Mac Technology Overview

Developer

Contents

Figures and Tables

Objective-CSwift

About Developing for Mac

The OS X operating system combines a stable core with advanced technologies to help you deliver world-class products on the Mac platform. Knowing what these technologies are, and how to use them, can help streamline your development process, while giving you access to key OS X features.

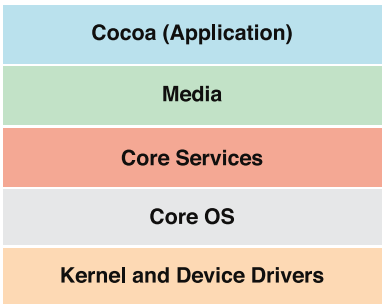
At a Glance

This guide introduces you to the range of possibilities for developing Mac software, describes the many technologies you can use for software development, and points you to sources of information about those technologies. It does not describe user-level system features or features that have no impact on software development.

OS X Has a Layered Architecture with Key Technologies in Each Layer

It's helpful to view the implementation of OS X as a set of layers. The lower layers of the system provide the fundamental services on which all software relies. Subsequent layers contain more sophisticated services and technologies that build on (or complement) the layers below.

Figure I-1 Layers of OS X



The lower the layer a technology is in, the more specialized are the services it provides. Generally, technologies in higher layers incorporate lower-level technologies to provide common app behaviors. A good rule of thumb is to use the highest-level programming interface that meets the goals of your app. Here is a brief summary of the layers of OS X.

- The Cocoa (Application) layer includes technologies for building an app's user interface, for responding to user events, and for managing app behavior.
- The Media layer encompasses specialized technologies for playing, recording, and editing audiovisual media and for rendering and animating 2D and 3D graphics.
- The Core Services layer contains many fundamental services and technologies that range from Automatic Reference Counting and low-level network communication to string manipulation and data formatting.
- The Core OS layer defines programming interfaces that are related to hardware and networking, including interfaces for running high-performance computation tasks on a computer's CPU and GPU.
- The Kernel and Device Drivers layer consists of the Mach kernel environment, device drivers, BSD library functions (`libSystem`), and other low-level components. The layer includes support for file systems, networking, security, interprocess communication, programming languages, device drivers, and extensions to the kernel.

Relevant Chapters: Cocoa Application Layer (page 26), Media Layer (page 35), Core Services Layer (page 53), Core OS Layer (page 68), Kernel and Device Drivers Layer (page 72)

You Can Create Many Different Kinds of Software for Mac

Using the developer tools and system frameworks, you can develop a wide variety of software for Mac, including the following:

- **Apps.** Apps help users accomplish tasks that range from creating content and managing data to connecting with others and having fun. OS X provides a wealth of system technologies such as app extensions and handoff, that you use to extend the capabilities of your apps and enhance the experience of your users.
- **Frameworks and libraries.** Frameworks and libraries enable code sharing among apps.
- **Command-line tools and daemons.** Command-line tools allow sophisticated users to manipulate data in the command-line environment of the Terminal app. Daemons typically run continuously and act as servers for processing client requests.
- **App plug-ins and loadable bundles.** Plug-ins extend the capabilities of other apps; bundles contain code and resources that apps can dynamically load at runtime.
- **System plug-ins.** System plug-ins, such as audio units, kernel extensions, I/O Kit device drivers, preference panes, Spotlight importers, and screen savers, extend the capabilities of the system.

Relevant Chapter: Creating Software Products for the Mac Platform (page 13)

When Porting a Cocoa Touch App, Be Aware of API Similarities and Differences

The technology stacks on which Cocoa and Cocoa Touch apps are based have many similarities. Some system frameworks are identical (or nearly identical) in each platform, including Foundation, Core Data, and AV Foundation. This commonality of API makes some migration tasks—for example, porting the data model of your Cocoa Touch app—easy.

Other migration tasks are more challenging because they depend on frameworks that reflect the differences between the platforms. For example, porting controller objects and revising the user interface are more demanding tasks because they depend on AppKit and UIKit, which are the primary app frameworks in the Cocoa and CocoaTouch layers, respectively.

Relevant Chapter: Migrating from Cocoa Touch (page 88)

See Also

Apple provides developer tools and additional information that support your development efforts.

Xcode, Apple's integrated development environment, helps you design, create, debug, and optimize your software. You can download Xcode from the Mac App Store.

For an overview of the developer tools for OS X, see the Xcode Apple Developer webpage. For an overview Xcode functionality, read *Xcode Overview*.

The OS X Developer Library contains the documentation, sample code, tutorials, and other information you need to write OS X apps. You can access the OS X Developer Library from the Apple Developer website or from Xcode. In Xcode, choose Help > Documentation and API Reference to view documents and other resources in the Organizer window.

In addition to the OS X Developer Library, there are other sources of information on developing different types of software for Mac:

- **Apple Open Source**. Apple makes major components of OS X—including the UNIX core—available to the developer community. To learn about Apple's commitment to Open Source development, visit Open Source Development Resources. To learn more about some specific Open Source projects, such as Bonjour and WebKit, visit Mac OS Forge.
- **BSD**. Berkeley Software Distribution (BSD) is an essential UNIX-based part of the OS X kernel environment. Several excellent books on BSD and UNIX are available in bookstores. But you can also find additional information on any of the websites that cover BSD variants—for example, The FreeBSD Project.
- **Third-party books**. Several excellent books on Mac app development can be found online and in the technical sections of bookstores.

Creating Software Products for the Mac Platform

Apps are the most common type of Mac software, but there are many other types of software that you can create, too. The following sections introduce the range of software products you can create for the Mac platform and suggest when you might consider doing so.

Apps

Apps are by far the predominant type of software created for Mac, or for any platform. You use Cocoa to build new Mac apps. To learn more about the features and frameworks available in Cocoa, see Cocoa Application Layer (page 26).

In general, there are three basic styles of Mac apps:

- **The single-window utility app.** A single-window utility app helps users perform the primary task within one window. Although a single-window utility app might also open an additional window—such as a preferences window—the user remains focused on the main window. Calculator is an example of a single-window utility app.
- **The single-window "shoebox" app.** The defining characteristic of a shoebox app is the way it gives users an app-specific view of their content. For example, iPhoto users don't find or organize their photos in the Finder; instead, they manage their photo collections entirely within the app.
- **The multiwindow document-based app.** A multiwindow document-based app, such as Pages, opens a new window for each document the user creates or views. This style of app does not need a main window (although it might open a preferences or other auxiliary window).

App Extensions

No matter what type of app you write, you use app extensions to extend the functionality and content of that app to other parts of the system, or even to other apps. Types of extensions include:

- **Today.** Display information from your app, or perform a quick task in the Today view of Notification Center.
- **Share.** Share information with others by posting information to a website or social service, or sending data out in some other way.
- **Action.** Create a context allowing the user to manipulate or view items from your app inside another app.
- **Finder.** Show the sync state information in Finder.

App Store

Regardless of the app style you choose, your goal is probably to get your app into the Mac App Store. The development process that helps you achieve this goal includes a mix of coding and administrative tasks. Some of these tasks are:

- Becoming a Mac developer
- Deciding whether you need to code sign your app (apps that use iCloud or are sandboxed must be code signed)
- Configuring your Xcode project
- Taking advantage of the latest Mac technologies
- Submitting your app to Apple for approval

This document gives you an overview of Mac technologies that you can incorporate into your app. To learn more about the other tasks involved in the development process, read *Developing for the App Store*.

Development Languages

The tools for OS X supports many different development languages. In addition to Objective-C, Swift, C++, C, and other such languages, Xcode provides support for many scripting languages. For more information, see Scripts (page 22) below.

The most common languages used for development for OS X are Objective-C and Swift. The following sections call out key features in some of these environments.

Objective-C

Objective-C is a C-based programming language with object-oriented extensions. It is a primary development language for Cocoa apps. Unlike C++ and some other object-oriented languages, Objective-C comes with its own dynamic runtime environment. This runtime environment makes it much easier to extend the behavior of code at runtime without having access to the original source.

Objective-C 2.0 supports the following features, among many others:

- Blocks (which are described in Block Objects (page 56))
- Declared properties, which offer a simple way to declare and implement an object's accessor methods
- A `for` operator syntax for performing fast enumerations of collections
- Formal and informal protocols, which allow you to declare methods that are independent of any specific class, but which any class might implement

- Categories and extensions, both of which allow you to add methods to an existing class

To learn about the Objective-C language, see *The Objective-C Programming Language*.

Swift

Swift is new programming language for Cocoa and Cocoa Touch with a concise and expressive syntax. Swift incorporates research on programming language combined with decades of experience building Apple platforms. Code written in Swift co-exists with existing classes written in Objective-C, allowing for easy adoption.

Some of the key features of Swift are:

- Closures unified with function pointers
- First class functions
- Tuples and multiple return values
- Generics
- Fast and concise iteration over a range or collection
- Structs and Enums that support methods, extensions, and protocols
- Functional programming patterns
- Type safety and type inference with restricted access to direct pointers

Swift also supports the use of Playgrounds, an interactive environment for real time evaluation of code. Use playgrounds for designing a new algorithm, creating and verifying new tests, or learning about the language and APIs.

To learn more about Swift, see *The Swift Programming Language*, or for a quick overview, see *Welcome to Swift*.

Other Types of Software

There are many other types of software you can develop for Mac. Most of these software products have no user interface (UI) and instead provide services that extend the capabilities of other software, such as third-party apps or the system itself.

Frameworks

A framework is a special type of bundle used to distribute shared resources, including library code, resource files, header files, and reference documentation. Frameworks offer a more flexible way to distribute shared code—for example, image files and localized strings—that you might otherwise put into a dynamic shared library. Frameworks also have a version control mechanism that makes it possible to distribute multiple versions of a framework in the same framework bundle.

Apple uses frameworks to distribute the public interfaces of OS X (and iOS), which are packaged in software development kits. A software development kit (SDK) collects the frameworks, header files, tools, and other resources necessary for developing software targeted at a specific version of a platform. You, too, can use frameworks to distribute public code and interfaces that you create, or to develop private shared libraries to embed in your apps.

> **Note:** Although OS X also supports the concept of an "umbrella" framework, which encapsulates multiple subframeworks in a single package, this mechanism is used primarily for the distribution of Apple software. The creation of umbrella frameworks by third-party developers is not recommended.

You can use any programming language to create your own frameworks, but it's best to choose a language that makes it easy to update the framework later. Apple frameworks generally export programmatic interfaces in ANSI C, Objective-C, or Swift. Both of these languages have a well-defined export structure that makes it easy to maintain compatibility between different revisions of the framework.

To learn about the structure and composition of frameworks, see *Framework Programming Guide* . That document also describes how to use Xcode to create public and private frameworks.

Plug-ins

Plug-ins are the standard way to extend many apps and system behaviors. A plug-in is a bundle whose code is loaded dynamically into the runtime of an app. Because it's loaded dynamically, a plug-in can be added and removed by the user.

The app and system plug-ins listed below represent some of the many opportunities for developing plug-ins.

Address Book action plug-ins. An Address Book plug-in lets you add custom actions that act on the data in a person's Address Book card. For example, the existing Large Type action displays the selected phone number in large type. Each action plug-in performs a single action, which can open a simple window within the Address Book app. If an action needs to do anything else, it must launch your app to perform the action. To learn how to create an Address Book action plug-in, see Creating and Using Address Book Action Plug-ins.

App plug-ins. An app plug-in can extend the features of any app that supports a plug-in model. In addition to third-party apps, several Apple apps also support plug-ins, such as iTunes, Final Cut Pro, and Aperture. For information about developing plug-ins for Apple apps, visit the Apple Developer website.

Automator plug-ins. Using an Automator plug-in, you can expand the default set of actions available in Automator, a utility app that lets users assemble complex scripts using a palette of predefined actions. Automator plug-ins can be written in AppleScript or Objective-C, so you can write them for your own app's features or for the features of other scriptable apps. (It's a good idea to provide Automator plug-ins for your app's most common tasks because doing so gives users more ways to interact with your app.) To learn how to write an Automator plug-in, see *Automator Programming Guide*.

Core Audio plug-ins. A Core Audio plug-in can support the manipulation of audio streams during most processing stages. For example, you can use plug-ins to generate, process, or receive an audio stream or to interact with new types of audio-related hardware devices. To begin learning about Core Audio, read *Core Audio Overview*.

Image units. An image unit is a type of plug-in that you can use with the Core Image and Core Video technologies. An image unit consists of a collection of filters—each of which implements a specific manipulation for image data—packaged together in a single bundle. For example, you could write a set of filters that perform different kinds of edge detection and package them as one image unit. To learn how to create an image unit, see Creating Custom Filters.

Input methods. A common example of an input method is an interface for typing Japanese or Chinese characters using multiple keystrokes. Other examples of input methods include spelling checkers and pen-based gesture recognition systems. You can create input methods using Input Method Kit (`InputMethodKit.framework`). For information on how to use this framework, see *InputMethodKit Framework Reference*.

Metadata importers. Spotlight relies on metadata importers to gather information about the user's files and to build a systemwide index. Spotlight uses this index to help users find information by searching on attributes that make sense to them, such as the duration of a video or the dimensions of an image. If your app defines a custom file format, you should always provide a metadata importer for that file format. (If your app relies on commonly used file formats, such as JPEG, RTF, or PDF, the system provides a metadata importer for you.) To learn how to create metadata importers, see *Spotlight Importer Programming Guide*.

Quartz Composer plug-ins. Quartz Composer supports a plug-in mechanism that allows you to create a custom patch and make it available in the Quartz Composer workspace and to most Quartz Composer clients. (A *patch* is processing unit that performs a specific task, such as processing a string or rendering an OpenGL texture.) To learn how to create a Quartz Composer plug-in, see *Quartz Composer Custom Patch Programming Guide*.

Quick Look plug-ins. A Quick Look plug-in—also known as a Quick Look generator—converts a document from its native format into a format that Quick Look can display to users. If your app creates documents of a nonstandard or private type, it's a good idea to provide a Quick Look generator so that users can get previews of these documents in Quick Look. To learn how to create a Quick Look plug-in, see *Quick Look Programming Guide*.

Safari plug-ins. Safari supports the Netscape-style plug-in model for incorporating additional types of content in the web browser. In Safari in OS X v10.7 and later, these plug-ins run in their own process, which improves the stability and security of Safari. Netscape-style plug-ins include support for onscreen drawing, event handling, and networking and scripting functions.

> **Note:** Beginning in OS X v10.7, Safari does not support WebKit plug-ins because they are not compatible with the new process architecture. Going forward, you must convert WebKit plug-ins to Netscape-style plug-ins or Safari Extensions.

For information about creating Safari plug-ins with the Netscape API, see *WebKit Plug-In Programming Topics* and *WebKit Framework Reference*.

Safari Extensions

Use Safari extensions to add features both to the Safari web browser and to the content that Safari displays. For example, you can add custom buttons to the browser's toolbar, reformat webpages, block unwanted sites, and create contextual menu items. Extensions let you inject scripts and style sheets into pages of web content.

A Safari extension is a collection of HTML, JavaScript, and CSS files with support for both HTML5 and CSS3. Safari extensions are supported in both OS X and Windows systems running Safari 5.0 and later.

To learn more about Safari extensions, read *Safari Extensions Development Guide* in the Safari Developer Library.

Dashboard Widgets

A Dashboard widget is a lightweight web app that helps users perform a common task, such as checking a stock price or getting a weather report. Widgets reside in the Dashboard environment, which can appear over the user's current desktop or as a separate space. OS X includes several built-in widgets, and users can download third-party widgets from Apple - Downloads - Dashboard.

In effect, widgets are HTML-based apps with optional JavaScript code to provide dynamic behavior. Dashboard uses WebKit to provide the environment for displaying the HTML and running the JavaScript code. Your widgets can take advantage of several extensions provided by that environment, including a way to render content using Quartz-like JavaScript functions.

The Dashcode app provides a streamlined environment for developing widgets and includes several templates that help you get started. To learn how to use Dashcode, see *Dashcode User Guide*. To learn more about the technologies you can use in a widget, see *Dashboard Programming Topics*.

Agent Applications

An agent is a special type of application that typically runs in the background, providing information as needed to the user or to another app. For example, the Dock is an agent application that is run by OS X.

An agent can be launched by the user but is more likely to be launched by the system or another app. As a result, agents do not show up in the Dock or the window displayed by the Force Quit menu command. Although agents might occasionally come to the foreground and display a user interface, they do not have a menu bar for choosing commands. All user interaction with an agent application is brief and focused on a specific goal, such as setting preferences or requesting information.

To create an agent application, you create a bundled app and include the `LSUIElement` key in its information property list (`Info.plist`) file. For more information on using this key, see *Information Property List Key Reference*.

Screen Savers

Screen savers are small programs that take over the screen after a certain period of idleness. Screen savers provide entertainment and also prevent the screen image from being burned into the surface of a display. OS X supports both slideshows and programmatically generated screen-saver content.

A slideshow is a simple type of screen saver that does not require any code to implement. To create a slideshow, you create a bundle with an extension of `.slideSaver`. Inside this bundle, you place a Resources directory that contains the images you want to display in your slideshow. Your bundle should also include an information property list that specifies basic information about the bundle, such as its name, identifier string, and version.

OS X includes several slideshow screen savers you can use as templates for creating your own. These screen savers are located in `/System/Library/Screen Savers`. You should put your own slideshows in either `/Library/Screen Savers` or in the `~/Library/Screen Savers` directory of a user.

A programmatic screen saver is a screen saver that continuously generates content to appear on the screen. You can use this type of screen saver to create animations or to create a screen saver with user-configurable options. The bundle for a programmatic screen saver ends with the `.saver` extension.

You create programmatic screen savers using Cocoa with the Objective-C language or with Swift. Specifically, you create a custom subclass of `ScreenSaverView` that provides the interface for displaying the screen saver content and options. The information property list of your bundle provides the system with the name of your custom subclass. For information on creating programmatic screen savers, see *Screen Saver Framework Reference*.

Services

Services are not separate programs that you write; instead, they are features exported by your app for the benefit of other apps. Services let you share the resources and capabilities of your app with other apps in the system. Users access services through the Services menu that is available in every app's application menu. (Services replace the contextual menu plug-in functionality that was available in earlier versions of OS X.)

A service typically acts on the currently selected data. When the user initiates a service, the app that holds the selected data places it on the pasteboard. The app whose service was selected then takes the data, processes it, and puts the results (if any) back on the pasteboard for the original app to retrieve. For example, a user might select a folder in the Finder and choose a service that compresses the folder contents and replaces them with the compressed version. Services can represent one-way actions as well. For example, a service could take the currently selected text in a window and use it to create the content of a new email message. For information on how to provide and use services in your app, see *Services Implementation Guide*.

Preference Panes

Preference panes are used primarily to modify system preferences for the current user. Preference panes are implemented as plug-ins and installed in `/Library/PreferencePanes`. App developers can also take advantage of these plug-ins to manage per-user app preferences; however, most apps provide their own UI to manage preferences.

You might need to create preference panes if you create:

- Hardware devices that are user configurable
- Systemwide utilities, such as virus protection programs, that require user configuration

If you're an app developer, you might want to reuse preference panes intended for the System Preferences app or use the same model to implement your app preferences. To learn how to create and manage preference panes, read *Preference Pane Programming Guide*.

Dynamic Websites and Web Services

OS X supports a variety of techniques and technologies for creating web content. In addition to Identity Services (page 60) and Dashboard Widgets (page 18), dynamic websites and web services offer web developers ways to deliver their content quickly and easily.

OS X provides support for creating and testing dynamic content in web pages. If you are developing CGI-based web apps, you can create websites using a variety of scripting technologies, including Perl and the PHP Hypertext Preprocessor (a complete list of scripting technologies is provided in Scripts (page 22)). You can also create and deploy more complex web apps using JBoss, Tomcat, and WebObjects. To deploy your webpages, use the built-in Apache HTTP web server.

Safari provides standards-compliant support for viewing pages that incorporate numerous technologies, including HTML, XML, XHTML, DOM, CSS, Java, and JavaScript. You can also use Safari to test pages that contain multimedia content created for QuickTime, Flash, and Shockwave.

The Simple Object Access Protocol (SOAP) is an object-oriented protocol that defines a way for programs to communicate over a network. XML-RPC is a protocol for performing remote procedure calls between programs. In OS X, you can create clients that use these protocols to gather information from web services across the Internet. To create these clients, you use technologies such as PHP, JavaScript, AppleScript, and Cocoa.

Representational State Transfer (REST) is an alternative method for transferring data using URLs. OS X provides full support for building REST applications through NSURL, NSURLSession, and related classes. For more information, see *URL Loading System Programming Guide*.

If you want to provide your own web services in OS X, use WebObjects or implement the service using the scripting language of your choice. You then post your script code to a web server, give clients a URL, and publish the message format your script supports.

For information on how to create client programs using AppleScript, see *XML-RPC and SOAP Programming Guide*. For information on how to create web services, see *WebObjects Web Services Programming Guide*.

Mail Stationery

The Mail app provides templates that give users prebuilt email messages that are easily customized. Because templates are HTML based, they can incorporate images and advanced formatting to give the user's email a much more stylish and sophisticated appearance.

Developers and web designers can create custom template packages for external or internal users. Each template consists of an HTML page, a property list file, and a set of images which are packaged together in a bundle and then stored in the Mail app's stationery folder. The HTML page and images define the content of the email message and can include drop zones for custom user content. The property list file gives Mail information about the template, such as its name, ID, and the name of its thumbnail image. To learn how to create new stationery templates, see *Mail Programming Topics*.

Command-Line Tools

Command-line tools are simple programs that manipulate data through a text-based interface. These tools do not use windows, menus, or other user interface elements traditionally associated with apps. Instead, they run from the command-line environment of the Terminal app. Because command-line tools require less explicit knowledge of the system to develop, they are often simpler to write than many other types of software. However, command-line tools are best suited to technically savvy users who are familiar with the conventions and syntax of the command-line interface.

Xcode supports the creation of command-line tools from several initial code bases. For example, you can create a simple and portable tool using standard C or C++ library calls, or you can create a tool more specific to OS X using frameworks such as Core Foundation, Core Services, or Cocoa Foundation.

Launch Items and Daemons

Launch items are special programs that launch other programs or perform one-time operations during startup and login periods. Daemons are programs that run continuously and act as servers for processing client requests. You typically use launch items to launch daemons or perform periodic maintenance tasks, such as checking the hard drive for corrupted information.

Launch items should not be confused with the login items found in the Accounts system preferences. Login items are typically agent applications that run within a given user's session and can be configured by that user. Launch items are not user-configurable.

Few developers should ever need to create launch items or daemons. These programs are reserved for special situations in which you need to guarantee the availability of a particular service. For example, OS X provides a launch item to run the DNS daemon. Similarly, a virus-detection program might install a launch item to launch a daemon that monitors the system for virus-like activity. In both cases, the launch item would run its daemon in the root session, which provides services to all users of the system. To learn more about launch items and daemons, see *Daemons and Services Programming Guide*.

Scripts

A script is a set of text commands that are interpreted at runtime and turned into a sequence of actions. Most scripting languages provide high-level features that make it easy to implement complex workflows quickly. Scripting languages are often very flexible, letting you call other programs and manipulate the data they return. Some scripting languages are also portable across platforms, so that you can use your scripts anywhere.

Table 1-1 lists many of the scripting languages available in OS X.

Table 1-1 Scripting languages available in OS X

Language	Description
AppleScript	An English-based language for controlling scriptable apps in OS X. Use it to tie together apps involved in a custom workflow or repetitive job. For more information, see *AppleScript Overview*.
`bash`	A Bourne-compatible shell script language used to build programs on UNIX-based systems.
`csh`	The C shell script language used to build programs on UNIX-based systems.
Perl	A general-purpose scripting language supported on many platforms. Perl provides an extensive set of features suited for text parsing and pattern matching and also has some object-oriented features. For more information, see The Perl Programming Language website.
PHP	A cross-platform, general-purpose scripting language that is especially suited for web development. For more information, see PHP: Hypertext Preprocessor.
Python	A general-purpose, object-oriented scripting language implemented for many platforms. For more information, see Python Programming Language. To learn about using Python with the Cocoa scripting bridge, see *Ruby and Python Programming Topics for Mac*.
Ruby	A general-purpose, object-oriented scripting language implemented for many platforms. For more information, see Ruby Programming Language. To learn about using Ruby with the Cocoa scripting bridge, see *Ruby and Python Programming Topics for Mac*.
`sh`	The Bourne shell script language used to build programs on UNIX-based systems.
Tcl	A general-purpose language implemented for many platforms. Tcl (Tool Command Language) is often used to create graphical interfaces for scripts. For more information, see Tcl Developer Site.
`tcsh`	A variant of the C shell script language used to build programs on UNIX-based systems.
`zsh`	The Z shell script language used to build programs on UNIX-based systems.

