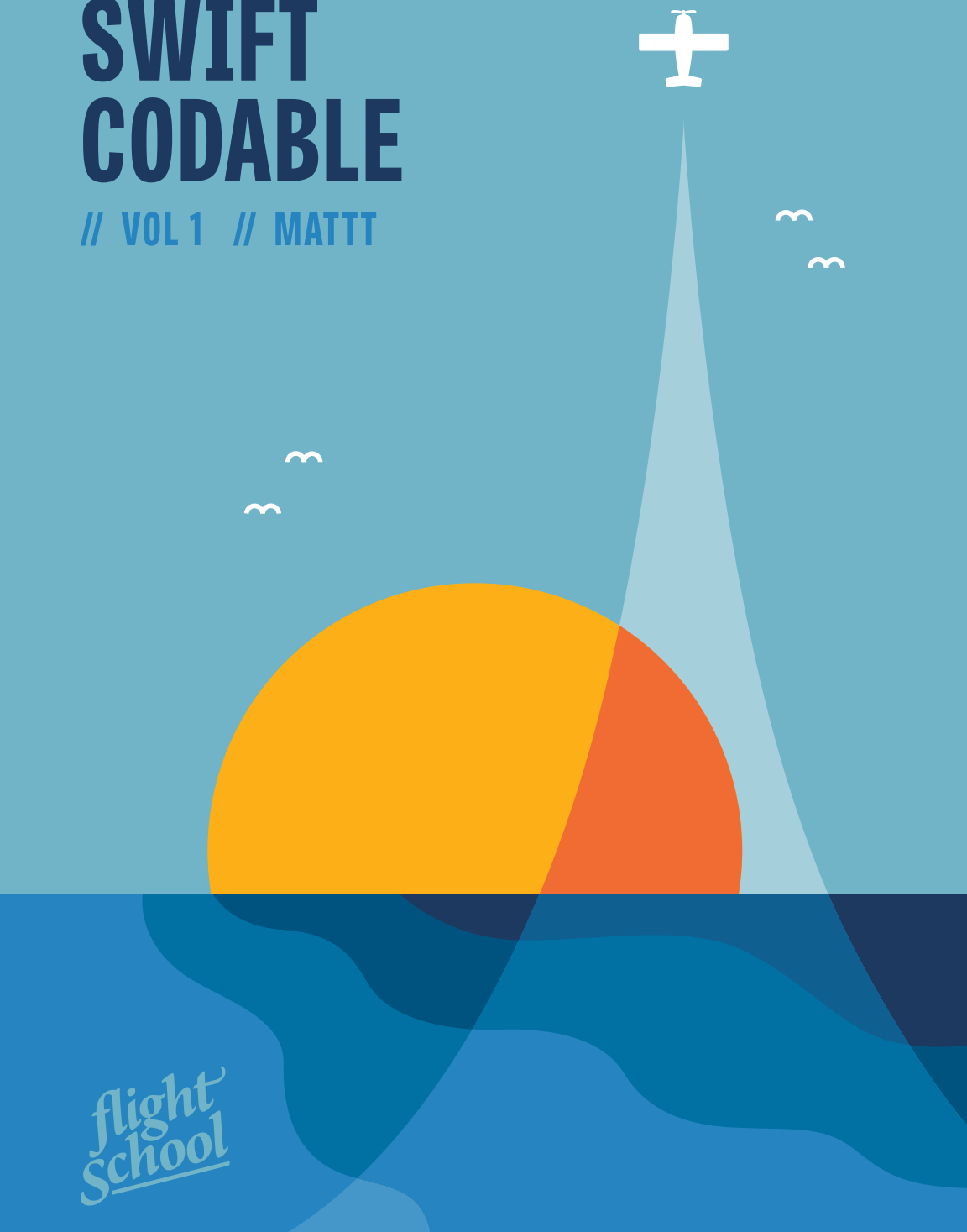
GUIDE TO SWIFT CODABLE

// VOL 1 // MATTT



flight School

Flight School
An imprint of Read Evaluate Press, LLC
Portland, Oregon
https://readeval.press

Publisher	Mattt Zmuda
Illustrator	Lauren Mendez
Cover Designer	Mette Hornung Rankin
Editor	Morgan Tarling

ISBN 978-1-949080-00-1

Printed in IBM Plex, designed by Mike Abbink at IBM in collaboration with Bold Monday.
Additional glyphs provided by Google Noto Sans.
Twemoji graphics made by Twitter and other contributors.
Cover type set in Tablet Gothic, designed by José Scaglione & Veronika Burian.

Contents

First Flight 1
Holding Patterns 18
Taking the Controls 31
Building a Music Store App with iTunes Search API 50
Building an In-Flight Service App with User Defaults 71
Building a Baggage Scanning App with Core Data 89
Implementing a MessagePack Encoder 101


flight
School

Chapter 1:
First Flight

For your first flight with `Codable`, let's take a round-trip journey by decoding a JSON payload into a model object and then encoding that model object back into JSON. We'll then circle around and try out these maneuvers in different array formations. Finally, we'll do a quick run-through of our emergency procedures to get a better idea of how to handle errors, should they arise.

By performing these simple drills, you'll get a 10,000-foot view of how to use `Codable`, and be ready to take on more advanced exercises in the next chapters.

So without further ado, let's head over to the ramp and check out our aircraft!

Inspecting the Airplane

Prior to engaging in any operation, it's customary to perform a visual inspection of the plane, starting at the left wingtip and moving around clockwise.

```
{
  "manufacturer": "Cessna",
  "model": "172 Skyhawk",
  "seats": 4
}
```

This is a Cessna 172 Skyhawk, represented as JSON. The opening and closing curly braces ({}), aptly reminiscent of a fuselage, denote the start and end of the aircraft. Inside this top-level object, there are three attributes listed as key/value pairs:

Key	Value
`"manufacturer"`	“Cessna”
`"model"`	“172 Skyhawk”
`"seats"`	4

Note:

JSON, or **J**ava**S**cript **O**bject **N**otation, is a text-based format for representing information. It’s easy for both humans and computers to read and write, which has helped make it a ubiquitous standard on the web.

Building Out a Model

To interact programmatically with this aircraft (and others like it), we create a Swift type that fits the JSON representation, matching up each JSON value to its counterpart in Swift’s type system.

JSON can represent ordered structures (*arrays*) and unordered structures (*objects*), each of which may contain any combination of strings, numbers, or the values `true`, `false`, and `null`. Fortunately for us, each of these JSON types maps rather nicely onto something in Swift.

JSON	Swift Types
object	`Dictionary`[1]
array	`Array`
`true` / `false`	`Bool`
`null`	`Optional`[2]
string	`String`
number	`Int`, `Double`, et. al.[3]

1. Objects are unordered key-value pairs. They’re analogous to a Swift `Dictionary` with `String` key type and can be converted to and from types that conform to `Codable`.
2. `Codable` can automatically map `null` values to `nil` for properties with an `Optional` type.
3. JSON represents numbers as a sequence of digits, agnostic of any semantics; no distinction is made between integer and floating-point, fixed and variable-width, or binary and decimal formats.

   Each implementation chooses how to interpret number values. `JSONDecoder` (and its underlying

We start by defining a new structure, `Plane`, to correspond to the top-level object in the payload. In this example, we use a structure, but a class would work just as well. Each JSON attribute becomes a Swift property with the key as the property name and the value type as the property type. Both the `manufacturer` and `model` attributes have string values and map onto `String` properties with the same name. The `seats` attribute has a number value and maps onto the `seats` property of type `Int`.

```
{
  "manufacturer": "Cessna",
  "model": "172 Skyhawk",
  "seats": 4
}
```

```
struct Plane {
    var manufacturer: String
    var model: String
    var seats: Int
}
```

Now that we've defined a `Plane` structure, the next objective is to decode one from our JSON payload.

Introducing the `Codable` Protocol

Swift 4.0 introduces a new language feature called `Codable`, which vastly improves the experience of converting objects to and from a representation.

The best way to understand `Codable` is to look at its declaration:

```
typealias Codable = Decodable & Encodable
```

`Codable` is a *composite type* consisting of the `Decodable` and `Encodable` protocols.

---

processor, `JSONSerialization`) interprets number values using `NSNumber`, and provides conversions into most number types in the Swift Standard Library.

The `Decodable` protocol defines a single initializer:

```
init(from decoder: Decoder) throws
```

Types that conform to the `Decodable` protocol can be initialized by any `Decoder` type.

The `Encodable` protocol defines a single method:

```
func encode(to encoder: Encoder) throws
```

If a type conforms to the `Encodable` protocol, any `Encoder` type can create a representation for a value of that type.

Let's return to our `Plane` model and take `Decodable` out for a spin.

Adopting `Decodable` in the Model

You *adopt* a protocol by adding the protocol's name after the type's name, separated by a colon.

```
struct Plane: Decodable {
    var manufacturer: String
    var model: String
    var seats: Int
}
```

A type is said to *conform* to a protocol if it satisfies all the requirements of that protocol. For `Decodable`, the only requirement is for the type to have an implementation of `init(from:)`.

The `init(from:)` initializer takes a single `Decoder` argument. `Decoder` is a protocol that specifies the requirements for decoding a `Decodable` object from a representation. In order to accommodate a variety of data interchange formats, including JSON and property lists, both decoders and encoders use an abstraction called *containers*. A container is something that holds a value. It can hold a single value, or it can hold multiple values — either keyed, like a dictionary, or unkeyed, like an array.

Because the JSON payload has an object at the top level, we'll create a keyed container and then decode each property value by its respective key.

`Codable` expects a type called `CodingKeys` that conforms to the `CodingKey` protocol, which defines a mapping between Swift property names and container keys. This type is typically an enumeration with a `String` raw value, because keys are unique and represented by string values.

Let's create a `CodingKeys` enumeration for `Plane`:

```
struct Plane: Decodable {
    // ...

    private enum CodingKeys: String, CodingKey {
        case manufacturer
        case model
        case seats
    }
}
```

We don't need to provide an explicit raw value for any of the enumeration cases. because the names of each property are the same as the corresponding JSON key.

Next, in the `init(from:)` initializer, we create a keyed container by calling the `container(keyedBy:)` method on `decoder` and passing our `CodingKeys` type as an argument.[4]

4. The postfix `self` expression allows types to be used as values.

Finally, we initialize each property value by calling the `decode(_:,forKey:)` method on `container`:

```
init(from decoder: Decoder) throws {
  let container =
    try decoder.container(keyedBy: CodingKeys.self)

  self.manufacturer =
    try container.decode(String.self,
                         forKey: .manufacturer)
  self.model =
    try container.decode(String.self,
                         forKey: .model)
  self.seats =
    try container.decode(Int.self,
                         forKey: .seats)
}
```

We have a `Plane` model, and by conforming to the `Decodable` protocol, we can now create a `Plane` object from a JSON representation. We're now ready for take-off.

Decoding JSON Into a Model Object

Start by importing Foundation. We'll need it for `JSONDecoder` and `JSONEncoder`.

```
import Foundation
```

Apps typically load JSON from a network request or a local file. For simplicity, we can define the JSON directly in source code using another feature added in Swift 4.0: *multi-line string literals*.

```
let json = """
{
    "manufacturer": "Cessna",
    "model": "172 Skyhawk",
    "seats": 4,
}
""".data(using: .utf8)!
```

Multi-line string literals are delimited by triple quotation marks (`"""`). Unlike conventional string literals, the multi-line variant allows newline returns as well as unescaped quotation marks (`"`). This makes it perfectly suited for representing JSON in source code.

The `data(using:)` method converts a string into a `Data` value that can be passed to a decoder. This method returns an optional, and in most cases, a `guard` statement with a conditional assignment (`if-let`) would be preferred. However, because we're calling this method on a literal string value that we know will always produce valid UTF-8 data, we can use the forced unwrapping postfix operator (`!`) to get a nonoptional `Data` value.

Next, we create a `JSONDecoder` object and use it to call the `decode(_:from:)` method[5] . This method can throw an error. For expediency, we're using `try!` for now instead of doing proper error handling (we'll do it the right way on the next go around).

```
let decoder = JSONDecoder()

let plane = try! decoder.decode(Plane.self, from: json)
```

We have liftoff! Go ahead and check it out for yourself.

```
print(plane.manufacturer)
// Prints "Cessna"

print(plane.model)
// Prints "172 Skyhawk"

print(plan.seats)
// Prints "4"
```

5. Swift could infer the type of the decode method without supplying the type as the first argument, but a design decision was made to make this explicit to reduce ambiguity about what's happening.

Encoding a Model Object Into JSON

In order to complete the loop, we need to update `Plane` to adopt `Encodable`.

```
struct Plane: Decodable, Encodable { }
```

As we saw earlier, `Codable` is defined as a typealias for the composition of `Decodable` and `Encodable`. Therefore, we can simplify this further:

```
struct Plane: Codable { }
```

Next, we implement the `encode(to:)` method required by `Encodable`. As you might expect, `encode(to:)` reads much like a mirror image of `init(from:)`. Start by creating a container by calling the `container(keyedBy:)` method on `encoder` and passing `CodingKeys.self` as we did before. Here, `container` is a variable (`var` instead of `let`), because this method populates the `encoder` argument, which requires modification. For each property, we call `encode(_:forKey:)`, passing the property's value and its corresponding key.

```
func encode(to encoder: Encoder) throws {
    var container = encoder.container(keyedBy: CodingKeys.self)
    try container.encode(self.manufacturer,
                         forKey: .manufacturer)
    try container.encode(self.model,
                         forKey: .model)
    try container.encode(self.seats,
                         forKey: .seats)
}
```

We've done all of the necessary setup, and all systems are go. Let's make for our final approach and complete our loop:

```
let encoder = JSONEncoder()
let reencodedJSON = try! encoder.encode(plane)

print(String(data: reencodedJSON, encoding: .utf8)!)
// Prints "{"manufacturer":"Cessna",
//          "model":"172 Skyhawk",
//          "seats":4}"
```

Aces! Other than a lack of whitespace, what we got back from `JSONEncoder` is exactly what we originally put into `JSONDecoder`.[6]

We could keep going around and around, converting back and forth between model and representation to our heart's content. If you think about the demands of a real app as it brokers information requests between client and server, this is exactly the kind of guarantee we need to ensure that nothing is lost in translation.

Deleting Unnecessary Code

Now we're going to do something a bit unexpected: delete all of the code we've written so far to conform to `Codable`. This should leave only the structure and property declarations.

```
struct Plane: Codable {
    var manufacturer: String
    var model: String
    var seats: Int
}
```

Go ahead and try running the code to decode and encode JSON. What changed? *Absolutely nothing.* Which leads us to the true killer feature of `Codable`:

---

6. JSON isn't whitespace sensitive, but you might be. If you find compressed output to be aesthetically challenging, set the `outputFormatting` property of `JSONEncoder` to `.prettyPrinted`.

Swift automatically synthesizes conformance for `Decodable` and `Encodable`.

So long as a type adopts the protocol in its declaration — that is, not in an extension — and each of its properties has a type that conforms, everything is taken care of for you.

We can see for ourselves that `Plane` meets all of the criteria for synthesizing conformance to `Codable`[7] :

- ☑ `Plane` adopts `Codable` in its type declaration.
- ☑ Each of the properties in `Plane` has types conforming to `Codable`.

Most built-in types in the Swift Standard Library and many types in the Foundation framework conform to `Codable`. So most of the time, it's just up to you to keep the `Codable` party going.

> **Note:**
>
> Aspiring pilots may be alarmed when — halfway into their introductory flight, cruising at a few thousand feet — their flight instructor tells them to cut power to the engine to see what happens. (Just as here, absolutely nothing. You just glide.) Not all CFIs do this, but it's a visceral demonstration of the physics of flight.

If you're miffed about writing all of that code when it wasn't necessary, please be advised: there are occasions in which you do need to implement conformance manually. We'll be covering that in Chapter 3.

For now, let's collect ourselves and get ready for another loop.

---

7. To be more precise, Swift automatically synthesizes conformance `Decodable` and `Encodable`. Think of `Codable` synthesis as two separate passes: one for `Decodable` and another for `Encodable`.

Flying in Formation

Objects don't always fly solo. In fact, it's quite common for payloads to have arrays of objects in them.

For example, a JSON response may have an array as its top-level value:

```
[
  {
    "manufacturer": "Cessna",
    "model": "172 Skyhawk",
    "seats": 4
  },
  {
    "manufacturer": "Piper",
    "model": "PA-28 Cherokee",
    "seats": 4
  }
]
```

Decoding this into an array of `Plane` objects couldn't be simpler: take our call to `decode(_:from:)` from before, and replace `Plane.self` with `[Plane].self`. This changes the decoded type from a `Plane` object to an array of `Plane` objects.

```
let planes = try! decoder.decode([Plane].self, from: json)
```

`[Plane]` is syntactic shorthand for `Array<Plane>`, or an `Array` with the generic constraint that its elements are `Plane` objects. Why does this work? It's thanks to *conditional conformance* — another feature of Swift 4.

```
// swift/stdlib/public/core/Codable.swift.gyb
extension Array : Decodable where Element : Decodable {
    // ...
}
```

Although top-level arrays are technically valid JSON, it's considered best practice to always have an object at the top level instead. For example, the following JSON has an array keyed off the top-level object at `"planes"`:

```
{
  "planes": [
    {
      "manufacturer": "Cessna",
      "model": "172 Skyhawk",
      "seats": 4
    },
    {
      "manufacturer": "Piper",
      "model": "PA-28 Cherokee",
      "seats": 4
    }
  ]
}
```

Like `Array`, the `Dictionary` type conditionally conforms to `Decodable` if its associated `KeyType` and `ValueType` conform to `Decodable`:

```
// swift/stdlib/public/core/Codable.swift.gyb
extension Dictionary : Decodable where Key : Decodable,
                                       Value : Decodable {
    // ...
}
```

Because of this, you can change the same call to `decode(_:from:)` from before and pass `[String: [Plane]].self` (shorthand for `Dictionary<String, Array<Plane>>`) instead:

```
let keyedPlanes = try! decoder.decode([String: [Plane]].self,
                                      from: json)
let planes = keyedPlanes["planes"]
```

Alternatively, you can create a new type that conforms to `Decodable` and has a property whose name matches the JSON key and has a value of type `[Plane]`, and pass that to `decode(_:from:)`:

```
struct Fleet: Decodable {
    var planes: [Plane]
}

let fleet = try! decoder.decode(Fleet.self, from: json)
let planes = fleet.planes
```

You typically do this when the structure of your payload is semantically meaningful. If objects are nested arbitrarily, such as to namespace keys, you should probably just use a standard collection class.

