Implementing iOS and macOS Documents with the Files App

Managing Files and Ensuring Compatibility

Jesse Feiler

Apress®

Implementing iOS and macOS Documents with the Files App

Managing Files and Ensuring Compatibility

Jesse Feiler

Apress®

Implementing iOS and macOS Documents with the Files App: Managing Files and Ensuring Compatibility

Jesse Feiler
Plattsburgh, NY, USA

ISBN-13 (pbk): 978-1-4842-4491-3 ISBN-13 (electronic): 978-1-4842-4492-0
https://doi.org/10.1007/978-1-4842-4492-0

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Aaron Black
Development Editor: James Markham
Coordinating Editor: Jessica Vakili

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media, New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/978-1-4842-4491-3. For more detailed information, please visit www.apress.com/source-code.

Printed on acid-free paper

Table of Contents

About the Author

Jesse Feiler is a developer, consultant, and author specializing in database technologies and location-based apps. Jesse's apps include NP Risk, Minutes Machine, Utility Smart, Cyber Continuity, and Saranac River Trail. He has worked for organizations as varied as the Federal Reserve Bank of New York (Chief, Special Projects Staff in Systems Development), the Albers and Archipenko foundations (data management), and a number of database projects typically using FileMaker. His apps are available in the App Store and are published by Champlain Arts Corp (`champlainarts.com`). Jesse is heard regularly on WAMC Public Radio for the Northeast's The Roundtable. He is a founder of Friends of Saranac River Trail, Inc. A native of Washington DC, he has lived in New York City and currently lives in Plattsburgh, NY.

About the Technical Reviewer

Charles Cruz is a mobile application developer for the iOS, Windows Phone, and Android platforms. He graduated from Stanford University with B.S. and M.S. degrees in engineering. He lives in Southern California and runs a photography business with his wife (www.bellalentestudios.com) and enjoys backpacking. Charles can be reached at codingandpicking@gmail.com.

CHAPTER 1

Using Documents

We use documents to store and organize data in the apps that we use.

This is a simple description of how and why we use documents with mobile apps built with macOS and iOS. There's much more than this simple description to consider when you start working with documents, and this chapter goes into the basic details you need to consider. You can find many books and articles dealing with documents, but the key points are described here.

Describing a Document

When you use an app, you sometimes need to store data for the app (that's the basic description just mentioned). Storing data turns out to be far from simple because when we talk about storing data, we almost always mean storing *and retrieving* data on demand. For that store-and-retrieve process to be useful to developers and users, you need to be able to *identify* the data to be stored and retrieved, such as the current temperature.

Just to make things a little more complex, you need to be able to store and retrieve data that you can identify in two different ways:

- You need to be able to identify the physical location of the data to be stored and retrieved.
- You need to be able to identify the logical characteristics of the data to be stored and retrieved.

J. Feiler, *Implementing iOS and macOS Documents with the Files App*,
https://doi.org/10.1007/978-1-4842-4492-0_1

Putting this together means that you need to be able to store, retrieve, and identify data by its location and characteristics (such as a name).

Keeping Track of a Document and Its Data

We are accustomed to thinking of documents as static objects: once a document is written or printed, it doesn't appear to change. You can make changes or edits to documents, but those changes are typically visible in one way or another so that the initial document is modified. In the digital world, changes can be continual, and thinking of a document as a static object is misleading, to say the least.

When you use word processing tools, you can often track changes to documents so that instead of a static document you may have a multitude of changed documents. This multitude of changed documents can proliferate quickly not only with word processing documents but also with changes using tools such as Git or GitHub.

Structuring a Document

Documents can be structured in any way that the developer chooses. As you will see in Chapter 2, you can use structures that you create or common structures that are defined by others. The structure of a document provides a structure (or format) for the data that the document will contain. When you know a document's structure, you can read or write its data.

At least that is the idea. Document structures can change over time so in practice you need to know not only the structure of a document but the specific variation of the structure in use.

Note The variation of a document's structure is often referred to as a *version.*

Handling Document Versions

A common way of handling the issue of document versions is to create a document structure that has at least two components: one is the version identifier and the second is everything else. For example, a document can start with the version identifier, which might be something as simple as a string or even an integer. In that way, your app will know to read a single integer or a string of X characters from the beginning of a document's data. That integer or string lets your app identify the version; having done that, your app can read the data for that version. This strategy is commonly used in macOS and iOS using a *file manager* (described in Chapter 2).

Comparing Documents and Files

Documents store data for an app in a known location from which it can be retrieved (or to which it can be stored). This location is typically a *file*—an object that is managed by the operating system. Like documents themselves, files can also have versions. A significant difference between a file and a document is that in many cases, the operating system manages a file's opening, closing, and storage. A document in many cases is inside a file.

Note This is a simplification and generalization.

Structuring a Document and an App

Apps that are based on data are easy to build or convert to document-based apps. There are two common ways of building such apps. In the first way, developers start from a data structure and add functionality to it. In the other way, developers start from functionality and add data to it.