Scripting Additions for AppleScript

A scripting addition delivers additional functionality for AppleScript scripts by adding systemwide support for new commands or data types. Developers who need features not available in the current command set can use scripting additions to implement those features and make them available to all apps. For example, one of

the built-in scripting additions extends the basic file-handling commands to support the reading and writing of file contents from an AppleScript script. For information on how to create a scripting addition, see Technical Note TN1164, "Scripting Additions for OS X."

Kernel Extensions

Kernel extensions are code modules that load directly into the kernel process space and therefore bypass the protections offered by the OS X core environment. Most developers have little need to create kernel extensions. The situations in which you might need a kernel extension are the following:

- Your code needs to handle a primary hardware interrupt.
- The client of your code is inside the kernel.
- A large number of apps require a resource your code provides. For example, you might implement a file-system stack using a kernel extension.
- Your code has special requirements or needs to access kernel interfaces that are not available in the user space.

Although a device driver is a type of kernel extension, by convention the term *kernel extension* refers to a code module that implements a new network stack or file system. You would not use a kernel extension to communicate with an external device such as a digital camera or a printer. (For information on communicating with external devices, see Device Drivers (page 24).)

> **Note:** Kernel data structures have an access model that makes it possible to write nonfragile kernel extensions—that is, kernel extensions that do not break when the kernel data structures change. Developers are highly encouraged to use the kernel-extension API for accessing kernel data structures.

For information about writing kernel extensions, see *Kernel Programming Guide*.

Device Drivers

Device drivers are a special type of kernel extension that enable OS X to communicate with many hardware devices, including mice, keyboards, and FireWire drives. Device drivers communicate hardware status to the system and facilitate the transfer of device-specific data to and from the hardware. OS X provides default drivers for many types of devices, but these might not meet the needs of all hardware developers.

Although developers of mice and keyboards might be able to use the standard drivers, many other developers require custom drivers. Developers of hardware such as scanners, printers, AGP cards, and PCI cards typically have to create custom device drivers because these devices require more sophisticated data handling than is

usually needed for mice and keyboards. Hardware developers also tend to differentiate their hardware by adding custom features and behavior, which makes it difficult for Apple to provide generic drivers to handle all devices.

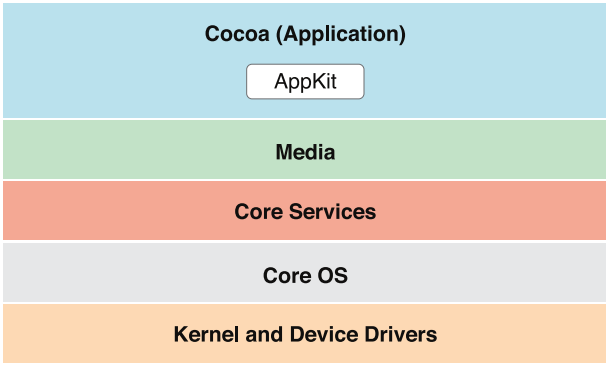
Apple provides code you can use as the basis for your custom drivers. The I/O Kit provides an object-oriented framework for developing device drivers using C++. To learn more about the I/O Kit, see *IOKit Fundamentals* .

Cocoa Application Layer

The Cocoa application layer is primarily responsible for the appearance of apps and their responsiveness to user actions. In addition, many of the features that define the OS X user experience—such as Notification Center, full-screen mode, and Auto Save—are implemented by the Cocoa layer.

> **Note:** In this book, *Cocoa* usually refers to the application layer of OS X. In other Apple technical documents, *Cocoa* frequently refers to all programmatic interfaces that you might use to develop an app, regardless of the layer in which those interfaces reside.

The term *Aqua* refers to the overall appearance and behavior of OS X. The Aqua look and feel is characterized by consistent, user-friendly behaviors combined with a masterful use of layout, color, and texture. Although much of the Aqua look and feel comes for free when you use Cocoa technologies to develop your app, there are still many steps you should take to distinguish your app from the competition. To create a beautiful, compelling app that users will love, be sure to follow the guidance provided in *OS X Human Interface Guidelines*.



High-Level Features

The Cocoa (Application) layer implements many features that are distinctive aspects of the OS X user experience. Users expect to find these features throughout the system, so it's a good idea to support all the features that make sense in your app.

Notification Center

Notification Center provides a way for users to receive and view app notifications in an attractive, unobtrusive way. For each app, users can specify how they want to be notified of an item's arrival; they can also reveal Notification Center to view all the items that have been delivered.

The Notification Center APIs, which help you configure the user-visible portions of a notification item, schedule items for delivery, and find out when items have been delivered. You can also determine whether your app has launched as a result of a notification, and if it has, whether that notification is a local or remote (that is, push) notification.

To learn about integrating the Notification Center into your app, see *NSUserNotificationCenter Class Reference* and *NSUserNotification Class Reference*. To ensure that your app gives users the best Notification Center experience, read Notification Center. In addition, you can add items to the Today view using a Today extension, see Today in the *App Extension Programming Guide*.

Game Center

Game Center accesses the same social-gaming network as on iOS, allowing users to track scores on a leaderboard, compare their in-game achievements, invite friends to play a game, and start a multiplayer game through automatic matching. Game Center functionality is provided in three parts:

- The Game Center app, in which users sign in to their account, discover new games and new friends, add friends to their gaming network, and browse leaderboards and achievements.
- The Game Kit framework, which contains the APIs developers use to support live multiplayer or turn-based games and adopt other Game Center features, such as in-game voice chat and leaderboard access.
- The online Game Center service supported by Apple, which performs player authentication, provides leaderboard and achievement information, and handles invitations and automatching for multiplayer games. You interact with the Game Center service only indirectly, using the Game Kit APIs.

> **Note:** To support Game Center in a game developed for OS X, you must sign the app with a provisioning profile that enables Game Center. To learn more about provisioning profiles, see Provisioning Your System.

In your game, use the Game Kit APIs to post scores and achievements to the Game Center service and to display leaderboards in your user interface. You can also use Game Kit APIs to help users find others to play with in a multiplayer game.

To learn more about adding Game Center support to your app, see *Game Center Programming Guide*.

Sharing

The sharing service provides a consistent user experience for sharing content among many types of services. For example, a user might want to share a photo by posting it in a Twitter message, attaching it to an email, or sending it to another Mac user via AirDrop.

Use the AppKit `NSSharingService` class to get information about available services and share items with them directly. As a result, you can display a custom UI to present the services. You can also use the `NSSharingServicePicker` class to display a list of sharing services (including custom services that you define) from which the user can choose. When a service is performed, the system-provided sharing window is displayed, where the user can comment or add recipients.

Resume

Resume is a systemwide enhancement of the user experience that supports app persistence. A user can log out or shut down the operating system, and on next login or startup, OS X automatically relaunches the apps that were last running and restores the windows that were last opened. If your app provides the necessary support, reopened windows have the same size and location as before; in addition, window contents are scrolled to the previous position and selections are restored.

To support app persistence, you must also implement automatic and sudden app termination, user interface preservation, and Auto Save. See, Automatic and Sudden Termination of Apps, User Interface Preservation, and Documents Are Automatically Saved in the *Mac App Programming Guide*.

Full-Screen Mode

When an app enters full-screen mode it opens its frontmost app or document window in a separate space. Enabling full-screen mode adds an Enter Full Screen menu item to the View menu or, if there is no View menu, to the Window menu. When a user chooses this menu item, the frontmost app or document window fills the entire screen.

The AppKit framework provides support for customizing the appearance and behavior of full-screen windows. For example, you can set a window-style mask and can implement custom animations when an app enters and exits full-screen mode.

You enable and manage full-screen support through methods of the `NSApplication` and `NSWindow` classes and the `NSWindowDelegate Protocol` protocol. To find out more about this feature, read Implementing the Full-Screen Experience in *Mac App Programming Guide*.

Cocoa Auto Layout

Cocoa Auto Layout is a rule-based system designed to implement the layout guidelines described in *OS X Human Interface Guidelines*. It expresses a larger class of relationships and is more intuitive to use than springs and struts.

Using Auto Layout brings you a number of benefits:

- Localization through swapping of strings alone, instead of also revamping layouts
- Mirroring of UI elements for right-to-left languages such as Hebrew and Arabic
- Better layering of responsibility between objects in the view and controller layers

  A view object usually knows best about its standard size and its positioning within its superview and relative to its sibling views. A controller can override these values if something nonstandard is required.

The entities you use to define a layout are Objective-C objects called *constraints*. You define constraints by combining attributes—such as leading, trailing, left, right, top, bottom, width, and height—that encapsulate the relationships between UI elements. (Leading and trailing are similar to left and right, but they are more expressive because they automatically mirror the constraint in a right-to-left environment.) In addition, you can assign priority levels to constraints, to identify the constraints that are most important to satisfy.

You can use Interface Builder to add and edit constraints for your interface. When you need more control, you can work with constraints programatically.

For more information on Auto Layout, see *Auto Layout Guide*.

Popovers

A popover is a view that displays additional content related to existing content onscreen. AppKit provides the `NSPopover` class to support popovers. AppKit automatically positions a popover relative to the view containing the existing content—known as the *positioning view*—and it moves the popover when the popover's positioning view moves.

You configure the appearance and behavior of a popover, including which user interactions cause the popover to close. And by implementing the appropriate delegate method, you can configure a popover to detach itself and become a separate window when a user drags it.

For more information, see *NSPopover Class Reference* and *NSPopoverDelegate Protocol Reference*. For guidelines on using popovers, see Popovers in *OS X Human Interface Guidelines*.

Software Configuration

OS X programs commonly use property list files (also known as *plist files*) to store configuration data. A property list is a text or binary file used to manage a dictionary of key-value pairs. Apps use a special type of property list file, called an *information property list* (`Info.plist`) *file*, to communicate key attributes of the app—such as the app's name, unique identification string, and version information—to the system. Apps also use property list files to store user preferences or other custom configuration data.

The advantage of property list files is that they are easy to edit and modify from outside the runtime environment of your app. Xcode includes a built-in property list editor for editing your app's `Info.plist` file. To learn more about information property list files and the keys you put in them, see *Runtime Configuration Guidelines* and *Information Property List Key Reference*. To learn how to edit a property list file in Xcode, see Edit Keys and Values.

Inside your app, you can read and write property list files programmatically using facilities found in both Core Foundation and Cocoa. For more information on creating and using property lists programmatically, see *Property List Programming Guide* or *Property List Programming Topics for Core Foundation*.

Accessibility

Accessibility is the successful access to information and information technologies by the millions of people who have some type of disability or special need. OS X provides many built-in features and assistive technologies that help users with special needs benefit from the Mac. OS X also provides software developers with the functions they need to create apps that are accessible to all users.

Apps that use Cocoa interfaces receive significant support for accessibility automatically. For example, apps get the following support for free:

- Zoom features let users increase the size of onscreen elements.
- Sticky keys let users press keys sequentially instead of simultaneously for keyboard shortcuts.
- Mouse keys let users control the mouse with the numeric keypad.
- Full keyboard access mode lets users complete any action using the keyboard instead of the mouse.
- Speech recognition lets users speak commands rather than type them.
- Text-to-speech reads text to users with visual disabilities.
- VoiceOver provides spoken user interface features to assist visually impaired users.

Although Cocoa integrates accessibility support into its APIs, there might still be times when you need to provide more descriptive information about your windows and controls. The Accessibility section of the Xcode Identity inspector makes it easy to provide custom accessibility information about the UI elements in your app. Or you can use the appropriate accessibility interfaces to change the settings programmatically.

For more information about accessibility, see *Accessibility Programming Guide for OS X*.

AppleScript

OS X employs AppleScript as the primary language for making apps scriptable. With AppleScript, users can write scripts that link together the services of multiple scriptable apps.

When designing new apps, you should consider AppleScript support early in the process. The key to a good design that supports AppleScript is choosing an appropriate data model for your app. The design must not only serve the purposes of your app but also make it easy for AppleScript implementers to manipulate your content. After you settle on a data model, you can implement the Apple event code needed to support scripting.

To learn how to support AppleScript in your programs, see Applescript Overview.

Spotlight

Spotlight provides advanced search capabilities for apps. The Spotlight server gathers metadata from documents and other relevant user files and incorporates that metadata into a searchable index. The Finder uses this metadata to provide users with more relevant information about their files. For example, in addition to listing the name of a JPEG file, the Finder can also list its width and height in pixels.

App developers use Spotlight in two different ways. First, you can search for file attributes and content using the Spotlight search API. Second, if your app defines its own custom file formats, you should incorporate any appropriate metadata information in those formats and provide a Spotlight importer plug-in to return that metadata to Spotlight.

> **Note:** You should not use Spotlight for indexing and searching the general content of a file. Spotlight is intended for searching only the metainformation associated with files. To search the actual contents of a file, use Search Kit. For more information on Search Kit, see Other Frameworks in the Core Services Layer (page 67).

For more information on using Spotlight in your apps, see *Spotlight Overview*.

Ink Services

Ink Services provides handwriting recognition for apps that support the Cocoa and WebKit text systems and any text system that supports input methods. The automatic support is for text and handwriting gestures (which are defined in the Ink panel). The Ink framework offers several features that you can incorporate into your apps, including the following:

- Enabling or disabling handwriting recognition programmatically

- Accessing Ink data directly
- Supporting either deferred recognition or recognition on demand
- Supporting the direct manipulation of text by means of gestures

The Ink Services feature is implemented by the Ink framework (`Ink.framework`). The Ink framework is not intended solely for developers of end-user apps. Hardware developers can also use it to implement a handwriting recognition solution for a new input device. You might also use the Ink framework to implement your own correction model to provide users with a list of alternate interpretations for handwriting data.

The Ink framework is a subframework of `Carbon.framework`; you should link to it directly with the umbrella framework, not with `Ink.framework`. For more information on using Ink Services in Cocoa apps, see *Using Ink Services in Your Application*.

Frameworks

The Cocoa (Application) layer includes the frameworks described in the following sections.

Cocoa Umbrella Framework

The Cocoa umbrella framework (`Cocoa.framework`) imports the core Objective-C frameworks for app development: AppKit, Foundation, and Core Data.

- **AppKit** (`AppKit.framework`). This is the only framework of the three that is actually in the Cocoa layer. See AppKit (page 33) for a summary of AppKit features and classes.
- **Foundation** (`Foundation.framework`). The classes of the Foundation framework (which resides in the Core Services layer) implement data management, file access, process notification, network communication, and other low-level features. AppKit has a direct dependency on Foundation because many of its methods and functions either take instances of Foundation classes as parameters, or return the instances as values.

  To find out more about Foundation, see Foundation and Core Foundation (page 64).
- **Core Data** (`CoreData.framework`). The classes of the Core Data framework (which also resides in the Core Services layer) manage the data model of an app based on the Model-View-Controller design pattern. Although Core Data is optional for app development, it is recommended for apps that deal with large data sets.

  For more information about Core Data, see Core Data (page 63).

AppKit

AppKit is the key framework for Cocoa apps. The classes in the AppKit framework implement the user interface (UI) of an app, including windows, dialogs, controls, menus, and event handling. They also handle much of the behavior required of a well-behaved app, including menu management, window management, document management, Open and Save dialogs, and pasteboard (Clipboard) behavior.

In addition to having classes for windows, menus, event handling, and a wide variety of views and controls, AppKit has window- and data-controller classes and classes for fonts, colors, images, and graphics operations. A large subset of classes comprise the Cocoa text system, described in Text, Typography, and Fonts (page 37). Other AppKit classes support document management, printing, and services such as spellchecking, help, speech, and pasteboard and drag-and-drop operations.

Apps can participate in many of the features that make the user experience of OS X such an intuitive, productive, and rewarding experience. These features include the following:

- **Gestures**. Users appreciate being able to use fluid, intuitive Multi-Touch gestures to interact with OS X. AppKit classes make it easy to adopt these gestures in your app and to provide a better zoom experience without redrawing your content. For example, `NSScrollView` includes built-in support for the smart zoom gesture (that is, a two-finger double-tap on a trackpad). When you provide the semantic layout of your content, `NSScrollView` can intelligently magnify the content under the pointer. You can also use this class to respond to the lookup gesture (that is, a three-finger tap on a trackpad). To learn more about the gesture support that `NSScrollView` provides, see *NSScrollView Class Reference*.
- **Spaces**. Spaces lets the user organize windows into groups and switch back and forth between groups to avoid cluttering up the desktop. AppKit provides support for sharing windows across spaces through the use of collection behavior attributes on the window. For information about setting these attributes, see *NSWindow Class Reference*.
- **Fast User Switching**. With this feature, multiple users can share access to a single computer without logging out. One user's session can continue to run, while another user logs in and accesses the computer. To support fast user switching, be sure that your app avoids doing anything that might affect another version of the app running in a different session. To learn how to implement this behavior, see *Multiple User Environment Programming Topics*.

Xcode includes Interface Builder, a user interface editor that contains a library of AppKit objects, such as controls, views, and controller objects. With it, you can create most of your UI (including much of its behavior) graphically rather than programatically. With the addition of Cocoa bindings and Core Data, you can also implement most of the rest of your app graphically.

For an overview of the AppKit framework, see the introduction to the *AppKit Framework Reference*. *Mac App Programming Guide* offers a practical discussion of how you use mostly AppKit classes to implement an app's user interface, its documents, and its overall behavior.

Game Kit

The Game Kit framework (`GameKit.framework`) provides APIs that allow your app to participate in Game Center. For example, you can use Game Kit classes to display leaderboards in your game and to give users the opportunity to share their in-game achievements and play multiplayer games.

To learn more about using Game Kit in your app, see *GameKit Framework Reference*.

Preference Panes

The Preference Panes framework (`PreferencePanes.framework`) lets you create plug-ins containing a user interface for setting app preferences. At runtime, the System Preferences app (or your app) can dynamically load the plug-in and present the settings UI to users. In System Preferences, each icon in the Show All view represents an individual preference pane plug-in. You typically implement preference pane plug-ins when your app lacks its own user interface or has a very limited UI but needs to be configurable. In these cases, you create both the plug-in and the app-specific code that reads and writes the preference settings.

For more information about creating preference-pane plug-ins, see *Preference Pane Programming Guide*.

Screen Saver

The Screen Saver framework (`ScreenSaver.framework`) contains classes for creating dynamically loadable bundles that implement screen savers. Users can select your screen saver in the Desktop & Screen Saver pane of the System Preferences app. Screen Saver helps you implement the screen saver view and preview and manage screen saver preferences.

To learn more about creating screen savers, see *Screen Saver Framework Reference*. Also read the technical note *Building Screen Savers for Snow Leopard* for additional information.

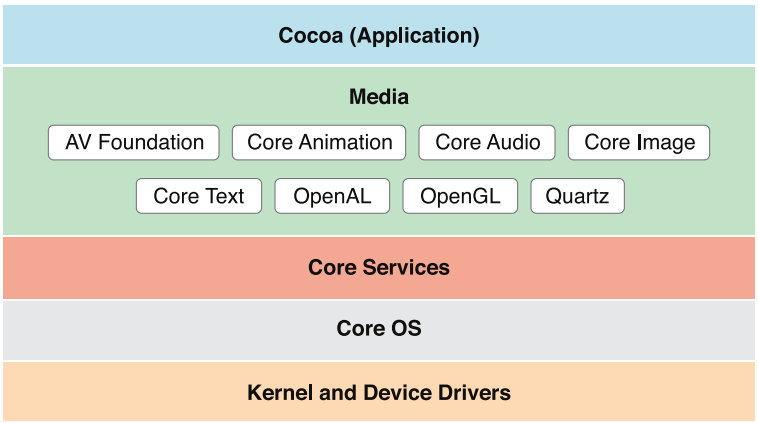
Security Interface

The Security Interface framework (`SecurityInterface.framework`) contains classes that provide UI elements for programs implementing security features such as authorization, access to digital certificates, and access to items in keychains. There are classes for creating custom views and standard security controls, for creating panels and sheets for presenting and editing certificates, for editing keychain settings, and for presenting and allowing selection of identities.

For more information about the Security Interface framework, see *Security Interface Framework Reference*.

Media Layer

Beautiful graphics and high-fidelity multimedia are hallmarks of the OS X user experience. Take advantage of the technologies of the Media layer to incorporate 2D and 3D graphics, animations, image effects, and professional-grade audio and video functionality into your app.



Supported Media Formats

OS X supports more than 100 media types, covering a range of audio, video, image, and streaming formats. Table 3-1 lists some of the more common supported file formats.

Table 3-1 Partial list of formats supported in OS X

Image formats	PICT, BMP, GIF, JPEG, TIFF, PNG, DIB, ICO, EPS, PDF
Audio file and data formats	AAC, AIFF, WAVE, uLaw, AC3, MPEG-3, MPEG-4 (`.mp4`, `.m4a`), `.snd`, `.au`, `.caf`, Adaptive multi-rate (`.amr`)
Video file formats	AVI, AVR, DV, M-JPEG, MPEG-1, MPEG-2, MPEG-4, AAC, OpenDML, 3GPP, 3GPP2, AMC, H.264, iTunes (`.m4v`), QuickTime (`.mov`, `.qt`)
Web streaming protocols	HTTP, RTP, RTSP

Graphics Technologies

A distinctive quality of any OS X app is high-quality graphics in its user interface. And on a Retina display, users are more aware than ever of your app's graphics.

The simplest, most efficient, and most common way to ensure high-quality graphics in your app is to use the standard views and controls of the AppKit framework, along with prerendered images in different resolutions. In this way, you let the system do the work of rendering the app's UI appropriately for the current display. Occasionally, you might need to go beyond off-the-shelf views and simple graphics. In these situations, you can take advantage of the powerful OS X graphics technologies. The following sections describe some of these technologies; for summaries of all technologies see Media Layer Frameworks (page 41).

Graphics and Drawing

OS X offers several system technologies for graphics and drawing. Many of these technologies provide support for making your rendered content look good at different screen resolutions. To learn how to make sure that your app looks good on a high-resolution display, see *High Resolution Guidelines for OS X*.

Cocoa Drawing

The AppKit framework provides object-oriented wrappers for many of the features found in Quartz 2D. Cocoa provides support for drawing primitive shapes such as lines, rectangles, ovals, arcs, and Bezier paths. It supports drawing in both standard and custom color spaces and it supports content manipulations using graphics transforms. Drawing calls made from Cocoa are composited along with all other Quartz 2D content. You can even mix Quartz 2D drawing calls (and drawing calls from other system graphics technologies) with Cocoa calls in your code.

The AppKit framework is described in AppKit (page 33). For more information on how to draw using Cocoa features, see *Cocoa Drawing Guide*.

Other Frameworks for Graphics and Drawing

In addition to AppKit (specifically, its Cocoa drawing interface), there are several other important frameworks for graphics and drawing. By design, Cocoa drawing integrates well with the other graphics and drawing technologies listed next.

- **Core Graphics** (`CoreGraphics.framework`). Core Graphics (also known as *Quartz 2D*) offers native 2D vector- and image-based rendering capabilities that are resolution- and device-independent. These capabilities include path-based drawing, painting with transparency, shading, drawing of shadows, transparency layers, color management, antialiased rendering, and PDF document generation. The Core Graphics framework is in the Application Services umbrella framework.

Quartz is at the heart of the OS X graphics and windowing environment. It provides rendering support for 2D content and combines a rich imaging model with on-the-fly rendering, compositing, and antialiasing of content. It also implements the windowing system for OS X and provides low-level services such as event routing and cursor management (for more information, see Core Graphics (page 42)).

