**App Life Cycle**

* Apps are a sophisticated interplay between our custom code and the system frameworks.
* The system frameworks provide the basic infrastructure that all apps need to run, and you provide the code required to customize that infrastructure and give the app the look and feel we want.
* To do that effectively, it helps to understand a little bit about the iOS infrastructure and how it works.
* iOS frameworks rely on design patterns such as model-view-controller and delegation in their implementation.
* Understanding those design patterns is crucial to the successful creation of an app.

**The Structure of an App:**

* During startup, the [UIApplicationMain](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIKitFunctionReference/index.html" \l "//apple_ref/c/func/UIApplicationMain" \t "_self) function sets up several key objects and starts the app running.
* At the heart of every iOS app is the [UIApplication](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIApplication_Class/index.html" \l "//apple_ref/occ/cl/UIApplication" \t "_self) object, whose job is to facilitate the interactions between the system and other objects in the app.
* The first thing to notice is that iOS apps use a model-view-controller architecture.
* This pattern separates the app’s data and business logic from the visual presentation of that data.
* This architecture is crucial to creating apps that can run on different devices with different screen sizes.
* The Model-View-Controller (MVC) design pattern assigns objects in an application one of three roles: model, view, or controller.
* The pattern defines not only the roles objects play in the application, it defines the way objects communicate with each other.
* Each of the three types of objects is separated from the others by abstract boundaries and communicates with objectss of the other types across those boundaries.
* MVC is central to a good design for a Cocoa application.
* The benefits of adopting this pattern are numerous.
* Many objects in these applications tend to be more reusable, and their interfaces tend to be better defined.
* Applications having an MVC design are also more easily extensible than other applications.

## Model Objects:

* Model objects encapsulate the data specific to an application and define the logic and computation that manipulate and process that data.
* For example, a model object might represent a character in a game or a contact in an address book.
* A model object can have to-one and to-many relationships with other model objects, and so sometimes the model layer of an application effectively is one or more object graphs.
* Much of the data that is part of the persistent state of the application (whether that persistent state is stored in files or databases) should reside in the model objects after the data is loaded into the application.
* Because model objects represent knowledge and expertise related to a specific problem domain, they can be reused in similar problem domains.
* Ideally, a model object should have no explicit connection to the view objects that present its data and allow users to edit that data—it should not be concerned with user-interface and presentation issues.

**Communication**:

* User actions in the view layer that create or modify data are communicated through a controller object and result in the creation or updating of a model object.
* When a model object changes (for example, new data is received over a network connection), it notifies a controller object, which updates the appropriate view objects.

## View Objects:

* A view object is an object in an application that users can see.
* A view object knows how to draw itself and can respond to user actions.
* A major purpose of view objects is to display data from the application’s model objects and to enable the editing of that data.
* Despite this, view objects are typically decoupled from model objects in an MVC application.
* Because we typically reuse and reconfigure them, view objects provide consistency between applications.
* Both the UIKit and AppKit frameworks provide collections of view classes, and Interface Builder offers dozens of view objects in its Library.

**Communication**:

* View objects learn about changes in model data through the application’s controller objects and communicate user-initiated changes—for example, text entered in a text field—through controller objects to an application’s model objects.

## Controller Objects

* A controller object acts as an intermediary between one or more of an application’s view objects and one or more of its model objects.
* Controller objects are thus a conduit through which view objects learn about changes in model objects and vice versa.
* Controller objects can also perform setup and coordinating tasks for an application and manage the life cycles of other objects.

**Communication**:

* A controller object interprets user actions made in view objects and communicates new or changed data to the model layer.
* When model objects change, a controller object communicates that new model data to the view objects so that they can display it.