Trying Our Best with Error Handling

Our first couple go-arounds with `Codable` went well, but our technique was a bit sloppy with those `try!` keywords. For this last lap, let's take a moment to show how the pros do it in real applications.

Swift functions and initializers marked with the `throws` keyword may generate an error instead of returning a value. Both the `Decodable` initializer `init(from:)` and the `Encodable` method `encode(to:)` are marked with the `throws` keyword — which makes sense, because it may not always be possible to complete the operation successfully. When decoding a representation into an object, the data might be corrupted, a key might be missing, there might be a type mismatch, or a nonoptional value might be missing. It's less typical for there to be problems during encoding; it really depends on the format.

Swift requires all errors to be handled one way or another. So when you call an expression marked with the `throws` keyword, you need to prefix it with the `try` operator or either of its variants.

```
do {
    let plane = try decoder.decode(Plane.self, from: json)
} catch {
    print("Error decoding JSON: \(error)")
}
```

Note:

For information about how to communicate errors to users, see the Human Interface Guidelines.

You can use the optional-try operator (`try?`) to convert a throwing expression into an optional expression. That is, the expression returns `nil` if an error occurs, otherwise, it returns the expression value in an optional. If you aren't doing anything meaningful in your error handling, this can be a useful shorthand.

```
if let plane = try? decoder.decode(Plane.self, from: json) {
    print(plane.model)
}
```

Alternatively, there's the forced-try operator (`try!`), which we've been using up until this point. Following Swift's language conventions, the trailing *"bang"*, or exclamation mark (`!`), indicates unsafe behavior. If you use the forced-try operator on an expression that results in an error, the program exits immediately. Because sudden app termination makes for a lousy user experience, you generally avoid `try!` in app code. However, for the purposes of exploring a new concept — such as in a Playground — failing fast can actually be to your advantage.

Thus concludes our first flight with `Codable`. If you feel like your head is spinning, don't worry — you'll get more comfortable with practice.

Things get even more exciting in Chapter 2. We'll apply what we learned here to a more challenging payload, and learn strategies for maximizing how much the compiler does for us.

Recap

- `Codable` is a *composite type* consisting of the `Decodable` and `Encodable` protocols.
- Swift automatically synthesizes conformance for `Decodable` and `Encodable` if a type adopts conformance in its declaration, and each of its stored properties also conforms to that protocol.
- An `Array` or `Dictionary` conforms to `Codable` if its associated element type is `Codable`-conforming.

Chapter 2:
Holding Patterns

In the previous chapter, you saw how `Codable` works under ideal conditions: each property name matched its respective key name, each property type was directly encodable in JSON, and all values were guaranteed.

Of course, you can't expect things to always go so smoothly.

In this chapter, you'll learn to adapt `Codable` to extenuating circumstances, including how to decode dates and enumerated values, how to access nested keys, and what to do about `null` values.

Translating between Swift models and JSON representations isn't the most interesting work, but it's necessary for getting where we want to go. If you want to do anything interesting, you'll need to share information with others. This is as true in programming as it is in flying, which segues nicely into our example for this chapter: flight plans.

Making a Flight Plan

The Federal Aviation Administration requires pilots to submit a flight plan prior to take off. A flight plan includes information like aircraft identification, estimated departure time, and the flight route.

To file a flight plan, you can fill out and submit a hard-copy of FAA Form 7233-1 to your local flight service station. You can also call Flight Services (1-800-WX-BRIEF) and talk to a flight specialist to file your flight plan with them. But if the mere mention of paperwork causes your hands to cramp preemptively, or if the very idea of communicating with a human on the phone fills you with dread, fear not, dear reader — there's a third option: You can file flight plans electronically!

Not content to use something off the shelf, let's take a look at how we might implement a critical piece of functionality using what we learned about `Codable` so far.

Here's a simplified flight plan, represented as JSON:

```
{
    "aircraft": {
        "identification": "NA12345",
        "color": "Blue/White"
    },
    "route": ["KTTD", "KHIO"],
    "flight_rules": "VFR",
    "departure_time": {
        "proposed": "2018-04-20T14:15:00-07:00",
        "actual": "2018-04-20T14:20:00-07:00"
    },
    "remarks": null
}
```

Compared to our last JSON payload, this is quite a bit more complicated. Here we have nested objects, a nested array, timestamps, and a `null` value. Despite how complex this payload may look, it's still possible to create an implementation of `FlightPlan` that automatically synthesizes conformance to `Codable`.

Tip:

When you need a type to conform to `Decodable` and/or `Encodable`, try to avoid implementing conformance manually. Every line of code that you have to write has an opportunity cost — that is, there's always something more important you could be doing instead. There's also an associated risk; especially for a rote task like this, each line of code is an opportunity to introduce a subtle error, whether by mistyping the name of a key, or copy-pasting and not making all the necessary changes between similar lines.

For this example, we'll look at how to create a `FlightPlan` structure that conforms to `Decodable`. Let's traverse the JSON payload to get a lay of the land.

Decoding a Nested Object

Nested in the top-level object at the `"aircraft"` key is an object of its own.

```
"aircraft": {
    "identification": "NA12345",
    "color": "Blue/White"
}
```

Unlike the `Plane` in the previous example, the relevant details for a flight plan are those that help an aircraft be identified.

Private aircraft are identified by their registration number (commonly known as the "N" Number in the US because registration numbers begin with the letter N). Providing some guidance about visual appearance vis-à-vis color makes it easier for an aircraft to be matched visually on the ground or in the sky.

Note:

Commercial aircraft use the operating agency and flight number, such as "AA10" for American Airlines Flight 10 (LAX → JFK). Military aircraft have a tactical call sign, such as "SWIFT41".

We could define `aircraftIdentification` and `aircraftColor` properties on `FlightPlan`, but it's both easier and more convenient to define a new `Aircraft` structure that can be decoded from the nested representation.

```
struct Aircraft: Decodable {
    var identification: String
    var color: String
}
```

The corresponding property on `FlightPlan` is also straightforward:

```
var aircraft: Aircraft
```

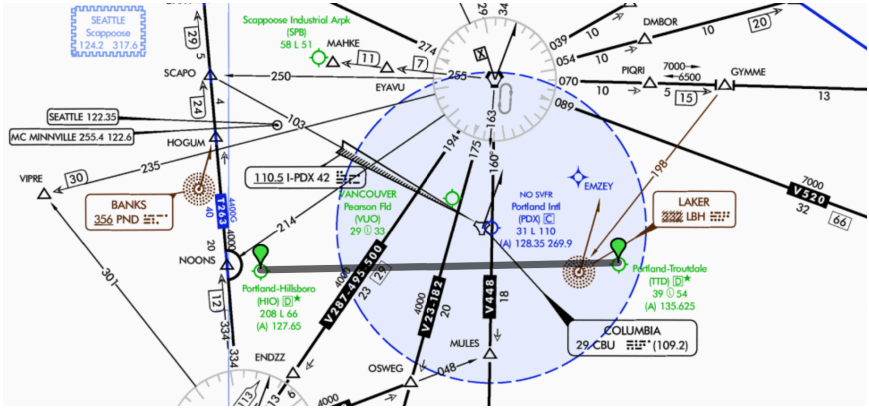
When the decoder encounters the key `"aircraft"`, it will call the synthesized `init(from:)` initializer using the JSON object value to create a new `Aircraft` value, and then set that value to the `aircraft` property.

Decoding an Array of Strings

```
"route": ["KTTD", "KHIO"]
```

Each point in a flight route is identified by a 4-character code.[1]

KTTD and KHIO are both regional airports near Portland, Oregon. KTTD is the code for Portland-Troutdale Airport, located 10 miles east of the city, and KHIO is the code for Portland-Hillsboro Airport, about 15 miles to the southwest. This route takes us over the heart of the city, offering gorgeous views of Forest Park and the Willamette River along the way.



Flight Route Data Courtesy of SkyVector

1. The term is “airdrome” if you want to be technical about it (or “aerodrome” if you want to be British about it).

As we saw when decoding an array of `Plane` objects, Swift automatically decodes arrays of decodable types. The same is true when that array is a property of a type.

```
var route: [String]
```

Decoding an Enumeration from a String

```
"flight_rules": "VFR"
```

There are generally two sets of regulations for flying an aircraft: VFR, or *visual flight rules*, and IFR, or *instrument flight rules*.

> **Note:**
>
> VFR applies on clear days with good visibility and limited cloud cover, when the pilot can adjust altitude and direction based on what they see outside the cockpit. In the US, any certified pilot can operate an aircraft under visual flight rules; instrument flight typically requires additional certification.

Because flight rules specify how aircraft are expected to operate, pilots need to declare their intention on their flight plan.

In the JSON payload, flight rules are encoded as a string associated with the `flight_rules` key. Because there are a small, closed set of possible values, flight rules are a great candidate for an enumerated type.

```
enum FlightRules: String, Decodable {
    case visual = "VFR"
    case instrument = "IFR"
}
```

The `FlightRules` enumeration declares a `String` raw value type and has cases for both visual and instrument flight rules. By having a raw value type and adopting `Decodable` in its declaration, `FlightRules` automatically conforms to `Decodable` without any additional code.

When decoding from a representation, the `FlightRules` type will attempt to initialize by raw value. If the value is anything other than `"VFR"` or `"IFR"`, initialization and decoding will fail.

```
var flightRules: FlightRules
```

Handling Key / Property Name Mismatches

Something else about the `"flight_rules"` key may have caught your attention: the key itself.

A common convention for web services is to use `snake_case` for JSON object keys — that is, lowercase with underscores (_) instead of spaces. In contrast, the Swift API Design Guidelines prescribe `UpperCamelCase` case conventions for types and `lowerCamelCase` for everything else.

By popular demand, Swift 4.1 added the `keyDecodingStrategy` property to `JSONDecoder` as a convenient way to convert between various conventions. In this example, we could specify the `convertFromSnakeCase` key decoding strategy to automatically map JSON keys like `"flight_rules"` to the property name `flightRules`.

```
var decoder = JSONDecoder()
decoder.keyDecodingStrategy = .convertFromSnakeCase
```

This convenience, however, comes at the price of trusting JSON responses to be consistent which, not to disparage our fine colleagues working on the back-end, is not a wager you'd be wise to take. This also introduces some cognitive overhead by splitting concerns across multiple call sites. It may be old-fashioned to say so, but sometimes a model's got to look after itself.

A more sustainable strategy is to create an explicit mapping using `CodingKeys`. When synthesizing `Decodable` conformance, Swift looks for a subtype named `CodingKeys` that conforms to the `Coding`

Key protocol. Most often, this is an enumeration marked `private` with `String` raw values that are explicitly set for any cases that have a name mismatch.

```
extension FlightPlan {
    private enum CodingKeys: String, CodingKey {
        case aircraft
        case flightRules = "flight_rules"
        case route
        case departureDates = "departure_time"
        case remarks
    }
}
```

You won't feel particularly clever doing it this way but think of this small amount of boilerplate as an investment in your future sanity. `Codable` already does so much of the work for you, so why not put in the extra 30 seconds to get things right?

Decoding Dates from Timestamps

JSON doesn't have a native date type. Instead, there are two common ways to represent dates:

1. As a string with a formatted timestamp
2. As a number with seconds (or milliseconds or nanoseconds) since the epoch

A formatted timestamp is typically the way to go for most applications. For a few extra bytes (~25 compared to 8 for a number), you get something that's human-readable, adaptable, and unambiguous.

That said, clients rarely get any say in the matter, and have to work with what they're given.

Lucky for us, `JSONDecoder` provides a convenient mechanism for interpreting dates no matter how they're represented. Just set the `dateDecodingStrategy` property according to what you'd expect.

In our case, dates are formatted as ISO 8601 timestamps, so we use the `.iso8601` date decoding strategy.

```
var decoder = JSONDecoder()
decoder.dateDecodingStrategy = .iso8601
```

The additional challenge with the departure times is that the values are nested in the JSON response.

```
"departure_time": {
    "proposed": "2018-04-20T14:15:00-07:00",
    "actual": "2018-04-20T14:20:00-07:00"
}
```

Whereas the JSON object for `"aircraft"` signifies a distinct entity, the `"departure_time"` object merely organizes related fields. An alternative to creating an intermediate type is to declare a property of type `[String: Date]` and define computed properties for the specific keys you're interested in.

```
private var departureDates: [String: Date]

var proposedDepartureDate: Date? {
    return departureDates["proposed"]
}

var actualDepartureDate: Date? {
    return departureDates["actual"]
}
```

By making `departureDates` private and exposing `proposedDepartureDate` and `actualDepartureDate`, we hide the implementation details from callers.

Decoding Null Values with `Optional` or Default Values

JSON objects can specify a `null` value for a key. Such is the case for the `"remarks"` attribute.

But that's nothing to worry about, because Swift has first-class support for handling null values: `Optional`. Use an `Optional` for the expected type, and the synthesized implementation of `Decodable` will work automatically.

```
var remarks: String?
```

Final Approach

Now that we've covered all of the edge cases, let's put everything together and make our final approach with `FlightPlan`.

```
struct FlightPlan: Decodable {
    var aircraft: Aircraft

    var route: [String]

    var flightRules: FlightRules

    private var departureDates: [String: Date]

    var proposedDepartureDate: Date? {
        return departureDates["proposed"]
    }

    var actualDepartureDate: Date? {
        return departureDates["actual"]
    }

    var remarks: String?

    private enum CodingKeys: String, CodingKey {
        case aircraft
        case flightRules = "flight_rules"
        case route
        case departureDates = "departure_time"
        case remarks
    }
}
```

Now let's bring the JSON payload in. Just as before, we take advantage of the Swift multi-line string literal to define our JSON directly in our Swift source code.

```
let json = """
{
    "aircraft": {
        "identification": "NA12345",
        "color": "Blue/White"
    },
    "route": ["KTTD", "KHIO"],
    "departure_time": {
        "proposed": "2018-04-20T14:15:00-07:00",
        "actual": "2018-04-20T14:20:00-07:00"
    },
    "flight_rules": "IFR",
    "remarks": null
}
""".data(using: .utf8)!
```

We then create a `JSONDecoder` and set the `dateDecodingStrategy` to `.iso8601`.

```
var decoder = JSONDecoder()
decoder.dateDecodingStrategy = .iso8601
```

Finally, we call the `decode(_:from:)` method on `decoder` to create a `FlightPlan` from the JSON data:

```
let plan = try! decoder.decode(FlightPlan.self, from: json)

print(plan.aircraft.identification)
// Prints "NA12345"

print(plan.actualDepartureDate!)
// Prints "2018-04-20T14:20:00-07:00"
```

With working code in hand, we're ready to file our flight plan and spread our knowledge of `Codable` to even the most far-flung reaches of our code bases.

We wrap up our introduction to `Codable` in the next chapter by discussing the rare but annoying situations in which you have to implement `Decodable` or `Encodable` conformance yourself.

Recap

- When you need a type to conform to `Decodable` and/or `Encodable`, try to avoid implementing conformance manually.
- Define types for nested objects when it makes sense to do so.
- Reconcile differences in property names and key names by defining a private `CodingKeys` enumeration and overriding case raw values as needed.
- Use computed properties to coordinate access of decoded properties that have an inconvenient structure or need to be processed further.

Chapter 3:
Taking the Controls

As with any useful abstraction, `Codable` makes simple tasks easy, and complex tasks possible. When you declare a type that adopts `Encodable` or `Decodable` and the compiler synthesizes conformance, it's like magic.

Sure, you may need to nudge things a bit on occasion by specifying custom `CodingKey` values or mediating access to decoded information through a computed property. But these small adjustments are an order of magnitude better than writing all that boilerplate code.

However, even the best abstraction has its breaking point, and in this respect, `Codable` is no exception. Sometimes you have no choice but to write `init(from:)` or `encode(to:)` yourself (usually it's the fault of a misbehaving web service).

It's not that implementing either of these requirements is particularly labor-intensive, but if you're used to having everything taken care of for you by the compiler, you may need a little guidance on what needs to happen.

This chapter dives into some of the scenarios in which implementing `Codable` manually is necessary. Armed with this knowledge, you'll be sure to breeze through your next patch of rough skies with plenty of fuel and motivation to spare.

Decoding Unknown Keys

One unfortunate pattern that you might encounter in the wild is to have dynamic keys in a JSON payload. For example, consider this representation of a flight route:

```
{
  "points": ["KSQL", "KWVI"],
  "KSQL": {
    "code": "KSQL",
    "name": "San Carlos Airport"
  },
  "KWVI": {
    "code": "KWVI",
    "name": "Watsonville Municipal Airport"
  }
}
```

The `"points"` key has an array of string values that corresponds to the other keys in the top-level object. This pattern may be convenient in some languages, but not as much in Swift with `Codable`. That said, there's still a reasonably nice way to deal with this.