- **Core Animation.** Core Animation enables your app to create fluid animations using advanced compositing effects. It defines a hierarchical view-like abstraction that mirrors a hierarchy of views and is used to perform complex animations of user interfaces. Core Animation is implemented by the Quartz Core framework (`QuartzCore.framework`) (for more information, see Core Animation (page 48)).
- **SpriteKit** (`SpriteKit.framework`). SpriteKit provides the tools and methods for creating and rendering and animating textured images, or sprites. You use graphical editors for creating sprites, and then use those sprites in scenes that simulate game physics. In addition to sprites, you can add lights, emitters, and different kinds of fields to scenes. SpriteKit animates your scene and calls back to your code for events such as collisions. To learn more, see Sprite Kit (page 50).
- **Scene Kit** (`SceneKit.framework`). Scene Kit provides a high-level, Objective-C graphics API that you can use to efficiently load, manipulate, and render 3D scenes. Powerful and easy-to-use Scene Kit integrates well with Core Animation and SpriteKit, allowing you to use built-in materials or custom GLSL shaders to render your 3D scenes (for more information, see Scene Kit (page 50)).
- **OpenGL** (`OpenGL.framework`). OpenGL is an open, industry-standard technology for creating and animating real-time 2D and 3D graphics. It is primarily intended for games and other apps with real-time rendering needs. To learn more about OpenGL in OS X, see OpenGL (page 46).
- **GLKit** (`GLKit.framework`). GLKit provides libraries of commonly needed functions and classes that reduce the effort required to create shader-based apps or to port existing apps that rely on fixed-function vertex or fragment processing provided by earlier versions of OpenGL ES or OpenGL. To learn more about the GLKit framework, see GLKit (page 45).

Text, Typography, and Fonts

OS X provides extensive support for advanced typography for Cocoa apps. With this support, your app can control the fonts, layout, typesetting, text input, and text storage when managing the display and editing of text. For the most basic text requirements, you can use the text fields, text views, and other text-displaying objects provided by the AppKit framework.

There are two technologies to draw upon for more sophisticated text, font, and typography needs: the Cocoa text system and the Core Text API. Unless you need low-level access to the layout manager routines, the Cocoa text system should provide most of the features and performance you need. If you need a lower-level API for drawing any kind of text into a `CGContext`, then you should consider using the Core Text API.

- **Cocoa text system.** AppKit provides a collection of classes, known as the *Cocoa text system* , that together provide a complete set of high-quality typographical services. With these services, apps can create, edit, display, and store text with all the characteristics of fine typesetting, such as kerning, ligatures, line breaking, and justification. Use the Cocoa Text system to display small or large amounts of text and to customize the default layout manager classes to support custom layout. The Cocoa text system is the recommended technology for most apps that require text handling and typesetting capabilities.

  AppKit also offers a class (`NSFont`), a font manager, and a font panel. In addition, the Cocoa text system supports vertical text and linguistic tagging.

  For an overview of the Cocoa text system, see *Cocoa Text Architecture Guide* .

- **Core Text** (`CoreText.framework`). The Core Text framework contains low-level interfaces for laying out Unicode text and handling Unicode fonts. Core Text provides the underlying implementation for many features of the Cocoa text system.

  To learn more about the Core Text framework, see Core Text (page 42).

Images

Both AppKit and Quartz let you create objects that represent images (`NSImage` and `CGImageRef`) from various sources, draw these images to an appropriate graphics context, and even composite one image over another according to a given blending mode. Beyond the native capabilities of AppKit and Core Graphics, you can do other things with images using the following frameworks:

- **Image Capture Core** (`ImageCaptureCore.framework`). The Image Capture Core framework enables your app to browse locally connected or networked scanners and cameras, to list and download thumbnails and images, to take scans, rotate and delete images and, if the device supports it, to take pictures or control the scan parameters. For more information, see Other Media Layer Frameworks (page 50).
- **Core Image.** Core Image is an image processing technology that allows developers to process images with system-provided image filters, create custom image filters, and detect features in an image. Examples of built-in filters are those that crop, blur, and warp images. Core Image is implemented by the Quartz Core framework (`QuartzCore.framework`). For more information, see Core Image (page 49).
- **Image Kit.** Image Kit is built on top of the Image Capture Core framework. It provides views you can use in your app to help users connect to cameras and scanners, view and download images from these devices, and edit and process the images. For more information, see Image Kit (page 47).
- **Image I/O** (`ImageIO.framework`). The Image I/O framework helps you read image data from a source and write image data to a destination. Sources and destinations can be URLs, Core Foundation data objects, and Quartz data consumers and data providers. For more information, see Image I/O (page 42).

Color Management

ColorSync is the color management system for OS X. It provides essential services for fast, consistent, and accurate color reproduction, proofing, and calibration. It also provides an interface for accessing and managing systemwide settings for color devices such as displays, printers, cameras, and scanners.

In most cases, you do not need to call ColorSync functions at all. Quartz and Cocoa automatically use ColorSync to manage pixel data when drawing on the screen or printing. By design, the ICC (International Color Consortium) profiles embedded in the color data of images and PDF documents are fully respected. You are strongly encouraged to use appropriate calibrated color spaces—for example, those based on ICC profiles—when creating your own content. For very special needs, ColorSync also allows you to define a custom color management module (CMM) to use instead of a system-provided CMM to perform required color conversions based on the ICC profiles.

For information about creating custom color spaces, see Making Custom Color Spaces. For information on the ColorSync API, see *ColorSync on Mac OS X*, and the ColorSync header files in `/System/Library/Frameworks/ApplicationServices.framework/Frameworks/ColorSync.framework/Headers`.

Printing

OS X implements printing support using a collection of APIs and system services that are available to all app environments. Drawing on the capabilities of Quartz, the printing system delivers a consistent human interface and streamlines the development process for printer vendors. It also provides apps with a high degree of control over the user interface elements in printing dialogs. Table 3-2 describes some other features of the OS X printing system.

Table 3-2 Features of the OS X printing system

Feature	Description
AirPrint	Users can print to an AirPrint-enabled printer on their network without having to use a third-party driver.
CUPS	Common UNIX Printing System (CUPS), an open source architecture used to handle print spooling and other low-level features, provides the underlying support for printing.
Desktop printers	Desktop printers offer users a way to manage printing and print jobs from the Dock or desktop.
Fax support	Fax support means that users can fax documents directly from the Print dialog.
Native PDF support	PDF as a native data type lets any app easily save textual and graphical data to the device-independent PDF format.

Feature	Description
PostScript support	PostScript support allows apps to use legacy third-party drivers to print to PostScript Level 2–compatible and Level 3–compatible printers and to convert PostScript files directly to PDF.
Print preview	The print preview capability lets users see documents through a PDF viewer app prior to printing.
Printer discovery	Printer discovery enables users to detect, configure, and add to printer lists those printers that implement Bluetooth or Bonjour.
Raster printers (support for)	This support allows apps to print to raster printers using legacy third-party drivers.
Speedy spooling	Speedy spooling enables apps that use PDF to submit PDF files directly to the printing system instead of spooling individual pages.

To learn more about the Cocoa printing architecture, and about how to support printing in a Cocoa app, see *Printing Programming Guide for Mac*; for information about the Core Printing API, see *Core Printing Reference*.

Audio Technologies

OS X includes support for high-quality audio recording, synthesis, manipulation, and playback. The frameworks in the following list are ordered from high level to low level, with the AV Foundation framework offering the highest-level interfaces you can use. When choosing an audio technology, remember that higher-level frameworks are easier to use and, for this reason, are usually preferred. Lower-level frameworks offer more flexibility and control but require you to do more work.

- **AV Foundation** (`AVFoundation.framework`). AV Foundation supports audio playback, editing, analysis, and recording. Unless you are creating a fast-action game, a virtual music instrument, or a Voice over IP (VoIP) app, look first at AV Foundation. For more information, see AV Foundation (page 43).
- **OpenAL** (`OpenAL.framework`). OpenAL implements a cross-platform standard API for 3D audio. OpenAL also lets you add high-performance positional playback in games and other apps. For more information, see OpenAL (page 46).
- **Core Audio** (`CoreAudio.framework`). Core Audio consists of a set of frameworks that provide audio services that support recording, playback, synchronization, signal processing, format conversion, synthesis, and surround sound. Core Audio is well suited for adding VoIP and other high-performance audio features to your app. For more information, see Core Audio (page 44).

Video Technologies

Whether you are playing movie files from your app or streaming them from the network, OS X provides several technologies to play your video-based content. On systems with the appropriate hardware, you can also use these technologies to capture video and incorporate it into your app.

When choosing a video technology, remember that the higher-level frameworks simplify your work and are, for this reason, usually preferred. The frameworks in the following list are ordered from highest to lowest level, with the AV Foundation framework offering the highest-level interfaces you can use.

- **AVKit** (`AVKit.framework`). AV Kit supports playing visual content in your application using the standard controls.
- **AV Foundation** (`AVFoundation.framework`). AV Foundation supports playing, recording, reading, encoding, writing, and editing audiovisual media.
- **Core Media** (`CoreMedia.framework`). Core Media provides a low-level C interface for managing audiovisual media. With the Core Media I/O framework, you can create plug-ins that can access media hardware and that can capture video and mixed audio and video streams.
- **Core Video** (`CoreVideo.framework`). Core Video provides a pipeline model for digital video between a client and the GPU to deliver hardware-accelerated video processing while allowing access to individual frames. Use Core Video only if your app needs to manipulate individual video frames; otherwise, use AV Foundation.

For information about each of the video frameworks (as well as other frameworks) in the Media layer, see Media Layer Frameworks (page 41).

Media Layer Frameworks

OS X includes numerous technologies for rendering and animating 2D and 3D content (including text) and for playing, recording, and editing audiovisual media.

Application Services Umbrella Framework

The Application Services umbrella framework (`ApplicationServices.framework`) includes the following subframeworks. You should not link with these frameworks directly; instead link with (and import) `ApplicationServices.framework`.

Core Graphics

The Quartz 2D client API offered by the Core Graphics framework (`CoreGraphics.framework`) provides commands for managing the graphics context and for drawing primitive shapes, images, text, and other content. The Core Graphics framework defines the Quartz 2D interfaces, types, and constants you use in your apps.

Quartz 2D provides many important features to apps, including the following:

- High-quality rendering on the screen
- High-resolution UI support
- Antialiasing for all graphics and text
- Support for adding transparency information to windows
- Internal compression of data
- A consistent feature set for all printers
- Automatic PDF generation and support for printing, faxing, and saving as PDF
- Color management through ColorSync

For information about the Quartz 2D API, see *Quartz 2D Programming Guide*.

Core Text

Core Text is a C-based API that provides precise control over text layout and typography. Core Text provides a layered approach to laying out and displaying Unicode text. You can modify as much or as little of the system as suits your needs. Core Text also provides optimized configurations for common scenarios, saving setup time in your app.

The Core Text font API is complementary to the Core Text layout engine. Core Text font technology is designed to handle Unicode fonts natively and comprehensively, unifying disparate OS X font facilities so that developers can do everything they need to do without resorting to other APIs.

For more information about Core Text, see *Core Text Programming Guide* and *Core Text Reference Collection*.

Image I/O

The `NSImage` class takes advantage of all the capabilities of Image I/O and is the preferred method for loading and drawing image resources on OS X. It is particularly important to use `NSImage` if you're working with images that are visible in the UI, because this class handles multiresolution images.

Apps can use the Image I/O framework to read and write image data in most file formats in a highly efficient manner. It offers very fast image encoding and decoding facilities, supports image metadata and caching, provides color management, and is able to load image data incrementally.

The central objects in Image I/O are image sources and image destinations. Sources and destinations can be URLs (`CFURLRef`), data objects (`CFDataRef` and `CFMutableDataRef`), and data consumer and data provider objects (`CGDataConsumerRef` and `CGDataProviderRef`).

The Image I/O programmatic interfaces were once part of Quartz 2D (Core Graphics) but have been collected in an separate framework so that apps can use it independently of Core Graphics. To learn more about Image I/O, see *Image I/O Programming Guide*.

AV Foundation

The AV Foundation framework (`AVFoundation.framework`) provides services for capturing, playing, inspecting, editing, and reencoding time-based audiovisual media. The framework includes many Objective-C classes, with the core class being `AVAsset`; this class presents a uniform model and inspection interface for all forms and sources of audiovisual media. The services offered by this framework include the following:

- Movie or audio capture
- Movie or audio playback, including precise synchronization and audio panning
- Media editing and track management
- Media asset and metadata management
- Audio file inspection (for example, data format, sample rate, and number of channels)

For most audio recording and playback requirements, you can use the `AVAudioPlayer` and `AVAudioRecorder` classes.

AV Foundation is the recommended framework for all new development involving time-based audiovisual media. AV Foundation is also recommended for transitioning existing apps based on QuickTime or QTKit.

> **Note:** The AV Foundation frameworks for OS X and iOS are almost identical.

For more information about the classes of the AV Foundation framework, see *AV Foundation Programming Guide*.

ColorSync

The ColorSync framework (`ColorSync.framework`) implements the color management system for OS X. To find out more about ColorSync, see Color Management (page 39).

The ColorSync framework is a subframework of the Application Services umbrella framework. Your project cannot link to it directly; it must link instead to `ApplicationServices.framework`.

Core Audio

Core Audio consists of a group of frameworks that offer sophisticated services for manipulating multichannel audio. (Core Audio is also the name of one framework of this group.) You can use Core Audio to generate, record, mix, edit, process, and play audio. You can also use Core Audio to work with MIDI (Musical Instrument Digital Interface) data using both hardware and software MIDI instruments. The frameworks of Core Audio are as follows:

- **Core Audio** (`CoreAudio.framework`). Core Audio provides interfaces that allow your app to interact with audio hardware and obtain and convert the host's time base. It also defines data types used throughout the Core Audio frameworks.
- **Core Audio Kit** (`CoreAudioKit.framework`). Core Audio Kit contains Objective-C classes that provide user interfaces for audio units.
- **Audio Toolbox** (`AudioToolbox.framework`). Audio Toolbox provides general audio services to apps, such as audio converters and audio processing graphs, reading and writing audio data in files, managing and playing event tracks, and assigning and reading audio format metadata.
- **Audio Unit** (`AudioUnit.framework`). The Audio Unit framework defines programmatic interfaces for creating and manipulating audio units, plug-ins for handling or generating audio signals.
- **Core MIDI** (`CoreMIDI.framework`). Core MIDI provides MIDI support for apps, allowing them to communicate with MIDI devices, to configure or customize the global state of the MIDI system, and to create MIDI play-through connections between MIDI sources and destinations.

Most Core Audio interfaces are C based, providing a low-latency and flexible programming environment that is suitable for real-time apps. You can use Core Audio from any OS X program. Some of the benefits of Core Audio include the following:

- Built-in support for reading and writing a wide variety of audio file and data formats
- Plug-in support for custom formats, audio synthesis, and processing (audio DSP)
- A modular approach for constructing audio signal chains
- Easy MIDI synchronization
- Support for scheduled playback and synchronization with access to timing and control information
- A standardized interface to all built-in and external hardware devices, regardless of connection type (USB, Firewire, PCI, and so on)

Audio queue objects give you access to incoming or outgoing raw audio data while the system mediates playback and recording, employs codecs (if needed), and makes hardware connections. To intercept and process audio as it proceeds through an audio queue object, you use the real-time, callback-based feature known as *audio queue processing taps*. This feature works for input (such as for recording) as well as output (such as for playback). Audio queue processing taps let you insert audio units or other forms of audio processing, effectively tapping into and modifying the data stream within an audio queue. You can also use this technology in a *siphon mode*, in which you gain access to the stream but do not modify it.

To learn more about the Core Audio framework, see *Core Audio Framework Reference*.

GLKit

GLKit framework (`GLKit.framework`) helps reduce the effort required to create new shader-based apps or to port existing apps that rely on fixed-function vertex or fragment processing provided by earlier versions of OpenGL ES or OpenGL. In addition, the GLKit framework includes APIs that perform several optimized mathematical operations, reduce the effort in loading texture data, and provide standard implementations of commonly needed shader effects.

> **Note:** GLKit requires an OpenGL context that supports the OpenGL 3.2 Core Profile. GLKit is not available to 32-bit apps in OS X.

To learn more about using GLKit in your app, see *GLKit Framework Reference*.

Instant Messaging

OS X provides two technologies to support instant messaging:

- **Instant Message** (`InstantMessage.framework`). The Instant Message framework supports the detection and display of a user's online presence in apps other than iChat. You can find out the current status of a user connected to an instant messaging service, obtain the user's custom icon and status message, or obtain a URL to a custom image that indicates the user's status.

  You can use the Instant Message framework to support iChat Theater. This feature gives your app the ability to inject audio or video content into a running iChat conference.

- **Instant Message Service Plug-in** (`IMServicePlugin.framework`). The Instant Message Service Plug-in framework enables the creation of bundle plug-ins that let apps talk to instant-messaging services, such as iChat. It includes support for the buddy list, handle availability, handle icon, status messages, instant messaging, and group-chat messaging.

For more information about using the Instant Message framework, see *Instant Message Programming Guide*. For more information about the Instant Message Service Plug-in framework, see *IMServicePlugIn Protocol Reference* and *IMServicePlugInInstantMessagingSupport Protocol Reference*

OpenAL

The Open Audio Library (`OpenAL.framework`) is a cross-platform standard framework for delivering 3D audio. OpenAL lets you implement high-performance positional playback in games and other programs. Because it is a cross-platform standard, apps you write using OpenAL in OS X can be ported to run on many other platforms.

OpenAL in OS X supports audio capture, exponential and linear distance models, location offsets, and spatial effects such as reverb and occlusion. The OS X implementation of OpenAL also supports certain Core Audio features, such as mixer sample rates.

To learn about the OpenAL specification, visit OpenAL.

OpenGL

OpenGL is an industry wide standard for developing portable three-dimensional (3D) graphics apps. It is specifically designed for apps such as games that need a rich, robust framework for visualizing shapes in two and three dimensions. OpenGL is one of the most widely adopted graphics API standards, which makes code written for OpenGL portable and consistent across platforms. The OpenGL framework (`OpenGL.framework`) in OS X includes a highly optimized implementation of the OpenGL libraries that provides high-quality graphics at a consistently high level of performance.

OpenGL offers a broad and powerful set of imaging functions, including texture mapping, hidden surface removal, alpha blending (transparency), antialiasing, pixel operations, viewing and modeling transformations, atmospheric effects (fog, smoke, and haze), and other special effects. Each OpenGL command directs a drawing action or causes a special effect, and developers can create lists of these commands for repetitive effects. Although OpenGL is largely independent of the windowing characteristics of each operating system, the standard defines special glue routines to enable OpenGL to work in an operating system's windowing environment. The OS X implementation of OpenGL implements these glue routines to enable operation with Quartz Compositor.

OpenGL supports the ability to use multiple threads to process graphics data. OpenGL also supports pixel buffer objects, color-managed texture images in the sRGB color space, and 64-bit addressing. It also offers improvements in the shader programming API.

OS X v10.7 added support for Open GL 3.2 Core, which added more extensions as baseline features. Note that routines and mechanisms found in OpenGL 1.*n* and OpenGL 2.*n* are deprecated, including the removal of the fixed-function pipeline that was the main approach to developing OpenGL 1.*n* apps. An OpenGL 3.2 app

requires you to create your own shader strategy rather than assuming that a standard graphics pipeline will do everything. As a result, for simple apps you need to write more boilerplate code than was required by previous versions of OpenGL.

For information about using OpenGL in OS X, see *OpenGL Programming Guide for Mac*.

Quartz

The Quartz umbrella framework includes the following subframeworks. You should not link directly with any of the subframeworks; instead link with (and import) `Quartz.framework`.

Image Kit

The Image Kit (`ImageKit.framework`) is an Objective-C framework that makes it easy to incorporate powerful imaging services into your apps. This framework takes advantage of features in Quartz, Core Image, OpenGL, and Core Animation to provide an advanced and highly optimized development path for implementing the following features:

- Displaying images
- Rotating, cropping, and performing other image-editing operations
- Browsing for images
- Taking pictures using the built-in picture taker panel
- Displaying slideshows
- Browsing for Core Image filters
- Displaying custom views for Core Image filters

For more information on how to use Image Kit, see *ImageKit Programming Guide* and *ImageKit Reference Collection*.

PDF Kit

PDF Kit (`PDFKit.framework`, a subframework of the Quartz framework) is a Cocoa framework for managing and displaying PDF content directly from your app's windows and dialogs. You can embed a `PDFView` object in your window and give it a PDF file to display. The PDF view handles the rendering of the PDF content, handles copy-and-paste operations, and provides controls for navigating and setting the zoom level. Other classes let you get the number of pages in a PDF file, find text, manage selections, add annotations, and specify the behavior of some graphical elements, among other actions.

For more information on PDF Kit, see *PDFKit Programming Guide*.

Quartz Composer

The Quartz Composer framework (`QuartzComposer.framework`) provides programmatic support for working with Quartz Composer compositions. It enables apps to load, play, and control compositions and to integrate them with Cocoa views, Core Animation layers, or OpenGL rendering.

For more information, see *Quartz Composer Programming Guide*.

Quick Look UI

The Quick Look UI framework (`QuickLookUI.framework`) defines an interface for implementing a Quick Look preview panel, which displays the preview of a list of items. The framework also includes the capability for embedding a preview inside a view.

For an example of an app that implements Quick Look preview panels, see *QuickLookDownloader*.

Quartz Core

The Quartz Core framework (`QuartzCore.framework`) implements two important system technologies for graphics and imaging Core Animation and Core Image.

Core Animation

Core Animation is a set of Objective-C classes used for sophisticated 2D rendering and animation. Using Core Animation, you can create everything from basic window content to carousel–style user interfaces (such as the Front Row interface), and achieve respectable animation performance without having to tune your code using OpenGL or other low-level drawing routines. This performance is achieved using server-side content caching, which restricts the compositing operations performed by the server to only those parts of a view or window whose contents actually change.

At the heart of the Core Animation programming model are layer objects, which are similar in many ways to Cocoa views. As with views, you can arrange layers in hierarchies, change their size and position, and tell them to draw themselves. Unlike views, layers do not support event handling, accessibility, or drag and drop.

You can manipulate the layout of layers in more ways than the layout of traditional Cocoa views. In addition to positioning layers using a layout manager, you can apply 3D transforms to layers to rotate, scale, skew, or translate them in relation to their parent layer.

Layer content can be animated implicitly or explicitly depending on the actions you take. Modifying specific properties of a layer—such as its geometry, visual attributes, or children—typically triggers an implicit animation to transition from the property's old state to the new one. For example, adding a child layer triggers an animation that causes the child layer to fade gradually into view. You can also trigger animations explicitly in a layer by modifying its transformation matrix.

You can manipulate layers independent of, or in conjunction with, the views and windows of your app. Cocoa apps can take advantage of Core Animation's integration with the `NSView` class to add animation effects to windows. Layers can also support the following types of content:

- Quartz and Cocoa drawing content
- OpenGL content
- Quartz Composer compositions
- Core Image filter effects

For information about Core Animation, see *Animation Overview*.

Core Image

Core Image extends the basic graphics capabilities of the system to provide a framework for implementing complex visual behaviors in your app. Core Image uses GPU-based acceleration and 32-bit floating-point support to provide fast image processing and pixel-level accurate content. Its plug-in–based architecture lets you expand the capabilities of Core Image through the creation of image units, which implement the desired visual effects.

Core Image includes built-in image units that allow you to do the following:

- Crop, composite, blur, and sharpen images
- Correct color, including performing white-point adjustments
- Apply color effects, such as sepia tone
- Warp the geometry of an image by applying an affine transform or a displacement effect
- Generate color, checkerboard patterns, Gaussian gradients, and other pattern images
- Add transition effects to images or video
- Provide real-time control, such as color adjustment and support for sports mode, vivid mode, and other video modes
- Apply linear lighting effects, such as spotlights
- Face detection including mouth and eye bounds.

You can use both the built-in and custom image units in your app to implement special effects and perform other types of image manipulations.

For information about how to use Core Image or how to write custom image units, see *Core Image Programming Guide* and *Core Image Reference Collection*. For information about the built-in filters in Core Image, see *Core Image Filter Reference*.

QuickTime Kit

QTKit (`QTKit.framework`) is an Objective-C framework for manipulating QuickTime-based media. This framework lets you incorporate movie playback, movie editing, export to standard media formats, and other QuickTime behaviors easily into your apps.

> **Note:** AV Foundation is the recommended API for all new development involving time-based audiovisual media. AV Foundation is also recommended for transitioning existing apps that were based on QuickTime or QTKit. For more information, see AV Foundation (page 43).

Scene Kit

The Scene Kit framework (`SceneKit.framework`) provides a high-level Objective-C API that helps you to efficiently load, manipulate, and render 3D scenes in your app. Scene Kit allows you to import Digital Asset Exchange files (`.dae` files) that are created by popular content-creation applications and gives you access to the objects, lights, cameras, and geometry data that define a 3D scene. Using an approach based on scene graphs, Scene Kit makes it simple to modify, animate, and render your 3D scenes.

