Design Tips

On This Page

View controllers are an essential tool for apps running on iOS, and the view controller infrastructure of UIKit makes it easy to create sophisticated interfaces without writing a lot of code. When implementing your own view controllers, use the following tips and guidelines to ensure that you are not doing things that might interfere with the natural behavior expected by the system.

Use System-Supplied View Controllers Whenever Possible

Many iOS frameworks define view controllers that you can use as-is in your apps. Using these system-supplied view controllers saves time for you and ensures a consistent experience for the user.

Most system view controllers are designed for specific tasks. Some view controllers provide access to user data such as contacts. Others might provide access to hardware or provide specially tuned interfaces for managing media. For example, the `UIImagePickerController` class in UIKit displays a standard interface for capturing pictures and video and for accessing the user's camera roll.

Before you create your own custom view controller, look at the existing frameworks to see if a view controller already exists for the task you want to perform.

- The UIKit framework provides view controllers for displaying alerts, taking pictures and video, and managing files on iCloud. UIKit also defines many standard container view controllers that you can use to organize your content.
- The GameKit framework provides view controllers for matching players and for managing leaderboards, achievements, and other game features.
- The Address Book UI framework provides view controllers for displaying and picking contact information.
- The MediaPlayer framework provides view controllers for playing and managing video, and for choosing media assets from the user's library.
- The EventKit UI framework provides view controllers for displaying and editing the user's calendar data.
- The GLKit framework provides a view controller for managing an OpenGL rendering surface.
- The Multipeer Connectivity framework provides view controllers for detecting other users and inviting them to connect.
- The Message UI framework provides view controllers for composing emails and SMS messages.
- The PassKit framework provides view controllers for displaying passes and adding them to Passbook.
- The Social framework provides view controllers for composing messages for Twitter, Facebook, and other social media sites.
- The AVFoundation framework provides a view controller for displaying media assets.

> IMPORTANT
>
> Never modify the view hierarchy of system-provided view controllers. Each view controller owns its view hierarchy and is responsible for maintaining the integrity of that hierarchy. Making changes might introduce bugs into your code or prevent the owning view controller from operating correctly. In the case of system view controllers, always rely on the publicly available methods and properties of the view controller to make all modifications.

For information about using a specific view controller, see the reference documentation for the corresponding framework.

Make Each View Controller an Island

View controllers should always be self-contained objects. No view controller should have knowledge about the internal workings or view hierarchy of another view controller. In cases where two view controllers need to communicate or pass data back and forth, they should always do so using explicitly defined public interfaces.

The delegation design pattern is frequently used to manage communication between view controllers. With delegation, one object defines a protocol for communicating with an associated delegate object, which is any object that conforms to the protocol. The exact type of the delegate object is unimportant. All that matters is that it implements the methods of the protocol.

Use the Root View Only as a Container for Other Views

Use the root vie
as a container gives all of your views a common parent view, which makes many layout operations simpler. Many Auto Layout constraints require a common parent view to lay out the views properly.

On This Page

Know Where Your Data Lives

In the model-view-controller design pattern, a view controller's role is to facilitate the movement of data between your model objects and your view objects. A view controller might store some data in temporary variables and perform some validation, but its main responsibility is to ensure that its views contain accurate information. Your data objects are responsible for managing the actual data and for ensuring the overall integrity of that data.

An example of the separation of data and interface exists in the relationship between the `UIDocument` and `UIViewController` classes. Specifically, no default relationship exists between the two. A `UIDocument` object coordinates the loading and saving of data, while a `UIViewController` object coordinates the display of views onscreen. If you create a relationship between the two objects, remember that the view controller should only cache information from the document for efficiency. The actual data still belongs to the document object.

Adapt to Changes

Apps can run on a variety of iOS devices, and view controllers are designed to adapt to different-sized screens on those devices. Rather than use separate view controllers to manage content on different screens, use the built-in adaptivity support to respond to size and size class changes in your view controllers. The notifications sent by UIKit give you the opportunity to make both large-scale and small-scale changes to your user interface without having to change the rest of your view controller code.

For more information about handling adaptivity changes, see The Adaptive Model.