As always, we start with a solid model. Declare a `Route` type with a nested `Airport` type — both of which conform to `Decodable`.

```
struct Route: Decodable {
    struct Airport: Decodable {
        var code: String
        var name: String
    }

    var points: [Airport]
}
```

Up until this point, `CodingKeys` has been implemented as an enumeration of raw `String` values. However, the `CodingKeys` requirement can also be satisfied by a structure, and that structure can be conditionally initialized with arbitrary `Int` or `String` values.

```
private struct CodingKeys: CodingKey {
    var stringValue: String

    var intValue: Int? {
        return nil
    }

    init?(stringValue: String) {
        self.stringValue = stringValue
    }

    init?(intValue: Int) {
        return nil
    }

    static let points =
        CodingKeys(stringValue: "points")!
}
```

In this case, we aren't using integer-based keys, so we stub out just enough implementation to fulfill the needs of the protocol.

Now, in the `init(from:)` initializer, we can dynamically build up a list of airports based on the array of codes decoded for the `.points` key.

```
init(from coder: Decoder) throws {
    let container =
        try coder.container(keyedBy: CodingKeys.self)

    var points: [Airport] = []
    let codes = try container.decode([String].self,
                                     forKey: .points)
    for code in codes {
        let key = CodingKeys(stringValue: code)!
        let airport =
            try container.decode(Airport.self,
                                 forKey: key)
        points.append(airport)
    }
    self.points = points
}
```

Tip: You can initialize the `points` property in this example more succinctly by using the `map(_:)` method instead of `for-in` loop. This technique is discussed in Chapter 5.

Decoding Indeterminate Types

"Look — up in the sky! It's a bird! It's a plane! It's... a heterogeneous collection of `Decodable` elements!"

```
[
  {
    "type": "bird",
    "genus": "Chaetura",
    "species": "Vauxi"
  },
  {
    "type": "plane",
    "identifier": "NA12345"
  }
]
```

You know it's going to be a bad day when your JSON response has a `"type"` key in it. Playing loosey-goosey with the type system may be a calling card of web languages like JavaScript and Ruby, but Swift will have absolutely none of that.

Swift *demands* formal types. A bird is a `Bird`. A plane is a `Plane`.

```
struct Bird: Decodable {
    var genus: String
    var species: String
}
```

```
struct Plane: Decodable {
    var identifier: String
}
```

And when something can be either, then it is `Either`.

```
enum Either<T, U> {
    case left(T)
    case right(U)
}
```

One way to cope with Swift's *horror incognito* is to create a conditional extension on `Either` types whose associated types are `Decodable`:

```
extension Either: Decodable where T: Decodable,
                                  U: Decodable
{
    init(from decoder: Decoder) throws {
        if let value = try? T(from: decoder) {
            self = .left(value)
        } else if let value = try? U(from: decoder) {
            self = .right(value)
        } else {
            let context = DecodingError.Context(
                codingPath: decoder.codingPath,
                debugDescription:
                    "Cannot decode \(T.self) or \(U.self)"
            )
            throw DecodingError.dataCorrupted(context)
        }
    }
}
```

Now, when trapped in a dilemma, Swift can face its fears without relinquishing type safety:

```
let decoder = JSONDecoder()
let objects = try! decoder.decode([Either<Bird, Plane>].self,
from: json)

for object in objects {
    switch object {
    case .left(let bird):
        print("Poo-tee-weet? It's \(bird.genus)
\(bird.species)!")
    case .right(let plane):
        print("Vooooooooooooooom! It's \(plane.identifier)!")
    }
}
```

However, this approach is only suitable for two possibilities — and rare is the case when a dilemma could not also be a trilemma or more. In such a case, we may look for a more general solution that can decode any arbitrary type.

Decoding Arbitrary Types

If you're migrating from `JSONSerialization` to `JSONDecoder`, you may be struck by how inflexible `Codable` is by comparison. Whereas pretty much everything with `JSONSerialization` is a `[String : Any]` dictionary, you'd be hard-pressed to encounter one in a `Decodable` type.

Consider a `Report` type that has the properties `title`, `body` and `metadata`.

```
struct Report {
    var title: String
    var body: String
    var metadata: [String: Any]
}
```

Tip: Every `metadata` or `userInfo` property is a compromise between correctness, completeness, and flexibility — an escape hatch for unthinkable use cases; a concession to one's own hubris.

Ideally, any attribute that might be specified in a `metadata` dictionary could be expressed through properties. But if there's no way to know what might be specified, or you've just given up on the whole endeavor, you might relent and add a junk drawer to your model.

The `[String: Any]` type doesn't conform to `Codable`, because its value type, `Any`, doesn't (because otherwise, every type would be `Codable`, and that's just not the case). Unfortunately, `[String : Codable]` doesn't work either, because a concrete value type is needed.

If you knew that all metadata values were the same type, like `String`, you could simply change the type to `[String: String]`. However, if you need to be able to encode heterogenous values — including nested arrays and dictionaries — then you need something else.

One possible solution is to create a type-erased, `Decodable`-conforming type with an interface similar to `AnyHashable`. Call it `AnyDecodable`.[1] With this type, you can have `Codable` conformance synthesized without diminishing the functionality of the `metadata` property.

```
struct Report: Decodable {
    var title: String
    var body: String
    var metadata: [String: AnyDecodable]
}
```

1. A reference implementation for `AnyDecodable` can be found here.

Decoding Multiple Representations

If your app incorporates information from multiple data sources, each data provider is likely to have their own conventions for how keys are named and how values are formatted.

For example, an app that checks aviation fuel prices at North American airports may consult a variety of American and Canadian web services to get this information. Aside from the inevitable variance in how data is structured, you may have to account for price rates in both USD/gallon and CAD/liter.

```
// American
// (prices in USD/gallon)
[
  {
    "fuel": "100LL",
    "price": 5.6
  },
  {
    "fuel": "Jet A",
    "price": 4.1
  }
]
```

```
// Canadian
// (prices in CAD/liter)
{
  "fuels": [
    {
      "type": "100LL",
      "price": 2.54
    },
    {
      "type": "Jet A",
      "price": 3.14
    },
    {
      "type": "Jet B",
      "price": 3.03
    }
  ]
}
```

When faced with the problem of modeling disparate information sources, a good first step is to extract whatever the data sources agree on — anything that has a universal standard or strong conventions. In this case, the small, fixed set of fuel options is nicely modeled as an enumeration:

```
enum Fuel: String, Decodable {
    case jetA = "Jet A"
    case jetB = "Jet B"
    case oneHundredLowLead = "100LL"
}
```

To fully take advantage of `Codable`, create a model for each kind of representation that your app encounters. Even when conformance can't be synthesized, it's nice to isolate those concerns.

```
struct AmericanFuelPrice:
        Decodable
{
  let fuel: Fuel

  /// USD / gallon
  let price: Decimal
}
```

```
struct CanadianFuelPrice:
        Decodable
{
  let type: Fuel

  /// CAD / liter
  let price: Decimal
}
```

From here, it's just a question of how to reconcile these types.

One option is to create a canonical structure and define overloaded initializers that accept these representative types:

```
struct FuelPrice {
  let type: Fuel
  let pricePerLiter: Decimal
  let currency: String
}
```

```
extension FuelPrice {
  init(_ other:
         AmericanFuelPrice)
  {
    self.type = other.fuel
    self.pricePerLiter =
        other.price /
        3.78541
    self.currency = "USD"
  }
}
```

```
extension FuelPrice {
  init(_ other:
         CanadianFuelPrice)
  {
    self.type = other.type
    self.pricePerLiter =
        other.price
    self.currency = "CAD"
  }
}
```

This works best for a small number of representations.

If you're working across many different data sources you might instead define a canonical `FuelPrice` type as a protocol, and then conform each representative `Decodable` type to it in an extension:

```
protocol FuelPrice {
  var type: Fuel { get }
  var pricePerLiter: Decimal { get }
  var currency: String { get }
}
```

```
extension AmericanFuelPrice:
          FuelPrice
{
  var type: Fuel {
    return self.fuel
  }

  var pricePerLiter: Decimal
{
    return self.price /
             3.78541
  }

  var currency: String {
    return "USD"
  }
}
```

```
extension CanadianFuelPrice:
          FuelPrice
{
  var pricePerLiter: Decimal
{
    return self.price
  }

  var currency: String {
    return "CAD"
  }
}
```

Tip: When specifying prices, always specify the currency and store values using the `Decimal` type rather than `Double` to avoid loss of precision.

For an in-depth discussion of how to work with money in your app, please refer to our *Guide to Swift Numbers*.

Decoding Inherited Types

Class types don't always get along with `Decodable`.

This is an unfortunate consequence of Swift's rather complex initialization behavior. Because the `Decodable` protocol requires

a designated initializer, `init(from:)`, and designated initializers can't be declared in an extension, you're pretty much limited to adopting `Decodable` in the class declaration.[2]

Adding inheritance into the mix only complicates the matter further.

Consider the following models for an `EconomySeat` and `PremiumEconomySeat`:

```
class EconomySeat: Decodable {
    var number: Int
    var letter: String
}

class PremiumEconomySeat: EconomySeat {
    var mealPreference: String?
}
```

If we were to attempt to decode a `PremiumEconomySeat` object from JSON containing expected, nonnull values for the keys `"number"`, `"letter"`, and `"mealPreference"`, the resulting `PremiumEconomySeat` would have a `nil` value for `mealPreference`. Why? Because `PremiumEconomySeat` inherits the synthesized implementation of `init(from:)` directly from its superclass, without any accommodations for the additional property in the subclass.

2. Alternatively, a class declared as `final` can implement `init(from:)` as a convenience initializer in an extension.

In order to get the correct behavior, you must implement `CodingKeys` and `init(from:)` for the subclass.

```
class EconomySeat: Decodable {
  var number: Int
  var letter: String
}

class PremiumEconomySeat: EconomySeat {
  var mealPreference: String?

  private enum CodingKeys: String, CodingKey {
      case mealPreference
  }

  required init(from decoder: Decoder) throws {
      let container =
        try decoder.container(keyedBy: CodingKeys.self)
      self.mealPreference =
        try container.decodeIfPresent(
                String.self,
                forKey: .mealPreference
        )
      try super.init(from: decoder)
  }
}
```

Note: When traveling internationally in economy class, select the *Hindu Vegetarian* meal option within 24 hours of departure. It's often much better than the standard fare — and, as an added bonus, you get your meal before everyone else!

Decoding from Different Types of Values

For all the flack that developers on the client-side might give our back-end colleagues, we can at least credit them for *not* being a servant of *Xel'lotath, other god of madness and insanity*.

Because if they *were*, and indeed sought to *render all existence into chaos*, then you'd definitely see more JSON payloads like this:

```
{
  "coordinates": [
    {
      "latitude": 37.332,
      "longitude": -122.011
    },
    [-122.011, 37.332],
    "37.332, -122.011"
  ]
}
```

If we were to encounter such an abhorrent payload, how would we handle values that could be objects, arrays, or single values?

Actually, it's not that difficult — it just takes a bit of effort:

First, try to create a keyed container.

```
if let container =
    try? decoder.container(keyedBy: CodingKeys.self)
{
    self.latitude =
        try container.decode(Double.self, forKey: .latitude)
    self.longitude =
        try container.decode(Double.self, forKey: .longitude)
    self.elevation =
        try container.decodeIfPresent(Double.self,
                                      forKey: .elevation)
}
```

If that doesn't work, try an unkeyed container.

```
else if var container = try? decoder.unkeyedContainer() {
    self.longitude = try container.decode(Double.self)
    self.latitude = try container.decode(Double.self)
    self.elevation = try container.decodeIfPresent(Double.self)
}
```

Otherwise, try a single value container.

```
else if let container = try? decoder.singleValueContainer() {
    let string = try container.decode(String.self)
    let components =
        string.trimmingCharacters(
            in: CharacterSet.decimalDigits.inverted
        ).components(separatedBy: ",")
    guard components.count == 2,
        let longitude = Double(components[0]),
        let latitude = Double(components[1])
    else {
        throw DecodingError.dataCorruptedError(
                in: container,
                debugDescription: "Invalid coordinate string"
                )
    }

    self.latitude = latitude
    self.longitude = longitude
    self.elevation = nil
}
```

And if all else fails, give up and throw an error.

```
else {
    let context = DecodingError.Context(
        codingPath: decoder.codingPath,
        debugDescription: "Unable to decode Coordinate"
    )
    throw DecodingError.dataCorrupted(context)
}
```

Once you have the appropriate container, it's just a matter of extracting values as appropriate.

Configuring How Encodable Types Are Represented

It's not uncommon to work with models that have multiple standard or conventional ways to be represented.

When decoding such values, you may attempt to determine the format before parsing them or you may specify a strategy for interpreting values, as is the case for `dateDecodingStrategy` for `JSONDecoder`.

When encoding these kinds of values, though, you may want the option to specify which format to use.

Consider the following `Pixel` type that represents a color value by encoding the intensities of red, green, and blue light.

```
struct Pixel {
    var red: UInt8
    var green: UInt8
    var blue: UInt8
}
```

There are many ways to represent colors:

RGB hexadecimal	`#FA8072`
RGB functional	`rgb(216, 191, 216)`
HSL functional	`hsl(210.1, 100%, 50%)`
Name	`LemonChiffon`

To customize how `Pixel` values are encoded, start by declaring a `ColorEncodingStrategy` enumeration.

```
enum ColorEncodingStrategy {
    case rgb
    case hexadecimal(hash: Bool)
}
```

Next, extend `CodingUserInfoKey` and add a new type constant, `.colorEncodingStrategy`.

```
extension CodingUserInfoKey {
    static let colorEncodingStrategy =
        CodingUserInfoKey(rawValue: "colorEncodingStrategy")!
}
```

You can pass a desired `ColorEncodingStrategy` to the `Encodable` type by setting the `.colorEncodingStrategy` on the encoder's `userInfo` property.

```
let encoder = JSONEncoder()
encoder.userInfo[.colorEncodingStrategy] =
    ColorEncodingStrategy.hexadecimal(hash: true)

let cyan = Pixel(red: 0, green: 255, blue: 255)
let magenta = Pixel(red: 255, green: 0, blue: 255)
let yellow = Pixel(red: 255, green: 255, blue: 0)
let black = Pixel(red: 0, green: 0, blue: 0)

let json = try! encoder.encode([cyan, magenta, yellow, black])
// ["#00FFFF", "#FF00FF", "#FFFF00", "#000000"]
```

The implementation of `encode(to:)` can then read this key on `userInfo` and configure its behavior accordingly.

```
extension Pixel: Encodable {
  func encode(to encoder: Encoder) throws {
    var container = encoder.singleValueContainer()

    switch encoder.userInfo[.colorEncodingStrategy]
            as? ColorEncodingStrategy
    {
    case let .hexadecimal(hash)?:
      try container.encode(
          (hash ? "#" : "") +
          String(format: "%02X%02X%02X",
                 red, green, blue)
      )
    default:
      try container.encode(
          String(format: "rgb(%d, %d, %d)",
                 red, green, blue)
      )
    }
  }
}
```

Tip: The `case` statement in the `switch` statement above uses the `?` suffix to pattern match the `Optional` type returned when attempting to look up the `.colorEncodingStrategy` key in `userInfo`.

These are just some of the issues you may encounter in your journeys with `Codable`. Much of the time, there's a reasonable way to adapt `Codable` to the particulars of your situation with minimal implementation.

If you find that `Encodable` or `Decodable` just isn't working for you in a situation, know that you can always pull the ripcord and use `JSONSerialization`.

To quote Chuck Yeager:

> *If you can walk away from a landing, it's a good landing.*
> *If you use the airplane the next day, it's an outstanding landing.*

Recap

- It's not always possible for the compiler to synthesize `Encodable` or `Decodable` conformance. In these situations, you're responsible for satisfying the protocol requirements.
- Create a custom `CodingKeys` structure to decode keys that aren't known at build time.
- Use `Either` or `AnyDecodable` to decode representations of heterogeneous collections.
- Create a `Decodable` type for each representation format your app works with, and define a canonical type that reconciles differences between these formats.
- If a subclass of an `Encodable` or `Decodable` type has additional stored properties, remember to override the inherited implementation of `init(from:)` or `encode(to:)`.
- Define custom `userInfo` coding keys to pass configuration options to models through the encoder.

Chapter 4:

Building a Music Store App with iTunes Search API

Between gates 17 and 19 in concourse D of Cupertino International Airport lies an unassuming little kiosk that sells music. Though celebrated in its day, it stands as a relic of a time that now seems but a distant memory, before streaming services and ubiquitous cloud storage and LTE networks... A time when you could fit 1000 songs in your pocket and be content with that. Insofar as music can be bought and sold any more than one might buy an emotion or the wind, this store offers millions of tracks for a flat price of 99¢ each.

The shopping experience is simple and intuitive: You enter a search term, and a list of results comes up moments later. Select any one of those results, and you can hear a free preview of the track before deciding whether or not to purchase it.

In this chapter, we'll build out the core functionality of such an app using `Codable` to process results from the Apple iTunes Search API. We'll start by getting acquainted with the API through its documentation. Once we know what endpoints are available and what generally to expect back, we can translate our understanding into code, and use what we've learned about `Codable` to parse API responses into Swift model objects. And then to wrap things up, we'll create a simple UI around this functionality to make something that we can show off to users.

By working with a real web service, you'll see what it's like to interact with a 3rd party API for the first time, and learn some strategies for solving problems when they arise.

Getting Acquainted with the iTunes Search API

> *You can't buy what you can't find what you can't buy what you find what you can't buy what you can't what you can't buy what you can't find what you can't...*
>
> – *The Raconteurs "Store Bought Bones"*

Apple offers an API to search the iTunes Store, App Store, iBooks Store, and Mac App Store as part of their affiliate program. You can use it to create apps that earn a commission on music, app, or book sales. If a user makes a purchase after clicking a link from your app or website, you could get a small percentage of the cost.