Scene Kit integrates with Image Kit, SpriteKit, and Core Animation, so you do not need advanced 3D graphical programming skills. For example, you can embed a 3D scene into a layer and then use Core Animation compositing capabilities to add overlays and backgrounds. You can also use Core Animation layers as textures for your 3D objects in 3D scenes. To learn how to use Scene Kit in your app, see *Scene Kit Programming Guide*.

Sprite Kit

The Sprite Kit framework (`SpriteKit.framework`) provides a graphics rendering and animation infrastructure that you can use to animate arbitrary textured images, or sprites. You use graphical tools to create the sprites. You then create scenes including sprites, light sources, emitters, and physics fields. Sprite Kit animates the scene you specify using a traditional animation loop, doing the work to render frames of animation efficiently using the graphics hardware.

Sprite kit includes basic sound playback support in addition to physics simulation. In addition, you can create complex special effects and texture atlases directly in Xcode. This combination of framework and tools makes Sprite Kit a good choice for games and other apps that require similar kinds of animation. Animated images created with Sprite Kit work well with SceneKit. To learn how to use Sprite Kit, see *SpriteKit Programming Guide*.

Other Media Layer Frameworks

The Media layer of OS X also has the following frameworks:

- **Audio Video Bridging** (`AudioVideoBridging.framework`). The Audio Video Bridging framework supports Audio Video Bridging (AVB) and implements the IEEE P1722.1 draft standard. AVB enhances the quality of service and provides guaranteed latency and bandwidth for media streams over an AVB network. Audio Video Bridging gives you access to the Entity discovery and control protocols of IEEE P1722.1 over an Ethernet network.
- **Core Media** (`CoreMedia.framework`). The Core Media framework provides low-level audiovisual media objects and tools for managing them. It also defines the fundamental time representation used uniformly throughout AV Foundation.

  For information about using the Core Media framework, see *Core Media Framework Reference*.
- **Core Media I/O** (`CoreMediaIO.framework`). The Core Media I/O framework publishes the Device Abstraction Layer (DAL) plug-in API. This technology enables you to create plug-ins that can access media hardware and capture video and "muxed" (video combined with audio) streams. Core Media I/O is a replacement for the QuickTime VDig API.
- **Core Video** (`CoreVideo.framework`). Core Video creates a bridge between QuickTime and the graphics card's GPU to deliver hardware-accelerated video processing. Benefits of Core Video include filters and effects, per-pixel accuracy, and hardware scalability.

  For information about using the Core Video framework, see *Core Video Programming Guide*.
- **Disc Recording** (`DiscRecording.framework`). Disc Recording gives apps the ability to burn and erase CDs and DVDs. Your app specifies the content to be burned but the framework (`DiscRecording.framework`) takes over the process of buffering the data, generating the proper file format information, and communicating everything to the burner.

  The Disc Recording UI framework (`DiscRecordingUI.framework`) provides a complete, standard set of windows for gathering information from the user and displaying the progress of the burn operation.

  For more information about the Disc Recording frameworks, see *Disc Recording Framework Reference* and *Disc Recording UI Framework Reference*.
- **DVD Playback** (`DVDPlayback.framework`). The DVD Playback framework embeds DVD viewer capabilities into an app. You use the framework to control various aspects of playback, including menu navigation, viewer location, angle selection, and audio track selection. You can play back DVD data from disc or from a local `VIDEO_TS` directory.

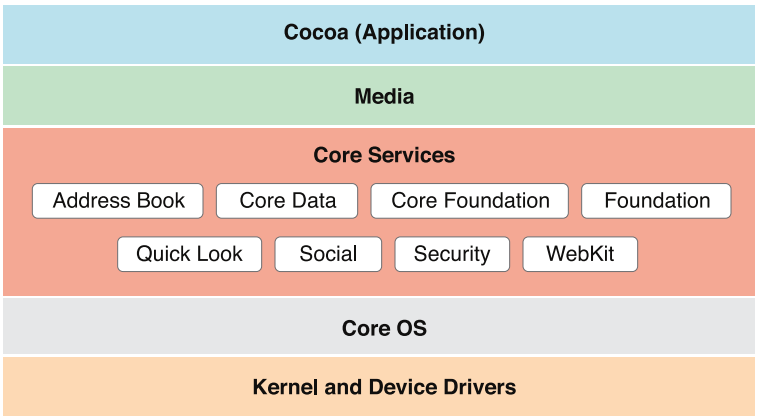
  For more information about using the DVD Playback framework, see *DVD Playback Services Programming Guide*.
- **Image Capture Core** (`ImageCaptureCore.framework`). Image Capture Core helps you capture image data from scanners and digital cameras. The interfaces of the framework are device independent, so you can use them to gather data from any devices connected to the system. You can get a list of devices, retrieve information about a specific device or image, and retrieve the image data itself.

This framework works in conjunction with the Image Capture Devices framework (`ICADevices.framework`) to communicate with imaging hardware. For information on using these frameworks, see *Image Capture Applications Programming Guide*.

- **Video Toolbox** (`VideoToolbox.framework`). The Video Toolbox framework comprises the 64-bit replacement for the QuickTime Image Compression Manager. Video Toolbox provide services for video compression and decompression, and for conversion between raster image formats stored in Core Video pixel buffers.

Core Services Layer

The technologies in the Core Services layer are called *core services* because they provide essential services to apps but have no direct bearing on the app's user interface. In general, these technologies are dependent on frameworks and technologies in the two lowest layers of OS X—that is, the Core OS layer and the Kernel and Device Drivers layer.



High-Level Features

The following sections describe some of the key technologies available in the Core Services layer.

Social Media Integration

Two frameworks that make it easy for users to share content with various social networking services from within your app:

- **Accounts** (`Accounts.framework`). The Accounts framework provides access to supported account types that are stored in the Accounts database.
- **Social** (`Social.framework`). The Social framework provides an API for sending requests to supported social media services that can perform operations on behalf of your users.

When you use the Accounts framework, your app does not need to be responsible for storing account login information because an `ACAccount` object stores the users login credentials for services such as Twitter and Facebook. Users can grant your app permission to use their account login credentials, bypassing the need to

enter their user name and password. If no account for a particular service exists in the user's Accounts database, you can help them create and save an account from within your app. To learn more about the Accounts framework, see *Accounts Framework Reference*.

The Social framework provides a simple interface for accessing the user's social media accounts, including Facebook and Sina Weibo, a Chinese microblogging website. Apps can use this framework to post status updates and images to a user's account. The Social framework works with the Accounts framework to provide a single sign-on model for the user and to ensure that access to the user's account is approved. For more information about the Social API, see *Social Framework Reference*.

iCloud Storage

From a user's perspective, iCloud is a simple feature that automatically makes their personal content available on all of their devices. When you adopt iCloud, OS X initiates and manages uploading and downloading of data for the devices associated with an iCloud account.

There are three types of iCloud storage that your app can take advantage of:

- **Document storage**. Document storage is for user-visible file-based content, such as presentations or documents, or for other substantial file-based content, such as the state of a complex game.
- **Key-value storage**. Key-value storage is for sharing small amounts of data—such as preferences or bookmarks—among instances of your app.
- **Core Data storage**. Core Data storage supports server-based, multidevice database solutions for structured content. (Core Data storage is built on document storage.)

Many apps can benefit from using more than one type of storage. For example, a complex strategy game could employ document storage for its game state, and key-value storage for scores and achievements.

> **Important:** To use iCloud storage in your app, you need to get an appropriate provisioning profile for your development device and request the appropriate entitlements in your Xcode project. To learn more about these tasks, see Provisioning Your System and Configuring Entitlements in *Tools Workflow Guide for Mac*.

To learn more about adding iCloud storage to your app, read *iCloud Design Guide*.

CloudKit

CloudKit provides apps with more control over when and how data is stored in iCloud. Unlike iCloud Storage, CloudKit is a complimentary service that works with your apps existing data structures. CloudKit records include support for:

- Saving, searching, and fetching data for specific to an individual user

- Saving, searching, and fetching data in a public area shared by all users

CloudKit has minimal caching and relies on a network connection. To learn more about using CloudKit, see the *CloudKit Framework Reference*

File Coordination

File coordination eliminates file-system inconsistencies due to overlapping read and write operations from competing processes. When you use the `NSDocument` class from the AppKit framework, you get file coordination with very little effort. To use file coordination explicitly, you employ the `NSFileCoordinator` class and the `NSFilePresenter` protocol, both from the Foundation framework.

The file coordination APIs let you assert your app's ownership of files and directories. When another process attempts access, you have a chance to respond. For example, if another app attempts to read a document that your app is editing, you have a chance to write unsaved changes to disk before the other app is allowed to do its reading.

You use file coordination only with files that users conceivably might share with other users, not (for example) with files written to a cache or other temporary locations. File coordination is an enabling technology for automatic document saving, App Sandbox, and other features introduced in OS X v10.7. For more information on file coordination, see Coordinating File Access With Other Processes in *Mac App Programming Guide*.

Bundles and Packages

A feature integral to OS X software distribution is the bundle mechanism. Bundles encapsulate related resources in a hierarchical file structure but present those resources to the user as a single entity. Programmatic interfaces make it easy to find resources inside a bundle. These same interfaces form a significant part of the OS X internationalization strategy.

Apps and frameworks are only two examples of bundles in OS X. Plug-ins, screen savers, and preference panes are also implemented using the bundle mechanism. Developers can also use bundles for their document types to make it easier to store complex data.

Packages are another technology, like bundles, that make distributing software easier. A package—also referred to as an *installation package*—is a directory that contains files and directories in well-defined locations. The Finder displays packages as files. Double-clicking a package launches the Installer app, which then installs the contents of the package on the user's system.

For an overview of bundles and to learn how they are constructed, see *Bundle Programming Guide*.

Internationalization and Localization

Localization (which is the process of adapting your app for use in another region) is necessary for success in many foreign markets. Users in other countries are much more likely to buy your software if the text and graphics reflect their own language and culture. Before you can localize an app, though, you must design it in a way that supports localization, a process called internationalization. Properly internationalizing an app makes it possible for your code to load localized content and display it correctly.

Internationalizing an app involves the following steps:

- Use Unicode strings for storing user-visible text.
- Extract user-visible text into "strings" resource files.
- Use nib files to store window and control layouts whenever possible.
- Use international or culture-neutral icons and graphics whenever possible.
- Use Cocoa or Core Text to handle text layout.
- Support localized file-system names (also known as *display names*).
- Use formatter objects in Core Foundation and Cocoa to format numbers, currencies, dates, and times based on the current locale.

For details on how to support localized versions of your software, see *Internationalization and Localization Guide*. For information on Core Foundation formatters, see *Data Formatting Guide for Core Foundation*.

Block Objects

Block objects, or *blocks*, are a C-level mechanism that you can use to create an ad hoc function body as an inline expression in your code. In other languages and environments, a block is sometimes called a *closure* or a *lambda*. You use blocks when you need to create a reusable segment of code but defining a function or method might be a heavyweight (and perhaps inflexible) solution. For example, blocks are a good way to implement callbacks with custom data or to perform an operation on all the items in a collection. Many OS X technologies—for example Game Kit, Core Animation, and many Cocoa classes—use blocks to implement callbacks.

The compiler provides support for blocks using the C, C++, and Objective-C languages. For more information about how to create and use blocks, see *Blocks Programming Topics*.

Grand Central Dispatch

Grand Central Dispatch (GCD) provides a simple and efficient API for achieving the concurrent execution of code in your app. Instead of providing threads, GCD provides the infrastructure for executing any task in your app asynchronously using a dispatch queue. Dispatch queues collect your tasks and work with the kernel to facilitate their execution on an underlying thread. A single dispatch queue can execute tasks serially or concurrently, and apps can have multiple dispatch queues executing tasks in parallel.

There are several advantages to using dispatch queues over traditional threads. One of the most important is performance. Dispatch queues work more closely with the kernel to eliminate the normal overhead associated with creating threads. Serial dispatch queues also provide built-in synchronization for queued tasks, eliminating many of the problems normally associated with synchronization and memory contention normally encountered when using threads.

In addition to providing dispatch queues, GCD provides three other dispatch interfaces to support the asynchronous design approach offered by dispatch queues:

Dispatch sources provide a way to handle the following types of kernel-level events that is more efficient than BSD alternatives:

- Timer notifications
- Signal handling
- Events associated with file and socket operations
- Significant process-related events
- Mach-related events
- Custom events that you define and trigger
- Asynchronous I/O through dispatch data and dispatch I/O

Dispatch groups allow one thread (or *task*) to block while it waits for one or more other tasks to finish executing.

Dispatch semaphores provide a more efficient alternative to the traditional semaphore mechanism.

For more information about how to use GCD in your apps, see *Concurrency Programming Guide*.

Bonjour

Bonjour is Apple's implementation of the zero-configuration networking architecture, a powerful system for publishing and discovering services over an IP network. It is relevant to both software and hardware developers.

Incorporating Bonjour support into your software improves the overall user experience. Rather than prompt the user for the exact name and address of a network device, you can use Bonjour to obtain a list of available devices and let the user choose from that list. For example, you could use it to look for available printing services, which would include any printers or software-based print services, such as a service to create PDF files from print jobs.

Developers of network-based hardware devices are strongly encouraged to support Bonjour. Bonjour alleviates the need for complicated setup instructions for network-based devices such as printers, scanners, RAID servers, and wireless routers. When plugged in, these devices automatically publish the services they offer to clients on the network.

For information on how to incorporate Bonjour services into a Cocoa app, see *Bonjour Overview*. To incorporate Bonjour into a non-Cocoa app, see *DNS Service Discovery Programming Guide*.

Security Services

OS X security is built upon several open source technologies—including BSD and Kerberos—adds its own features to those technologies. The Security framework (`Security.framework`) implements a layer of high-level services to simplify your security solutions. These high-level services provide a convenient abstraction and make it possible for Apple and third parties to implement new security features without breaking your code. They also make it possible for Apple to combine security technologies in unique ways.

OS X provides high-level interfaces for the following features:

- User authentication
- Certificate, Key, and Trust Services
- Authorization Services
- Secure Transport
- Keychain Services
- Smart cards with the CryptoTokenKit framework

Security Transforms, provide a universal context for all cryptographic work. A cryptographic unit in Security Transforms, also known as a *transform*, can be used to perform tasks such as encryption, decryption, signing, verifying, digesting, and encoding. You can also create custom transforms. Transforms are built upon GCD and define a data-flow model for processing data that allows high throughput on multicore machines.

OS X supports many network-based security standards; for a complete list of network protocols, see Standard Network Protocols (page 78). For more information about the security architecture and security-related technologies of OS X, see *Security Overview*.

Maps

MapKit provides a way for embedding maps into your windows and views. There is support for annotation, overlays, and reverse-geocoding lookup using coordinates. For more information, see *MapKit Framework Reference*

Address Book

Address Book is technology that encompasses a centralized database for contact and group information, an app for viewing that information, and a programmatic interface for accessing that information in your app. The database contains information such as user names, street addresses, email addresses, phone numbers, and distribution lists. Apps that support the Address Book framework can use this data as is or extend it to include app-specific information. They can also share user records with system apps, such as Contacts and Mail. For more information about the Address Book framework, see Address Book (page 63).

Address Book gives users control over their contacts data by requiring your app to get permission before it can access the Address Book database. When users have enabled iCloud, Address Book keeps their data synchronized across all their devices by using the CardDAV protocol. To learn how to integrate Address Book into your app, see *Address Book Programming Guide for Mac*.

Speech Technologies

OS X contains speech technologies that recognize and speak U.S. English.

Speech recognition is the ability for the computer to recognize and respond to a person's speech. Using speech recognition, users can accomplish tasks comprising multiple steps with one spoken command. Because users control the computer by voice, speech-recognition technology is very important for people with special needs. You can take advantage of the speech engine and API included with OS X to incorporate speech recognition into your apps.

Speech synthesis, also called text-to-speech (TTS), converts text into audible speech. TTS provides a way to deliver information to users without forcing them to shift attention from their current task. For example, the computer could deliver messages such as "Your download is complete" and "You have email from your boss; would you like to read it now?" in the background while you work. TTS is crucial for users with vision or attention disabilities. As with speech recognition, TTS provides an API and several user interface features to help you incorporate speech synthesis into your apps. You can also use speech synthesis to replace digital audio files of spoken text. Eliminating these files can reduce the overall size of your software bundle.

For more information, see *Speech Synthesis Programming Guide* and *NSSpeechRecognizer Class Reference*.

Identity Services

Identity Services encompasses features located in the Collaboration and Core Services frameworks. Identity Services provides a way to manage groups of users on a local system. In addition to standard login accounts, administrative users can now create sharing accounts, which use access control lists (ACLs) to restrict access to designated system or app resources. An access control list (ACL) gives fine-grained access to file-system objects. Sharing accounts do not have an associated home directory on the system and have much more limited privileges than traditional login accounts.

For more information about the features of Identity Services and how you use those features in your apps, see *Identity Services Programming Guide* and *Identity Services Reference Collection*.

Time Machine Support

Time Machine protects user data from accidental loss by automatically backing up data to a different hard drive. Included with this feature is a set of programmer-level functions that you can use to exclude unimportant files from the backup set. For example, you might use these functions to exclude your app's cache files or any files that can be recreated easily. Excluding these types of files improves backup performance and reduces the amount of space required to back up the user's system.

For information about using the Backup Core API, see *Backup Core Reference*.

Keychain Services

Keychain Services provides a secure way to store passwords, keys, certificates, and other sensitive information associated with a user. Users often have to manage multiple user IDs and passwords to access various login accounts, servers, secure websites, instant messaging services, and so on. A keychain is an encrypted container that holds passwords for multiple apps and secure services. Access to the keychain is provided through a single master password. Once the keychain is unlocked, Keychain Services–aware apps can access authorized information without bothering the user.

If your app handles passwords or sensitive information, you should support Keychain Services in your app. For more information on this technology, see *Keychain Services Programming Guide*.

XML Parsing

OS X includes several XML parsing technologies. Most apps should use these Foundation classes: `NSXMLParser` for parsing XML streams, and the NSXML classes (for example, `NSXMLNode`) for representing XML documents internally as tree structures. Core Foundation also provides a set of functions for parsing XML content.

The open source `libXML2` library allows your app to parse or write arbitrary XML data quickly. The headers for this library are located in the `/usr/include/libxml2` directory.

For information on parsing XML from a Cocoa app, see *Event-Driven XML Programming Guide* and *Tree-Based XML Programming Guide*.

SQLite Database

The SQLite library lets you embed a SQL database engine into your apps. Programs that link with the SQLite library can access SQL databases without running a separate RDBMS process. You can create local database files and manage the tables and records in those files. The library is designed for general purpose use but is still optimized to provide fast access to database records.

The SQLite library is located at `/usr/lib/libsqlite3.dylib`, and the `sqlite3.h` header file is in `/usr/include`. A command-line interface (`sqlite3`) is also available for communicating with SQLite databases using scripts. For details on how to use this command-line interface, see the `sqlite3` man page.

For more information about using SQLite, go to SQLite Home Page.

Notification Center

Apps can create and manage extensions in the Today view of the Notification Center. Extensions are used to give the user a summary of important pieces of information and can perform simple actions or launch the app. For more information, see *Notification Center Framework Reference*

Distributed Notifications

A distributed notification is a message posted by any process to a per-computer notification center, which in turn broadcasts the message to any processes interested in receiving it. Included with the notification is the ID of the sender and an optional dictionary containing additional information. The distributed notification mechanism is implemented by the Core Foundation `CFNotificationCenter` object and by the Cocoa `NSDistributedNotificationCenter` class.

Distributed notifications are ideal for simple notification-type events. For example, a notification might communicate the status of a certain piece of hardware, such as the network interface. Notifications should not be used to communicate critical information to a specific process because delivery of a notification to every registered received is not guaranteed.

Distributed notifications are true notifications because there is no opportunity for the receiver to reply to them. There is also no way to restrict the set of processes that receive a distributed notification. Any process that registers for a given notification may receive it. Because distributed notifications use a string for the unique registration key, there is a potential for namespace conflicts.

For information on Core Foundation support for distributed notifications, see *CFNotificationCenter Reference*. For information about Cocoa support for distributed notifications, see *Notification Programming Topics*.

Core Service Frameworks

OS X includes several core services that make developing apps easier. These technologies range from utilities for managing your internal data structures to high-level frameworks for speech recognition and accessing calendar data. This section summarizes the technologies in the Core Services layer that are relevant to developers—that is, that have programmatic interfaces or have an impact on how you write software.

Core Services Umbrella Framework

The Core Services umbrella framework (`CoreServices.framework`) includes the following frameworks:

- **Launch Services** (`LaunchServices.framework`). Launch Services gives you a programmatic way to open apps, documents, URLs, or files with a given MIME type in a way similar to the Finder or the Dock. The Launch Services framework also provides interfaces for programmatically registering the document types your app supports. Launch Services is in the Core Services umbrella framework. For information on how to use Launch Services, see *Launch Services Programming Guide*.
- **Metadata** (`Metadata.framework`). The Metadata framework helps you to create Spotlight importer plug-ins. It also provides a query API that you can use in your app to search for files based on metadata values and then sort the results based on certain criteria. (The Foundation framework offers an Objective-C interface to the query API.) For more information on Spotlight importers, querying Spotlight, and the Metadata framework, see *Spotlight Importer Programming Guide* and *File Metadata Search Programming Guide*.
- **Search Kit** (`SearchKit.framework`). Search Kit lets you search, summarize, and retrieve documents written in most human languages. You can incorporate these capabilities into your app to support fast searching of content managed by your app. This framework is part of the Core Services umbrella framework. For detailed information about the available features, see *SearchKit Reference*.
- **Web Services Core** (`WebServicesCore.framework`). Web Services Core provides support for the invocation of web services using CFNetwork. The available services cover a wide range of information and include things such as financial data and movie listings. Web Services Core is part of the Core Services umbrella framework. For a description of web services and information on how to use the Web Services Core framework, see *Web Services Core Programming Guide*.
- **Dictionary Services** (`DictionaryServices.framework`). Dictionary Services lets you create custom dictionaries that users can access through the Dictionary app. Through these services, your app can also access dictionaries programmatically and can support user access to dictionary look-up through a contextual menu. For more information, see *Dictionary Services Programming Guide*.

You should not link directly to any of these subframeworks; instead link to (and import) `CoreServices.framework`.

Accounts

The Accounts framework (`Accounts.framework`) provides a single sign-on model for supported account types such as Twitter and Facebook. Single sign-on improves the user experience because it prevents your app from having to prompt a user separately for login information related to an account. It also simplifies the development model for you by managing the account authorization process for your app.

Address Book

The Address Book framework (`AddressBook.framework`) uses a centralized database to store the user's contacts and other personal information. With the user's permission, your app can use the Address Book to access Exchange and CardDAV contacts and allow users to display and edit contacts in a standardized user interface.

The Address Book framework supports yearless dates and several instant-messaging services, in addition to the creation of custom plug-ins. For more information about using the Address Book APIs, see either *Address Book Objective-C Framework Reference for Mac* or *Address Book C Framework Reference for Mac*.

Automator

The Automator framework (`Automator.framework`) enables your app to run workflows. Workflows string together the actions defined by various apps to perform complex tasks automatically. Unlike AppleScript, which uses a scripting language to implement the same behavior, workflows are constructed visually, requiring no coding or scripting skills to create.

For information about incorporating workflows into your own apps, see *Automator Framework Reference*.

Core Data

The Core Data framework (`CoreData.framework`) manages the data model of an app in terms of the Model-View-Controller design pattern. Instead of defining data structures programmatically, you use the graphical tools in Xcode to build a schema representing your data model. At runtime, entities are created, managed, and made available through the Core Data framework with little or no coding on your part.

Core Data provides the following features:

- Storage of object data in mediums ranging from an XML file to a SQLite database
- Management of undo/redo operations beyond basic text editing
- Support for validation of property values
- Support for propagating changes and ensuring that the relationships between objects remain consistent

- Grouping, filtering, and organizing data in memory and transferring those changes to the user interface through Cocoa bindings

Core Data also includes incremental storage, a work concurrency model, and nested managed object contexts.

- Using the incremental store classes (`NSIncrementalStore` and `NSIncrementalStoreNode`), you can create Core Data stores that connect to any possible data source.
- The work concurrency model API enables your app to share unsaved changes between threads across the object graph efficiently.
- You can create nested managed object contexts, in which fetch and save operations are mediated by the parent context instead of a coordinator. This pattern has a number of usage scenarios, including performing background operations on a second thread or queue and managing discardable edits, such as in an inspector window or view.

For more information, see *Core Data Programming Guide*.

