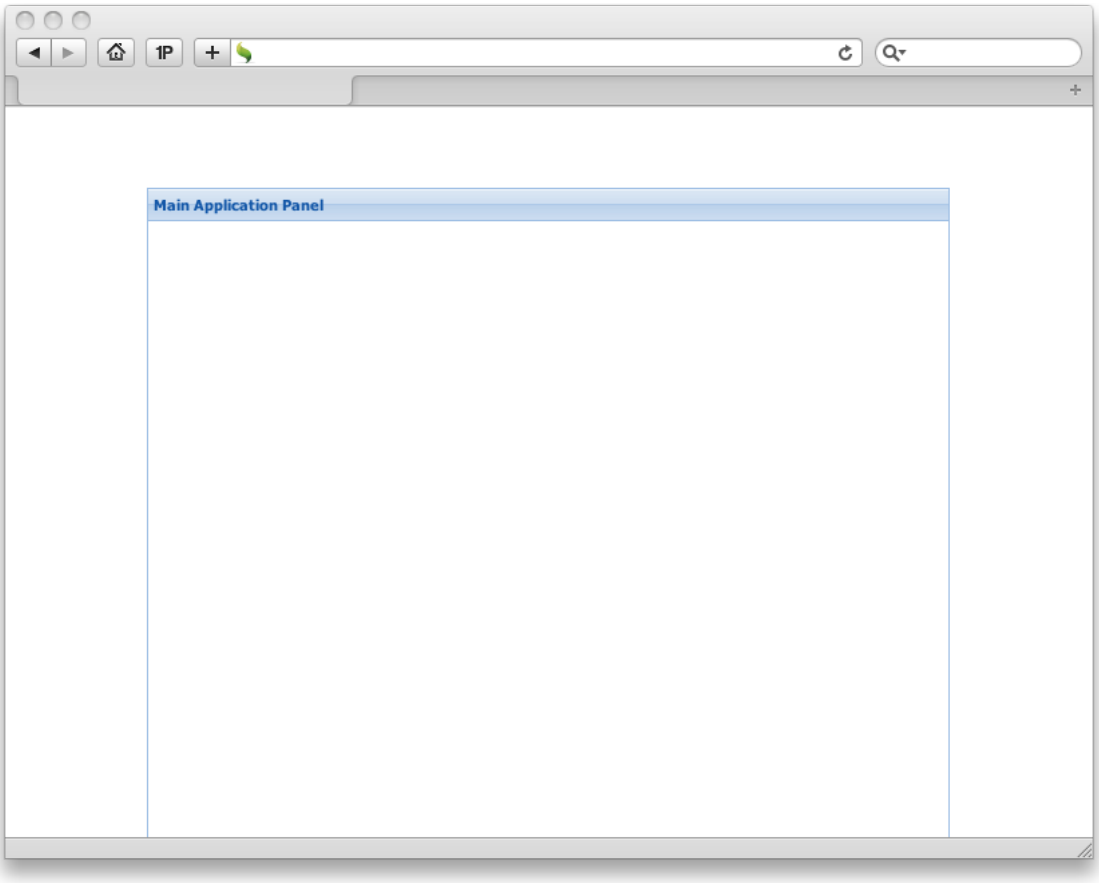
The *Viewport* is a great foundation to hold your UI components. When combined with the right Container Layouts and the full suite of Ext JS components you can achieve just about any interface design imaginable.

When you start combining the Viewport with different Container Layouts and then start nesting components within your layouts it can become easy to lose track of what component, layout or combination is responsible for affecting the scrolling behavior of your application.

Since Viewport extends *Ext.container.Container* it makes sense to think of a Viewport a special type of Container, with one important distinction. The Viewport is designed to **render into the BODY of your HTML document** and will **automatically resize as the browser resizes** to take up the **full space of the browser window**. Taking that functionality into account, by default the Viewport will clip any items that are outside of the viewable browser window just like a regular Container does. Also like a Container setting the *autoScroll* configuration option will provide scrollbars as needed as long as the layout your using supports scrolling.

The next example uses a Viewport with a VBoxLayout to demonstrate these concepts.

