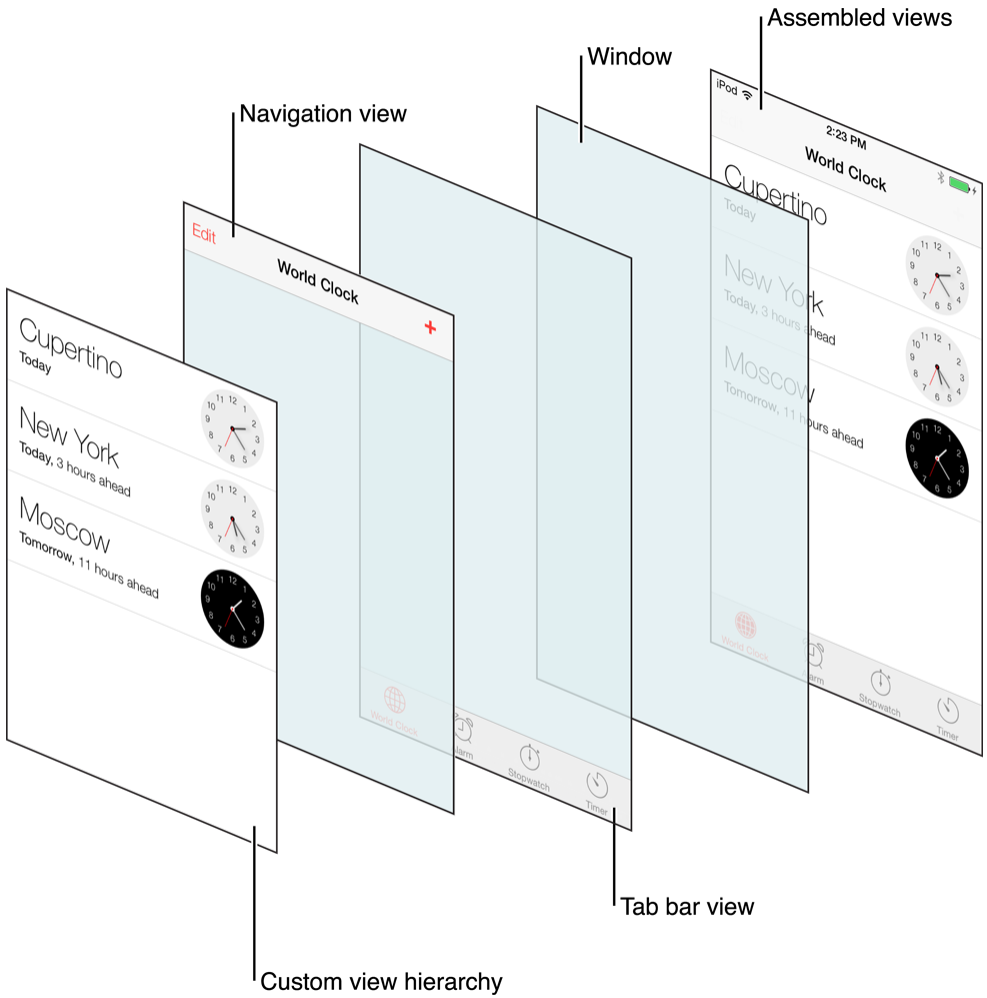
About Views

On This Page

Views are the building blocks for constructing your user interface. Rather than using one view to present your content, you are more likely to use several views, ranging from simple buttons and text labels to more complex views such as table views, picker views, and scroll views. Each view represents a particular portion of your user interface and is generally optimized for a specific type of content. By building view upon view, you get a view hierarchy.



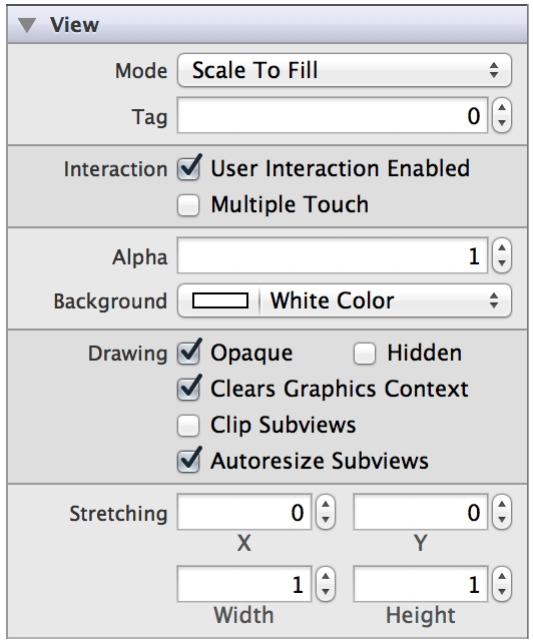
Purpose. Views allow users to:

- Experience app content
- Navigate within an app

Implementatior

On This Page

Configuration. Configure views in Interface Builder, in the View section of the Attributes Inspector. A few configurations cannot be made through the Attributes Inspector, so you must make them programmatically. You can set other configurations programmatically, too, if you prefer.