Event Kit

Event Kit (`EventKit.framework`) provides an interface for accessing a user's calendar events and reminder items. You can use the APIs in this framework to get existing events and to add new events to the user's calendar. Events that are created using Event Kit are automatically propagated to the CalDAV or Exchange calendars on other devices, which allows your app to display up-to-date calendar information without requiring users to open the Calendar app. (Calendar events can include configurable alarms with rules for when they should be delivered.)

You can also use Event Kit APIs to access reminder lists, create new reminders, add an alarm to a reminder, set the due date and start date of a reminder, and mark a reminder as complete. To learn more about the Event Kit APIs, see *EventKit Framework Reference*.

Foundation and Core Foundation

The Foundation and Core Foundation frameworks are essential to most types of software developed for OS X. The basic goals of both frameworks are the same:

- Define basic object behavior and introduce consistent conventions for object mutability, object archiving, notifications, and similar behaviors.
- Define basic object types representing strings, numbers, dates, data, collections, and so on.
- Support internationalization with bundle technology and Unicode strings.
- Support object persistence.

- Provide utility classes that access and abstract system entities and services at lower layers of the system, including ports, threads, processes, run loops, file systems, and pipes.

The difference between Foundation (`Foundation.framework`) and Core Foundation (`CoreFoundation.framework`) is the language environment in which they are used. Foundation is an Objective-C framework and is intended to be used with all other Objective-C frameworks that declare methods taking Foundation class types as parameters or with return values that are Foundation class types. Along with the AppKit and Core Data frameworks, Foundation is considered to be a core framework for app development (see Cocoa Umbrella Framework (page 32)). Core Foundation, on the other hand, declares C-based programmatic interfaces; it is intended to be used with other C-based frameworks, such as Core Graphics.

> **Note:** Although you can use Core Foundation objects and call Core Foundation functions in Objective-C or Swift projects, there is rarely a reason for doing so.

In its implementation, Foundation is based on Core Foundation. And, although it is C based, the design of the Core Foundation interfaces are more object-oriented than C. As a result, the opaque types (often referred to as *objects*) you create with Core Foundation interfaces operate seamlessly with the Foundation interfaces. Specifically, most equivalent Foundation classes and Core Foundation opaque types are toll-free bridged; this means that you can convert between object types through simple typecasting.

Foundation and Core Foundation provide basic data types and data management features, including the following:

- Collections
- Bundles and plug-ins
- Strings
- Raw data blocks
- Dates and times
- Preferences
- Streams
- URLs
- XML data
- Locale information
- Run loops
- Ports and sockets
- Notification Center interaction

- Interprocess communication between apps using XPC

For an overview of Foundation, read the introduction to *Foundation Framework Reference*. For an overview of Core Foundation, read *Core Foundation Design Concepts*.

Quick Look

Quick Look enables apps such as Spotlight and the Finder to display thumbnail images and full-size previews of documents. If your app defines custom document formats that are different from the system-supported content types, you should provide a Quick Look generator for those formats. (Generators are plug-ins that convert documents of the appropriate type from their native format to a format that Quick Look can display as thumbnails and previews to users.) Quick Look also allows users of your app to preview the contents of a document that's embedded as a text attachment without requiring them to leave the app.

For information about supporting Quick Look for your custom document types, see *Quick Look Programming Guide* and *Quick Look Framework Reference for Mac*. The related Quick Look UI framework is briefly described in Quick Look UI (page 48).

Social Framework

The Social framework (`Social.framework`) helps you integrate supported social networking services into your app by providing a template for creating HTTP requests and a generalized interface for posting requests on the user's behalf. You can also use the Social framework to retrieve information for integrating a user's social networking accounts into your app.

To learn more about the Social API, see *Social Framework Reference*.

Store Kit

Store Kit (`StoreKit.framework`) enables you to request payment from a user to purchase additional app functionality or content from the Mac App Store.

Store Kit handles the financial aspects of a transaction, processing the payment request through the user's iTunes Store account. Store Kit then provides your app with information about the purchase. Your app handles the other aspects of the transaction, including the presentation of a purchasing interface and the downloading (or unlocking) of the appropriate content. This division of labor gives you control over the user experience. You also decide on the delivery mechanism that works best for your app.

For more information about Store Kit, read *In-App Purchase Programming Guide* and *StoreKit Framework Reference*.

WebKit

The WebKit framework (`WebKit.framework`) enables your app to display HTML content. It has two subframeworks: Web Core and JavaScript Core. Web Core is for parsing and displaying HTML content; JavaScript Core is for parsing and executing JavaScript code.

WebKit also lets you create text views containing editable HTML. With this basic editing support, users can replace text and manipulate the document text and attributes, including CSS properties. WebKit also supports creating and editing content at the DOM level of an HTML document. For example, you could extract the list of links on a page, modify them, and replace them prior to displaying the document in a web view.

For information on how to use WebKit, see *WebKit Objective-C Programming Guide*.

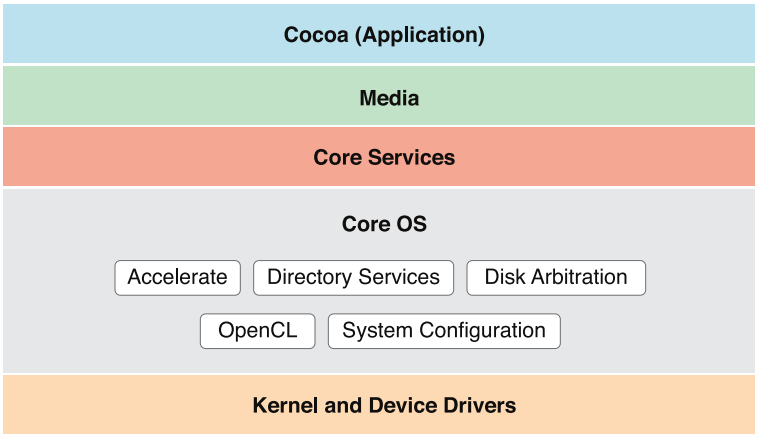
Other Frameworks in the Core Services Layer

The Core Services layer of OS X also includes the following frameworks:

- **Collaboration** (`Collaboration.framework`). The Collaboration framework provides a set of Objective-C classes for displaying sharing account information and other identity-related user interfaces. With this API, apps can display information about users and groups and display a panel for selecting users and groups during the editing of access control lists. For related information, see Identity Services (page 60).
- **Input Method Kit** (`InputMethodKit.framework`). Input Method Kit helps you build input methods for Chinese, Japanese, and other languages. The framework handles tasks such as connecting to clients and running candidate windows. To learn more, see *InputMethodKit Framework Reference*.
- **Latent Semantic Mapping** (`LatentSemanticMapping.framework`). Latent Semantic Mapping supports the classification of text and other token-based content into developer-defined categories, based on semantic information latent in the text. Products of this technology are maps, which you can use to analyze and classify arbitrary text—for example, to determine, if an email message is consistent with the user's interests. For information about the Latent Semantic Mapping framework, see *Latent Semantic Mapping Reference*.

Core OS Layer

The technologies and frameworks in the Core OS layer provide low-level services related to hardware and networks. These services are based on facilities in the Kernel and Device Drivers layer.



High-Level Features

The Core OS layer implements features related to app security.

Gatekeeper

Gatekeeper, allows users to block the installation of software that does not come from the Mac App Store and identified developers. If your app is not signed with a Developer ID certificate issued by Apple, it will not launch on systems that have this security option selected. If you plan to distribute your app outside of the Mac App Store, be sure to test the installation of your app on a Gatekeeper enabled system so that you can provide a good user experience.

Xcode supports most of the tasks that you need to perform to get a Developer ID certificate and code sign your app. To learn how to submit your app to the Mac App Store—or test app installation on a Gatekeeper enabled system—read *Tools Workflow Guide for Mac*.

App Sandbox

App Sandbox provides a last line of defense against stolen, corrupted, or deleted user data if malicious code exploits your app. App Sandbox also minimizes the damage from coding errors. Its strategy is twofold:

- App Sandbox enables you to describe how your app interacts with the system. The system then grants your app only the access it needs to get its job done, and no more.
- App Sandbox allows the user to transparently grant your app additional access by using Open and Save dialogs, drag and drop, and other familiar user interactions.

You describe your app's interaction with the system by setting entitlements in Xcode. For details on all the entitlements available in OS X, see *Entitlement Key Reference*.

When you adopt App Sandbox, you must code sign your app (for more information, see Code Signing (page 69)). This is because entitlements, including the special entitlement that enables App Sandbox, are built into an app's code signature.

For a complete explanation of App Sandbox and how to use it, read *App Sandbox Design Guide*.

Code Signing

OS X employs the security technology known as *code signing* to allow you to certify that your app was indeed created by you. After an app is code signed, the system can detect any change to the app—whether the change is introduced accidentally or by malicious code. Various security technologies, including App Sandbox and parental controls, depend on code signing.

In most cases, you can rely on Xcode automatic code signing, which requires only that you specify a code signing identity in the build settings for your project. The steps to take are described in Code Signing Your App in *Tools Workflow Guide for Mac*. If you need to incorporate code signing into an automated build system or if you link your app against third-party frameworks, refer to the procedures described in *Code Signing Guide*.

For a complete explanation of code signing in the context of App Sandbox, read App Sandbox in Depth in *App Sandbox Design Guide*.

Core OS Frameworks

The following technologies and frameworks are in the Core OS layer of OS X:

Accelerate

The Accelerate framework (`Accelerate.framework`) contains APIs that help you accelerate complex operations—and potentially improve performance—by using the available vector unit. Hardware-based vector units boost the performance of any app that exploits data parallelism, such as those that perform 3D graphic imaging, image processing, video processing, audio compression, and software-based cell telephony. (Because Quartz and QuickTime Kit incorporate vector capabilities, any app that uses these APIs can tap into this hardware acceleration without making any changes.)

The Accelerate framework is an umbrella framework that wraps the vecLib and vImage frameworks into a single package. The vecLib framework contains vector-optimized routines for doing digital signal processing, linear algebra, and other computationally expensive mathematical operations. The vImage framework supports the visual realm, adding routines for morphing, alpha-channel processing, and other image-buffer manipulations.

For information on how to use the components of the Accelerate framework, see *vImage Programming Guide*, *vImage Reference Collection*, and *vecLib Reference*. For general performance-related information, see *Performance Overview*.

Disk Arbitration

The Disk Arbitration framework (`DiskArbitration.framework`) notifies your app when local and remote volumes are mounted and unmounted. It also furnishes other updates on the status of remote and local mounts and returns information about mounted volumes. For example, if you provide the framework with the BSD disk identifier of a volume, you can get the volume's mount-point path.

For more information on Disk Arbitration, see *Disk Arbitration Framework Reference*.

OpenCL

The Open Computing Language (OpenCL) makes the high-performance parallel processing power of GPUs available for general-purpose computing. The OpenCL language is a general purpose computer language, not specifically a graphics language, that abstracts out the lower-level details needed to perform parallel data computation tasks on GPUs and CPUs. Using OpenCL, you create compute kernels that are then offloaded to a graphics card or CPU for processing. Multiple instances of a compute kernel can be run in parallel on one or more GPU or CPU cores, and you can link to your compute kernels from Cocoa, C, or C++ apps.

For tasks that involve data-parallel processing on large data sets, OpenCL can yield significant performance gains. There are many apps that are ideal for acceleration using OpenCL, such as signal processing, image manipulation, and finite element modeling. The OpenCL language has a rich vocabulary of vector and scalar operators and the ability to operate on multidimensional arrays in parallel.

For information about OpenCL and how to write compute kernels, see *OpenCL Programming Guide for Mac*.

Open Directory (Directory Services)

Open Directory is a directory services architecture that provides a centralized way to retrieve information stored in local or network databases. Directory services typically provide access to collected information about users, groups, computers, printers, and other information that exists in a networked environment (although they can also store information about the local system). You use Open Directory to retrieve information from these local or network databases. For example, if you're writing an email app, you can use Open Directory to connect to a corporate LDAP server and retrieve the list of individual and group email addresses for the company.

Open Directory uses a plug-in architecture to support a variety of retrieval protocols. OS X provides plug-ins to support LDAPv2, LDAPv3, NetInfo, AppleTalk, SLP, SMB, DNS, Microsoft Active Directory, and Bonjour protocols, among others. You can also write your own plug-ins to support additional protocols.

The Directory Services framework (`DirectoryServices.framework`) publishes a programmatic interface for accessing Open Directory services.

For more information on this technology, see *Open Directory Programming Guide*. For information on how to write Open Directory plug-ins, see *Open Directory Plug-in Programming Guide*.

System Configuration

System Configuration (`SystemConfiguration.framework`) is a framework that helps apps configure networks and determine if networks can be reached prior to connecting with them. The framework includes calls for a user experience when interacting with a captive network. (A captive network, such as a public Wi-Fi hotspot, requires user interaction before providing Internet access.)

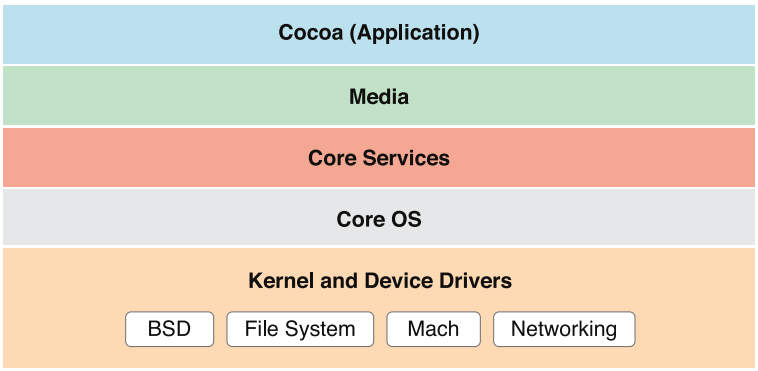
Use System Configuration APIs to determine and set configuration settings and respond dynamically to changes in that information. You can also use these APIs to help you determine whether a remote host is reachable and, if it is, to request a network connection so it can provide content to its users. To assist in this, System Configuration does the following:

- It provides access to current network configuration information.
- It allows apps to determine the reachability of remote hosts and start PPP-based connections.
- It notifies apps when there are changes in network status and network configuration.
- It provides a flexible schema for defining and accessing stored preferences and the current network configuration.

To learn more about System Configuration, see *System Configuration Programming Guidelines*.

Kernel and Device Drivers Layer

The lowest layer of OS X includes the kernel, drivers, and BSD portions of the system and is based primarily on open source technologies. OS X extends this low-level environment with several core infrastructure technologies that make it easier for you to develop software.



High-Level Features

The following sections describe features in the Kernel and Device Drivers layer of OS X.

XPC Interprocess Communication and Services

XPC is an OS X interprocess communication technology that complements App Sandbox by enabling privilege separation. Privilege separation, in turn, is a development strategy in which you divide an app into pieces according to the system resource access that each piece needs. The component pieces that you create are called *XPC services*.

You create an XPC service as an individual target in your Xcode project. Each service gets its own sandbox—specifically, it gets its own container and its own set of entitlements. In addition, an XPC service that you include with your app is accessible only by your app. These advantages add up to making XPC the best technology for implementing privilege separation in an OS X app.

XPC is integrated with Grand Central Dispatch (GCD). When you create a connection, you associate it with a dispatch queue on which message traffic executes.

When the app is launched, the system automatically registers each XPC service it finds into the namespace visible to the app. An app establishes a connection with one of its XPC services and sends it messages containing events that the service then handles.

For more on XPC Services, read Creating XPC Services in *Daemons and Services Programming Guide*. To learn more about App Sandbox, read *App Sandbox Design Guide*.

Caching API

The `libcache` API is a low-level purgeable caching API. Aggressive caching is an important technique in maximizing app performance. However, when caching demands exceed available memory, the system must free up memory as necessary to handle new demands. Typically, this means paging cached data to and from relatively slow storage devices, sometimes even resulting in systemwide performance degradation. Your app should avoid potential paging overhead by actively managing its data caches, releasing them as soon as it no longer needs the cached data.

In the wider system context, your app can also help by creating caches that the operating system can simply purge on a priority basis as memory pressure necessitates. The `libcache` library and Foundation framework's `NSCache` class help you to create these purgeable caches.

For more information about the functions of the `libcache` library, see *libcache Reference*. For more information about the `NSCache` class, see *NSCache Class Reference*.

In-Kernel Video Capture

I/O Video provides a kernel-level C++ programming interface for writing video capture device drivers. I/O Video replaces the QuickTime sequence grabber API as a means of getting video into OS X.

I/O Video consists of the `IOVideoDevice` class on the kernel side (along with various related minor classes) that your driver should subclass, and a user space device interface for communicating with the driver.

For more information, see the `IOVideoDevice.h` header file in the Kernel framework.

The Kernel

Beneath the appealing, easy-to-use interface of OS X is a rock-solid, UNIX-based foundation that is engineered for stability, reliability, and performance. The kernel environment is built on top of Mach 3.0 and provides high-performance networking facilities and support for multiple, integrated file systems.

The following sections describe some of the key features of the kernel and driver portions of Darwin.

Mach

Mach is at the heart of Darwin because it provides some of the most critical functions of the operating system. Much of what Mach provides is transparent to apps. It manages processor resources such as CPU usage and memory, handles scheduling, enforces memory protection, and implements a messaging-centered infrastructure for untyped interprocess communication, both local and remote. Mach provides the following important advantages to Mac computing:

- **Protected memory.** The stability of an operating system should not depend on all executing apps being good citizens. Even a well-behaved process can accidentally write data into the address space of the system or another process, which can result in the loss or corruption of data or even precipitate system crashes. Mach ensures that an app cannot write in another app's memory or in the operating system's memory. By walling off apps from each other and from system processes, Mach makes it virtually impossible for a single poorly behaved app to damage the rest of the system. Best of all, if an app crashes as the result of its own misbehavior, the crash affects only that app and not the rest of the system.
- **Preemptive multitasking.** With Mach, processes share the CPU efficiently. Mach watches over the computer's processor, prioritizing tasks, making sure activity levels are at the maximum, and ensuring that every task gets the resources it needs. It uses certain criteria to decide how important a task is and therefore how much time to allocate to it before giving another task its turn. Your process is not dependent on another process yielding its processing time.
- **Advanced virtual memory.** In OS X, virtual memory is "on" all the time. The Mach virtual memory system gives each process its own private virtual address space. For 64-bit apps, the theoretical maximum is approximately 18 exabytes, or 18 billion billion bytes. Mach maintains address maps that control the translation of a task's virtual addresses into physical memory. Typically only a portion of the data or code contained in a task's virtual address space resides in physical memory at any given time. As pages are needed, they are loaded into physical memory from storage. Mach augments these semantics with the abstraction of memory objects. Named memory objects enable one task (at a sufficiently low level) to map a range of memory, unmap it, and send it to another task. This capability is essential for implementing separate execution environments on the same system.
- **Real-time support.** This feature guarantees low-latency access to processor resources for time-sensitive media apps.

Mach also enables cooperative multitasking, preemptive threading, and cooperative threading.

64-Bit Kernel

As of v10.8, OS X requires a Mac that uses the 64-bit kernel. A 64-bit kernel provides several benefits:

- The kernel can support large memory configurations more efficiently.
- The maximum size of the buffer cache is increased, potentially improving I/O performance.

- Performance is improved when working with specialized networking hardware that emulates memory mapping across a wire or with multiple video cards containing over 2 GB of video RAM.

Because a 64-bit kernel does not support 32-bit drivers and kernel extensions (KEXTs), those items must be built for 64-bit. Fortunately, for most drivers and KEXTs, building for a 64-bit kernel is usually not as difficult as you might think. For the most part, transitioning a driver or KEXT to be 64-bit capable is just like transitioning any other piece of code. For details about how to make the transition, including what things to check for in your code, see *64-Bit Transition Guide*.

Device-Driver Support

Darwin offers an object-oriented framework for developing device drivers called the *I/O Kit framework*. This framework facilitates the creation of drivers for OS X and provides much of the infrastructure that they need. Written in a restricted subset of C++ and designed to support a range of device families, the I/O Kit is both modular and extensible.

Device drivers created with the I/O Kit acquire several important features:

- True plug and play
- Dynamic device management (“hot plugging”)
- Power management (for both desktops and portables)

If your device conforms to standard specifications—such as those for mice, keyboards, audio input devices, modern MIDI devices, and so on—it should just work when you plug it in. If your device doesn’t conform to a published standard, you can use the I/O Kit resources to create a custom driver to meet your needs. Devices such as AGP cards, PCI and PCIe cards, scanners, and printers usually require custom drivers or other support software in order to work with OS X.

For information on creating device drivers, see *IOKit Device Driver Design Guidelines*.

Network Kernel Extensions

Darwin allows kernel developers to add networking capabilities to the operating system by creating network kernel extensions (NKEs). The NKE facility allows you to create networking modules and even entire protocol stacks that can be dynamically loaded into the kernel and unloaded from it. NKEs also make it possible to configure protocol stacks automatically.

NKE modules have built-in capabilities for monitoring and modifying network traffic. At the data-link and network layers, they can also receive notifications of asynchronous events from device drivers, such as when there is a change in the status of a network interface.

For information on how to write an NKE, see *Network Kernel Extensions Programming Guide*.

BSD

Integrated with Darwin is a customized version of the Berkeley Software Distribution (BSD) operating system. Darwin's implementation of BSD includes much of the POSIX API, which higher-level apps can also use to implement basic app features. BSD serves as the basis for the file systems and networking facilities of OS X. In addition, it provides several programming interfaces and services, including:

- The process model (process IDs, signals, and so on)
- Basic security policies such as file permissions and user and group IDs
- Threading support (POSIX threads)
- Networking support (BSD sockets)

> **Note:** For more information about the FreeBSD operating system, go to The FreeBSD Project website. For more information about the boot process of OS X, including how it launches the daemons used to implement key BSD services, see *Daemons and Services Programming Guide*.

The following sections describe some of the key features of the BSD layer of OS X.

IPC and Notification Mechanisms

OS X supports the following technologies for interprocess communication (IPC) and for delivering notifications across the system:

- **File System Events.** File System Events (`FSEvents`) is a mechanism for notifying your app when changes occur in the file system, such as at the creation, modification, or removal of files and directories. The `FSEvents` API gives you a way to monitor many directories at once and detect general changes to a file hierarchy. For example, you might use this technology in backup software to detect what files changed. The `FSEvents` API is not intended for detecting fine-grained changes to individual files.

  To learn how to use the `FSEvents` API, see *File System Events Programming Guide*.

- **Kernel queues and kernel events.** These mechanisms allow you to intercept kernel-level events to receive notifications about changes to sockets, processes, the file system, and other aspects of the system. Kernel queues and events are part of the FreeBSD layer of the operating system and are described in the `kqueue` and `kevent` man pages.

- **BSD notifications.** BSD notifications have a few advantages over the notification services that are offered by Core Foundation and Foundation. For example, your program can receive BSD notifications through mechanisms that include Mach ports, signals, and file descriptors. Moreover, this technology is lightweight, efficient, and capable of coalescing notifications.

You can add support for BSD notifications to any type of program, including Cocoa apps. For more information, see *Mac Notification Overview* or the `notify` man page. For a discussion of Cocoa and Core Foundation interfaces for interprocess notification, see Distributed Notifications (page 61).

- **Sockets and ports.** Sockets and ports are portable mechanisms for interprocess communication. A socket represents one end of a communications channel between two processes either locally or across the network. A port is a channel between processes or threads on the local computer. Core Foundation and Foundation provide higher-level abstractions for ports and sockets that make them easier to implement and offer additional features. For example, you can use a `CFSocket` with a `CFRunLoop` to multiplex data received from a socket with data received from other sources (or more information, see *CFSocket Reference* and *CFRunLoop Reference*).
- **Streams**. A stream is mechanism for transferring data between processes when you are communicating using an established transport mechanism such as Bonjour or HTTP. Higher-level interfaces of Core Foundation and Foundation (which work with `CFNetwork`) provide a stream-based way to read and write network data and can be used with run loops to operate efficiently in a network environment.
- **Pipes**. A pipe is a communications channel typically created between a parent and a child process when the child process is forked. Data written to a pipe is buffered and read in first-in, first-out (FIFO) order. You can create named pipes (`pipe` function) for communication between related processes or named pipes for communication between any two processes. A named pipe must be created with a unique name known to both the sending and the receiving process.
- **Shared memory**. Shared memory is a region of memory that has been allocated by a process specifically for the purpose of being readable and possibly writable among several processes. You can create regions of shared memory using several BSD approaches, including the `shm_open` and `shm_unlink` routines, and the `mmap` routine. Access to shared memory is controlled through POSIX semaphores, which implement a kind of locking mechanism.