Figure 23. Centering a Panel in a Viewport



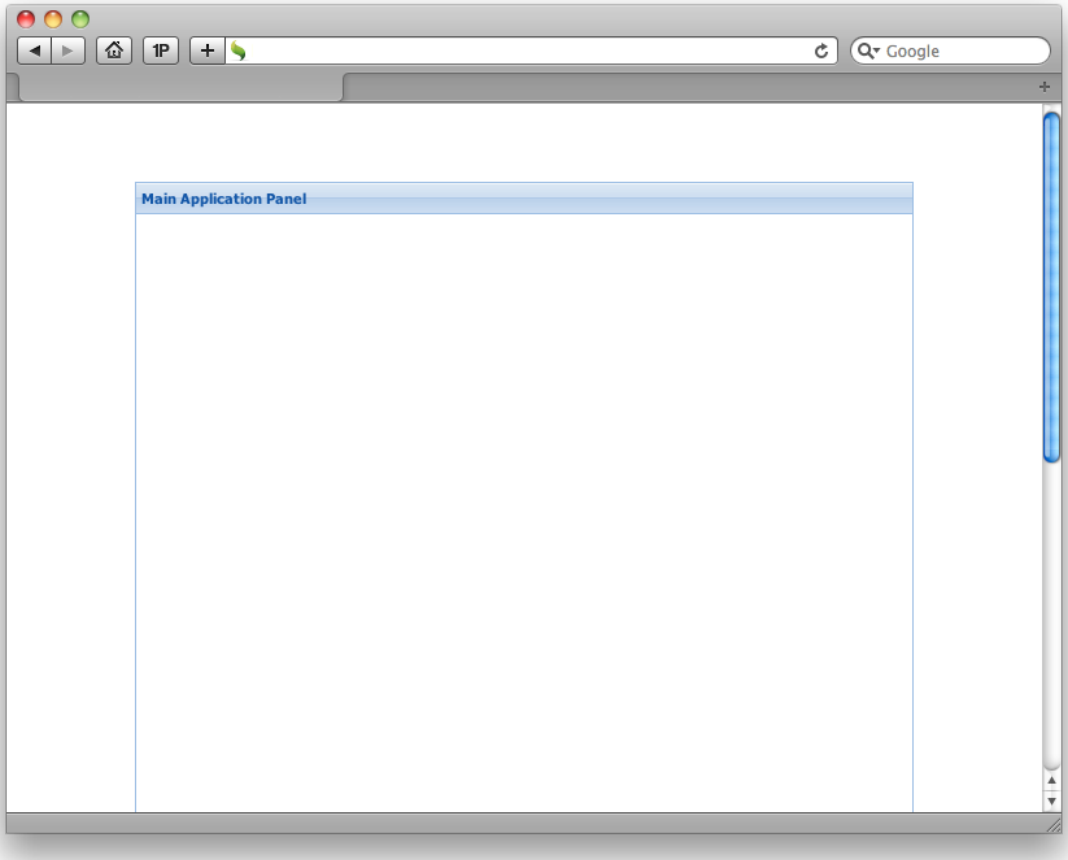

```
new Ext.Viewport({
 layout: {
  type:'vbox',
  align: 'center'
 },
 autoScroll: true,
 items: [{
  xtype: 'panel',
  title: 'Main Application Panel',
  width: 600,
  height: 900,
  margin: '60px 0 50px 0'
 }]
});
```

This example sets up a *Viewport* configured to use the *VBoxLayout*. Setting the *align* layout config option to *center* makes sure the Panel is always centered in the Viewport.

The *width* and *height* set for the Panel purposely exceed the available space in the browser window. Even though *autoScroll* has been set to true vertical scrollbars won't display in the browser. This is the default layout behavior of VBox which is expecting to manage the height of child components using the *flex* configuration option.

To get the desired vertical scrollbars to appear we need to add the *autoSize* Layout Configuration option.

Figure 24. Centering a Panel in a Viewport - With Vertical Scrolling




```
new Ext.Viewport({
 layout: {
  type:'vbox',
  align: 'center',
  autoSize: true   // provides vertical scrollbars
 },
 autoScroll: true,
 items: [{
  xtype: 'panel',
  title: 'Main Application Panel',
  width: 600,
  height: 900,
  margin: '60px 0 50px 0'
 }]
});
```

Setting the Layout Configuration option *autoSize* to *true* will automatically display vertical scrollbars when the height of child components exceeds the viewable area of the Viewport.