For the specific purposes of this book, it has a unique combination of traits that are hard to find among APIs:

- **No authentication is required.** You don't have to sign up for access, manage credentials, or juggle APIs in order to get started.
- **It has documentation.** The available endpoints and expected output are described in detail (though there are some quirks, as we'll see later).
- **Its domain is interesting and familiar.** Everyone likes music and has used iTunes at some point. Contrast this with what you typically encounter with open data.
- **It's good, not great.** As a result of being updated to accommodate the ever-increasing scope of what can be sold on iTunes, this API has accumulated a fair amount of cruft over the years.

Digging Into the Documentation

Start by opening up the Apple iTunes Search Documentation in your browser of choice.

Note: The section Searching the iTunes Store describes how to construct URLs from structured queries. For your convenience, we've provided the `AppleiTunesSearchURLComponents` structure for you to use in this example. You can find it in the Sources group of the `Chapter 4.playground` file.

We're going to focus on the section Understanding Search Results.

Ignoring for a moment a few disconcerting technical mistakes early on,[1] the documentation does a commendable job by providing both a raw JSON string and a table explaining the meaning of each key-value pair.

Comparing Expectation to Reality on the Command Line

There's a Russian proverb, *"Доверяй, но проверяй"*, or *"trust, but verify"*.

Think of it anytime you consult documentation.

Because no matter how correct or comprehensive it may be, documentation is always fighting a losing battle against entropy. Software changes over time, often in subtle and mysterious ways that evade detection. And with each small change, the documentation drifts ever further from reality.

To summarize the great philosophical tradition of Empiricism: *You can only arrive at truth through direct experience.*

When we open the documentation in Safari, we see that there's a table containing a list of keys in the JSON payload. We *could* copy-paste and manually edit the text to get what we need, but what's the fun in that? We're programmers, after all, and if there's a reasonable way to automate a task, then it's worth investigating.

---

1. From the documentation: "A collection of name/value pairs, also known as an object; this concept is similar to a Java Map object, a Javascript Dictionary, or a Pearl/Ruby hash."
   It's "JavaScript," not "Javascript," JavaScript doesn't have a `Dictionary` type, and it's "Perl," not "Pearl."

Let's crack open our toolbox and take advantage of the command line.[2] To start, open up Terminal.app or your preferred terminal emulator.

First, use `curl` to cache the HTML locally to a file.

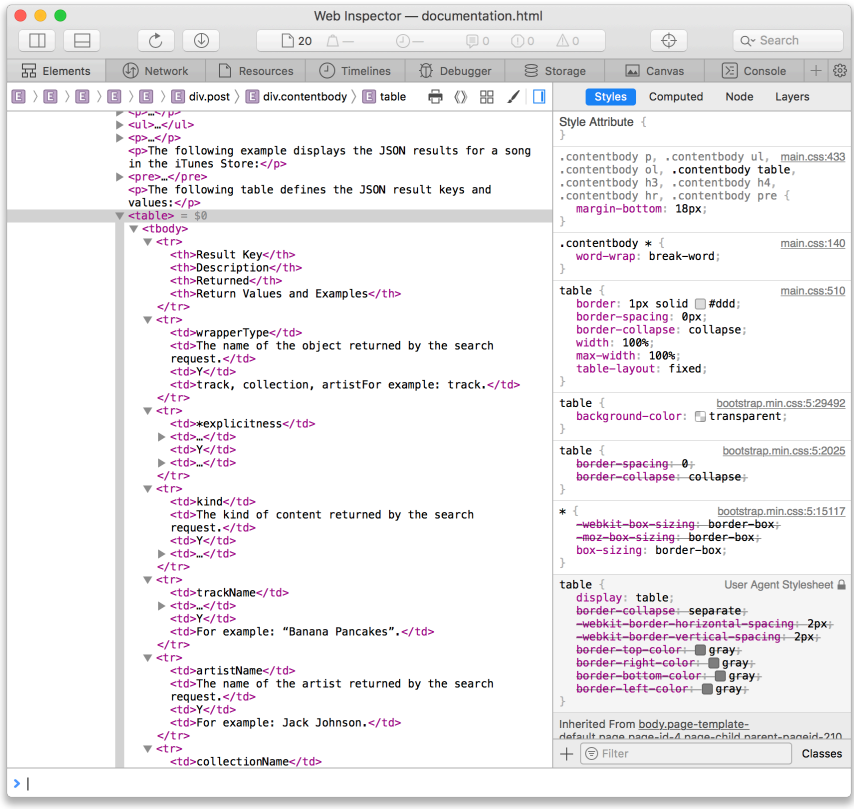
```
$ curl -o documentation.html \
    "https://affiliate.itunes.apple.com/resources/documentation/
itunes-store-web-service-search-api/"
```

Next, open that HTML file in Safari. We'll use the built-in Web Inspector to determine a reasonable query for accessing the information we're after.

```
$ open documentation.html
```

Select the "Develop ▸ Show Web Inspector" menu item (⌥⌘I) and find the table under "Understanding Search Results" on the page.

2. If you're unfamiliar with the command line, check out Apple's Command Line Primer for a quick introduction.



At the bottom of the inspector window, enter the following[3] :

```
> document.querySelectorAll("table")
< // NodeList [<table>, <table>, <table>, <table>] (4) = $1
```

This page has four `<table>` elements, so we need to be more specific.

We want only the last table, and we can get it by using the `:last-of-type` CSS pseudo-class.

```
> document.querySelectorAll("table:last-of-type td:first-child")
< // NodeList [<table>] (1) = $2
```

---

3. `document.querySelectorAll()` is a JavaScript function that takes a CSS selector and returns all nodes in the current document that match the query.

Let's proceed to get the first table cells in each row using the `:first-child` CSS pseudo-class.

```
> document.querySelectorAll("table:last-of-type td:first-child")
< // NodeList [<td>, <td>, <td>, <td>, <td>, <td>, …] (11) = $3
```

Now that we have a working selector, we can extract the information we're looking for into a text file.

HTML-XML-utils[4] is a suite of helpful utilities for manipulating HTML and XML on the command line. We'll use `hxnormalize` and `hxselect` from this package.

Send the cached HTML file to `hxnormalize` to fix markup errors and then pipe the result to `hxselect` to extract elements that match the CSS selector we determined from before. Use the `tr` command to strip newlines, punctuation, and whitespace, `sort` the results lexicographically, and then capture that output into a new file.

```
$ cat documentation.html \
    | hxnormalize -x \
    | hxselect -c \
               -s "\n" \
               -i "table:last-of-type td:first-child" \
    | tr , '\n' \
    | tr -d " " \
    | sort \
    > documented-attributes.txt
```

Finally, use `less` to view the contents of the file we just wrote.

```
$ less documented-attributes.txt
```

4. Install using Homebrew with `brew install html-xml-utils`.

- `*censoredName`
- `*explicitness`
- `*viewURL`
- `artistName`
- `artworkUrl100`
- `artworkUrl60`
- `collectionName`
- `kind`
- `previewUrl`
- `trackName`
- `trackTimeMillis`
- `wrapperType`

Success! We now have a list of the documented attributes. Now let's test these expectations against reality.

Use `curl` again to download the JSON response from the API for the example search URL provided in the documentation.

```
$ curl -o actual.json \
    "https://itunes.apple.com/search?term=jack+johnson&limit=1"
```

Use `less` to view the JSON we just downloaded.

```
$ less actual.json
```

```
{
 "resultCount": 1,
 "results": [...]
}
```

Already, we find something that the documentation failed to mention explicitly: results are nested at the `"results"` key of the top-level JSON object. We'll need to factor that into our implementation for decoding the response.

From here, let's get the keys of a result object so that we can compare them to our list of documented attributes.

We use `jq`[5] to select the keys of the JSON payload. We then pipe them through `tr` and `sed` to remove formatting artifacts, `sort` them lexicographically, and then finally capture the output into a new file.

```
$ cat actual.json \
    | jq ".results[0] | keys" \
    | tr -d '[] "," \
    | sed '/^$/d' \
    | sort \
    > actual-attributes.txt
```

`comm` is the lesser-known counterpart to `diff`, and it selects or rejects lines in common. We can use `comm` to get the difference between `actual-attributes.txt` and `documented-attributes.txt`. The `-1` option suppresses the entries in column 1, and the `-3` option suppresses the entries in column 3. What's left are the entries unique to the actual output, which we capture into a new file.

```
$ comm -1 -3 documented-attributes.txt actual-attributes.txt \
    > undocumented-attributes.txt
```

Now for the moment of truth: how many undocumented attributes are there?

Use `wc` to count the lines to find out.

```
$ wc undocumented-attributes.txt
⇢ 22
```

Twenty-two undocumented attributes. Wow. It's a good thing we checked!

---

5. Install using Homebrew with `brew install jq`.

Let's take a moment and synthesize what we know into something more meaningful. Here are the actual attributes again, organized into categories with emphasis added to any that were actually documented.[6]

Categorization

wrapperType, kind

General Information

releaseDate, primaryGenreName, country, currency

Track Information

trackCensoredName, trackCount, trackExplicitness, trackId, trackName, trackNumber, trackPrice, trackTimeMillis, trackViewUrl, isStreamable, previewUrl

Artist Information

artistId, artistName, artistViewUrl, artworkUrl100, artworkUrl30, artworkUrl60

Collection Information

collectionCensoredName, collectionExplicitness, collectionId, collectionName, collectionPrice, collectionViewUrl, discCount, discNumber

Artwork URLs

artworkUrl30, artworkUrl60, artworkUrl100

Of all the available attributes, we're interested in the following: `trackName`, `trackExplicitness`, `trackViewURL`, `previewURL`, `artistName`, `collectionName`, and `artworkURL100`.

6. `artistId`, `collectionId`, and `trackId` are present in the example JSON, but not in the documented keys table.

Defining the SearchResult type

With a clear sense of what attributes are available and which ones we want to use, we're ready to define the `SearchResult` type.

Because the iTunes Search API is read-only, we only need to conform to `Decodable`. And the properties aren't going to be modified, so we can make those constants by using `let` instead of `var`.

As you go through the documentation, copy-paste relevant information for each item into source code as a comment. Not only does this create a scaffolding for what you need to implement, but once you're finished, the end result is a fully-documented type.

```
/// A result returned from the iTunes Search API.
struct SearchResult: Decodable {
    /// The name of the track, song, video, TV episode, and so
on.
    let trackName: String?

    /// The explicitness of the track.
    let trackExplicitness: String?

    /// An iTunes Store URL for the content.
    let trackViewURL: URL?

    /// A URL referencing the 30-second preview file
    /// for the content associated with the returned media type.
    /// - Note: This is available when media type is track.
    let previewURL: URL?

    /// The name of the artist, and so on.
    let artistName: String?

    /// The name of the album, TV season, audiobook, and so on.
    let collectionName: String?

    /// A URL for the artwork associated with the returned
media type.
    let artworkURL100: URL?
}
```

As mentioned in Chapter 1, each property is named according to Swift API Design Guidelines, using `lowerCamelCase`. By coincidence, the

JSON keys follow the same convention (with the exception of the suffix `Url` not being capitalized), so we only need to override a few of the `CodingKeys` case values.

```
extension SearchResult {
    private enum CodingKeys: String, CodingKey {
        case trackName
        case trackExplicitness
        case trackViewURL = "trackViewUrl"
        case previewURL = "previewUrl"
        case artistName
        case collectionName
        case artworkURL100 = "artworkUrl100"
    }
}
```

Creating the SearchResponse Model

As we found by making that test request to the API, the JSON response nests the array of results at the `"results"` key.

Therefore, we define a `SearchResponse` type that conforms to `Decodable` to provide safe and convenient access to the results.

```
struct SearchResponse: Decodable {
    let results: [SearchResult]
}
```

Filtering Explicit Content

The iTunes Search API includes guidance about the content of search results from the Recording Industry Association of America, or RIAA.[7]

7. For more information, see Music Ratings on iTunes.

> *The Parental Advisory is a notice to consumers that recordings identified by this logo may contain strong language or depictions of violence, sex or substance abuse. Parental discretion is advised.*

The `trackExplicitness` attribute can have one of the following three values[8] :

- `explicit` (explicit lyrics, possibly explicit album cover),
- `cleaned` (explicit lyrics "bleeped out"),
- `notExplicit` (no explicit lyrics)

Anytime you have a small, exhaustive list of possible values, you have a great candidate for an enumeration:

```
enum Explicitness: String, Decodable {
    case explicit, clean, notExplicit
}
```

The `Explicitness` enumeration has a `String` raw value type, and adopts the `Decodable` protocol. The compiler automatically synthesizes `Decodable` conformance.

8. The documentation uses the notation `*explicitness` as a catch-all for `trackExplicitness` and `collectionExplicitness`, which isn't immediately clear until seeing the actual attributes.

Without any further modification, we can swap out the previous `String` type for the `trackExplicitness` property with the new `Explicitness` type.

```
let trackExplicitness: Explicitness?
```

For our use case, we'll want to filter out any explicit material before displaying results to the user. A good way to encapsulate this functionality is to define a `nonExplicitResults` computed property on the `SearchResponse` container model that filters based on `trackExplicitness`.

```
extension SearchResponse {
    var nonExplicitResults: [SearchResult] {
        return self.results.filter { (result) in
            result.trackExplicitness != .explicit
        }
    }
}
```

Reverse-Engineering Image URLs

In addition to the documented `artworkUrl60`, `artworkUrl100` attributes, we found the response JSON to also include an undocumented `artworkUrl30` attribute. Having 3 variations of the same URL is conspicuous. So let's investigate further to see what's going on.

Using the `jq` utility from before, we extract the values of the `artworkUrl30`, `artworkUrl60`, and `artworkUrl100` keys from the first element of the `results` array.

```
$ cat actual.json \
    | jq '.results[0] | .artworkUrl30, .artworkUrl60,
.artworkUrl100'
```

```
"http://is4.mzstatic.com/image/thumb/Music122/v4/6d/cd/ed/
6dcdedec-98ee-ca57-6c72-7f9f8d9dda6a/source/30x30bb.jpg"
"http://is4.mzstatic.com/image/thumb/Music122/v4/6d/cd/ed/
6dcdedec-98ee-ca57-6c72-7f9f8d9dda6a/source/60x60bb.jpg"
"http://is4.mzstatic.com/image/thumb/Music122/v4/6d/cd/ed/
6dcdedec-98ee-ca57-6c72-7f9f8d9dda6a/source/100x100bb.jpg"
```

From a quick visual inspection, we see that the URLs only differ in their last path components:

```
.../30x30bb.jpg
.../60x60bb.jpg
.../100x100bb.jpg
```

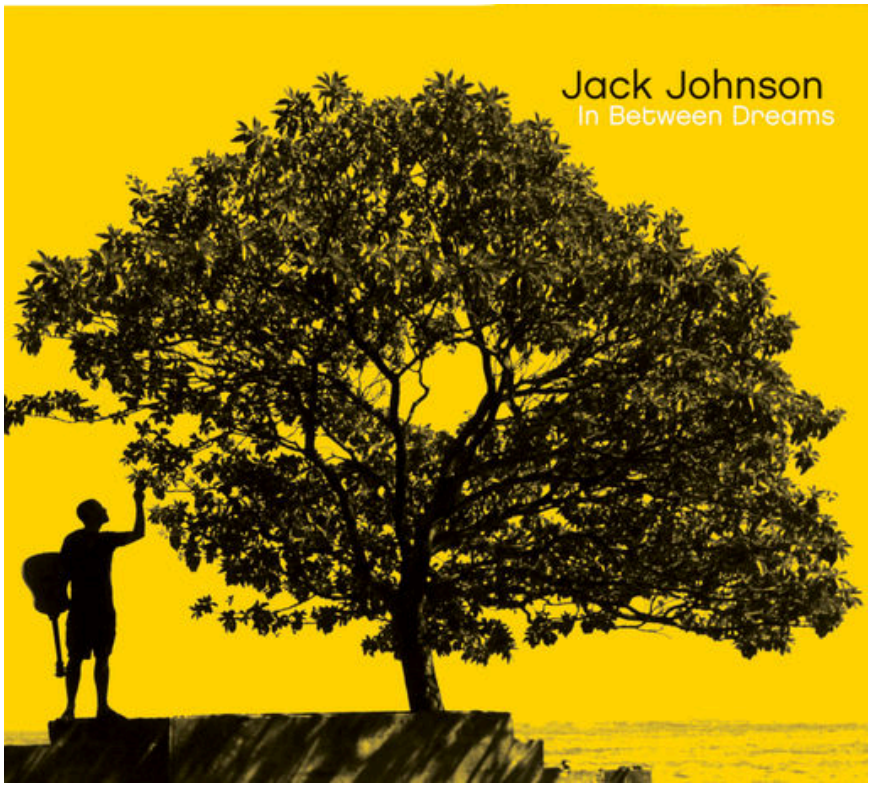
Wait, could it be as simple as swapping in our desired dimensions?

Get the value of the `artworkUrl100` attribute from `jq`, pipe that into `sed`[9] to replace the dimensions with a custom size, and then pipe the result to `xargs`[10] to open the resulting URL.

```
$ cat actual.json \
    | jq '.results[0].artworkUrl100' \
    | sed s/100x100/512x512/ \
    | xargs open
```

9. `sed` is a Unix utility that transforms text.

10. `xargs` is a Unix utility that converts piped input into arguments.



Halaka ukulele banana pancakes — it actually works! Now let's learn more about its behavior through experimentation:

`1x1bb.jpg` results in an image of size 1×1 px, so there's our lower bounds.

`10000x10000bb.jpg` results in an image of size 900×810 px, which seems to indicate that if the requested dimensions are too big, the API returns the largest size available. This may or may not be the same limit for every piece of artwork, but it's much larger than the 100×100 px returned by the API.

Now let's see what happens when the dimensions are different.