Content of Views

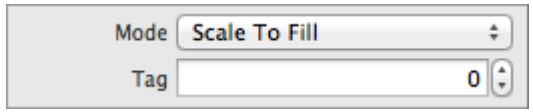
All views in UIKit are subclasses of the base class `UIView`. For example, UIKit has views specifically for presenting images, text, and other types of content. In places where the predefined views do not provide what you need, you can also define custom views and manage the drawing and event handling yourself.

Use the Mode (`contentMode`) field to specify how a view lays out its content when its bounds change. This property is often used to implement resizable controls. Instead of redrawing the contents of the view every

time its bounds change, you can use this property to specify that you want to scale the contents or pin them to a particular spot on the view.

The Tag (`tag`) field serves as an integer that you can use to identify view objects in your app.

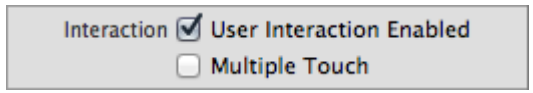
On This Page



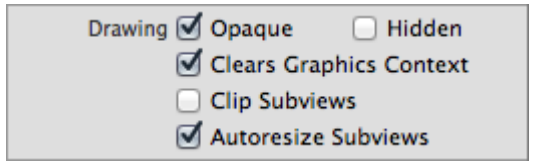
Behavior of Views

By default, the User Interaction Enabled (`userInteractionEnabled`) checkbox is selected, which means that user events—such as touch and keyboard—are delivered to the view normally. When the checkbox is unselected, events intended for the view are ignored and removed from the event queue.

The Multiple Touch (`multipleTouchEnabled`) checkbox is unselected by default, which means that the view receives only the first touch event in a multi-touch sequence. When selected, the view receives all touches associated with a multitouch sequence.



Views have a number of properties related to drawing behavior:



- The Opaque (`opaque`) checkbox tells the drawing system how it should treat the view. If selected, the drawing system treats the view as fully opaque, which allows the drawing system to optimize some

drawing operations and improve performance. If unselected, the drawing system composites the view normally with other content. You should always disable this checkbox if your view is fully or partially transparent.

- If the Hidd
  input ever

On This Page

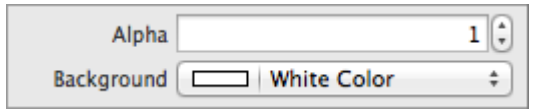
- When the Clears Graphics Context (`clearsContextBeforeDrawing`) checkbox is selected, the drawing buffer is automatically cleared to transparent black before the view is drawn. This behavior ensures that there are no visual artifacts left over when the view's contents are redrawn.
- Selecting the Clip Subviews (`clipsToBounds`) checkbox causes subviews to be clipped to the bounds of the view. If unselected, subviews whose frames extend beyond the visible bounds of the view are not clipped.
- When the Autoresize Subviews (`autoresizesSubviews`) checkbox is selected, the view adjusts the size of its subviews when its bounds change.

Appearance of Views

Background Color and Alpha

Adjusting the Alpha (`alpha`) field changes the transparency of the view as a whole. This value can range from `0.0` (transparent) to `1.0` (opaque). Setting a view's alpha value does not have an effect on embedded subviews.

Use the Background (`backgroundColor`) field to select a color to fill in the entire frame of the view. The background color appears underneath all other content in the view.



Appearance Proxies

You can use an appearance proxy to set particular appearance properties for all instances of a view in your application. For example, if you want all sliders in your app to have a particular minimum track tint color, you can specify this with a single message to the slider's appearance proxy.

There are two w
an instance of a container class.

On This Page

- To customize the appearance of all instances of a class, use `appearance` to get the appearance proxy for the class.

```
[[UISlider appearance] setMinimumTrackTintColor:[UIColor greenColor]];
```

- To customize the appearances for instances of a class when contained within an instance of a container class, or instances in a hierarchy, you use `appearanceWhenContainedIn:` to get the appearance proxy for the class.

```
[[UISlider appearanceWhenContainedIn:[UIView class], nil]
              setMinimumTrackTintColor:[UIColor greenColor]];
```

> NOTE
>
> You cannot use the appearance proxy with the `tintColor` property on `UIView`. For more information on using `tintColor`, see Tint Color.