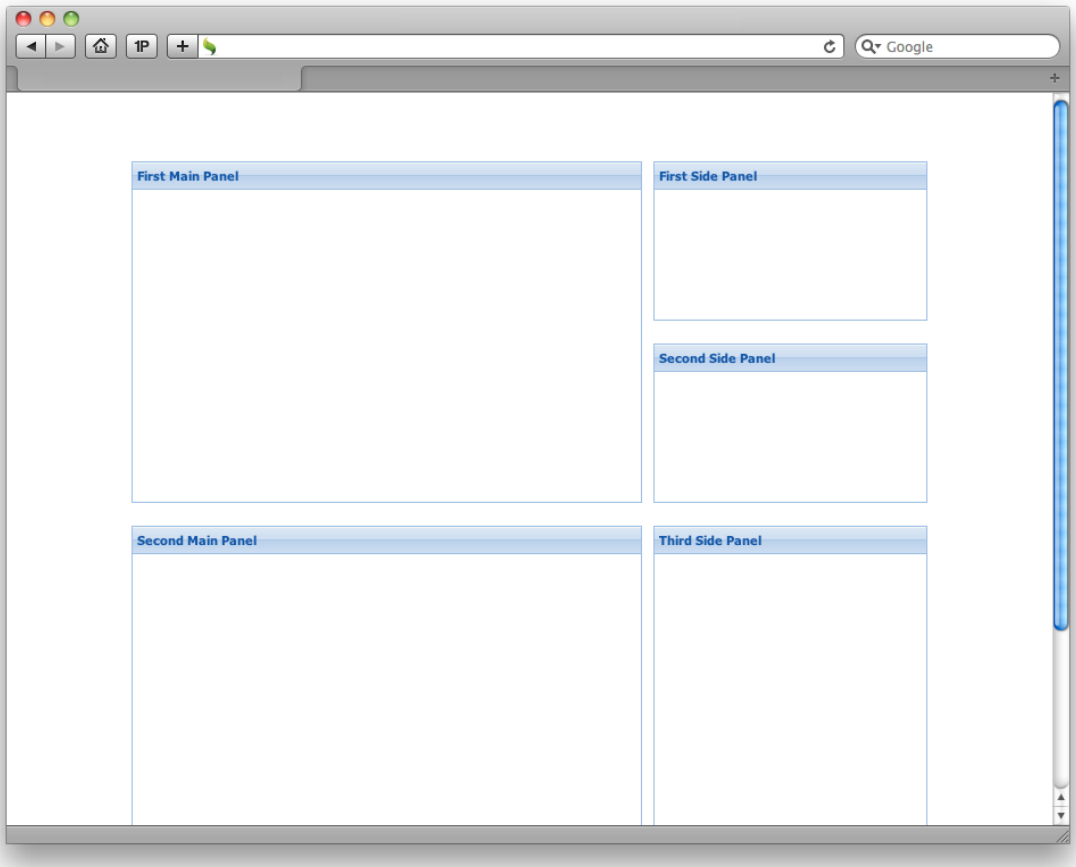
Note

Unlike *autoScroll* which is a **component configuration** option *autoSize* is a **layout configuration** option and needs to be placed inside your layout configuration object to function correctly.

ColumnLayout in a Viewport - With Vertical Scrolling

Combining the Viewport with a ColumnLayout is a great pattern for laying out column style application interfaces.

Figure 25.




```
new Ext.Viewport({
  layout: {
   type:'vbox',
   align: 'center',
   autoSize: true
  },
  autoScroll: true,
  items: [{
   xtype: 'container',
   width: 700,
```