100x200bb.jpg	200x100bb.jpg
	
100×90 px	111×100 px

`100x200bb.jpg` results in an image of size 100×90 px; for `200x100bb.jpg`, the image size is 111×100 px. Therefore, we can reasonably conclude that artwork is sized to fit its aspect ratio.

Being able to get different image sizes will be helpful in our app, because the provided presets aren't big enough for modern retina displays. Although this is undocumented behavior, it's too valuable not to use.

To provide a convenient and `Decodable`-compatible abstraction around all of this, we can make the `artworkURL100` property private, and define a new `artworkURL(size:)` function that constructs an artwork URL according to what we learned about the undocumented behavior.

```
private let artworkURL100: URL?
```

```
extension SearchResult {
    func artworkURL(size dimension: Int = 100) -> URL? {
        guard dimension > 0, dimension != 100,
            var url = self.artworkURL100 else {
            return self.artworkURL100
        }

        url.deleteLastPathComponent()

url.appendPathComponent("\(dimension)x\(dimension)bb.jpg")

        return url
    }
}
```

The `artworkURL(size:)` method replicates our Terminal command from before, swapping out `sed` for proper URL manipulation, care of the Foundation framework.

By making `artworkURL100` private, we direct API consumers to use `artworkURL(size:)` instead, which allows us to use our undocumented behavior to support large screen sizes. If that behavior should break in the future, the caller can still get the returned value for `artworkURL100` by calling the `artworkURL(size:)` with the default parameter value.

Fetching Search Results using URLSession

Having translated our mental model into a Swift model, we're ready to put our understanding to the test!

In our view controller, declare properties for a URL session data task and the current search results.

```
var dataTask: URLSessionDataTask? = nil
var results: [SearchResult] = []
```

We'll put all of our logic for loading results from the API into a search method.

```
func search<T>(for type: T.Type,
               with term: String) where T: MediaType {
    // ...
}
```

The first step of implementing the search method is to translate the `type` and `term` arguments into an HTTP request. For the iTunes Search API, all parameters are encoded into the URL query of a GET request.

```
let components = AppleiTunesSearchURLComponents<T>(term: term)
guard let url = components.url else {
    fatalError("Error creating URL")
}
```

Next, we cancel the existing data task, if one was pending. Then we create a new task using the `dataTask(with: url)` method on the shared `URLSession` instance. In the completion handler, we use `JSONDecoder` to decode a `SearchResponse` object from the received data. If everything checks out, the `results` property is updated and the table view is asynchronously told to update its contents.

```
self.dataTask?.cancel()

self.dataTask = URLSession.shared.dataTask(with: url) {
    (data, response, error) in

    guard let data = data, error == nil else {
        fatalError("Networking error \(error) \(response)")
    }

    do {
        let decoder = JSONDecoder()
        let searchResponse =
            try decoder.decode(SearchResponse.self, from: data)
        self.results = searchResponse.nonExplicitResults
    } catch {
        fatalError("Decoding error \(error)")
    }

    DispatchQueue.main.async {
        self.tableView.reloadData()
    }
}

self.dataTask?.resume()
```

Ahhh. Those pixels on the screen are like music to the ear.

From here, you can implement pretty much any feature you can think of as quickly as you can work out the UI. Displaying album artwork in rows? Easy. Adding a player for music previews? No problem. Sabotaging the results so that the only music you get back is “All Star” by Smash Mouth and the only movies you see are “Shrek”? I mean, yeah… you can do that too, I guess.

Either way, that’s the power of starting with a complete model.

There's a very real chance that, by the time you're reading this, none of this code works anymore. The days of buying and selling music *a la carte* are numbered, as are likely those of this API.

But the lessons we learned and the experience we gained from API — that… that doesn't go away.

Perhaps the lesson in all of this is that the process is ultimately more valuable than any code we write. Code erodes and breaks and rots over time. But our ability to approach problems with creativity and rigor only gets better with age.

Recap

- Remember, *“trust, but verify”*. Start with documentation as the foundation of your understanding and challenge assumptions through direct interaction with the system.
- Command-line tools offer a powerful way to explore web services without having to write Swift code.
- Writing code comes at the end of the process, not the start. Until you have a validated mental model, any line of code that you write is just a guess at how things work.


IN-FLIGHT MENU
10
11
12
13
14
A
B
C
D
E
ITEMS
1
Peanuts
Chips
Popcorn
kies

Chapter 5:

Building an In-Flight Service App with User Defaults

Catfish Airways prides itself on having the best in-flight crew around. Friendly and professional, they put passengers at ease and keep everyone safe.

Once a plane reaches its cruising altitude, flight attendants saunter down the center aisle, offering passengers snacks and beverages from their cart. In economy class, meals are available for purchase.

To streamline the purchasing process, each flight attendant is equipped with an app that keeps tab of all meals purchased. Because internet access is spotty at 30,000 feet, all data is saved locally on the device. Once the flight has landed, those devices sync their records to the server, which then charges passenger accounts on file and sends out receipts.

Up until this point, we've used `Codable` to work only with JSON. One of the enduring strengths in the implementation of `Codable` is that it works pretty much the same for any format that supports `Encoder` and/or `Decoder`. In addition to JSON, the Foundation framework provides a built-in decoder and encoder for property lists (`PropertyListDecoder` and `PropertyListEncoder`).

In this chapter, we'll look at how to use `Codable` with property lists to load inventory and to persist orders locally using `UserDefaults`.

Now, we request your full attention to the following paragraphs as we demonstrate the technical features of this data interchange format.

Introducing Property Lists

Property Lists are a data format that go back all the way to the days of NeXT. And though they're used extensively in macOS and iOS, their penetration outside the Apple ecosystem is quite limited.

If you're like many developers out there, your interactions with property lists are probably limited to opening the `Info.plist` or `Entitlements.plist` file in Xcode to configure your app. If so, the experience probably looked something like this:

Key	Type	Value
▾ Information Property List	Dictionary	*(14 items)*
Executable file	String	$(EXECUTABLE_NAME)
Bundle name	String	$(PRODUCT_NAME)
Bundle OS Type code	String	AAPL
Bundle version	String	1
Application requires iPhone…	Boolean	YES
▸ Required device capabilities	Array	*(1 item)*
…		

The trouble with this and other graphical interfaces *(*cough* Storyboards)*, is that they obscure the underlying information. Sure, GUIs are often nicer to interact with for everyday tasks, but there are some occasions where you just want to see what's actually going on.

If you instead control-click a `.plist` file in Xcode and select "Open As ▸ Source Code" (or just open the file in a different text editor) you'll see the real, honest-to-goodness content of the property list file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
 "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
        <key>CFBundleExecutable</key>
        <string>$(EXECUTABLE_NAME)</string>
        <key>CFBundleName</key>
        <string>$(PRODUCT_NAME)</string>
        <key>CFBundlePackageType</key>
        <string>APPL</string>
        <key>CFBundleVersion</key>
        <string>1</string>
        <key>LSRequiresIPhoneOS</key>
        <true/>
        <key>UIRequiredDeviceCapabilities</key>
        <array>
                <string>armv7</string>
        </array>
    <!-- snip -->
</dict>
</plist>
```

The first thing you'll notice is that property lists are XML files — or at least that's how they're typically represented. Property lists can also assume a more compact binary format, as well as an ASCII format affectionately referred to as "old-style" that was originally used in OpenStep frameworks. Around the first release of Mac OS X circa 2000, property lists adopted an XML format with a public DTD defined by Apple, and that's been the standard ever since. This schema defines the following mapping between data types and XML elements:

Type	XML Element
array	`<array>`
dictionary	`<dict>` (with `<key>`)
string	`<string>`
integer	`<integer>`
floating-point	`<real>`
binary data	`<data>`
date	`<date>`
Boolean	`<true/>` or `<false/>`

Note: You may have also noticed the long `CF` and `UI` constants. By default, Xcode uses descriptive, localized names. For example, the `CFBundleExecutable` key is shown as “Executable file”. You can toggle this behavior by selecting the “Editor ▸ Show Raw Keys & Values” menu item.

In contrast to JSON, property lists natively support both dates and binary data. They also differentiate between integer and decimal numbers, whereas JSON stores all numbers as double-precision floating-points. Because of this, properties in `JSONDecoder` for interpreting these kinds of values, such as `dateDecodingStrategy` and `nonConformingFloatDecodingStrategy`, aren’t necessary for `PropertyListDecoder`.

One syntactic quirk worth mentioning is how dictionaries are represented in property lists. Instead of using XML attributes, property lists use sequential pairs of elements consisting of one `<key>` element followed by one element for the value. This allows for dictionaries to nest more easily, and to provide explicit types for its values.

```
<!-- Conventional XML -->
<element key="value">

<!-- Property List -->
<key>key</key>
<string>value</string>
```

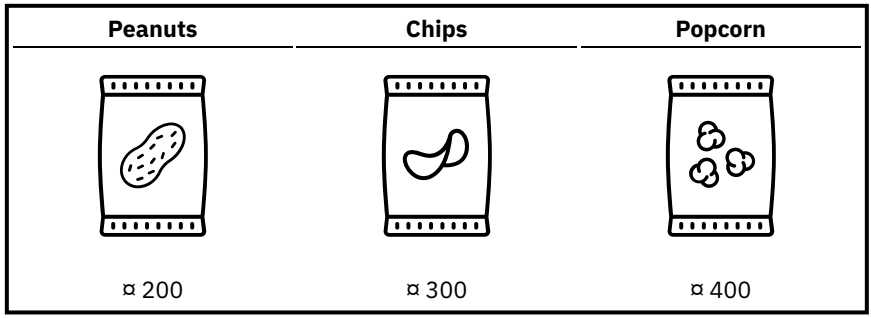
Defining the Item Model

The `Item` structure has a `name` property of type `String` and a `unitPrice` property of type `Int`. Because it adopts the `Codable` protocol in its declaration, and each of its properties have types that conform to `Codable`, the compiler automatically synthesizes conformance for `Item`.

```
struct Item: Codable {
    var name: String
    var unitPrice: Int
}
```

Creating and Adding Inventory.plist to Your Project

Our example app loads a list of items available for purchase from the `Inventory.plist` file, which is included as a resource in the app bundle. Each item has a name and a price per unit. On the menu for this flight are three items: Peanuts, Chips, and Popcorn.



```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
 "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
    <dict>
    <key>items</key>
    <array>
        <dict>
            <key>name</key>
            <string>Peanuts</string>
            <key>unitPrice</key>
            <integer>200</integer>
        </dict>
        <dict>
            <key>name</key>
            <string>Chips</string>
            <key>unitPrice</key>
            <integer>300</integer>
        </dict>
        <dict>
            <key>name</key>
            <string>Popcorn</string>
            <key>unitPrice</key>
            <integer>400</integer>
        </dict>
    </array>
    </dict>
</plist>
```

To load a property list file from your bundle, you need to first add it to your app target. When you add one or more files to the project navigator, Xcode displays a modal that prompts you to choose options for adding those files. At the bottom of that screen, you can select which targets those files are added to (the app target is selected by default). When a file is added to a target, it's added to the "Copy Bundle Resources" build phase, and becomes accessible to that target at run time.

Loading the Inventory from a Property List File

You can use the `Bundle` method `url(forResource:withExtension:)` to get a URL to that resource.

```
import Foundation

guard let url = Bundle.main.url(forResource: "Inventory",
withExtension: ".plist") else {
    fatalError("Inventory.plist missing from main bundle")
}
```

Use the `Data(contentsOf:options:)` initializer to read the contents of the property list, create a `PropertyListDecoder`, and then use it to decode an array of `Item` elements from the data.

```
var inventory: [Item]
do {
    let data = try Data(contentsOf: url)

    let decoder = PropertyListDecoder()
    let plist = try decoder.decode([String: [Item]].self, from:
data)
    inventory = plist["items"]!
} catch {
    fatalError("Cannot load inventory \(error)")
}
```

Now that we have an inventory of items, let's make it possible for passengers to order them.

Defining the `Order` Model

A flight attendant creates an order for each passenger by recording how many of each item they're served. Passengers are identified by their seat assignment, which is stored in the `seat` property as a `String` (for example, "17B"). The quantity for each item in the order is stored as an array of `LineItem` objects in the `lineItems` property. The `creationDate` property acts as an immutable timestamp that's set when the order is created.

```
import Foundation

struct Order: Codable {
    var seat: String

    struct LineItem: Codable {
        var item: Item
        var count: Int

        var price: Int {
            return item.unitPrice * count
        }
    }

    var lineItems: [LineItem]

    let creationDate: Date = Date()
}
```

Calculating the Total Price for an Order

We'll eventually want to calculate the total price for an order. You do this by initializing a running total to zero, iterating over each line item, and adding its price to the running total.

Here's an example of how to do this using a `for-in` loop:

```
// Conventional approach
var totalPrice = 0
for lineItem in lineItems {
    totalPrice += lineItem.price
}
```

As an alternative, consider the equivalent functional programming approach:

```
// Functional approach
lineItems.map{ $0.price }.reduce(0, +)
```

If you're unfamiliar with functional programming, this may look like an entirely different language. This single line of code may be dense (by necessity, since it's doing the equivalent of the preceding five lines of code), but everything can be understood when broken down into bite-sized pieces:

`map(_:)` is a method defined on the `Sequence` protocol. We can call this method on `lineItems` because `Array` conforms to `Collection` and `Collection` inherits `Sequence`. The `map(_:)` method takes a single closure argument, which is evaluated for each element in the sequence; it then takes the return value of each evaluated closure and returns the remaining values in an array.

The array that results from calling `map(_:)` then calls the `reduce(_:_:)` method. This practice of making successive method calls is known as *method chaining*.

The `reduce(_:_:)` method combines all the elements in a sequence into a single value. Like `map(_:)`, `reduce(_:_:)` is a method defined by `Sequence`, and we can call this method because `Array` conforms to `Collection` and `Collection` inherits `Sequence`.

`reduce(_:_:)` takes two arguments: an initial accumulating value and a closure. The method starts with an initial accumulating value and evaluates the closure with that initial value and the first element in the sequence to produce the next partial result. The closure then evaluates this partial value and the next element in the sequence to produce the next partial value. This process is repeated until no elements remain, at which point the method returns the last partial value.

To get a sum of all prices, `reduce(_:_:)` is called with an initial accumulating value of `0` and a closure that adds each element to the running total.

```
// Functional approach, without shorthand
let lineItemPrices = lineItems.map { (lineItem) in
    return lineItem.price
}
let totalPrice = lineItemPrices.reduce(0) { totalPrice,
lineItemPrice in
    return totalPrice + lineItemPrice
}
```

`$0` is a *shorthand argument name*. Shorthand argument names allow inline closure arguments to be referenced anonymously. For the call to `map(_:)`, `$0` refers to the first (and only) closure argument, the `LineItem` element.

Swift allows operators like + to be passed to inline closure arguments as syntactic shorthand for using the operator function. For the call to `reduce(_:_:)`, the + operator takes the sum of the two closure arguments: the accumulating value and the line item price.

Swift automatically uses the result of this statement as the return value for the closure, without needing the `return` keyword.

Defining a Convenience Initializer for Orders

While we're in the habit of enhancing the `Order` API, let's add a convenience initializer, `init(seat:itemCounts:)`, which sets the `lineItems` property from an `[Item: Int]` dictionary.

The conventional way to do this is to initialize a variable to an empty array of line items, iterate over each key-value pair in the dictionary, create a line item for each entry, and then append it to the mutable collection.

```
// Conventional approach
var lineItems: [LineItem] = []
for (item, count) in itemCounts {
   guard count > 0 else { continue }
   let lineItem = LineItem(item: item, count: count)
   lineItems.append(lineItem)
}
```

As an alternative, consider the equivalent functional programming approach:

```
// Functional approach
itemCounts.compactMap{ $1 > 0 ? LineItem(item: $0, count: $1) :
nil }
```

If this looks like pure gibberish, don't worry — we can use the same "divide-and-conquer" strategy from before to decipher it:

`compactMap(_:)`[1] is a method defined on the `Sequence` protocol. We can call this method on `itemCounts` because `Dictionary` conforms to `Collection` and `Collection` inherits `Sequence`.

The `compactMap(_:)` method takes a single closure argument, which is evaluated for each element in the sequence; it then takes the return value of each evaluated closure, removes any `nil` values, and returns the remaining values in an array. For a dictionary, the evaluated element is a tuple containing the key-value pair. In the case of `itemCounts`, the key is the item name, and the value is the count.

That's the functional part of it — the rest is syntactic trickery.

---

1. `flatMap(_:)` was renamed to `compactMap(_:)` in Swift 4.1.

To help make sense of what's happening, we can rewrite this in a way that doesn't rely on shorthand:

```
// Functional approach, without Shorthand
itemCounts.compactMap{ (key: String, value: Int) in
    if value > 0 {
        return LineItem(item: key, count: value)
    } else {
        return nil
    }
}
```

`$0` and `$1` are *shorthand argument names*, referring to the first and second closure arguments, respectively. In this case, `$0` refers to the key and `$1` refers to the value.

`$1 > 0` is an expression that compares the count to zero using the greater-than (>) operator. The result of this expression is passed to the *ternary conditional operator* (?), which is kind of like a shorthand `if-else` statement. If the expression evaluates to `true`, the expression to the left of the colon (`:`), `LineItem(item: $0, count : $1)`, is used. If the expression evaluates to `false`, the expression to the right of the colon (`:`), `nil`, is used.

Swift automatically uses the result of this statement as the return value for the closure, without needing the `return` keyword.

Introducing User Defaults

`UserDefaults` may be one of the most misunderstood parts of Foundation.

Its name suggests… well it's unclear, really. It interacts with the *user defaults system* (mystery solved), but what does that mean? Most documentation uses the terms "preference" and "default" interchangeably, and describes it both as a way to store user preferences and a way to store app data.

And its API is littered with dead-ends; the documentation itself includes the phrase "has no effect and shouldn't be used". No, seriously.

And yet, `UserDefaults` is *by far* the most convenient persistence mechanism on Apple platforms.

This is due in part to historically weak documentation (though recently it's improved quite a bit).

We'll let `UserDefaults` speak for itself, in its own words, courtesy of the header documentation:

> *[UserDefaults] is a hierarchical persistent interprocess (optionally distributed) key-value store, optimized for storing user settings.*

Let's break that down (though not in that exact order):

"Key-Value Store"

You can get and set arbitrary values by key, like a dictionary.

"Persistent"

Values you set are available the next time you launch the app.

"Interprocess"

The user defaults database is shared across all apps running on the system. You can choose to set values that can be read globally or by certain apps.

"Hierarchical"

When looking up the value for a key, a search list is consulted until a match is found. Values set higher up in the search path trump other values. For example, values passed as command line arguments override values set by the current app, which in turn override values set by other apps.

“Optimized for Storing User Settings”

The user defaults system works best when the data stored is small, read frequently, and written only occasionally. For large object graphs and/or data that is updated frequently, a different persistence mechanism like Core Data may be more appropriate.

Persisting Orders with User Defaults

User defaults aren’t property lists, and property lists aren’t user defaults. It just so happens that user defaults can only get and set “property list objects”, which is an informal distinction — that is, not represented in code — given to types that can be directly represented in property lists: arrays, dictionaries, numbers, Boolean values, dates, and binary data.

This is the reason we use `PropertyListEncoder` and `PropertyListDecoder` in conjunction with `UserDefaults`.

It also helps that property lists are more capable formats for encoding numbers, dates, and binary data than JSON. When we use property lists for serialized representations, we can be confident that what we put in is exactly what we get out.

Loading Orders

When the plane reaches cruising altitude and the flight attendants launch the app, we’ll load any orders that were stored during the last session. This allows us to pick up where we left off, should anything cause the app to be terminated.

We use the `object(forKey:)` method to fetch the serialized representation of any persisted orders. Then, we decode the data into an array of `Order` objects using `PropertyListDecoder`.