**Key Objects in IOS App**

MODEL



Data Objects

Document

Event Loop

UIWindow

View and UI Objects

View Controller

Application Delegate

a

UIApllication

VIEW

CONTROLLER

[**UIApplication**](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIApplication_Class/index.html#//apple_ref/occ/cl/UIApplication)**object:**

* The UIApplication object manages the event loop and other high-level app behaviors.
* It also reports key app transitions and some special events (such as incoming push notifications) to its delegate, which is a custom object we define.
* The UIApplication class provides a centralized point of control and coordination for apps running in iOS.
* Every app has exactly one instance of UIApplication (or, very rarely, a subclass of UIApplication).
* When an app is launched, the system calls the [UIApplicationMain](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIKitFunctionReference/index.html" \l "//apple_ref/c/func/UIApplicationMain) function; among its other tasks, this function creates a [singleton](https://developer.apple.com/library/ios/documentation/General/Conceptual/DevPedia-CocoaCore/Singleton.html#//apple_ref/doc/uid/TP40008195-CH49) UIApplication object.
* Thereafter we access the object by calling the [sharedApplication](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIApplication_Class/index.html" \l "//apple_ref/occ/clm/UIApplication/sharedApplication) class method.

**App delegateobject:**

* The app delegate is the heart of our custom code.
* This object works in tandem(one behind other) with the UIApplication object to handle app initialization, state transitions, and many high-level app events.
* This object is also the only one guaranteed to be present in every app, so it is often used to set up the app’s initial data structures.
* Delegation is a simple and powerful pattern in which one object in a program acts on behalf of, or in coordination with, another object.
* The delegating object keeps a reference to the other object—the delegate—and at the appropriate time sends a message to it.
* The message informs the delegate of an event that the delegating object is about to handle or has just handled.
* The delegate may respond to the message by updating the appearance or state of itself or other objects in the application, and in some cases it can return a value that affects how an impending event is handled.
* The main value of delegation is that it allows us to easily customize the behavior of several objects in one central object.

**Documents and data model objects:**

* Data model objects store our app’s content and are specific to our app.
* For example, a banking app might store a database containing financial transactions, whereas College database containing the student information and faculty information of respective departments.
* Apps can also use document objects(custom subclasses of [UIDocument](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIDocument_Class/index.html" \l "//apple_ref/occ/cl/UIDocument" \t "_self)) to manage some or all of their data model objects.
* Document objects are not required but offer a convenient way to group data that belongs in a single file or file package.

**View controller objects:**

* View controller objects manage the presentation of our app’s content on screen.
* A view controller manages a single view and its collection of subviews.
* When presented, the view controller makes its views visible by installing them in the app’s window.
* The [UIViewController](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIViewController_Class/index.html" \l "//apple_ref/occ/cl/UIViewController" \t "_self) class is the base class for all view controller objects.
* It provides default functionality for loading views, presenting them, rotating them in response to device rotations, and several other standard system behaviors.
* UIKit and other frameworks define additional view controller classes to implement standard system interfaces such as the image picker, tab bar interface, and navigation interface.
* The UIViewController class provides the infrastructure for managing the views of our iOS apps.
* A view controller manages a set of views that make up a portion of our app’s user interface.
* It is responsible for loading and disposing of those views, for managing interactions with those views, and for coordinating responses with any appropriate data objects.
* View controllers also coordinate their efforts with other controller objects—including other view controllers—and help manage our app’s overall interface.
* We rarely create instances of the UIViewController class directly.
* Instead, we create instances ofUIViewController subclasses and use those objects to provide the specific behaviors and visual appearances that we need.
* A view controller’s main responsibilities include the following:
* Updating the contents of the views, usually in response to changes to the underlying data.
* Responding to user interactions with views.
* Resizing views and managing the layout of the overall interface.
* A view controller is tightly bound to the views it manages and takes part in the responder chain used to handle events.
* View controllers are also [UIResponder](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIResponder_Class/index.html" \l "//apple_ref/occ/cl/UIResponder) objects and are inserted into the responder chain between the view controller’s root view and that view’s superview, which typically belongs to a different view controller.
* If none of the view controller’s views handle an event, the view controller has the option of handling the event or passing it along to the superview.