  Although shared memory permits any process with the appropriate permissions to read or write a shared memory region directly, it is very fragile—leading to the dangers of data corruption and security breaches—and should be used with care. It is best used only as a repository for raw data (such as pixels or audio), with the controlling data structures accessed through more conventional interprocess communication.

  For information about `shm_open`, `shm_unlink`, and `mmap`, see the `shm_open`, `shm_unlink`, and `mmap` man pages.
- **Apple event**. An Apple event is a high-level semantic event that an app can send to itself, to other apps on the same computer, or to apps on a remote computer. Apps can use Apple events to request services and information from other apps. To supply services, you define objects in your app that can be accessed using Apple events and then provide Apple event handlers to respond to requests for those objects.

  Although Apple events are not a BSD technology, they are a low-level alternative for interprocess communication. To learn how to use Apple events, see *Apple Events Programming Guide*.

Note: Mach ports are another technology for transferring messages between processes. However, messaging with Mach port objects is the least desirable way to communicate between processes. Mach port messaging relies on knowledge of the kernel interfaces, which might change in a future version of OS X. The only time you might consider using Mach ports directly is if you are writing software that runs in the kernel.

Network Support

OS X is one of the premier platforms for computing in an interconnected world. It supports the dominant media types, protocols, and services in the industry, as well as differentiated and innovative services from Apple.

The OS X network protocol stack is based on BSD. The extensible architecture provided by network kernel extensions, summarized in Network Kernel Extensions (page 75), facilitates the creation of modules implementing new or existing protocols that can be added to this stack.

Standard Network Protocols

OS X provides built-in support for a large number of network protocols that are standard in the computing industry. Table 6-1 summarizes these protocols.

Table 6-1 Network protocols

Protocol	Description
802.1x	802.1x is a protocol for implementing port-based network access over wired or wireless LANs. It supports a wide range of authentication methods, including TLS, TTLS, LEAP, MDS, and PEAP (MSCHAPv2, MD5, GTC).
DHCP and BOOTP	The Dynamic Host Configuration Protocol and the Bootstrap Protocol automate the assignment of IP addresses in a particular network.
DNS	Domain Name Services is the standard Internet service for mapping host names to IP addresses.
FTP and SFTP	The File Transfer Protocol and Secure File Transfer Protocol are two standard means of moving files between computers on TCP/IP networks.
HTTP and HTTPS	The Hypertext Transport Protocol is the standard protocol for transferring webpages between a web server and browser. OS X provides support for both the insecure and secure versions of the protocol.

Protocol	Description
LDAP	The Lightweight Directory Access Protocol lets users locate groups, individuals, and resources such as files and devices in a network, whether on the Internet or on a corporate intranet.
NBP	The Name Binding Protocol is used to bind processes across a network.
NTP	The Network Time Protocol is used for synchronizing client clocks.
PAP	The Printer Access Protocol is used for spooling print jobs and printing to network printers.
PPP	For dial-up (modem) access, OS X includes PPP (Point-to-Point Protocol). PPP support includes TCP/IP as well as the PAP and CHAP authentication protocols.
PPPoE	The Point-to-Point Protocol over Ethernet protocol provides an Ethernet-based dial-up connection for broadband users.
S/MIME	The Secure/Multipurpose Internet Mail Extensions protocol supports encryption of email and the attachment of digital signatures to validate email addresses.
SLP	Service Location Protocol is designed for the automatic discovery of resources (servers, fax machines, and so on) on an IP network.
SOAP	The Simple Object Access Protocol is a lightweight protocol for exchanging encapsulated messages over the web or other networks.
SSH	The Secure Shell protocol is a safe way to perform a remote login to another computer. Session information is encrypted to prevent unauthorized access of data.
TCP/IP and UDP/IP	OS X provides two transmission-layer protocols, TCP (Transmission Control Protocol) and UDP (User Datagram Protocol), to work with the network-layer Internet Protocol (IP). (OS X includes support for IPv6 and IPSec.)
XML-RPC	XML-RPC is a protocol for sending remote procedure calls using XML over the web.

OS X also implements a number of file-sharing protocols; see Table 6-4 (page 82) for a summary of these protocols.

Network Technologies

OS X supports the network technologies listed in Table 6-2.

Table 6-2 Network technology support

Technology	Description
Ethernet 10/100Base-T	For the Ethernet ports built into every new Macintosh.
Ethernet 1000Base-T	Also known as Gigabit Ethernet. For data transmission over fiber-optic cable and standardized copper wiring.
Jumbo Frame	This Ethernet format uses 9 KB frames for interserver links rather than the standard 1.5 KB frame. Jumbo Frame decreases network overhead and increases the flow of server-to-server and server-to-app data.
Serial	Supports modem and ISDN capabilities.
Wireless	Supports the 802.11b, 802.11g, and 802.11n wireless network technologies using AirPort and AirPort Extreme.
IP Routing/RIP	IP routing provides routing services for small networks. It uses Routing Information Protocol (RIP) in its implementation.
Multihoming	Enables a computer host to be physically connected to multiple data links that can be on the same or different networks.
IP aliasing	Allows a network administrator to assign multiple IP addresses to a single network interface.
Zero-configuration networking	See Bonjour (page 57).
NetBoot	Allows computers to share a single System folder, which is installed on a centralized server that the system administrator controls. Users store their data in home directories on the server and have access to a common Applications folder, both of which are also commonly installed on the server.
Personal web sharing	Allows users to share information with other users on an intranet, no matter what type of computer or browser they are using. The Apache web server is integrated as the system's HTTP service.

Network Diagnostics

Network diagnostics is a way of helping the user solve network problems. Although modern networks are generally reliable, there are still times when network services may fail. Sometimes the cause of the failure is beyond the ability of the desktop user to fix, but sometimes the problem is in the way the user's computer is configured. The network diagnostics feature provides a diagnostic app to help the user locate problems and correct them.

If your app encounters a network error, you can use the diagnostic interfaces of `CFNetwork` to launch the diagnostic app and attempt to solve the problem interactively. You can also choose to report diagnostic problems to the user without attempting to solve them.

For more information on using this feature, see Using Network Diagnostics.

File-System Support

The file-system component of Darwin is based on extensions to BSD and an enhanced Virtual File System (VFS) design. The file-system component includes the following features:

- Permissions on removable media. This feature is based on a globally unique ID registered for each connected removable device (including USB and FireWire devices) in the system.
- Access control lists, which support fine-grained access to file-system objects.
- URL-based volume mounts, which enable users (via a Finder command) to mount such things as AppleShare and web servers
- Unified buffer cache, which consolidates the buffer cache with the virtual-memory cache
- Long filenames (255 characters or 755 bytes, based on UTF-8)
- Support for hiding filename extensions on a per-file basis
- Journaling of all file-system types to aid in data recovery after a crash

Because of its multiple app environments and the various kinds of devices it supports, OS X handles file data in many standard volume formats. Table 6-3 lists the supported formats.

Table 6-3 Supported local volume formats

Volume format	Description
Mac OS Extended Format	Also called HFS (hierarchical file system) Plus, or HFS+. This is the default root and booting volume format in OS X. This extended version of HFS optimizes the storage capacity of large hard disks by decreasing the minimum size of a single file.
Mac OS Standard Format	Also called hierarchical file system, or HFS. This is the legacy volume format in Mac OS systems prior to Mac OS 8.1. HFS (like HFS+) stores resources and data in separate forks of a file and makes use of various file attributes, including type and creator codes.

Volume format	Description
UDF	Universal Disk Format, used for hard drives and optical disks, including most types of CDs and DVDs. OS X supports reading UDF revisions 1.02 through 2.60 on both block devices and most optical media, and it supports writing to block devices and to DVD-RW and DVD+RW media using UDF 2.00 through 2.50 (except for mirrored metadata partitions in 2.50). You can find the UDF specification at http://www.osta.org.
ISO 9660	The standard format for CD-ROM volumes.
NTFS	The NT File System, used by Windows computers. OS X can read NTFS-formatted volumes but cannot write to them.
UFS	UNIX File System, a flat (that is, single-fork) disk volume format, based on the BSD FFS (Fast File System), that is similar to the standard volume format of most UNIX operating systems; it supports POSIX file-system semantics, which are important for many server applications. Although UFS is supported in OS X, its use is discouraged.
MS-DOS (FAT)	The FAT file system is used by many Windows computers, digital cameras, video cameras, SD and SDHC memory cards, and other digital devices. OS X can read and write FAT-formatted volumes.
ExFAT	The ExFAT file system is an extension of the FAT file system, and is also used on Windows computers, some digital cameras and video cameras, SDXC memory cards, and other digital devices. OS X can read and write ExFAT-formatted volumes.

HFS+ volumes support aliases, symbolic links, and hard links, whereas UFS volumes support symbolic links and hard links but not aliases. Although an alias and a symbolic link are both lightweight references to a file or directory elsewhere in the file system, they are semantically different in significant ways. For more information, see Aliases and Symbolic Links in *File System Overview*.

Note: OS X does not support stacking in its file-system design.

Because OS X is intended to be deployed in heterogeneous networks, it also supports several network file-sharing protocols. Table 6-4 lists these protocols.

Table 6-4 Supported network file-sharing protocols

File protocol	Description
AFP	Apple Filing Protocol, the principal file-sharing protocol in Mac OS 9 systems (available only over TCP/IP transport).

File protocol	Description
NFS	Network File System, the dominant file-sharing protocol in the UNIX world.
WebDAV	Web-based Distributed Authoring and Versioning, an HTTP extension that allows collaborative file management on the web.
SMB/CIFS	SMB/CIFS, a file-sharing protocol used on Windows and UNIX systems.

Security

The roots of OS X in the UNIX operating system provide a robust and secure computing environment whose track record extends back many decades. OS X security services are built on top of BSD (Berkeley Software Distribution), an open-source standard. BSD is a form of the UNIX operating system that provides basic security for fundamental services, such as file and network access.

The CommonCrypto library, which is part of `libSystem`, provides raw cryptographic algorithms. It is intended to replace similar OpenSSL interfaces.

> **Note:** CDSA (Common Data Security Architecture) and OpenSSL are deprecated and their further use is discouraged. Consider using Security Transforms technology to replace CDSA and CommonCrypto to replace OpenSSL. Security Transforms, which are part of the Security framework, are described in Security Services (page 58).

OS X also includes the following security features:

- Adoption of Mandatory Access Control, which provides a fine-grained security architecture for controlling the execution of processes at the kernel level. This feature enables the “sandboxing” of apps, which lets you limit the access of a given app to only those features you designate.
- Support for code signing and installer package signing. This feature lets the system validate apps using a digital signature and warn the user if an app is tampered with.
- Compiler support for fortifying your source code against potential security threats. This support includes options to disallow the execution of code located on the stack or other portions of memory containing data.
- Support for putting unknown files into quarantine. This is especially useful for developers of web browsers or other network-based apps that receive files from unknown sources. The system prevents access to quarantined files unless the user explicitly approves that access.

For an introduction to OS X security features, see *Security Overview*.

Scripting Support

Darwin includes all of the scripting languages commonly found in UNIX-based operating systems. In addition to the scripting languages associated with command-line shells (such as `bash` and `csh`), Darwin also includes support for Perl, Python, Ruby, Ruby on Rails, and others.

OS X provides scripting bridges to the Objective-C classes of Cocoa. These bridges let you use Cocoa classes from within your Python and Ruby scripts. For information about using these bridges, see *Ruby and Python Programming Topics for Mac*.

Threading Support

OS X provides full support for creating multiple preemptive threads of execution inside a single process. Threads let your program perform multiple tasks in parallel. For example, you might create a thread to perform some lengthy calculations in the background while a separate thread responds to user events and updates the windows in your app. Using multiple threads can often lead to significant performance improvements in your app, especially on computers with multiple CPU cores. Multithreaded programming is not without its dangers though. It requires careful coordination to ensure your app's state does not get corrupted.

All user-level threads in OS X are based on POSIX threads (also known as pthreads). A pthread is a lightweight wrapper around a Mach thread, which is the kernel implementation of a thread. You can use the pthreads API directly or use any of the threading packages offered by Cocoa. Although each threading package offers a different combination of flexibility versus ease-of-use, all packages offer roughly the same performance.

In general, you should try to use Grand Central Dispatch or operation objects to perform work concurrently. However, there might still be situations where you need to create threads explicitly. For more information about threading support and guidelines on how to use threads safely, see *Threading Programming Guide*.

X11

The X11 windowing system is provided as an optional installation component for the system. This windowing system is used by many UNIX applications to draw windows, controls, and other elements of graphical user interfaces. The OS X implementation of X11 uses the Quartz drawing environment to give X11 windows a native OS X feel. This integration also makes it possible to display X11 windows alongside windows from native apps written in Cocoa.

Software Development Support

The following sections describe some additional features of OS X that affect the software development process.

Binary File Architecture

The underlying architecture of OS X executables was built from the beginning with flexibility in mind. This flexibility became important as Mac computers have transitioned from using PowerPC to Intel CPUs and from supporting only 32-bit apps to 64-bit apps. The following sections provide an overview of the types of architectures you can support in your OS X executables along with other information about the runtime and debugging environments available to you.

Hardware Architectures

When OS X was first introduced, it was built to support a 32-bit PowerPC hardware architecture. With Apple's transition to Intel-based Mac computers, OS X added initial support for 32-bit Intel hardware architectures. In addition to 32-bit support, OS X v10.4 added some basic support for 64-bit architectures as well and this support was expanded in OS X v10.5. This means that apps and libraries can now support two different architectures:

- 32-bit Intel (`i386`)
- 64-bit Intel (`x86_64`)

Although apps can support all of these architectures in a single binary, doing so is not required. The ability to create "universal binaries" that run natively on all supported architectures gives OS X the flexibility it needs for the future.

Supporting multiple architectures requires careful planning and testing of your code for each architecture. There are subtle differences from one architecture to the next that can cause problems if not accounted for in your code. For example, some built-in data types have different sizes in 32-bit and 64-bit architectures. Accounting for these differences is not difficult but requires consideration to avoid coding errors.

Xcode provides integral support for creating apps that support multiple hardware architectures. For information about tools support and creating universal binaries. For information about 64-bit support in OS X, including links to documentation for how to make the transition, see 64-Bit Support (page 85).

64-Bit Support

OS X was initially designed to support binary files on computers using a 32-bit architecture. In OS X v10.4, however, support was introduced for compiling, linking, and debugging binaries on a 64-bit architecture. This initial support was limited to code written using C or C++ only. In addition, 64-bit binaries could link against the Accelerate framework and `libSystem.dylib` only.

Starting in OS X v10.5, most system libraries and frameworks are 64-bit ready, meaning they can be used in both 32-bit and 64-bit apps. Frameworks built for 64-bit means you can create apps that address extremely large data sets, up to 128 TB on the current Intel-based CPUs. On Intel-based Macintosh computers, some 64-bit apps may even run faster than their 32-bit equivalents because of the availability of extra processor resources in 64-bit mode.

There are a few technologies that have not been ported to 64-bit. Development of 32-bit apps with these APIs is still supported, but if you want to create a 64-bit app, you must use alternative technologies. Among these APIs are the following:

- The entire QuickTime C API (not deprecated, but developers should use QuickTime Kit instead in 64-bit apps)
- HIToolbox, Window Manager, and most other user interface APIs (not deprecated, but developers should use Cocoa UI classes and other alternatives); see *64-Bit Guide for Carbon Developers* for the list of specific APIs and transition paths.

OS X uses the LP64 model that is in use by other 64-bit UNIX systems, which means fewer headaches when porting from other operating systems. For general information on the LP64 model and how to write 64-bit apps, see *64-Bit Transition Guide*. For Cocoa-specific transition information, see *64-Bit Transition Guide for Cocoa*.

Object File Formats

OS X is capable of loading object files that use several different object-file formats. Mach-O format is the format used for all native OS X app development.

For information about the Mach-O file format, see *OS X ABI Mach-O File Format Reference*. For additional information about using Mach-O files, see *Mach-O Programming Topics*.

Debug File Formats

Whenever you debug an executable file, the debugger uses symbol information generated by the compiler to associate user-readable names with the procedure and data address it finds in memory. Normally, this user-readable information is not needed by a running program and is stripped out (or never generated) by the compiler to save space in the resulting binary file. For debugging, however, this information is very important to be able to understand what the program is doing.

OS X supports two different debug file formats for compiled executables: Stabs and DWARF. The Stabs format is present in all versions of OS X and until the introduction of Xcode 2.4 was the default debugging format. Code compiled with Xcode 2.4 and later uses the DWARF debugging format by default. When using the Stabs format, debugging symbols, like other symbols are stored in the symbol table of the executable; see *OS X ABI Mach-O File Format Reference*. With the DWARF format, debugging symbols are stored either in a specialized segment of the executable or in a separate debug-information file.

For information about the DWARF standard, go to The DWARF Debugging Standard; for information about the Stabs debug file format, see *STABS Debug Format*. For additional information about Mach-O files and their stored symbols, see *Mach-O Programming Topics*.

Runtime Environments

Since its first release, OS X has supported several different environments for running apps. The most prominent of these environments is the dynamic link editor (`dyld`) environment, which is also the only environment supported for active development. Most of the other environments provided legacy support during the transition from Mac OS 9 to OS X and are no longer supported for active development. The following sections describe the runtime environments you may encounter in various versions of OS X.

dyld Runtime Environment

The `dyld` runtime environment is the native environment in OS X and is used to load, link, and execute Mach-O files. At the heart of this environment is the `dyld` dynamic loader program, which handles the loading of a program's code modules and associated dynamic libraries, resolves any dependencies between those libraries and modules, and begins the execution of the program.

Upon loading a program's code modules, the dynamic loader performs the minimal amount of symbol binding needed to launch your program and get it running. This binding process involves resolving links to external libraries and loading them as their symbols are used. The dynamic loader takes a lazy approach to binding individual symbols, doing so only as they are used by your code. Symbols in your code can be strongly linked or weakly linked. Strongly linked symbols cause the dynamic loader to terminate your program if the library containing the symbol cannot be found or the symbol is not present in the library. Weakly linked symbols terminate your program only if the symbol is not present and an attempt is made to use it.

For more information about the dynamic loader program, see the `dyld` man page. For information about building and working with Mach-O executable files, see *Mach-O Programming Topics*.

Language Support

The tools that come with OS X provide direct support for developing software using the C, C++, Objective-C, Objective-C++, and Swift languages along with numerous scripting languages. Support for other languages may also be provided by third-party developers. For more information on the key features of Swift and Objective-C, see Development Languages (page 14)

Migrating from Cocoa Touch

SwiftObjective-C

If you've developed an iOS app, many of the frameworks available in OS X should already seem familiar to you. The basic technology stack in iOS and OSX are identical in many respects. But, despite the similarities, not all of the frameworks in OS X are exactly the same as their iOS counterparts. This chapter describes the differences you may encounter as you create Mac apps and explains how you can adjust your code to handle some of the more significant differences.

General Migration Notes

In OS X, apps typically have a screen that is larger, and resources that are greater, than in iOS. As a developer, you have more frameworks at your disposal in OS X development and (generally) more programmatic possibilities. This greater range of possibilities may be a pleasant prospect, but it also requires different ways of thinking about user expectations and app design.

As you work on migrating your app, pay attention to the idioms and metaphors that are different on each platform. For example, you would not base your OS X app on a stack of view controllers or include a feature that requires a gyroscope. It's important that each version of your app looks and behaves as if it's been designed for the platform it's running on.

If your Cocoa Touch app is already factored according to the Model-View-Controller design pattern, it should be relatively easy to migrate key portions of your app to OS X.

Migrating the Data Model

Cocoa Touch apps whose data model is based on classes in the Foundation and Core Foundation frameworks can be brought over to OS X with little or no modification. OS X includes both frameworks, and they are virtually identical to their iOS counterparts. Most of the differences that do exist are relatively minor or are related to features that are not present in iOS. For example, iOS apps do not support AppleScript. For a detailed list of differences, see Foundation Framework Differences (page 100).

If your Cocoa Touch app is built on top of Core Data, you can easily migrate that data model to an OS X app. The Core Data framework in OS X supports binary and SQLite data stores (as it does in iOS), and it also supports XML data stores. For the supported data stores, you can copy your Core Data resource files to your OS X app project and use them as is. For information on how to use Core Data in your app, see *Core Data Programming Guide*.

Because OS X apps can display much more data on the screen than iOS apps can, you can expand your data model as part of your migration, as long as it doesn't degrade the user experience. In addition to having access to more display space, an OS X app also has access to more resources, including memory. Although you can work with larger data sets, be sure that your algorithms scale to the new platform so that you don't introduce inefficiencies when you migrate your app.

Migrating the User Interface

For various reasons, the structure and implementation of the user interface (UI) in an OS X app is very different from that in an iOS app. Adapting your app to these differences is the main work of migration. As you bring your iOS app to OS X, keep the following themes in mind.

- **Device differences impact the user experience.** In contrast to an iOS app, an OS X app has access to a larger screen, a more dependable supply of power, and a much larger pool of memory. These differences affect how an app presents information to users and how users interact with the app. For example, there is generally only one window per app in iOS, and that window has a fixed size, plays a limited role in the UI, and cannot be manipulated by users. On the other hand, an OS X app often has multiple windows. These windows can contain separate documents, or they might display auxiliary information, such as tool palettes or app preferences. Users can move, resize, close, minimize, and perform other operations with the windows in an OS X app.
- **Users interact with apps differently on each platform.** Because people use touch gestures to interact with iOS apps, the onscreen UI objects must be large enough to manipulate with a human finger. In OS X, people use a mouse, trackpad, or other input device to move the onscreen pointer and interact with UI objects. Because the pointer is much more precise than a finger, the layout of an OS X app tends to be very different from the layout of an iOS app.
- **Users are not always aware of the file system.** iOS users have no direct access to the file system; instead, an iOS app reads and writes files to prescribed locations in its sandbox. In OS X, users can access the file system using the Finder interface. Although not all OS X apps expose the file system to users, those that do must also be prepared to handle issues related to document format, file encoding, and so on. Some OS X apps—for example, iPhoto and iTunes—keep their databases in an app-specific location and provide users with ways to manage their content without interacting with the file system.

When you create a new OS X app project in Xcode, you don't get the same app templates to choose from that you do when you start an iOS app project. (Although OS X defines a few app types—described in Apps (page 13)—these types are not directly related to the project templates.) The Cocoa app template is the typical choice for developing a standard OS X app.

For comprehensive information about the UI design principles of OS X, see *OS X Human Interface Guidelines*. For programmatic information about the windows and views you use to build your interface, and the underlying architecture on which they are based, see *Mac App Programming Guide*.

Migration Strategies

When migrating your app from iOS to OS X, there are several approaches you can use to minimize the amount of code refactoring necessary in your view and controller classes. Each approach has its advantages and disadvantages, and is thus best used in specific instances, as described in Table 7-1.

Table 7-1 Comparison of migration strategies

Situation	Migration approach	Advantages	Disadvantages
When you have heavily platform-dependent code	Mirror your code across both platforms, customizing as necessary.	Offers flexibility	Code duplication, resulting in greater maintenance and testing costs
When a common, lower-level framework provides necessary functionality	Use the common framework. For example, drop down from `UIKit` and `AppKit` to a lower-level, shared framework, to maximize code re-use and and minimize maintenance.	Maximizes code reuse, minimizes maintenance	Significant refactoring of your existing code, and less functionality provided by the lower-level framework
When dealing with an underlying API that's significantly different between the two platforms (option 1)	Use an adapter pattern, a design pattern that allows the interface of an existing class to be accessed from another interface.	Maximizes code reuse, provides simplified interface, requires less maintenance	Additional code to write

Situation	Migration approach	Advantages	Disadvantages
When dealing with an underlying API that's significantly different between the two platforms (option 2)	Create an adapter using the `#define` preprocessor macro. For example, replace every instance of your color class with either `UIColor` or `NSColor`. This approach also gives you compile-time error checking.	Minimizes new code to write, provides compile-time error checking	Can only be used for supported classes: `UIColor` and `NSColor`; `UIFont` and `NSFont`; `UIImage` and `NSImage`; `UIBezierPath` and `NSBezierPath`. Limited API coverage within supported classes. Requires custom archiving approach.