```
var orders: [Order]
do {
    if let data = UserDefaults.standard.object(forKey:
"orders") as? Data {
        let decoder = PropertyListDecoder()
        orders = try decoder.decode([Order].self, from: data)
    }
} catch {
    print("Error decoding orders: \(error)")
}
```

Tip: If you're making extensive use of `UserDefaults`, you may consider defining string constants instead of simply using string literals.

If defaults don't exist for a particular key, `object(forKey:)` returns `nil`. We can reduce complexity in our logic by calling `register(defaults:)` when the app finishes launching:

```
UserDefaults.standard.register(defaults: ["orders": []])
```

This ensures that if no orders are found, we start with an empty array instead of nothing.

Saving Orders

The app can save orders both periodically and in response to a request for termination. All you need to do is call `set(_:forKey:)` with a property list representation of the orders.

`UserDefaults` has overrides for the `set(_:forKey:)` method that accept values of type `Float`, `Double`, `Int`, `Bool`, and `URL?`, as well as `Any?`. We use this last override, for `Any?` when we pass the representation that's created by `PropertyListEncoder` for our orders.

```
do {
    let encoder = PropertyListEncoder()
    UserDefaults.standard.set(try encoder.encode(orders),
forKey: "orders")
} catch {
    print("Error saving orders: \(error)")
}
```

Deleting Orders

After the plane has landed and taxied to the gate, the flight attendant hands off their device to the ground crew, who are responsible for transferring orders to the central servers and getting everything ready for the next flight.

```
UserDefaults.standard.removeObject(forKey: "orders")
```

Because we called `register(defaults:)` at launch, the next time we call the `object(forKey:)` method, it will return an empty array, rather than `nil`. Just as the ground crew cleans the plane after every flight so do we extend the same courtesy for the next launch.

With all the focus on web services these days, it's easy to forget the simple pleasures of local persistence. To that end, `Codable`, property lists, and `UserDefaults` make for a winning combination.

The ability for `Codable`-conforming types to be used with property lists as well as JSON (or any other supported formats) means that you can easily incorporate local persistence into your app.

Recap

- Property lists are a great choice for storing information in a bundle.
- Use `PropertyListEncoder` to store objects in user defaults, and `PropertyListDecoder` to retrieve them.
- Use the `registerDefaults(_:)` method to set initial values when the app finishes launching.
- Functional programming methods like `map(_:)` and `reduce(_:)` can make code smaller and more understandable.


JFK
JFK
JFK

Chapter 6:

Building a Baggage Scanning App with Core Data

Flyers on Zarra Airlines' international routes are entitled to up to two checked bags in addition to a carry-on and personal item.

Check-in is a streamlined process: First, passengers print out their boarding pass at one of the available kiosks. If they're checking a bag, they print out a tag, attach it to their luggage, and then bring it to the attendant at the Bag Drop.

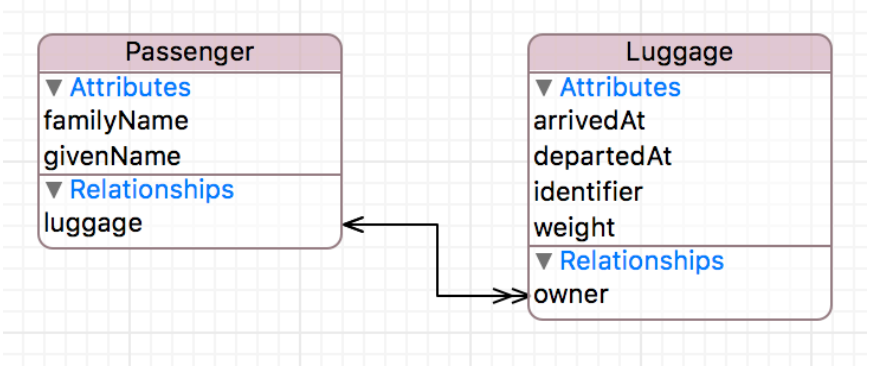
Luggage tags display the first initial of their given name and first three letters of their family name, as well as the airport codes for their point of departure and final destination. At the bottom, additional information is encoded as a barcode.

When a passenger leaves their luggage at Bag Drop, the baggage attendant places it on a conveyor belt that takes it past a rubber curtain and out of sight. That's the last that the passenger will see of it until (hopefully) reuniting with it at baggage claim at their destination. At every step of the way, attendants, handlers, and security personnel scan the barcode to ensure that all bags are present and accounted for.

In this chapter, we'll build a baggage tracking app using Core Data that creates managed objects from JSON using `Codable`. However, instead of sending requests over the network, the data is read from QR codes on luggage tags.

Defining the Managed Object Model Schema

This example defines a simple managed object model schema that consists of two entities: Luggage and Passenger.



Luggage

Each piece of luggage has a unique `identifier` that's generated at the time it's printed. The luggage's `weight` in pounds is also recorded. When the gate attendant at the point of departure scans the luggage, its `departedAt` attribute is set to the current time. Likewise, the `arrivedAt` attribute is updated when the luggage is scanned once it's unloaded at its final destination. Each piece of luggage has one `owner`, which is a reference to a `Passenger` record.

Attribute	Type	Properties
identifier	UUID	
weight	Float	Minimum = 0
departedAt	Date	Optional
arrivedAt	Date	Optional

Relationship	Type	Destination	Inverse
owner	To One	Passenger	luggage

Constraints
identifier

```
@objc(Luggage)
final class Luggage: NSManagedObject {
    @NSManaged var identifier: UUID
    @NSManaged var weight: Float

    @NSManaged var departedAt: NSDate?
    @NSManaged var arrivedAt: NSDate?

    @NSManaged var passenger: Passenger?
}
```

Passenger

For this example, each passenger is uniquely identified by the first initial of their `givenName` and the first three letters of their `familyName`. Each `passenger` record can be related to zero or more `Luggage` records through its `luggage` relationship.

Attribute	Type	Properties
givenName	String	Length = 1
familyName	String	Length = 3

Relationship	Type	Destination	Inverse	Properties
luggage	To Many	Luggage	passenger	Optional

Constraints
givenName, familyName

```
@objc(Passenger)
final class Passenger: NSManagedObject {
    @NSManaged var givenName: String?
    @NSManaged var familyName: String?

    @NSManaged var luggage: NSSet?
}
```

Note: In reality, personal names alone aren't a reliable way to distinguish between individuals — much less just a few initials. An actual system would use a combination of personal information and government-issued documents to uniquely identify passengers. Then again, an actual system wouldn't use iCloud or scan QR codes with iPhones, either. *Example code contrivances, amiright?*

Configuring a Winning Merge Strategy

For years, one of the greatest pain points for Core Data was the awkward "fetch-or-insert-and-update" dance that was necessary to prevent duplicate records when downloading records from an external data source. It became so ingrained in our minds that many of us may not have realized that it's no longer necessary.

As of iOS 9 and macOS El Capitan, you can add unique constraints for entities in the managed object model editor. If you save a `NSManagedObjectContext` that has its `mergePolicy` property configured to `NSMergeByPropertyObjectTrumpMergePolicy`, the persistent store won't insert a new object if an object exists with the same constraint values. Instead, it takes that object's attributes and updates the existing object *while maintaining any existing relationships*.

```
context.mergePolicy = NSMergeByPropertyObjectTrumpMergePolicy
```

Scanning a QR Code

QR code is a two-dimensional barcode format. You'd be forgiven for thinking that nobody actually uses them if your experience is limited to ill-conceived usage on product advertisements.

```swift
import CoreImage
import UIKit

func scanQRCode(in uiImage: UIImage) -> Data? {
    let context = CIContext()
    let options = [
        CIDetectorAccuracy: CIDetectorAccuracyHigh
    ]
    let detector = CIDetector(ofType: CIDetectorTypeQRCode,
                              context: context,
                              options: options)

    guard let ciImage = CIImage(image: uiImage),
        let features = detector?.features(in: ciImage) else {
            return nil
    }

    for case let (qrCode as CIQRCodeFeature) in features {
        if let data = qrCode.messageString?.data(using: .utf8) {
            return data
        }
    }

    return nil
}
```

Note: For simplicity, this example uses `CIDetector` to read QR codes from static images. You could instead use the AVFoundation framework to detect barcodes in real-time from device video capture.

For an example of this, see Apple's AVCamBarcode sample code project.

Scanning the QR code seen above produces the following JSON:

```
{
    "id": "F432BDB8-D84F-4461-8362-AFF89F6C493E",
    "owner": ["D", "ZMU"],
    "weight": 42.0
}
```

Implementing Decodable in NSManagedObject Subclasses

At first glance, it would seem that Core Data and `Decodable` are incompatible. The designated initializer for `NSManagedObject`, `init(entity:insertInto:)`, takes an `NSEntityDescription` argument and an optional `NSManagedObjectContext` argument, whereas `Decodable` requires an implementation of `init(from:)`.

```
// Core Data
init(entity: NSEntityDescription,
     insertInto context: NSManagedObjectContext?) {}

// Codable
init(from decoder: Decoder) throws {}
```

Fortunately, we can use the `userInfo` property on `Decoder` to pass the missing context where it's needed.

The `userInfo` property has the type `[CodingUserInfoKey: Any]`. To add a new user info key, you can define a new type constant in an extension to `CodingUserInfoKey`.

```
extension CodingUserInfoKey {
    static let context = CodingUserInfoKey(rawValue: "context")!
}
```

After creating a decoder, you can pass the managed object context by setting the custom `.context` key on its `userInfo` property.

```
let decoder = JSONDecoder()
decoder.userInfo[.context] = mainContext()
```

Then in the implementation of `init(from:)`, get the managed object context by accessing the `.context` key of the passed `decoder`. You can then use that context to create the required `NSEntityDescription` argument for the designated initializer. After invoking `init(entity:insertInto:)`, everything looks like a normal `Decodable` implementation, with properties set using values decoded from a keyed container.

```
extension Luggage: Decodable {
    private enum CodingKeys: String, CodingKey {
        case identifier = "id"
        case weight
    }

    convenience init(from decoder: Decoder) throws {
        guard let context = decoder.userInfo[.context] as?
NSManagedObjectContext else {
            fatalError("Missing context or invalid context")
        }

        guard let entity =
NSEntityDescription.entity(forEntityName: "Luggage", in:
context) else {
            fatalError("Unknown entity Luggage in context")
        }

        self.init(entity: entity, insertInto: context)

        let container = try decoder.container(keyedBy:
CodingKeys.self)
        self.identifier = try container.decode(UUID.self,
forKey: .identifier)
        self.weight = try container.decode(Float.self, forKey:
.weight)
        self.owner = try
container.decodeIfPresent(Passenger.self, forKey: .owner)
    }
}
```

The `Luggage` class implementation of `Decodable` delegates to the corresponding implementation for `Passenger` as a result of calling the `decodeIfPresent(_:,forKey:)` method.

```
extension Passenger: Decodable {
    public convenience init(from decoder: Decoder) throws {
        guard let context = decoder.userInfo[.context] as?
NSManagedObjectContext else {
            fatalError("Missing context or invalid context")
        }

        guard let entity =
NSEntityDescription.entity(forEntityName: "Passenger", in:
context) else {
            fatalError("Unknown entity Passenger in context")
        }

        self.init(entity: entity, insertInto: context)

        var container = try decoder.unkeyedContainer()
        self.givenName = try container.decode(String.self)
        self.familyName = try container.decode(String.self)
    }
}
```

The `init(from:)` initializer for the `Passenger` class also starts by getting the managed object context from the decoder, creating an entity description, and calling the designated initializer. The difference here is that `Passenger` decodes its name components from an unkeyed container.

Finding or Creating Luggage from a Tag

The `Luggage` is tagged and the `Passenger` is present and accounted for. Our last step is to implement a tag scanner so that we can send them both on their way.

The `LuggageTagScanner` class defines a single `scan` method that takes a `UIImage`, a point (origin or destination), and a managed object context.

```
class LuggageTagScanner {
    enum Point {
        case origin, destination
    }

    func scan(image uiImage: UIImage,
              at point: Point,
              in context: NSManagedObjectContext) throws
    {
        // Get JSON data from QR code

        let decoder = JSONDecoder()
        decoder.userInfo[.context] = context

        let luggage =
            try decoder.decode(Luggage.self, from: data)

        switch point {
        case .origin:
            luggage.departedAt = NSDate()
        case .destination:
            luggage.arrivedAt = NSDate()
        }
    }
}
```

Using the method described above, we first extract the JSON data in any QR code that's detected by the scanner.

We then pass the managed object context into the `userInfo` of the decoder in order for it to be used in the initializer for `Luggage`.

If this is its first time scanning this piece of luggage, the object inserted into the context as a new record. Otherwise, it gets merged with the existing record that shares the unique identifier — thanks to `NSMergeByPropertyObjectTrumpMergePolicy`.

Luggage in hand, we then update the `departedAt` or `arrivedAt` timestamp according to the point at which it was scanned.

When put into practice, our code is small enough to fit under the seat in front of us (with room to spare!).

```
do {
    try scanner.scan(image: image,
                     at: .origin,
                     in: context)
    try context.save()
} catch {
    fatalError("\(error)")
}
```

If you're a user of `Codable` and are weighing your persistence options, Core Data is certainly one way to go. The previous chapter showed that `UserDefaults` can be quite capable, too.

If you're already using Core Data, adopting `Codable` is a great way to reduce the complexity and size of your models. It's at least easier than, say, dealing with Core Data migrations.

Recap

- `Codable` conformance cannot be synthesized for `NSManagedObject` subclasses.
- To implement a decodable managed object, pass a managed object context to the decoder in its `userInfo` property using a custom `CodingUserInfoKey` value.
- Set the `mergePolicy` of managed object contexts to `NSMergeByPropertyObjectTrumpMergePolicy` to easily reconcile persisted objects with external data sources according to unique identifiers.

Chapter 7:

Implementing a MessagePack Encoder

After years of development, researchers at Furuhashi Labs unveiled their latest invention: a device that can shrink objects down by 50–70% — and more importantly, enlarge them back to their original size — all in the blink of an eye.

Carriers are already lining up to adopt this technology as a way to improve the efficiency of their operations.

"This is a game changer," said one developer at the event. *"With the amount of traffic we handle, transporting double or triple in the same capacity could be huge."*

However, the new technology isn't without its critics.

A spokesperson for the international trade group of schema-based serialization frameworks, Schéma, noted that *"In these troubling times, developers want to be confident about what data they're sending and receiving."*

Jason Industries also issued a press release, touting their recent partnership with Brotli Group: *"Together, we can match any benchmark, and do so without requiring our customers to completely retool their workflows."*

When reached for comment, a representative for SOAP, Inc. responded with a `<Fault>` element and declined to comment further.

The problem of data serialization and interchange is much larger than requesting JSON over HTTP. There's XML and its myriad permutations. You also have binary formats, like MessagePack, CBOR, and BSON. There are schema-based mechanisms like Protocol Buffers and Avro, and then there are more comprehensive

RPC frameworks like gRPC, Thrift, and SOAP. Even if you limit yourself to JSON served over HTTP, there's a lot of performance optimization to be had with streaming, and compression with gzip or Brotli.

Exploring the full extent of this space, however, requires you to leave the comfort of the Foundation framework. This isn't really a problem for the vast majority of users, for whom the provided coders for JSON and Property Lists are sufficient.

`Codable` isn't applicable for every data serialization technology, but we can adapt it to great effect for certain text and binary formats.

We round out our guide to `Codable` by showing you how to implement your own `Encoder` type. Our example uses MessagePack, but the same approach can be applied for other formats as well.

You probably won't ever need to do this yourself, but completing this exercise is the ultimate way to understand how `Codable` actually works. And knowing that, you'll be able to weather whatever bumps may come your way.

Understanding Encoders, Decoders, and Containers

An *encoder* is an object whose type conforms to the `Encoder` protocol. It's what is passed to the `encode(to:)` method that's required by the `Encodable` protocol. The `Encoder` protocol requires read access to the `codingPath` and `userInfo` properties, as well as methods that return keyed, unkeyed, and single-value containers.