[**UIWindow**](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIWindow_Class/index.html#//apple_ref/occ/cl/UIWindow)**object:**

* A [UIWindow](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIWindow_Class/index.html" \l "//apple_ref/occ/cl/UIWindow" \t "_self) object coordinates the presentation of one or more views on a screen.
* Most apps have only one window, which presents content on the main screen, but apps may have an additional window for content displayed on an external display.
* To change the content of our app, you use a view controller to change the views displayed in the corresponding window. We never replace the window itself.
* In addition to hosting views, windows work with the UIApplication object to deliver events to our views and view controllers.
* The UIWindow class defines an object known as a **window** that manages and coordinates the views an app displays on a device screen.
* Unless an app can display content on an external device screen, an app has only one window.

**View objects,control objects, and layer objects:**

* Views and controls provide the visual representation of our app’s content.
* A view is an object that draws content in a designated rectangular area and responds to events within that area.
* Controlsare a specialized type of view responsible for implementing familiar interface objects such as buttons, text fields, and toggle switches.
* The UIKit framework provides standard views for presenting many different types of content.
* We can also define our own custom views by subclassing[UIView](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIView_Class/index.html#//apple_ref/occ/cl/UIView) (or its descendants) directly.
* Layer objects are actually data objects that represent visual content.
* Views use layer objects intensively behind the scenes to render their content.
* We can also add custom layer objects to our interface to implement complex animations and other types of sophisticated visual effects.

**Different States for an app**

**Not Running**

**Suspended**

**BackGround**

**BackGround**

**ForeGround**

**Active**

**Inactive**

**Background**

* At any given moment, our app is in one of the states.
* The system moves our app from state to state in response to actions happening throughout the system.

The Different states of an app is.

1. **Not Running:**

The app has not been launched or was running but was terminated by the system.

1. **Inactive:**

The app is running in the foreground but is currently not receiving events. (It may be executing other code though.) An app usually stays in this state only briefly as it transitions to a different state.

1. **Active:**

The app is running in the foreground and is receiving events. This is the normal mode for foreground apps.

1. **Background:**

The app is in the background and executing code. Most apps enter this state briefly on their way to being suspended. However, an app that requests extra execution time may remain in this state for a period of time. In addition, an app being launched directly into the background enters this state instead of the inactive state.

1. **Suspended:**

The app is in the background but is not executing code. The system moves apps to this state automatically and does not notify them before doing so. While suspended, an app remains in memory but does not execute any code.

When a low-memory condition occurs, the system may purge suspended apps without notice to make more space for the foreground app.

Most state transitions are accompanied by a corresponding call to the methods of your app delegate object. These methods are your chance to respond to state changes in an appropriate way. These methods are listed below, along with a summary of how you might use them.

* [**application:willFinishLaunchingWithOptions:**](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIApplicationDelegate_Protocol/index.html#//apple_ref/occ/intfm/UIApplicationDelegate/application:willFinishLaunchingWithOptions:)—This method is your app’s first chance to execute code at launch time.
* [**application:didFinishLaunchingWithOptions**:](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIApplicationDelegate_Protocol/index.html#//apple_ref/occ/intfm/UIApplicationDelegate/application:didFinishLaunchingWithOptions:)—This method allows you to perform any final initialization before your app is displayed to the user.
* [**applicationDidBecomeActive:**](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIApplicationDelegate_Protocol/index.html#//apple_ref/occ/intfm/UIApplicationDelegate/applicationDidBecomeActive:)—Lets your app know that it is about to become the foreground app. Use this method for any last minute preparation.
* [**applicationWillResignActive**:](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIApplicationDelegate_Protocol/index.html#//apple_ref/occ/intfm/UIApplicationDelegate/applicationWillResignActive:)—Lets you know that your app is transitioning away from being the foreground app. Use this method to put your app into a quiescent state.
* [**applicationDidEnterBackground:**](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIApplicationDelegate_Protocol/index.html#//apple_ref/occ/intfm/UIApplicationDelegate/applicationDidEnterBackground:)—Lets you know that your app is now running in the background and may be suspended at any time.
* [**applicationWillEnterForeground:**](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIApplicationDelegate_Protocol/index.html#//apple_ref/occ/intfm/UIApplicationDelegate/applicationWillEnterForeground:)—Lets you know that your app is moving out of the background and back into the foreground, but that it is not yet active.
* [**applicationWillTerminate:**](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIApplicationDelegate_Protocol/index.html#//apple_ref/occ/intfm/UIApplicationDelegate/applicationWillTerminate:)—Lets us know that our app is being terminated. This method is not called if our app is suspended.