Video:: WWDC 2013 Bringing Your iOS Apps to OS X

WWDC 2014 Sharing code between iOS and OS X

Migrating the Controller Layer

Controller objects are a critical area of difference between app development for iOS and for OS X. In iOS, `UIViewController` objects are key components that support the presentation of data and handle many device-specific behaviors such as orientation changes. In OS X v10.10 there are three view controllers: `NSViewController`, `NSSplitViewController`, and `NSTableViewController`. Each view controller type participates in the event message chain and is able to respond to event messages.

Before OS X v10.10 there was only one view controller class, `NSViewController`, that do not receive event messages. In those earlier versions, the `NSWindowController` class is most similar to `UIViewController`. In OS X, a window controller manages a window and its current contents.

OS X has no equivalent to the iOS navigation controller which manages a stack of view controllers. If your iOS app uses a view controller stack to display the UI, you need to redesign your app to take advantage of the larger device display and multiple windows that are available in OS X.

In addition to the view and window controller classes, OS X also has the `NSController` class. Instances of this class (and its concrete subclasses) are controller objects that are used in the Cocoa bindings technology. (Cocoa bindings, which is not available in iOS, connects data in a model object with the presentation of that data in a view object, so that both objects are updated when the data changes.) Because controller objects manage an app's data model and not its views, they can be considered data controllers rather than view controllers.

For information about view controllers in OS X, see *NSViewController Class Reference*, and for information about window controllers see *NSWindowController Class Reference*. To learn more about `NSController` objects and Cocoa bindings, see *Cocoa Bindings Programming Topics*.

Differences Between the UIKit and AppKit Frameworks

In OS X, the AppKit framework provides the infrastructure for building graphical apps, managing the event loop, and performing other UI-related tasks. Although AppKit and UIKit (the corresponding iOS framework) provide similar support to apps, the implementation is very different. When you migrate an iOS app to OS X, you replace a significant number of UIKit classes with functionality provided by AppKit.

For information about the classes of AppKit, see *AppKit Framework Reference*.

User Events and Event Handling

Handling events in OS X differs significantly from handling events in iOS apps, mainly because the types of user events on each platform are different. If your iOS app does its own event handling, you will have to rewrite much if not all of the event-handling code when you migrate the app to OS X.

Unlike iOS, which defines touch events and motion events, OS X defines mouse events, keyboard events, trackpad events, tablet events, and tracking-area events. All of these event types relate to peripheral devices that can be attached to a computer. In addition, most of these event types include phases (for example, key-*up* and mouse-*down*) or modifiers (for example, pressing the Control key and a character key simultaneously) that event-handling code often has to test for.

Although the AppKit framework also defines touch (`NSTouch`) objects and gesture events, these events are appropriate only for laptops or desktop computers with supported trackpads. In the OS X version of your app, it's important to handle events from as many types of input devices as possible.

The basic techniques for handling events on each platform are similar. For example, a custom view must opt in to handle events. Then, to handle an event, a custom view must implement one or more methods declared by the responder class of the app framework.

Common examples of iOS event handlers that you need to replace in OS X are `UITapGestureRecognizer` and `UILongPressGestureRecognizer`. `UITapGestureRecognizer` can be replaced in OS X with the `mouseUp:` method of `NSResponder` or subclasses, while UILongPressGestureRecognizer is typically replaced by a right-click event, with a menu displayed using the `menuForEvent:` method of `NSView`.

You can do some event-handling tasks in OS X apps that have no parallel in iOS apps, such as tracking the movement of the pointer and monitoring incoming events in your own app or another app.

For more information about handling events in OS X apps, see *Cocoa Event Handling Guide*.

Windows

At a basic level, windows in AppKit play the same role they do in UIKit: They present graphical content in a rectangular area of the screen, they are at the root of a view hierarchy, they coordinate drawing operations, and they help to distribute events. In most other respects, windows and the framework objects that represent them in OS X and in iOS are different. Here are a few key differences between iOS and OS X windows.

- Most iOS apps have only one fixed size window. In contrast, an OS X app can have multiple windows of varying sizes and styles, and those windows share the screen space with the windows of other apps. Unlike an iOS window, which has no visual adornments, an OS X window usually displays a title and can include controls for moving, closing, resizing, and minimizing the window.
- An OS X app can have one or more main or secondary windows (an app that has multiple main windows is typically a document-based app). A main window is the principal focus of user events and presents an app's data. A secondary window generally serves an auxiliary function and might provide additional control over the data presented in the main window. Secondary windows are often panels—that is, instances of the `NSPanel` class.
- In UIKit, the `UIWindow` class is a subclass of `UIView`, but in AppKit, the `NSWindow` class is a subclass of `NSResponder`. Consequently, windows in AppKit are not backed by Core Animation layers as they are in UIKit. This difference means that you have to perform animation explicitly, at the view level. For a summary of Core Animation differences, see Table 7-3 (page 99).

Menus

Most OS X apps have a much larger command set than comparable iOS apps have. In OS X, apps use the systemwide menu bar to present these commands, whereas iOS apps present commands in UI elements such as toolbars, buttons, table views, and switches. As you migrate your iOS app to OS X, think about the best way to move many of the commands invoked by these elements into menus.

> **Note:** OS X apps can have pop-up lists, contextual menus, and menu-like controls such as the combo box in addition to menus in the menu bar.

By default, an OS X app's menu bar has some standard menus—such as the Apple menu, the app menu, File, Edit, and so on—and each of these menus contains standard menu items. When you create an OS X app project in Xcode, the nib-file template gives you a "starter set" of menus for your menu bar; remove and add menus and menu items to customize this set for your app.

Menu items use the target-action model to send commands (that is, *action messages*) to appropriate objects in the app. Some of the template menu items prewire their target and action properties. You can assign key equivalents to menu items that are most frequently invoked.

To learn more about menus, read *Application Menu and Pop-up List Programming Topics*.

Documents

A document-based app enables users to create, edit, save, and open multiple documents, such as word-processing or spreadsheet documents.

iOS provides a model for document-based apps through the abstract base class `UIDocument`. When you adopt the `UIDocument` approach, your app gets significant behavior with minimal coding effort on your part, including integration with iCloud storage, background reading and writing of data, and an optimized auto-save model.

OS X takes support for documents a step farther, defining an extensive architecture for document-based apps, because this content model is common for the platform. The OS X document architecture is closely based on the Model-View-Controller design pattern. Each document has its own main window and can have secondary windows for auxiliary functions. When you use Xcode to develop a document-based app, you get appropriately configured nib files and stub source files for the `NSDocument` subclass used to manage your documents.

To learn more about the document architecture, read *Mac App Programming Guide*.

Views and Controls

Some of the ready-to-use views that AppKit offers are similar to UIKit views in form, function, and interface. But even with these views, there are differences you should be aware of when migrating your code. Other UIKit views have no counterpart in AppKit because they would not work well in OS X; for these views, you must find a suitable alternative. For example, AppKit uses the `NSBrowser` class to manage the display of hierarchical information; in contrast, an iOS app would use navigation controllers. Some views that seem similar on both platforms have different inheritance characteristics. For example, in iOS `UITableView` inherits from the `UIScrollView` class, whereas in OS X `NSTableView` inherits from `NSControl`.

Although the view base classes—that is, `UIView` and `NSView`—are somewhat similar on both platforms, there are some fundamental differences between them.

- **Core Animation layers.** iOS views are layer backed by default. iOS apps typically manipulate views by changing properties of the view. In OS X, an app must opt in to make its views layer-backed. Consequently, it is more common for AppKit views to perform the same manipulations in their `drawRect:` method. Table 7-3 (page 99) gives more information about differences related to Core Animation layers.

- **Default coordinate system.** The default coordinate systems used in drawing operations are different in iOS and OS X. In iOS, the drawing origin is at the upper-left corner of a view; in OS X, the drawing origin is at the lower-left corner of a view. See Graphics, Drawing, and Printing (page 96) for additional information.
- **Use of cells for controls.** Because AppKit views can incur significant overhead, some AppKit controls use cells (that is, `NSCell` objects) as a lightweight alternative to views. A cell holds the information that its control needs in order to send an action message; a cell also draws itself when commanded by its control. Cells make possible controls such as a matrix object and a table view that have two-dimensional arrays of active subregions.
- **Drawing in relation to view bounds.** `UIView` subviews can draw outside their view bounds. By default, `NSView` subviews clip to view bounds, which is typically the desired behavior.

For functional descriptions of the views and controls available in OS X, along with information on how to use them in your app, see *OS X Human Interface Guidelines*. To learn about the common characteristics and behavior of AppKit views, see *View Programming Guide*.

File System

Many OS X apps let users locate files and directories in the file system, save files to specific file-system locations, open files, and do other file-system operations. An iOS app, on the other hand, must perform all file-reading and file-writing operations within the confines of its sandbox. In OS X v10.7 and later, Mac apps can also be "sandboxed," and this can restrict the file-system operations they can perform (for more information, see App Sandbox (page 69)). Nonetheless, even these apps might have to be prepared to open and save files in the file system as the user directs (assuming that the user has the necessary file permissions).

An OS X app enables these file-system behaviors largely through the Open and Save panels (implemented by the AppKit classes `NSOpenPanel` and `NSSavePanel`). Through instances of these objects, the app can present file-system browsers to users, prevent files it doesn't recognize from being selected, and obtain users' choices. You can also attach custom accessory views to these browsers.

In addition to making use of the Open and Save panels, your app can call methods of the `NSFileManager` and `NSWorkspace` classes for many file-system interactions. `NSFileManager` lets your app create, move, copy, remove, and link file-system items. It also offers other capabilities, such as discovering directory contents, and getting and setting file and directory attributes. Operations of the `NSWorkspace` class augment those of `NSFileManager`; these operations include launching apps, opening specific files, mounting local volumes and removable media, setting the Finder information of files and directories, and tracking file-system changes. (Sandboxed apps can't use `NSWorkspace` in many situations.)

If your iOS app writes and reads files in the `Documents` directory, it uses the `NSSearchPathForDirectoriesInDomains` function to get the proper directory path in the app sandbox. In OS X, you use the `URLsForDirectory:inDomains:` method of the `NSFileManager` class instead; for

this platform you might need to specify domains other than the user domain, and standard directory locations other than `Documents`. For example, your app might want to write or read data in the user's home directory, in `Library/Application Support`.

> **Note:** OS X apps should always store files they create in an appropriate location in the user's `Library` directory; they should not store files in `~/Documents` unless the user selects that location.

To learn more about file-system domains, standard file-system locations, filename extensions, BSD file permissions, AppKit facilities for managing file-system operations, and other information related to the OS X file system, read *File System Programming Guide*.

Graphics, Drawing, and Printing

There are many parallels between the graphics and drawing APIs of AppKit and those of UIKit, as shown in Table 7-2. Both frameworks have classes whose instances represent images, colors, and fonts. Both have classes for drawing Bezier paths and categories for drawing strings. Both have functions for stroking and filling rectangles. Both have programmatic facilities for obtaining and transforming graphics contexts of various types. In some cases, you can migrate code that uses UIKit methods and functions to corresponding AppKit methods and functions with little more than name changes.

Table 7-2 Comparison of graphics, drawing, and printing APIs

	iOS (UIKit)	OS X (AppKit)	Comments
Images	`UIImage`	`NSImage`	`NSImage` can render an image from source data appropriate to an output destination.
Colors	`UIColor`	`NSColor`, `NSColorSpace`, color-related view classes	Apps can use `NSColorSpace` to more precisely define the colors represented by `NSColor` objects.
Bezier paths	`UIBezierPath`	`NSBezierPath`	
Graphics contexts	Functions declared in `UIGraphics.h`.	`NSGraphicsContext`	
PDF	Functions declared in `UIGraphics.h`.	PDF Kit framework	OS X provides richer PDF support than iOS does.

	iOS (UIKit)	OS X (AppKit)	Comments
Printing	Multiple classes	Multiple classes	The classes and techniques for printing in each platform are completely different.

On both platforms, you can call Core Graphics functions when the framework-supplied methods or functions don't suffice for a particular purpose.

The drawing model for AppKit views is nearly identical to the drawing model for UIKit views, with one exception. UIKit views use a coordinate system where the origin for windows and views is in the upper-left corner by default, with positive axes extending down and to the right. In AppKit, the default origin point is in the bottom-left corner and the positive axes extend up and to the right. This is the *default coordinate system* of AppKit, which happens to coincide with the default coordinate system of Core Graphics. To change the default origin of an Appkit view, override the view's `isFlipped` method and return `YES`. The following types of views are are already flipped by default: `NSButton`, `NSScrollView`, `NSSplitView`, `NSTabView`, and `NSTableView`.

> **Note:** Whereas UIKit uses Core Graphics data types for rectangles, points, and other geometric primitives, AppKit uses its own defined types for the same purpose—for example, `NSRect` and `NSPoint`.

For information about graphics and drawing in OS X, see *Cocoa Drawing Guide*. To learn more about printing in OS X, see *NSPrintInfo Class Reference*.

Text

AppKit offers apps a sophisticated system for performing text-related tasks ranging from simple text entry to custom text layout and typesetting. Because the Cocoa text system is based on the Core Text framework and provides a comparable set of behaviors, Cocoa apps rarely need to use Core Text directly.

The native text support of UIKit is limited. Still there is some correspondence between that support and the support offered by AppKit—namely, text views, text fields, font objects, string drawing, and HTML content. If your iOS app uses Core Text to draw and manage fonts and text layout, you can migrate much of that code to your OS X app.

For an introduction to the text system of AppKit, see *Cocoa Text Architecture Guide*.

Table Views

Table views in AppKit are structurally different from table views in UIKit. In UIKit, a table view has any number of rows, and one or more sections but it has only one column. In AppKit, a table view can have any number of rows and columns (and there is no formal notion of sections). An iOS app typically uses a series of UIKit table views, each in their own screen, to present a hierarchical data set. An OS X app, on the other hand, typically uses a single AppKit table view to present all of a data set at the same time.

The structures of these table views differ because of the differences between the platforms. Table views, perhaps the most versatile UI object in iOS, are ideal for displaying data on a smaller screen. Apps use them to navigate hierarchies of data, to present master-detail relationships among data, to facilitate quick retrieval of indexed items, and to serve as lists from which users can select options.

Table views in OS X exist largely to present tabular data in a larger window. AppKit provides other UI objects that are suitable for some of the roles played by UIKit table views—for example, lists (pop-up lists, checkboxes, and radio buttons) and navigation of data hierarchies (browsers and outline views). Consequently, when you migrate your app's table views to OS X, you should first consider whether another AppKit view better suits your app's needs than a table view.

Table views in AppKit are `NSTableView` objects. Cells occupy the region where the rows and columns of the table view intersect. A table view's cells can be based on `NSCell` objects or, in OS X v10.7 and later, `NSView` objects. View-based table views are the preferred alternative. Populating AppKit table views is similar to populating UIKit table views: A data source is queried for the number of rows in the table view and is then asked for the value to put into each cell. Table views in OS X have these other parallels with UIKit table views:

- **Cell reuse**. The delegate of a view-based table view can return a view to use for a cell; the view has an identifier. On the first request, the table view loads this view from a nib file and then caches the view for subsequent requests.
- **Animated operations**. Both `NSCell`-based and `NSView`-based table views have methods for inserting, removing, and moving rows and items, optionally with an animation.

`NSOutlineView`, a subclass of `NSTableView`, has many of these same behaviors. For more information about creating and using table views, see *Table View Programming Guide for Mac* . NSStackView is an Auto Layout–based view that creates and manages the constraints needed to create horizontal or vertical stacks of views. For more information about creating and using stack views, see *View Programming Guide* .

Other Interface Differences

When migrating your app, you should keep in mind other technology differences between UIKit and AppKit; Table 7-3 summarizes these differences.

Table 7-3 Differences between UIKit and AppKit in interface technologies

Difference	Discussion
Core Animation layers	Every drawing surface in OS X can be backed by a Core Animation layer (as in iOS), but an app has to explicitly request this backing for its views. Once this request is made, animation is supported for changes in the properties of the views. In AppKit you don't get the same easy-to-use view-based animation support that you do in UIKit. AppKit also includes the animation features of layer hosting, animation proxies, and classes for animating multiple windows and views. For information about the animation capabilities and features of OS X, see *Animation Overview*.
Target-action model	Target-action in AppKit defines only one form for method signatures, unlike UIKit, which has three forms. Controls in AppKit send their action messages in response to a discrete user action, such as a mouse click; the notion of multiple actions associated with multiple interaction phases does not exist on the platform. However, a control composed of multiple cells can send a different action message for each cell. For more information about controls and the target-action model in OS X apps, see *Control and Cell Programming Topics*.
Responder chain	The responder chain in OS X differs slightly from the responder chain in iOS. As of OS X v10.10, view controllers are added as part of the chain for either user events or action messages. In earlier versions of OS X, view controllers are *not* part of the chain. For action messages, the responder chain in OS X includes window controllers and the view hierarchies of key windows and main windows. AppKit also uses the responder chain for cooperative error handling. To learn more about the responder chain, see *Cocoa Event Handling Guide*.
User preferences	In OS X, all preferences belong in your app; there is no separation of preferences between those managed by your app and those managed by the system. OS X has nothing comparable to the Settings bundle used by iOS apps to specify user preferences presented by the Settings app. Instead apps must create a secondary window for the user preferences. Users open this window by choosing Preferences from the app menu. In addition, OS X integrates user preferences into Cocoa bindings and enables command-line access to the underlying defaults system. One commonality is that both AppKit and UIKit apps use the `NSUserDefaults` class to retrieve user preferences. For more information, see *Preferences and Settings Programming Guide*.

Difference	Discussion
Accessor methods versus properties	UIKit makes extensive use of properties throughout its class declarations, but AppKit mostly declares accessor methods instead of properties.

Foundation Framework Differences

A slightly different version of the Foundation framework in iOS is available in OS X. Most of the classes you would expect to be present are available in both versions—for example, both framework versions provide support for managing values, strings, collections, threads, and many other common types of data. There are, however, some technologies that are present in Foundation in OS X but not included in iOS. These technologies are listed in Table 7-4.

Table 7-4 Foundation technologies available in OS X but not in iOS

Technology	Notes
Spotlight metadata management	Spotlight is a technology for organizing and accessing information on a computer using file metadata. (Metadata is data about a file, rather than the actual file contents.) If you want your app to create Spotlight queries and interact with the results, you use special query and predicate objects. For more information, see *Spotlight Overview*.
Cocoa bindings	Cocoa bindings is a technology that lets you, during development, establish a connection between an item of data encapsulated by a model object and the presentation of that data in a view. It eliminates the need for glue code in the controller layer of an app. For more information, see *Cocoa Bindings Programming Topics*.
Cocoa scripting (AppleScript)	Using certain classes of Foundation along with supporting technology, you can make an app scriptable. A scriptable app is one that responds to commands in AppleScript scripts. To learn more about this technology, see *Cocoa Scripting Guide*.

Technology	Notes
Distributed objects and port name server management	Distributed objects is an interprocess messaging technology. With distributed objects, an object in an app can send a message to an object in a different Cocoa app in the same network or in a different network. The port name server is an object that provides a port-registration service to distributed objects. For information about this technology, see *Distributed Objects Programming Topics*.

The Foundation framework in OS X provides support for both event-driven and tree-based XML processing. The `NSXMLParser` class (also available in iOS) supports the parsing of a stream of XML. In addition, the framework provides the NSXML classes (so called because the names of these classes begin with "NSXML"). Instances of these classes represent an XML document as a tree structure of nodes, including elements and attributes.

For a list of the specific classes that are available in OS X but not in iOS, see the class hierarchy diagram in The Foundation Framework in *Foundation Framework Reference*.

Differences in the Audio and Video Frameworks

In OS X and iOS the primary framework for audiovisual media is AV Foundation. The programmatic interfaces of the framework are almost the same in each platform. Just about any code you write using the iOS version of the framework should be valid with the OS X version. The presentation of audiovisual media, however, is different on the two platforms—specifically, OS X does not have the Media Player framework.

There are substantial differences between the audio frameworks of iOS and the audio frameworks of OS X. The following are the audio technologies of OS X that are not present in iOS:

- Programmatic interfaces in the Audio Unit framework for creating dynamically loadable plug-ins for audio processing (audio units) and conversion (audio codecs), along with bundled user interfaces for audio units
- Additional built-in audio units and audio codecs, along with additional capabilities in like-named audio units
- Programmatic interfaces in the Core Audio framework for interacting with audio hardware
- Programmatic interfaces in the I/O Kit framework for creating and using audio drivers
- Additional capabilities in the OpenAL framework, such as the ability to record audio and to use multichannel audio and surround sound
- Programmatic interfaces in the Core MIDI framework for music sequencing

- Audio development tools (AU Lab, `auval`, `afconvert`)

> **Note:** AV Foundation includes classes for playing and recording audio. These classes are adequate to the audio needs of many apps.

iOS, on the other hand, has audio technologies that are not present in OS X—for example, the Media Player framework, the vibration capabilities of the Audio Toolbox framework, the programmatic interfaces for input clicks, and the audio-session interfaces in the Audio Toolbox and AV Foundation frameworks, including the notions of audio session categories, interruptions, and route changes.

OS X also offers Quick Time, a legacy set of multimedia technologies for playing, creating, editing, importing, compressing, and streaming media of various types. Quick Time is available in both OS X and Windows systems.

To learn more about AV Foundation, see *AV Foundation Programming Guide*.

Differences in Other Frameworks Common to Both Platforms

Table 7-5 lists the key differences in other OS X frameworks from their counterparts in iOS.

Table 7-5 Differences in frameworks common to iOS and OS X

Framework	Differences
`AddressBook.framework`	Contains the interfaces for accessing user contacts. Although it shares the same name, the OS X version of this framework is very different from its iOS counterpart. OS X has no framework for presenting an interface for contacts, whereas iOS does. For more information, see *Address Book Programming Guide for Mac*.
`CFNetwork.framework`	Contains the Core Foundation Network interfaces. In OS X, the CFNetwork framework is a subframework of an umbrella framework, Core Services. Most of the interfaces, however, are the same for iOS and OS X. For more information, see *CFNetwork Framework Reference*.

Framework	Differences
`CoreGraphics.framework`	Contains the Quartz interfaces. You can use Quartz to create paths, gradients, shadings, patterns, colors, images, and bitmaps in exactly the same way you do in iOS. The OS X version of Quartz has features not present in iOS, including PostScript support, image sources and destinations, Quartz Display Services support, and Quartz Event Services support. In OS X, the Core Graphics framework is a subframework of the Application Services umbrella framework and is not a top-level framework, as it is in iOS. For more information, see *Quartz 2D Programming Guide*.
`EventKit.framework`	Provides classes for accessing and manipulating calendar events and reminders. The iOS version does not include reminder items.
`GameKit.framework`	Provides APIs that help you incorporate Game Center, peer-to-peer connectivity, and in-game voice chat into your game. Some of the classes that present UI are different on iOS and OS X.
`GLKit.framework`	Provides functions and classes that reduce the effort required to create new shader-based apps or to port existing apps that rely on fixed-function vertex or fragment processing provided by earlier versions of OpenGL ES or OpenGL. The iOS version includes classes that simplify the creation of OpenGL ES-aware views and view controllers. In OS X, `NSOpenGLView` class subsumes the `GLKView` and `GLKViewController` classes in iOS.
`OpenGL.framework`	OS X uses OpenGL instead of the OpenGL ES framework used in iOS. This fuller-featured version of OpenGL is intended for desktop systems. The programmatic interface of OpenGL is much larger than the one for OpenGLES. OpenGL has many extensions that are not available in the embedded version of the framework. For information about the OpenGL support in OS X, see *OpenGL Programming Guide for Mac*.
`QuartzCore.framework`	Contains the Core Animation, Core Image, and Core Video interfaces. Most of the Core Animation interfaces are the same for OS X and iOS. However, the iOS version of the framework lacks the API support for layout constraints and Core Image filters found in the OS X version. For more information, see *Quartz Core Framework Reference*.

Framework	Differences
`Security.framework`	Contains the security interfaces. Its OS X version includes more capabilities and programmatic interfaces. It has authentication and authorization interfaces and supports the display of certificate contents. In addition, its keychain interfaces are more comprehensive than the ones used in iOS. For information about the security support in OS X, see *Security Overview*.
`System-Configuration.framework`	Contains networking interfaces. The OS X version contains the complete interfaces, not just the reachability interfaces you find in iOS. For more information, see *System Configuration Programming Guidelines*.

OS X Frameworks

The OS X frameworks provide the interfaces you need to write software for Mac. Some of these frameworks contain simple sets of interfaces while others contain multiple subframeworks. Where applicable, the tables in this appendix list the key prefixes used by the classes, methods, functions, types, or constants of the framework. You should avoid using any of the specified prefixes in your own symbol names.