```
protocol Encoder {
    var codingPath: [CodingKey] { get }
    var userInfo: [CodingUserInfoKey : Any] { get }

    func container<Key>(keyedBy type: Key.Type) ->
        KeyedEncodingContainer<Key> where Key : CodingKey
    func unkeyedContainer() -> UnkeyedEncodingContainer
    func singleValueContainer() -> SingleValueEncodingContainer
}
```

A *decoder* conforms to the `Decoder` protocol. It's what gets passed into the `init(from:)` initializer that's required by the `Decodable` protocol. The `Decoder` protocol has the same requirements as `Encoder`, except that its methods return decoding containers, rather than encoding containers.

```
protocol Decoder {
    var codingPath: [CodingKey] { get }
    var userInfo: [CodingUserInfoKey : Any] { get }

    func container<Key>(keyedBy type: Key.Type) throws ->
        KeyedDecodingContainer<Key> where Key : CodingKey
    func unkeyedContainer() throws -> UnkeyedDecodingContainer
    func singleValueContainer() throws ->
SingleValueDecodingContainer
}
```

A *container* is an object that stores one or more values. Containers may be used for encoding Swift values into a representation or for initializing Swift values from a decoded representation.

There are three kinds of containers:

- **Single-value containers**, which store a single encodable or decodable value
- **Unkeyed containers**, which store a collection of encodable or decodable values, similar to an `Array` or `Set`
- **Keyed containers**, which store a collection of encodable or decodable key-value pairs, similar to a `Dictionary`

Containers are created by calling the `singleValueContainer()`, `unkeyedContainer()`, or `container(keyedBy:)` methods on an encoder or decoder.

Containers provide the essential functionality for encoding or decoding values for a particular data interchange format.

Encoding containers are responsible for implementing `encode(_:)`, or `encode(_:forKey:)` for keyed containers. These containers are then used to encode property values in the `encode(to:)` method. For example:

```
func encode(to encoder: Encoder) throws {
    var container = encoder.container(keyedBy: CodingKeys.self)
    try container.encode(self.manufacturer, forKey:
.manufacturer)
    try container.encode(self.model, forKey: .model)
    try container.encode(self.seats, forKey: .seats)
}
```

Decoding containers are responsible for implementing `decode(_:)`, or `decode(_:forKey:)` for keyed containers, which are used to decode and initialize property values in `init(from:)`. For example:

```
init(from decoder: Decoder) throws {
    let container = try decoder.container(keyedBy:
CodingKeys.self)
    self.manufacturer = try container.decode(String.self,
forKey: .manufacturer)
    self.model = try container.decode(String.self, forKey:
.model)
    self.seats = try container.decode(Int.self, forKey: .seats)
}
```

Container protocols require a number of methods that are overloaded for various types. Take, for example, the methods required by the `SingleValueEncodingContainer` protocol:

```
mutating func encodeNil() throws
mutating func encode(_ value: Bool) throws
mutating func encode(_ value: Int) throws
mutating func encode(_ value: Int16) throws
mutating func encode(_ value: Int32) throws
mutating func encode(_ value: Int64) throws
mutating func encode(_ value: Int8) throws
mutating func encode(_ value: UInt) throws
mutating func encode(_ value: UInt16) throws
mutating func encode(_ value: UInt32) throws
mutating func encode(_ value: UInt64) throws
mutating func encode(_ value: UInt8) throws
mutating func encode(_ value: Float) throws
mutating func encode(_ value: Double) throws
mutating func encode(_ value: String) throws
mutating func encode<T>(_ value: T) throws where T : Encodable
```

Each of these methods is called `encode(_:)`, but differ in what type of argument they take. This is an example of *method overloading*.

Containers also require a few variants of these overloaded methods, including `encodeConditional(_:)` for reference types and `encodeIfPresent(_:forKey:)` for optional types in keyed containers. Fortunately, default implementations of these are provided through a protocol extension, so you don't typically implement them yourself.

Introducing MessagePack

MessagePack[1] is a binary serialization format. It's similar to JSON, in that it can represent the same kinds of structures and values, but MessagePack can do so faster and smaller. However, as a binary format, MessagePack isn't meant to be human-readable and this can make it more difficult to work with than a text-based format like JSON.

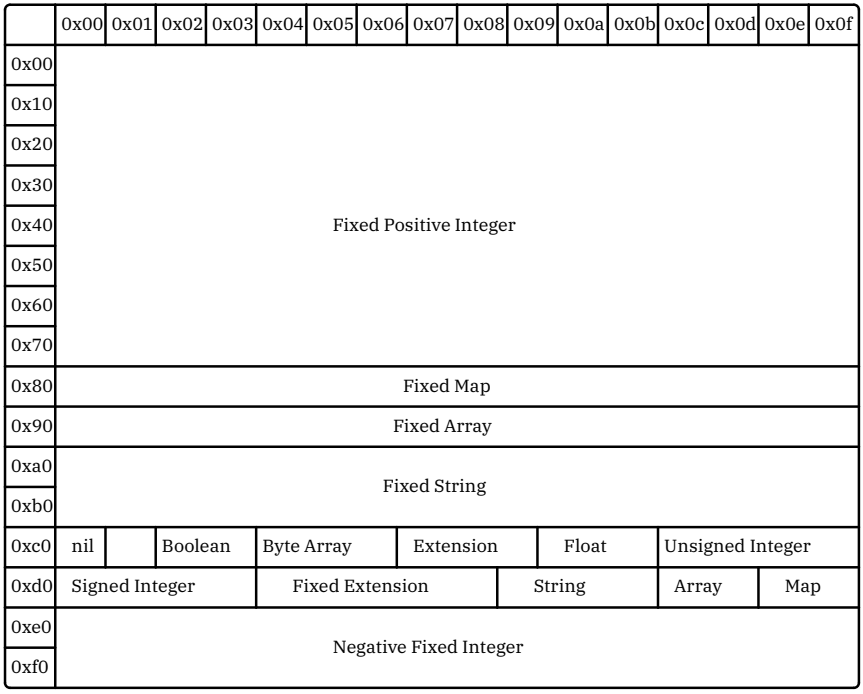
---

1. Also known as "msgpack" — a nod to its compression efficiency.

How does MessagePack represent information so efficiently? It's all thanks to a clever encoding scheme, which is described in the MessagePack Specification.

MessagePack payloads encode an object. The first byte of the payload is the *format*, which indicates the type of object and how to decode the value. Numbers, for example, indicate the number of additional bytes to read to get the value. Strings and other types may read additional bytes to determine their length and then decode their value accordingly. And some objects, like `nil`, `true`, and `false`, are encoded entirely in the format byte.

The following diagram shows the correspondence between format bytes and the types of value they represent:



There are 8 bits in a byte ($2^8 = 256$ possible values).

The 8 bits in a byte can be expressed in 8 binary (base-2) digits (0b11110000), or more conveniently, in 2 hexadecimal (base-16) digits (0xF0).

Each column in the table corresponds to the low 4 bits of the byte, or the first hexadecimal digit from the right. Each row corresponds to the high 4 bits, or the second hexadecimal digit from the right. For example, the cell labeled “nil” corresponds to the byte value 0xc0, or expressed in binary, 0b11000000. The cell labeled “Signed Integer” spans the byte values 0xd0 to 0Xd3.

One of the key features of MessagePack’s design is how it takes advantage of the full 255 values in a byte.

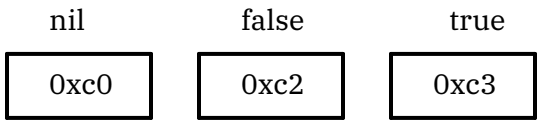
Smaller numbers are more common than larger ones, so the numbers 0 to 127 are encoded directly in the low bits of the format byte (ditto for negative numbers down to -31). Likewise, strings are more likely to be short, so similar affordances are made for strings up to 32 characters in length. The same approach is also used for arrays containing up to 15 elements and for maps with as many key-value pairs.

In order to get a better sense of how this all works in practice, let’s take a look at some examples:

Tip: If you’re less familiar with binary representations, all of this can be pretty overwhelming. However, you’re encouraged to take a few moments to try and make sense of the byte diagrams that follow. The moment everything clicks will be well worth the effort.

Nil and Boolean Values

- If the format byte is 0xc0, the value is nil.
- If the format byte is 0xc2, the value is false.
- If the format byte is 0xc3, the value is true.

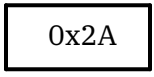


Numbers

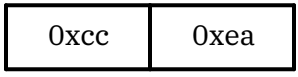
MessagePack supports signed and unsigned integers, as well as single and double precision floating point numbers.

- If the format is `positive fixint` (0x00 – 0x7f), the value is a 7-bit integer decoded from the low bits.
- If the format is `negative fixint` (0xe0 – 0xff), the value is a 5-bit negative integer decoded from the low bits.
- If the format is `int 8` (0xd0), or `uint 8` (0xcc), the value is encoded in the next 1 byte.
- If the format is `int 16` (0xd1), or `uint 16` (0xcd), the value is encoded in the next 2 bytes.
- If the format is `int 32` (0xd2), `uint 32` (0xce), or `float 32` (0xca), the value is encoded in the next 4 bytes.
- If the format is `int 64` (0xd3), `uint 64` (0xcf), or `float 64` (0xcb), the value is encoded in the next 8 bytes.

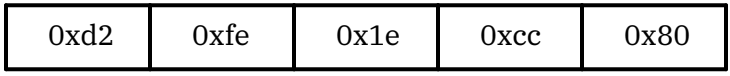
fixint (42)



uint8 (234)



int32 (-31536000)



Strings

Strings are encoded as UTF-8 bytes.

- If the format is `fixstr` (0xa0 – 0xbf), the length is a 5-bit integer decoded from the low bits.
- If the format is `str 8` (0xd9), `str 16` (0xda), or `str 32` (0xdb), the length is encoded in the next 1, 2, or 4 bytes.

fixstr (Hello)

0xa5	0x48	0x65	0x6C	0x6C	0x6F

Code	Glyph	Decimal	Name
U+0048	H	72	Latin Capital letter H
U+0065	e	101	Latin Small Letter E
U+006C	l	108	Latin Small Letter L
U+006F	o	111	Latin Small Letter O

Arrays

- If the format is `fixarray` (0x90 – 0x9f), the length is a 4-bit integer decoded from the low bits.
- If the format is `array 16` (0xdc) or `array 32` (0xdd), the length is encoded in the next 2 or 4 bytes.

fixarray ([1, 2, 3, 4, 5])

0x95	0x01	0x02	0x03	0x04	0x05

Maps

Maps are stored like arrays, with their elements alternating between keys and values.

- If the format is `fixmap` (0x80 – 0x8f), the length is double the 4-bit integer decoded from the low bits.
- If the format is `map 16` (0xde) or `map 32` (0xdf), the length is double the value encoded in the next 2 or 4 bytes.

fixmap ([a: 1, b: 2])

0x82	0xA1	0x61	0x01	0xA1	0x62	0x02

Byte Arrays

- If the format is `bin 8` (0xc4), `bin 16` (0xc5), or `bin 32` (0xc6), the length is encoded in the next 1, 2, or 4 bytes, and the raw binary content is encoded in that length of bytes.

bin 8 [␀]

0xc4	0x01	0xc0

Extension Types

- If the format is `fixext 1` (0xd4), `fixext 2` (0xd5), `fixext 4` (0xd6), `fixext 8` (0xd7), or `fixext 16` (0xd8), the next byte indicates the extension type, and the contents are a byte array consisting of the subsequent 1, 2, 4, 8, or 16 bytes.
- If the format is `ext8` (0xc7), `ext16` (0xc8), or `ext32` (0xc9), the next byte indicates the extension type, the length is encoded in the next 1, 2, or 4 bytes, and the contents are a byte array of that length.

Note: You may use the format values from 0 to 127 (0x00 – 0x7F) for any application-specific types. The values from -128 to -1 (0x80 – 0xFF) are reserved for predefined types. Currently, the only predefined type is -1 for timestamps.

Timestamps

Timestamps are an extension defined in the MessagePack specification. They're encoded as the number of seconds and nanoseconds since the Unix epoch (1970-01-01 00:00:00 GMT).

- If the format is `fixext 4` (0xd6) and the extension format is -1 (0xFF), the next 4 bytes encode an unsigned integer value for the number of seconds.
- If the format is `fixext 8` (0xd6) and the extension format is -1 (0xFF), the next 8 bytes encode two unsigned integer values for the number of seconds and nanoseconds.
- If the format is `ext8` (0xc7), the length is 12 (0x0c), and the extension format is -1 (0xFF), the next 4 and 8 bytes encode unsigned integer values for the number of seconds and nanoseconds.

timestamp 32 (2018-04-20 12:00:00 -0700)

0xd6	0xff	0x5a	0xda	0x38	0xb0

Now that we have a solid understanding of the MessagePack format, we can implement it for `Codable`.

Tip: For the purposes of this discussion, we'll show how to implement an encoder. The process of creating a decoder is pretty much the same in reverse and is left as an exercise for the reader. A full implementation of the MessagePack encoder and decoder is available on GitHub.

Creating Your Own Encoder

You could read all about the `Decoder` and `Encoder` protocols and their associated container types, and still have no clue where to start. Conspicuously absent in all this abstraction is anything that seems vaguely related to actually reading or writing data.

So where do we actually *do* anything?

The documentation isn't much help. There's plenty of discussion about using `Codable` for your own types with the JSON and Property

List coders provided by Foundation, but no information about `Encoder`, `Decoder`, or containers beyond what you can gather by looking at their names.

But why read the docs, anyway? Especially when you can just look at the source…

Meh. Never mind.

The implementation details for `JSONEncoder` are obscured by layers of boxing and unboxing between NS objects and Swift values using `JSONSerialization`. Same for Property List coders, which amounts to a lot of ceremony around calls to `PropertyListSerialization` class methods.

At least we can always depend on Xcode to save the day.

😐

If we declare a type that conforms to, say, `SingleValueEncodingContainer`, Xcode warns that the type doesn't conform to that protocol and offers to stub out the missing properties and methods. Problem is, only a few of them are added, resulting in even more "fix-it" warnings from Xcode to stub out missing properties and methods.

On top of all of this — the vague documentation, the muddled reference implementation, the ineffectual tooling — there are a number of pitfalls relating to how things are named.

For one, `JSONEncoder` doesn't actually conform to the `Encoder` protocol. Likewise for `JSONDecoder` and the `Decoder` protocol.

Instead, each of these classes delegates their core functionality to private, underscore-prefixed types that satisfy the protocol requirements. This misdirection is perhaps the most significant source of misunderstanding for how `Codable` really works.

That `decode(_:from:)` method that you're so familiar with? Not actually a part of `Codable`, but rather a conspiracy between `JSONDecoder` and `PropertyListDecoder`.

Consider also this delightful inconsistency:

`SingleValueEncodingContainer`	Protocol
`UnkeyedEncodingContainer`	Protocol
`KeyedEncodingContainer`	Structure

What you're actually meant to implement is `KeyedEncodingContainerProtocol`, and then use this type to initialize the type-erased structure that's returned by the `container(keyedBy:)` method on `Encoder`.

Defining the Public and Private Types for an Encoder

We define a public `MessagePackEncoder` type that provides a convenient API for encoding objects into data, just like the `JSONEncoder` or `PropertyListEncoder` classes in Foundation.[2]

Internally, the `_MessagePackEncoder` class is responsible for conforming to the `Encoder` protocol. The three container types are nested in that internal type. This namespacing allows them to be concise, and not collide with the protocols they adopt.

```
public class MessagePackEncoder {
    public func encode<T>(_ value: T) throws -> Data
                        where T : Encodable
}

class _MessagePackEncoder: Encoder {
    class SingleValueContainer: SingleValueEncodingContainer
    class UnkeyedContainer: UnkeyedEncodingContainer
    class KeyedContainer<Key>: KeyedEncodingContainerProtocol
            where Key: CodingKey
}
```

2. Just because we're admonishing the pattern doesn't mean we won't follow it. Consistency is important, after all.

This is a skeleton of what we're going to build. We'll start with the single-value container, and then complete the other container types. With all those in place, the public and internal encoder types quickly fall into place.

Tip: Containers are implemented as classes because they act as shared, mutable state between the encoder that creates them, and the `Encodable` type that interacts with them in the `encode(_:)` method.

Defining a Common Container Protocol

It will eventually be helpful for us to reason about each of the three MessagePack container types on similar terms. To do that, we'll define a `MessagePackEncodingContainer` protocol.

The core responsibility for each container type is to return the representation of their contents. For MessagePack, the encoded representation is `Data`.

```
protocol MessagePackEncodingContainer {
    var data: Data { get }
}
```

Look for how each container defines its `data` property, and how everything ties together in the encoder implementation at the end.

Implementing a Single-Value Encoding Container

The `SingleValueContainer` class is responsible for encoding values like `nil`, `true`, `false`, integers, floating-point numbers, strings, dates (as timestamps), and data (as byte arrays).

Each `SingleValueContainer` object is initialized with the `codingPath` and `userInfo` of the encoder. The `codingPath` property tracks the internal state of the encoder, which can be helpful for debugging.

The `userInfo` property offers a mechanism by which users can pass additional configuration options to change the behavior of the encoder.

```
class SingleValueContainer: SingleValueEncodingContainer {
    var codingPath: [CodingKey]
    var userInfo: [CodingUserInfoKey: Any]

    init(codingPath: [CodingKey],
         userInfo: [CodingUserInfoKey : Any])
    {
        self.codingPath = codingPath
        self.userInfo = userInfo
    }

    // ...
}
```

Managing Internal Storage

When a value is encoded to a single-value container, its data is appended to a private `storage` property. A `checkCanEncode(value:)` method and a corresponding Boolean property are defined as a way to ensure that only one value is written to `storage`.