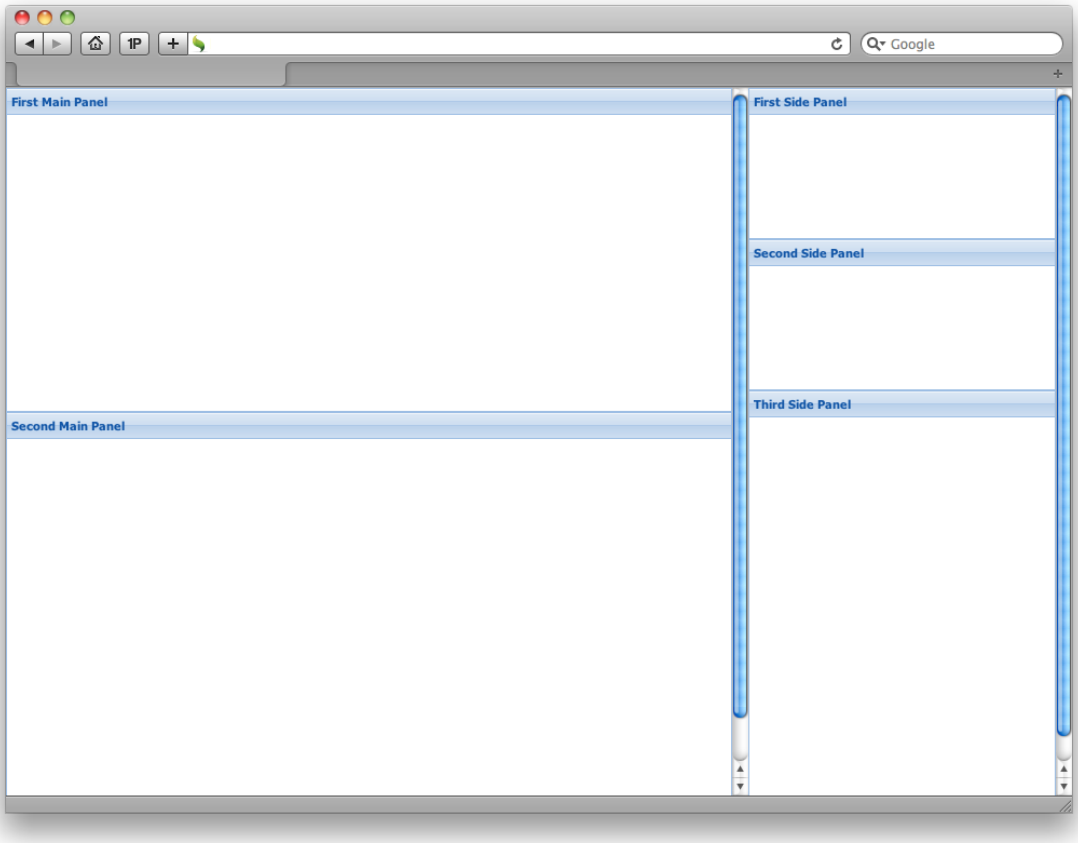
```
  margin: '60px 0 50px 0',
  layout: {
   type: 'column'
  },
  items:[{
   xtype: 'container',
   columnWidth: .65,
   items:[{
    xtype: 'panel',
    title: 'First Main Panel',
    height: 300,
    margin: '0 0 20px 0'
   }, {
    xtype: 'panel',
    title: 'Second Main Panel',
    height: 400
   }]
  },{
   xtype: 'container',
   width: 10
  },{
   xtype: 'container',
   columnWidth: .35,
   items: [{
    xtype: 'panel',
    title: 'First Side Panel',
    height: 140,
    margin: '0 0 20px 0'
   },{
    xtype: 'panel',
    title: 'Second Side Panel',
    height: 140,
    margin: '0 0 20px 0'
   },{
    xtype: 'panel',
    title: 'Third Side Panel',
    height: 400
   }]
  }]
 }]
});
```

ColumnLayout will automatically provide vertical scrollbars when *autoScroll* is set to *true*.

Scrolling Applications - BorderLayout

The BorderLayout is modeled after multi-pane desktop application interface layouts. Scrolling within the BorderLayout works just like it does in a desktop application, each region is responsible for creating and managing its own set of scrollbars.

Figure 26. BorderLayout in a Viewport - Region Scrolling




```
new Ext.Viewport({
 layout: {
  type:'border'
 },
 items:[{
  xtype: 'container',
  region: 'center',
  autoScroll: true,
  items:[{
   xtype: 'panel',
   title: 'First Main Panel',
   height: 300
  }, {
   xtype: 'panel',
   title: 'Second Main Panel',
   height: 400
  }]
```

```
 },{
  xtype: 'container',
  region: 'east',
  autoScroll: true,
  width: 300,
  items: [{
   xtype: 'panel',
   title: 'First Side Panel',
   height: 140
  },{
   xtype: 'panel',
   title: 'Second Side Panel',
   height: 140
  },{
   xtype: 'panel',
   title: 'Third Side Panel',
   height: 400
  }]
 }]
});
```

Setting *autoScroll* to *true* for the *center* and *east* regions automatically displayed scrollbars when their contents exceeded the viewable browser area.

Note

If you'd like your applications to scroll like more traditional web applications considering using VBox or an HBoxLayouts along with the ColumnLayout to create a similar design as BorderLayout regions.