Summary

In this chapter, you saw an overview of documents, versions, and the differences between documents and files. From here you will move onto the details of documents and how to use them effectively.

Today, JSON (JavaScript Object Notation) and the `Codable` protocols are commonly used for managing app data. Previously, a technology referred to as *coding* was commonly used to store and manage data that is internal to an app. The basic process was to convert data that is identified by keys to and from `NSData` objects. The operating systems support `NSData`, and you don't have to worry about the implementation: it is fast and efficient. The only limitation is that not every type of data can be archived.

If you are building an app that needs to manage persistent data, chances are that Codable is the way to go. If you are modifying an existing app, you may want to continue using the archiving code that already exists. (Using both is perfectly feasibly, but it can become a maintenance nightmare.)

CHAPTER 2

Looking Inside a Document

In Chapter 1, you learned how to describe and structure a document. You now know that you, as the designer and developer of an app and its documents, control what data is stored, where and how it is stored, and how to identify and reference it.

You can decide that the data will be stored as a sequence of integers or as a single long string, whatever matters to you and the data you will use. In practice, it makes sense to structure the data inside a document if only to be able to access it easily. This chapter shows how to structure the data within a document using JSON encoding. This structure and encoding provides an easy-to-use format for data that relies on Unicode strings that can represent basic types recognized by JSON.

Using JSON Encoding

What matters most for JSON is the fact that the format is text-based (as opposed, for example, to a binary or digital representation) and the fact that each element can be named (as opposed to being identified by location or sequence).

J. Feiler, *Implementing iOS and macOS Documents with the Files App*,
https://doi.org/10.1007/978-1-4842-4492-0_2

A location- or sequence-based coding style lets you specify the format of each element in the encoding sequence. Knowing the format of an element means that you know how much space it will take up, and this will let you read or write the data using the standard read/write syntax in any programming language.

The disadvantage of sequence- or location-based coding is that if you change the sequence of data elements or the format of a data element, you break any read/write code that you already have. JSON encoding relies on names of data elements rather than their formats or sequence. Thus, you avoid the frequent problem of breaking read/write code when you modify a format of a single data element or when you change the order of the data elements.

Introducing JSON

JSON starts as a text format for serialization of structured data. In this sense, *serialization* means converting the strings or other objects into a format that can be read or written. JSON starts from four primitive types, the meanings of which are common to many programming languages:

- A *string* is an ordered collection of Unicode characters.
- A *number* is just that; the most basic JSON number is a double.
- A *Boolean* is true or false.
- The final primitive value in JSON is *null*, an object that has no value.

In JSON, these types can be combined into *objects*, which are unordered collections of name/value pairs; a JSON *array* is an ordered collection of name/value pairs.

JSON and Swift

Swift goes beyond the basic JSON types with its `JSONSerialization` class (part of the Foundation framework). `JSONSerialization` converts `JSON` into array and dictionary Swift data types in addition to the basic JSON string, number, and Bool data types.

Note Swift bridges Boolean and bool (C) types into Bool types. This is handled automatically for you.

Using Swift Structs

JSON is a flexible and easy-to-use notation tool. On the other hand, Swift is designed to be a powerful tool for building apps, particularly those using the model-view-controller (MVC) design pattern, which is more complex than JSON. One area that demonstrates this well is the Swift struct type. You may often declare structs in Swift that you will use throughout your app (or not at all). When you work strictly with JSON, it is uncommon to declare a struct that is not used to store data. This section explains how to create and use Swift structs with JSON.

Listing 2-1 shows how to create a Swift struct for a `Student` object or model (the terms are interchangeable in this section) using a playground.

Listing 2-1. Swift Struct

```
import Foundation

struct Student {
  var name: String
  var studentID: Int
}
```

What matters here is that the `Student` struct contains two var elements: `name` and `studentID`. Also worth noting is the fact that in this playground the Foundation framework must be imported because it will be used for working with JSON data. The other elements of the struct are standard Swift elements.

Tip Note that the Swift style is to capitalize names of objects such as structs, so the name of the `Student` struct is capitalized.

With the struct shown in Listing 2-1, you can create an instance of the struct using code such as the following:

```
let student1 = Student(name: "John Appleseed", studentID: 154)
```

You can integrate JSON with Swift by using an `encode (to: encoder)` function to encode data along with an `init (from decoder:)` to do the reverse. To do this, you need to create keys to identify the elements that you will be coding and decoding. The first step is to declare coding keys as an `enum CodingKeys element`, as shown in Listing 2-2.

Listing 2-2. Swift Extension for Coding Keys

```
  enum CodingKeys: String, CodingKey {
    case studentID = "studentID"
    case name
  }
}
```

Note that they are the keys you will use to encode and decode the data for the `name` and `studentID` variables. With the keys established along with the variables, you can now create an `encode (to: encoder)` function, as shown in Listing 2-3. Note that this extension indicates that the `Student` struct conforms to the `Encodable` protocol.