System Frameworks

Table A-1 describes the frameworks located in the `/System/Library/Frameworks` directory and lists the first version of OS X in which each became available.

Table A-1 System frameworks

Name	First available	Prefixes	Description
`Accelerate.framework`	10.3	`cblas`, `vDSP`, `vv`	Umbrella framework for vector-optimized operations. See Accelerate Framework (page 117).
`Accounts.framework`	10.8	`AC`	Provides access to user accounts stored in the Accounts database.
`AddressBook.framework`	10.2	`AB`, `ABV`	Provides access to the Address Book, which is a centralized database of user contact information.
`AGL.framework`	10.0	`AGL`, `GL`, `glm`, `GLM`, `glu`, `GLU`	Contains Carbon interfaces for OpenGL.
`AppKit.framework`	10.0	`NS`	Contains classes and methods for the Cocoa user-interface layer. In general, link to `Cocoa.framework` instead of this framework.
`AppKitScripting.framework`	10.0	N/A	Deprecated. Use `AppKit.framework` instead.

Name	First available	Prefixes	Description
AppleScriptKit.framework	10.0	ASK	Contains interfaces for creating AppleScript plug-ins.
AppleScriptObjC.framework	10.6	NS	Contains Objective-C extensions for creating AppleScript plug-ins.
Application-Services.framework	10.0	AE, AX, ATSU, CG, CT, LS, PM, QD, UT	Umbrella framework for several app-level services. See Application Services Framework (page 117).
AudioToolbox.framework	10.0	AU, AUMIDI	Contains interfaces for getting audio stream data, routing audio signals through audio units, converting between audio formats, and playing back music.
AudioUnit.framework	10.0	AU	Contains interfaces for defining Core Audio plug-ins.
AudioVideo-Bridging.framework	10.8	AVB	Supports Audio Video Bridging (AVB) and implements the IEEE P1722.1 draft standard.
Automator.framework	10.4	AM	Umbrella framework for creating Automator plug-ins. See Automator Framework (page 117).
AVFoundation.framework	10.7	AV	Provides interfaces for playing, recording, inspecting, and editing audiovisual media.
AVKit.framework	10.9	AV	Provides API for media playback including user controls, chapter navigation, subtitles, and closed captioning. See *AVKit Framework Reference*
CalendarStore.framework	10.5	Cal	Deprecated. Use Event Kit instead. See Event Kit (page 64).

Name	First available	Prefixes	Description
Carbon.framework	10.0	HI, HR, ICA, ICD, Ink, Nav, OSA, PM, SFS,SR	Umbrella framework for Carbon-level services. See Carbon Framework (page 118).
CloudKit.framework	10.10	CK	Provides a conduit for moving data between your app and iCloud that can be used for all types of data. It also gives you control of when transfers occur. See *CloudKit Quick Start* or *CloudKit Framework Reference*.
CFNetwork.framework	10.3	CF	Contains interfaces for network communication using HTTP, sockets, and Bonjour.
Cocoa.framework	10.0	NS	Wrapper for including the Cocoa frameworks AppKit.framework, Foundation.framework, and CoreData.framework.
Collaboration.framework	10.5	CB	Contains interfaces for managing identity information.
CoreAudio.framework	10.0	Audio	Contains the hardware abstraction layer interface for manipulating audio.
CoreBluetooth.framework	10.10	CB	Contains the classes used for communicating with Bluetooth Low Energy devices.
CoreAudioKit.framework	10.4	AU	Contains Objective-C interfaces for audio unit custom views.
CoreData.framework	10.4	NS	Contains interfaces for managing your app's data model.

Name	First available	Prefixes	Description
CoreFoundation.framework	10.0	CF	Provides fundamental software services, including abstractions for common data types, string utilities, collection utilities, plug-in support, resource management, preferences, and XML parsing.
CoreGraphics.framework	10.0	CG	Contains the Quartz interfaces for creating graphic content and rendering that content to the screen.
CoreLocation.framework	10.6	CL	Provides interfaces for determining the geographical location of a computer.
CoreMedia.framework	10.7	CM	Contains low-level interfaces for for managing and and playing audio-visual media in an app.
CoreMediaIO.framework	10.7	CMIO	Contains interfaces of the Device Abstraction Layer (DAL) used for creating plug-ins that can access media hardware.
CoreMIDI.framework	10.0	MIDI	Contains utilities for implementing MIDI client programs.
CoreMIDIServer.framework	10.0	MIDI	Contains interfaces for creating MIDI drivers to be used by the system.
CoreServices.framework	10.0	CF, DCS, MD, SK, WS	Umbrella framework for system-level services. See Core Services Framework (page 119).
CoreText.framework	10.5	CT	Contains the interfaces for performing text layout and display.
CoreVideo.framework	10.5	CV	Contains interfaces for managing video-based content.
CoreWLAN.framework	10.6	CW	Contains interfaces for managing wireless networks.

Name	First available	Prefixes	Description
CryptoTokenKit.framework	10.10	TK	Contains interface for using smart cards.
Directory-Service.framework	10.0	ds	Contains interfaces for supporting network-based lookup and directory services in your app. You can also use this framework to develop directory service plug-ins.
DiscRecording.framework	10.2	DR	Contains interfaces for burning data to CDs and DVDs.
DiscRecordingUI.framework	10.2	DR	Contains the user interface layer for interacting with users during the burning of CDs and DVDs.
DiskArbitration.framework	10.4	DA	Contains interfaces for getting information related to local and remote volumes.
DrawSprocket.framework	10.0	DSp	Contains the game sprocket component for drawing content to the screen.
DVComponentGlue.framework	10.0	IDH	Contains interfaces for communicating with digital video devices, such as video cameras.
DVDPlayback.framework	10.3	DVD	Contains interfaces for embedding DVD playback features into your app.
EventKit.framework	10.8	EK	Provides an interface for accessing a user's calendar events and reminder items.
Exception-Handling.framework	10.0	NS	Contains exception-handling classes for Cocoa apps.
FinderSync.framework	10.10	FI	Provides API for enhancing the Finder's user interface by adding badges, shortcut menu items, and toolbar buttons. See *Finder Sync Framework Reference*

Name	First available	Prefixes	Description
ForceFeedback.framework	10.2	FF	Contains interfaces for communicating with force feedback–enabled devices.
Foundation.framework	10.0	NS	Contains the classes and methods for the Cocoa Foundation layer. If you are creating a Cocoa app, linking to the Cocoa framework is preferable.
FWAUserLib.framework	10.2	FWA	Contains interfaces for communicating with FireWire-based audio devices.
GameController.framework	10.9	GC	A collection of classes for discovering and interacting with connected game controllers. See *GameControllerProgrammingGuide*
GameKit.framework	10.8	GK	Provides APIs that allow your app to participate in Game Center.
GLKit.framework	10.8	GLK	Provides functions and classes that reduce the effort required to create new shader-based apps or to port existing apps that rely on fixed-function vertex or fragment processing provided by earlier versions of OpenGL ES or OpenGL.
GLUT.framework	10.0	glut, GLUT	Contains interfaces for the OpenGL Utility Toolkit, which provides a platform-independent interface for managing windows.
GSS.framework	10.7	gss	Contains interfaces for Generic Security Services Application Program Interface (GSSAPI).
ICADevices.framework	10.3	ICD	Contains low-level interfaces for communicating with digital devices such as scanners and cameras. See Carbon Framework (page 118).

Name	First available	Prefixes	Description
ImageCapture-Core.framework	10.6	IC	Contains Objective-C interfaces for communicating with digital devices such as scanners and cameras.
IMCore.framework	10.6	IM	Do not use.
IMServicePlugIn.framework	10.7	IM	Contains interfaces for building third-party plug-ins for Chat services. Umbrella framework for IMServicePlugIn-Support.framework.
InputMethodKit.framework	10.5	IMK	Contains interfaces for developing new input methods, which are modules that handle text entry for complex languages.
Installer-Plugins.framework	10.4	IFX	Contains interfaces for creating plug-ins that run during software installation sessions.
InstantMessage.framework	10.4	FZ, IM	Contains interfaces for obtaining the online status of an instant messaging user.
IOBluetooth.framework	10.2	IO	Contains interfaces for communicating with Bluetooth devices.
IOBluetoothUI.framework	10.2	IO	Contains the user interface layer for interacting with users manipulating Bluetooth devices.
IOKit.framework	10.0	IO, IOBSD, IOCF	Contains the main interfaces for creating user-space device drivers and for interacting with kernel-resident drivers from user space.
IOSurface.framework	10.6	IO	Contains low-level interfaces for sharing graphics surfaces between apps.

Name	First available	Prefixes	Description
JavaFrame-Embedding.framework	10.5	N/A	Contains interfaces for embedding Java frames in Objective-C code.
JavaScriptCore.framework	10.5	JS	Contains the library and resources for executing JavaScript code within an HTML page. (Prior to OS X v10.5, this framework was part of WebKit.framework.
JavaVM.framework	10.0	JAWT, JDWP, JMM, JNI, JVMDI, JVMPI, JVMTI	Contains the system's Java Development Kit resources.
Kerberos.framework	10.0	GSS, KL, KRB, KRB5	Contains interfaces for using the Kerberos network authentication protocol.
Kernel.framework	10.0	*numerous*	Contains the interfaces for kernel-extension development, including Mach, BSD, libkern, I/O Kit, and the various families built on top of I/O Kit.
LatentSemantic-Mapping.framework	10.5	LSM	Contains interfaces for classifying text based on latent semantic information.
LDAP.framework	10.0	N/A	Do not use.
Local-Authentication.framework	10.10	LA	Contains API for requesting authentication from users using specified policies. See *Local Authentication Framework Reference*
MapKit.framework	10.9	MK	Classes and protocols for embedding maps into the windows and views of your apps. Includes support for annotations, overlays, and reverse-geocoding lookups. See *MapKit Framework Reference*

Name	First available	Prefixes	Description
`Media-Accessibility.framework`	10.9	MA	Provides API to access user preferences for captions shown with media, closed captioning for example. See *Media Accessibility Framework Reference*
`MediaLibrary.framework`	10.9	ML	Provides a read-only data model representing a user's collections of images, audio, and video. See *Media Library Framework Reference*
`Message.framework`	10.0	`AS`, `MF`, `PO`, `POP`, `RSS`, `TOC`, `UR`, `URL`	Contains Cocoa extensions for mail delivery.
`Multipeer-Connectivity.framework`	10.10	MC	Contains API for finding and communicating with services provided by nearby devices using infrastructure Wi-Fi networks, peer-to-peer WiFi, and Bluetooth personal area networks. See **!!! Multipeer Connectivity Framework Reference**
`NetFS.framework`	10.6	`NetFS`	Contains interfaces for working with network file systems.
`Notification-Center.framework`	10.10	NC, NS	Contains the interfaces for creating and managing extensions in the Today view of the Notification Center. See *Notification Center Framework Reference*
`OpenAL.framework`	10.4	`AL`	Contains the interfaces for OpenAL, a cross-platform 3D audio delivery library.
`OpenCL.framework`	10.6	`CL`, `cl`	Contains the interfaces for distributing general-purpose computational tasks across the available GPUs and CPUs of a computer.

Name	First available	Prefixes	Description
OpenDirectory.framework	10.6	OD	Contains Objective-C interfaces for managing Open Directory information.
OpenGL.framework	10.0	CGL, GL, glu, GLU	Contains the interfaces for OpenGL, which is a cross-platform 2D and 3D graphics rendering library.
OSAKit.framework	10.4	OSA	Contains Objective-C interfaces for managing and executing OSA-compliant scripts from Cocoa apps.
PCSC.framework	10.0	MSC, Scard, SCARD	Contains interfaces for interacting with smart card devices.
PreferencePanes.framework	10.0	NS	Contains interfaces for implementing custom modules for the System Preferences app.
PubSub.framework	10.5	PS	Contains interfaces for subscribing to RSS and Atom feeds.
QTKit.framework	10.4	QT	Contains Objective-C interfaces for manipulating QuickTime content.
Quartz.framework	10.4	GF, PDF, QC, QCP	Umbrella framework for Quartz services. See Quartz Framework (page 119).
QuartzCore.framework	10.4	CA,CI, CV	Contains the interfaces for Core Image, Core Animation, and Core Video.
QuickLook.framework	10.5	QL	Contains interfaces for generating thumbnail previews of documents.
QuickTime.framework	10.0	N/A	Contains interfaces for embedding QuickTime multimedia into an app.
Ruby.framework	10.5	N/A	Contains interfaces for the Ruby scripting language.

Name	First available	Prefixes	Description
SceneKit.framework	10.8	SCN	Provides a high-level, Objective-C API to efficiently load, manipulate, and render 3D scenes in an app.
ScreenSaver.framework	10.0	N/A	Contains interfaces for writing screen savers.
Scripting.framework	10.0	NS	Deprecated. Use Foundation.framework instead.
ScriptingBridge.framework	10.5	SB	Contains interfaces for running scripts from Objective-C code.
Security.framework	10.0	CSSM, Sec	Contains interfaces for system-level user authentication and authorization.
Security-Foundation.framework	10.3	Sec	Contains Cocoa interfaces for authorizing users.
Security-Interface.framework	10.3	PSA, SF	Contains the user interface layer for authorizing users in Cocoa apps.
Service-Management.framework	10.6	SM	Contains interfaces for loading, unloading and managing launchd services.
Social.framework	10.8	SL	Provides an API for sending requests to supported social networking services that can perform operations on behalf of users.
SpriteKit.framework	10.9	SK	Provides API for animating arbitrary textured images, or sprites. It includes sound playback, a physics engine, and a rendering loop. See *SpriteKit Programming Guide*
StoreKit.framework	10.7	SK	Supports requesting payment from a user to purchase additional functionality or content from the Mac App Store.
SyncServices.framework	10.4	ISync	Deprecated in OS X v10.7.

Name	First available	Prefixes	Description
System.framework	10.0	N/A	Do not use.
SystemConfiguration.framework	10.0	SC	Contains interfaces for accessing network configuration and reachability. information.
Tcl.framework	10.3	Tcl	Contains interfaces for accessing the system's Tcl interpreter from an app.
Tk.framework	10.4	Tk	Contains interfaces for accessing the system's Tk toolbox from an app.
TWAIN.framework	10.2	TW	Contains interfaces for accessing TWAIN-compliant image-scanning hardware.
vecLib.framework	10.0	N/A	Deprecated. Use Accelerate.framework instead. See Accelerate Framework (page 117).
VideoDecodeAcceleration.framework	10.7	VDA	Contains interfaces for using hardware resources for accelerated video decoding.
VideoToolbox.framework	10.8	VT	Comprises the 64-bit replacement for the QuickTime Image Compression Manager.
WebKit.framework	10.2	DOM, Web	Umbrella framework for rendering HTML content. See WebKit Framework (page 120).
XgridFoundation.framework	10.4	XG	Deprecated in OS X v10.8. Contains interfaces for connecting to and managing computing cluster software.

OS X contains several umbrella frameworks for major areas of functionality. Umbrella frameworks group several related frameworks into a larger framework that can be included in your project. When writing software, link your project against the umbrella framework; do not try to link directly to any of its subframeworks. The following sections describe the contents of the umbrella frameworks in OS X.

Accelerate Framework

Table A-2 lists the subframeworks of the Accelerate framework (`Accelerate.framework`). If you are developing apps for earlier versions of OS X, `vecLib.framework` is available as a standalone framework.

Table A-2 Subframeworks of the Accelerate framework

Subframework	Description
`vecLib.framework`	Contains vector-optimized interfaces for performing math, big-number, and DSP calculations, among others.
`vImage.framework`	Contains vector-optimized interfaces for manipulating image data.

Application Services Framework

Table A-3 lists the subframeworks of the Application Services framework (`ApplicationServices.framework`). These frameworks provide C-based interfaces and are intended primarily for Carbon apps, although other programs can use them. The listed frameworks are available in all versions of OS X unless otherwise noted.

Table A-3 Subframeworks of the Application Services framework

Subframework	Description
`ATS.framework`	Contains interfaces for font layout and management using Apple Type Services.
`ColorSync.framework`	Contains interfaces for color matching using ColorSync.
`HIServices.framework`	Contains interfaces for accessibility, Internet Config, the pasteboard, the Process Manager, and the Translation Manager. Available in OS X 10.2 and later.
`ImageIO.framework`	Contains interfaces for importing and exporting image data.
`LangAnalysis.framework`	Contains the Language Analysis Manager interfaces.
`PrintCore.framework`	Contains the Core Printing Manager interfaces.
`QD.framework`	Contains the QuickDraw interfaces.
`SpeechSynthesis.framework`	Contains the Speech Manager interfaces.

Automator Framework

Table A-4 lists the subframeworks of the Automator framework (`Automator.framework`).

Table A-4 Subframeworks of the Automator framework

Subframework	Description
`MediaBrowser.framework`	Contains private interfaces for managing Automator plug-ins.

Carbon Framework

Table A-5 lists the subframeworks of the Carbon framework (`Carbon.framework`). The listed frameworks are available in all versions of OS X unless otherwise noted.

Table A-5 Subframeworks of the Carbon framework

Subframework	Description
`CarbonSound.framework`	Contains the Sound Manager interfaces. Whenever possible, use Core Audio instead.
`CommonPanels.framework`	Contains interfaces for displaying the Font window, Color window, and some network-related dialogs.
`Help.framework`	Contains interfaces for launching and searching Apple Help.
`HIToolbox.framework`	Contains interfaces for the Carbon Event Manager, HIToolbox object, and other user interface–related managers.
`HTMLRendering.framework`	Contains interfaces for rendering HTML content. The WebKit framework is the preferred framework for HTML rendering. See WebKit Framework (page 120).
`ImageCapture.framework`	Contains interfaces for capturing images from digital cameras. This framework works in conjunction with the Image Capture Devices framework (`ICADevices.framework`).
`Ink.framework`	Contains interfaces for managing pen-based input. (Ink events are defined with the Carbon Event Manager.)
`Navigation-Services.framework`	Contains interfaces for displaying file navigation dialogs.
`OpenScripting.framework`	Contains interfaces for writing scripting components and interacting with those components to manipulate and execute scripts.
`Print.framework`	Contains the Carbon Printing Manager interfaces for displaying printing dialogs and extensions.

Subframework	Description
SecurityHI.framework	Contains interfaces for displaying security-related dialogs.
Speech-Recognition.framework	Contains the Speech Recognition Manager interfaces.

Core Services Framework

Table A-6 lists the subframeworks of the Core Services framework (CoreServices.framework). These frameworks provide C-based interfaces and are intended primarily for Carbon apps, although other programs can use them. The listed frameworks are available in all versions of OS X unless otherwise noted.

Table A-6 Subframeworks of the Core Services framework

Subframework	Description
AE.framework	Contains interfaces for creating and manipulating Apple events and making apps scriptable.
CarbonCore.framework	Contains interfaces for many legacy Carbon Managers. Most of the APIs in this framework are deprecated in OS X v10.8 (for more information, see *Carbon Core Deprecations*).
DictionaryServices.framework	Provides dictionary lookup capabilities.
LaunchServices.framework	Contains interfaces for launching apps.
Metadata.framework	Contains interfaces for managing Spotlight metadata.
OSServices.framework	Contains interfaces for Open Transport and many hardware-related legacy Carbon managers.
SearchKit.framework	Contains interfaces for the Search Kit.

Quartz Framework

Table A-7 lists the subframeworks of the Quartz framework (Quartz.framework).

Table A-7 Subframeworks of the Quartz framework

Subframework	Description
ImageKit.framework	Contains Objective-C interfaces for finding, browsing, and displaying images.

Subframework	Description
PDFKit.framework	Contains Objective-C interfaces for displaying and managing PDF content.
QuartzComposer.framework	Contains Objective-C interfaces for playing Quartz Composer compositions in an app.
QuartzFilters.framework	Contains Objective-C interfaces for managing and applying filter effects to a graphics context.
QuickLookUI.framework	Contains Objective-C interfaces for creating and managing a Quick Look preview panel, which is a UI object that displays preview items.

WebKit Framework

Table A-8 lists the subframeworks of the WebKit framework (WebKit.framework).

Table A-8 Subframeworks of the WebKit framework

Subframework	Description
WebCore.framework	Contains the library and resources for rendering HTML content in an HTMLView control.

Xcode Frameworks

Xcode and all of its supporting tools and libraries reside in a portable directory structure. This directory structure makes it possible to have multiple versions of Xcode installed on a single system or to have Xcode installed on a portable hard drive that you plug in to your computer when you need to do development. This portability means that the frameworks required by the developer tools are installed in the <Xcode>/Library/Frameworks directory, where <Xcode> is the path to the Xcode installation directory. Table A-9 lists the frameworks that are located in this directory.

Table A-9 Xcode frameworks

Framework	First available	Prefixes	Description
XCTest.framework	Xcode 5	XC	Interfaces for implementing unit tests in Objective-C.

Framework	First available	Prefixes	Description
`InterfaceBuilderKit.framework`	10.5	`ib`, `IB`	Interfaces for writing plug-ins that work in Interface Builder 3.0 and later.
`SenTestingKit.framework`	10.4	`Sen`	Interfaces for implementing unit tests in Objective-C.

System Libraries

Some specialty libraries at the BSD level are not packaged as frameworks. Instead, OS X includes many dynamic libraries in the `/usr/lib` directory and its subdirectories. Dynamic shared libraries are identified by their `.dylib` extension. Header files for the libraries are located in the `/usr/include` directory.

OS X uses symbolic links to point to the most current version of most libraries. When linking to a dynamic shared library, use the symbolic link instead of a link to a specific version of the library. Library versions may change in future versions of OS X. If your software is linked to a specific version, that version might not always be available on the user's system.

Document Revision History

This table describes the changes to *Mac Technology Overview*.

Date	Notes
2014-10-16	Updates for OSX 10.10
2014-07-15	Updated the Migrating from Cocoa Touch chapter.
2013-10-22	Made minor formatting corrections.
2012-07-23	Described new technologies available in OS X v10.8. Added a section on Notification Center (page 27). Added a section on Game Center (page 27). Added a section on Sharing (page 28). Added a section on Social Media Integration (page 53). Added a section on Gatekeeper (page 68). Added sections on the following new frameworks: Game Kit (page 34), GLKit (page 45), Scene Kit (page 50), Audio Video Bridging and Video Toolbox (both described in Other Media Layer Frameworks (page 50)), Accounts (page 63), Event Kit (page 64), and Social Framework (page 66). Updated the sections on Apps (page 13), Resume (page 28), Cocoa Auto Layout (page 29), and iCloud Storage (page 54).
2012-01-09	Added descriptions for several new technologies first available in OS X v10.7. Added a section on iCloud Storage (page 54). Added a section on Automatic Reference Counting (page 59). Added a section on Popovers (page 29).

Date	Notes
	Added a section on Store Kit (page 66). Added a section on In-Kernel Video Capture (page 73). Updated the sections on Cocoa Auto Layout (page 29), File Coordination (page 55), App Sandbox (page 69), Code Signing (page 69), XPC Interprocess Communication and Services (page 72), AV Foundation (page 43), Core Data (page 63), and Foundation and Core Foundation (page 64). Removed the section on Sync Services, which described a framework that was deprecated in OS X v10.7. If you need information on Sync Services, see the Legacy OS X Developer Library.
2011-04-30	Updated document for OS X v10.7, with extensive reorganization and rewriting. Added the chapter "Migrating from Cocoa Touch."
2009-08-14	Updated for OS X v10.6.
2008-10-15	Removed outdated reference to jikes compiler. Marked the AppleShareClient framework as deprecated, which it was in OS X v10.5.
2007-10-31	Updated for OS X v10.5. The document was also reorganized.
2006-06-28	Associated in-use prefix information with the system frameworks. Clarified directories containing developer tools.
2005-10-04	Added references to "Universal Binary Programming Guidelines."
2005-08-11	Fixed minor typos. Updated environment variable inheritance information.
2005-07-07	Incorporated developer feedback. Added AppleScript to the list of app environments.
2005-06-04	Corrected the man page name for SQLite.
2005-04-29	Fixed broken links and incorporated user feedback.

Date	Notes
	Incorporated porting and technology guidelines from "Apple Software Design Guidelines." Added information about new system technologies. Changed "Rendezvous" to "Bonjour." Added new software types to list of development opportunities. Added a command-line primer. Added a summary of the available development tools. Updated the list of system frameworks.
2004-05-27	First version of *OS X Technology Overview*. Some of the information in this document previously appeared in *System Overview*.

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010