Tint Color

Views have a `tintColor` property that specifies the color of key elements within the view. Each subclass of `UIView` defines its own appearance and behavior for `tintColor`. For example, this property determines the color of the outline, divider, and icons on a stepper:

The `tintColor` property is a quick and simple way to skin your app with a custom color. Setting a tint color for a view automatically sets that tint color for all of its subviews. However, you can manually override this property for any of those subviews and its descendants. In other words, each view inherits its superview's tint

color if its own tint color is `nil`. If the highest level view in the view hierarchy has a `nil` value for tint color, it defaults to the system blue color.

On This Page

Template Images

In iOS 7, you can choose to treat any of the images in your app as a template—or stencil—image. When you elect to treat an image as a template, the system ignores the image's color information and creates an image stencil based on the alpha values in the image (parts of the image with an alpha value of less than `1.0` get treated as completely transparent). This stencil can then be recolored using `tintColor` to match the rest of your user interface.

By default, an image (`UIImage`) is created with `UIImageRenderingModeAutomatic`. If you have `UIImageRenderingModeAutomatic` set on your image, it will be treated as template or original based on its context. Certain UIKit elements—including navigation bars, tab bars, toolbars, segmented controls—automatically treat their foreground images as templates, although their background images are treated as original. Other elements—such as image views and web views—treat their images as originals. If you want your image to always be treated as a template regardless of context, set `UIImageRenderingModeAlwaysTemplate`; if you want your image to always be treated as original, set `UIImageRenderingModeAlwaysOriginal`.

To specify the rendering mode of an image, first create a standard image, and then call the `imageWithRenderingMode:` method on that image.

```
UIImage *myImage = [UIImage imageNamed:@"myImageFile.png"];
myImage = [myImage imageWithRenderingMode:UIImageRenderingModeAlwaysTemplate];
```

Using Auto Layout with Views

The Auto Layout system allows you to define layout constraints for user interface elements, such as views and controls. Constraints represent relationships between user interface elements. You can create Auto Layout constraints by selecting the appropriate element or group of elements and selecting an option from the menu in the bottom right corner of Xcode's Interface Builder.

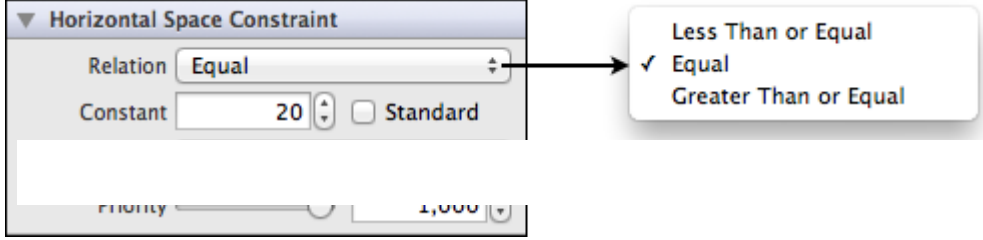
Auto layout contains two menus of constraints: pin and align. The Pin menu allows you to specify constraints that define some relationship based on a particular value or range of values. Some apply to the control itself

(width) while others define relationships between elements (horizontal spacing). The following tables describes what each group of constraints in the Auto Layout menu accomplishes:

On This Page

Constraint Nar	
Width Height	Sets the width or height of a single element.
Horizontal Spacing Vertical Spacing	Sets the horizontal or vertical spacing between exactly two elements.
Leading Space to Superview Trailing Space to Superview Top Space to Superview Bottom Space to Superview	Sets the spacing from one or more elements to the leading, trailing, top, or bottom of their container view. Leading and trailing are the same as left and right in English, but the UI flips when localized in a right-to-left environment.
Widths Equally Heights Equally	Sets the widths or heights of two or more elements equal to each other.
Left Edges Right Edges Top Edges Bottom Edges	Aligns the left, right, top, or bottom edges of two or more elements.
Horizontal Centers Vertical Centers Baselines	Aligns two or more elements according to their horizontal centers, vertical centers, or bottom baselines. Note that baselines are different from bottom edges. These values may not be defined for certain elements.
Horizontal Center in Container Vertical Center in Container	Aligns the horizontal or vertical centers of one or more elements with the horizontal or vertical center of their container view.

The “Constant” value specified for any Pin constraints (besides Widths/Heights Equally) can be part of a “Relation.” That is, you can specify whether you want the value of that constraint to be equal to, less than or equal to, or greater than or equal to the value.



On This Page

For more information, see *Auto Layout Guide*.

Making Views Accessible

To enhance the accessibility information for an item, select the object on the storyboard and open the Accessibility section of the Identity inspector.

For more information, see *Accessibility Programming Guide for iOS*.

Debugging Views

When debugging issues with views, watch for this common pitfall:

Setting conflicting opacity settings. You should not set the `opaque` property to `YES` if your view has an alpha value of less than `1.0`.