## App Termination:

* Apps must be prepared for termination to happen at any time and should not wait to save user data or perform other critical tasks.
* System-initiated termination is a normal part of an app’s life cycle.
* The system usually terminates apps so that it can reclaim memory and make room for other apps being launched by the user, but the system may also terminate apps that are misbehaving or not responding to events in a timely manner.
* Suspended apps receive no notification when they are terminated; the system kills the process and reclaims the corresponding memory.
* If an app is currently running in the background and not suspended, the system calls the [applicationWillTerminate:](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIApplicationDelegate_Protocol/index.html" \l "//apple_ref/occ/intfm/UIApplicationDelegate/applicationWillTerminate:" \t "_self) of its app delegate prior to termination.
* The system does not call this method when the device reboots.
* In addition to the system terminating your app, the user can terminate our app explicitly using the multitasking UI.
* User-initiated termination has the same effect as terminating a suspended app.
* The app’s process is killed and no notification is sent to the app.

**Threads and Concurrency:**

* The system creates our app’s main thread and we can create additional threads, as needed, to perform other tasks.
* For iOS apps, the preferred technique is to use Grand Central Dispatch (GCD), operation objects, and other asynchronous programming interfaces rather than creating and managing threads our self.
* Technologies such as GCD let us define the work we want to do and the order we want to do it in, but let the system decide how best to execute that work on the available CPUs.
* Letting the system handle the thread management simplifies the code we must write, makes it easier to ensure the correctness of that code, and offers better overall performance.
* Work involving views, Core Animation, and many other UIKit classes usually must occur on the app’s main thread.
* There are some exceptions to this rule—for example, image-based manipulations can often occur on background threads—but when in doubt, assume that work needs to happen on the main thread.
* Lengthy tasks (or potentially length tasks) should always be performed on a background thread.
* Any tasks involving network access, file access, or large amounts of data processing should all be performed asynchronously using GCD or operation objects.
* At launch time, move tasks off the main thread whenever possible.
* At launch time, our app should use the available time to set up its user interface as quickly as possible.
* Only tasks that contribute to setting up the user interface should be performed on the main thread.
* All other tasks should be executed asynchronously, with the results displayed to the user as soon as they are ready.

**The Main Run Loop:**

* An app’s main run loop processes all user-related events.
* The [UIApplication](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIApplication_Class/index.html" \l "//apple_ref/occ/cl/UIApplication" \t "_self) object sets up the main run loop at launch time and uses it to process events and handle updates to view-based interfaces.
* As the name suggests, the main run loop executes on the app’s main thread.
* This behavior ensures that user-related events are processed serially in the order in which they were received.
* As the user interacts with a device, events related to those interactions are generated by the system and delivered to the app via a special port set up by UIKit.
* Events are queued internally by the app and dispatched one-by-one to the main run loop for execution.
* The UIApplication object is the first object to receive the event and make the decision about what needs to be done.
* A touch event is usually dispatched to the main window object, which in turn dispatches it to the view in which the touch occurred.
* Other events might take slightly different paths through various app objects.