```
private var storage: Data = Data()

fileprivate var canEncodeNewValue = true
fileprivate func checkCanEncode(value: Any?) throws {
    guard self.canEncodeNewValue else {
        let context =
            EncodingError.Context(
                codingPath: self.codingPath,
                debugDescription: "Cannot encode multiple
values."
            )
        throw EncodingError.invalidValue(value as Any, context)
    }
}
```

Each encoding method in `SingleValueContainer` starts with a call to `checkCanEncode(value:)` and a deferred assignment of `canEncodeNewValue` to be executed when the method returns.

```
try checkCanEncode(value: nil)
defer { self.canEncodeNewValue = false }
```

Tip: For brevity, these repeated lines are omitted in the code listings that follow.

Encoding nil, true, and false

Encoding `nil` takes no effort at all. Simply append the corresponding format byte, 0xc0.

```
func encodeNil() throws {
    storage.append(0xc0)
}
```

Booleans are also a piece of cake. Here we use a `switch` statement to make it abundantly clear which format byte corresponds to which Boolean value.

```
func encode(_ value: Bool) throws {
    switch value {
    case false:
        storage.append(0xc2)
    case true:
        storage.append(0xc3)
    }
}
```

Encoding Numbers

In MessagePack, numbers are encoded in big-endian order, with the most significant byte (the byte with the largest value) at the beginning. For convenience, we can add a computed `bytes` property to signed and unsigned integer types in an extension on `FixedWidthInteger`.

```
extension FixedWidthInteger {
    var bytes: [UInt8] {
        let capacity = MemoryLayout<Self>.size
        var mutableValue = self.bigEndian
        return withUnsafePointer(to: &mutableValue) {
            return $0.withMemoryRebound(to: UInt8.self,
capacity: capacity) {
                return Array(UnsafeBufferPointer(start: $0,
count: capacity))
            }
        }
    }
}
```

We can then define corresponding properties on `Float` and `Double`.

```
extension Float {
    var bytes: [UInt8] {
        return self.bitPattern.bytes
    }
}

extension Double {
    var bytes: [UInt8] {
        return self.bitPattern.bytes
    }
}
```

According to the MessagePack specification:[3]

3. The “SHOULD” used here is defined by RFC 2119.

> *If an object can be represented in multiple possible output formats, serializers SHOULD use the format which represents the data in the smallest number of bytes.*

For numeric types with specified widths, it makes sense to encode them as is, even if they could be expressed by a smaller type.

```
func encode(_ value: Int8) throws {
    storage.append(0xd0)
    storage.append(contentsOf: value.bytes)
}

func encode(_ value: Int16) throws {
    storage.append(0xd1)
    storage.append(contentsOf: value.bytes)
}

// ... etc.
```

However, for a type like `Int` or `UInt`, we should attempt to find the smallest encoding — which may be a fixed positive or negative integer if it falls within range.

```
func encode(_ value: Int) throws {
    if let int8 = Int8(exactly: value) {
        if (int8 >= 0 && int8 <= 127) {
            self.storage.append(UInt8(int8))
        } else if (int8 < 0 && int8 >= -31) {
            self.storage.append(0xe0 +
                (0x1f & UInt8(truncatingIfNeeded: int8))
            )
        } else {
            try encode(int8)
        }
    } else if let int16 = Int16(exactly: value) {
        try encode(int16)
    } else if let int32 = Int32(exactly: value) {
        try encode(int32)
    } else if let int64 = Int64(exactly: value) {
        try encode(int64)
    } else {
        let context = EncodingError.Context(
            codingPath: self.codingPath,
            debugDescription: "Cannot encode integer \(value)."
        )
        throw EncodingError.invalidValue(value, context)
    }
}
```

Encoding Strings

Strings are the most complex of the single-value types to encode. Start by encoding the string into UTF-8 data, append the corresponding format byte (and length, if necessary), and then append the string data.

```
func encode(_ value: String) throws {
  guard let data = value.data(using: .utf8) else {
    let context = EncodingError.Context(
      codingPath: self.codingPath,
      debugDescription: "Cannot encode as UTF-8."
    )
    throw EncodingError.invalidValue(value, context)
  }

  let length = data.count
  if let uint8 = UInt8(exactly: length) {
    if (uint8 <= 31) {
      self.storage.append(0xa0 + uint8)
    } else {
      self.storage.append(0xd9)
      self.storage.append(contentsOf: uint8.bytes)
    }
  } else if let uint16 = UInt16(exactly: length) {
      self.storage.append(0xda)
      self.storage.append(contentsOf: uint16.bytes)
  } else if let uint32 = UInt32(exactly: length) {
      self.storage.append(0xdb)
      self.storage.append(contentsOf: uint32.bytes)
  } else {
      let context = EncodingError.Context(
        codingPath: self.codingPath,
        debugDescription:
          "Can't encode string with length \(length)."
      )
      throw EncodingError.invalidValue(value, context)
  }

  self.storage.append(data)
}
```

Computing the MessagePack Representation

For `SingleValueContainer`, the `data` property requirement of the `MessagePackEncodingContainer` protocol is simply an alias of its `storage` property.

```
extension SingleValueContainer:
    MessagePackEncodingContainer
{
    var data: Data {
        return self.storage
    }
}
```

The main reason for not simply calling the property `storage` rather than `data` is for consistency with the two remaining container types.

Implementing an Unkeyed Encoding Container

The `UnkeyedContainer` class starts much as the same as `SingleValueContainer`, with `codingPath` and `userInfo` properties that are specified in the initializer.

```
class UnkeyedContainer: UnkeyedEncodingContainer {
    var codingPath: [CodingKey]
    var userInfo: [CodingUserInfoKey: Any]

    init(codingPath: [CodingKey],
         userInfo: [CodingUserInfoKey : Any])
    {
        self.codingPath = codingPath
        self.userInfo = userInfo
    }

    // ...
}
```

Defining an Index CodingKey Type

Despite being an *un*-keyed container, there are still benefits to defining a custom `CodingKey` type. The nested `Index` type is

initialized with an `Int` and passed to any nested containers it creates as part of its coding path. Should encoding fail for a particular value, the coding path would help identify exactly where.

```
struct UnkeyedContainer.Index: CodingKey {
    var intValue: Int?

    var stringValue: String {
        return "\(self.intValue!)"
    }

    init?(intValue: Int) {
        self.intValue = intValue
    }

    init?(stringValue: String) {
        return nil
    }
}
```

A `nestedCodingPath` property serves as a convenience method when it comes time to initialize nested containers.

```
private var nestedCodingPath: [CodingKey] {
    return self.codingPath + [Index(intValue: self.count)!]
}
```

Storing Containers

Whereas the storage for a `SingleValueContainer` was a `Data` buffer, `UnkeyedContainer` uses an `Array`. The required `count` property is implemented as a computed property that returns the length of `storage`.

```
var storage: [MessagePackEncodingContainer] = []

var count: Int {
    return storage.count
}
```

Creating Nesting Containers

The `UnkeyedEncodingContainer` protocol requires methods for creating nested keyed and unkeyed containers. In both cases, we use the `nestedCodingPath` property we defined to compute a new coding path for the current index.

After creating a container, it's appended to the `storage` property. In the case of the `nestedContainer(keyedBy:)` method, note that the return value is an instance of the type-erased `KeyedEncodingContainer` structure discussed previously.

```
func nestedUnkeyedContainer() -> UnkeyedEncodingContainer {
  let container = UnkeyedContainer(
                    codingPath: self.nestedCodingPath,
                    userInfo: self.userInfo
                  )
  self.storage.append(container)

  return container
}

func nestedContainer<NestedKey>(keyedBy keyType:
NestedKey.Type) ->
        KeyedEncodingContainer<NestedKey>
        where NestedKey : CodingKey
{
  let container = KeyedContainer<NestedKey>(
                    codingPath: self.nestedCodingPath,
                    userInfo: self.userInfo
                  )
  self.storage.append(container)

  return KeyedEncodingContainer(container)
}
```

Encoding Values into an Unkeyed Container

Since we already keep track of nested keyed and unkeyed containers, we may as well do the same for single-value containers. In fact, it presents a rather elegant solution. By delegating responsibility for encoding single values to `SingleValueContainer`, we can consolidate all of that logic into one location.

```
func nestedSingleValueContainer()
    -> SingleValueEncodingContainer
{
  let container = SingleValueContainer(
                    codingPath: self.nestedCodingPath,
                    userInfo: self.userInfo
                  )
  self.storage.append(container)

  return container
}
```

The `UnkeyedEncodingContainer` protocol also requires a daunting list of `encode(_:)` methods, but we can satisfy all of those requirements in a single generic overload:

```
func encode<T>(_ value: T) throws where T : Encodable {
    var container = self.nestedSingleValueContainer()
    try container.encode(value)
}
```

Nice and easy, right?

We still have to implement `encodeNil()`, though, as it's the one method not covered by the generic overload.

```
func encodeNil() throws {
    var container = self.nestedSingleValueContainer()
    try container.encodeNil()
}
```

What happens to all of these single values? They'll all be accounted for when it comes time to compute the final representation.

Computing the MessagePack Representation

For `UnkeyedContainer`, we satisfy the `data` property requirement of `MessagePackEncodingContainer` with a computed property that dynamically builds up a representation from its contents.

The format byte for a MessagePack array depends on the number elements it contains. For 15 and under, the count can be stored in the low bits of the format byte; otherwise, it's stored in a 16- or 32-bit unsigned integer. After writing the format and length, it's just a matter of appending the data of each stored container.

```
extension UnkeyedContainer: MessagePackEncodingContainer {
  var data: Data {
    var data = Data()

    let length = storage.count
    if let uint16 = UInt16(exactly: length) {
      if uint16 <= 15 {
        data.append(UInt8(0x90 + uint16))
      } else {
        data.append(0xdc)
        data.append(contentsOf: uint16.bytes)
      }
    } else if let uint32 = UInt32(exactly: length) {
      data.append(0xdc)
      data.append(contentsOf: uint32.bytes)
    } else {
      fatalError()
    }

    for container in storage {
      data.append(container.data)
    }

    return data
  }
}
```

Implementing a Keyed Encoding Container

The implementation for `KeyedContainer` is nearly identical to that of `UnkeyedContainer`.

Like any container, it has `codingPath` and `userInfo` properties that are set in the initializer.

```
class KeyedContainer<Key> where Key: CodingKey {
    var codingPath: [CodingKey]
    var userInfo: [CodingUserInfoKey: Any]

    init(codingPath: [CodingKey],
         userInfo: [CodingUserInfoKey : Any])
    {
        self.codingPath = codingPath
        self.userInfo = userInfo
    }
}
```

As you might extrapolate from `UnkeyedContainer` using an `Array` for `storage`, `KeyedContainer` uses a `Dictionary`.

```
var storage: [String: MessagePackEncodingContainer] = [:]
```

Note: The keys for `storage` use `String` instead of the associated `Key` type because `CodingKey` doesn't conform to `Hashable`, and it's less work to use a key's `stringValue` property than to provide conformance through an extension.

Like the `nestedCodingPath` property on `UnkeyedContainer`, the `nestedCodingPath(forKey:)` method will make it easier to create nested containers.

```
private func nestedCodingPath(forKey key: CodingKey) ->
    [CodingKey]
{
    return self.codingPath + [key]
}
```

Storing Containers by Key

The implementation for creating nested containers is pretty much the same as for `UnkeyedContainer`, except that containers are stored in a dictionary by key instead of by being appended to

an array. We can safely do this without checking for existing membership because, unlike arbitrary `String` values, the provided `Key` values should be unique in this context.

```
func nestedUnkeyedContainer(forKey key: Key) ->
    UnkeyedEncodingContainer
{
  let container =
        UnkeyedContainer(
          codingPath: self.nestedCodingPath(forKey: key),
          userInfo: self.userInfo
        )
  self.storage[key.stringValue] = container

  return container
}

func nestedContainer<NestedKey>(
        keyedBy keyType: NestedKey.Type,
        forKey key: Key
     ) -> KeyedEncodingContainer<NestedKey>
     where NestedKey : CodingKey
{
    let container = KeyedContainer<NestedKey>(
        codingPath: self.nestedCodingPath(forKey: key),
        userInfo: self.userInfo
    )
    self.storage[key.stringValue] = container

    return KeyedEncodingContainer(container)
}
```

Encoding Single Values by Key

We can reuse our trick from `UnkeyedContainer` of delegating to `SingleValueContainer` for single values. A `nestedSingleValueContainer(forKey:)` convenience method creates and returns the container after first storing it.

```
private func nestedSingleValueContainer(forKey key: Key)
    -> SingleValueEncodingContainer
{
    let container = SingleValueContainer(
        codingPath: self.nestedCodingPath(forKey: key),
        userInfo: self.userInfo
    )
    self.storage[key.stringValue] = container

    return container
}
```

Once again, a generic override makes short work of satisfying the `encode(_:forKey:)` requirements of the protocol.

```
func encode<T>(_ value: T, forKey key: Key) throws where T :
Encodable {
    var container = self.nestedSingleValueContainer(forKey: key)
    try container.encode(value)
}
```

...with one exception, of course (pesky `nil`).

```
func encodeNil(forKey key: Key) throws {
    var container =
        self.nestedSingleValueContainer(forKey: key)
    try container.encodeNil()
}
```

Computing the MessagePack Representation

To satisfy the `data` property required by the `MessagePackEncoding Container` protocol, we again define a computed property that builds up the MessagePack representation by iterating over the containers in `storage`.

The format byte for a MessagePack map depends on the number elements it contains. If there are 15 or fewer key-value pairs, the `fixmap` format (0x80) can be used with the count stored in the low bits. If not, then the length is stored in a 16- or 32-bit unsigned integer. After that, keys and values are written in alternating order until the collection is exhausted.

```
extension UnkeyedContainer: MessagePackEncodingContainer {
  var data: Data {
    var data = Data()

    let length = storage.count
    if let uint16 = UInt16(exactly: length) {
      if length <= 15 {
          data.append(0x80 + UInt8(length))
      } else {
          data.append(0xdc)
          data.append(contentsOf: uint16.bytes)
      }
    } else if let uint32 = UInt32(exactly: length) {
      data.append(0xdd)
      data.append(contentsOf: uint32.bytes)
    } else {
      fatalError()
    }

    for (key, container) in self.storage {
      let keyContainer =
          SingleValueContainer(
            codingPath: self.codingPath,
            userInfo: self.userInfo
          )
      try! keyContainer.encode(key)
      data.append(keyContainer.data)

      data.append(container.data)
    }

    return data
  }
}
```

The one awkward part of this implementation is the extra container needed to encode the key. There are alternative approaches that may be cleaner, but this is fine for now.

Implementing Encoder

With all the prerequisites in place, it's time to implement the `Encoder` itself. Let's start with that all-too-familiar refrain of `codingPath` and `userInfo` initialization — this time seeing where all it all starts.

```
class _MessagePackEncoder: Encoder {
    var codingPath: [CodingKey] = []
    var userInfo: [CodingUserInfoKey : Any] = [:]
}
```

Creating Containers

...or rather, container. Singular. Each instance of an `Encoder` type should have at most a single top-level container.

Per the documentation:

> *You must use only one kind of top-level encoding container.*

It's unclear whether this actually means "no more than one top-level encoding container, period". For MessagePack, it makes sense to encode only a single top-level object, so we'll add an assertion in `willSet` to guarantee that at most one container is ever created.

```
fileprivate var container:
    MessagePackEncodingContainer?
{
    willSet {
        precondition(self.container == nil)
    }
}
```

The actual implementations for the required `singleValueContainer()`, `unkeyedContainer()`, and `container(keyedBy:)` methods are pretty boring — just a matter of creating a container, storing it in a property, and returning it.

```
func singleValueContainer() ->
    SingleValueEncodingContainer
{
    let container = SingleValueContainer(
                        codingPath: self.codingPath,
                        userInfo: self.userInfo
                    )
    self.container = container

    return container
}
```

```
func unkeyedContainer() ->
    UnkeyedEncodingContainer
{
    let container = UnkeyedContainer(
                        codingPath: self.codingPath,
                        userInfo: self.userInfo
                    )
    self.container = container

    return container
}
```

```
func container<Key>(keyedBy type: Key.Type) ->
        KeyedEncodingContainer<Key>
        where Key : CodingKey
{
    let container = KeyedContainer<Key>(
                        codingPath: self.codingPath,
                        userInfo: self.userInfo
                    )
    self.container = container

    return KeyedEncodingContainer(container)
}
```

Computing Encoded Data

This is it: the moment we've all been waiting for. Now we finally implement the linchpin that ties everything together. Here's the method for getting the MessagePack representation of a container:

```
var data: Data {
    return container?.data ?? Data()
}
```

Pretty underwhelming, right?

`Encoder` types have it pretty rough. They sulk around in the shadows of their user-visible, non-underscored counterparts. But when it comes time to actually do something, all of the interesting work is delegated to containers.

What kind of capricious designer would deign to weigh such burdens on its own creation? Itai Ferber, Michael LeHew, and Tony Parker, you say? Well then. Carry on. (Keep up the great work, gents!)

Wrapping Things Up with MessagePackEncoder

We're just a small step away from trying out our MessagePack encoder.

To match the expected interface of its JSON and Property List compatriots, let's create a `MessagePackEncoder` class with an `encode(_:)` method that returns `Data`:

```
class MessagePackEncoder {
  func encode(_ value: Encodable) throws -> Data {
    let encoder = _MessagePackEncoder()
    try value.encode(to: encoder)
    return encoder.data
  }
}
```

We could tack on some additional properties for configuration, but right now we're too excited to test this out. Also, MessagePack doesn't leave much to be desired as far as configuration.[4]

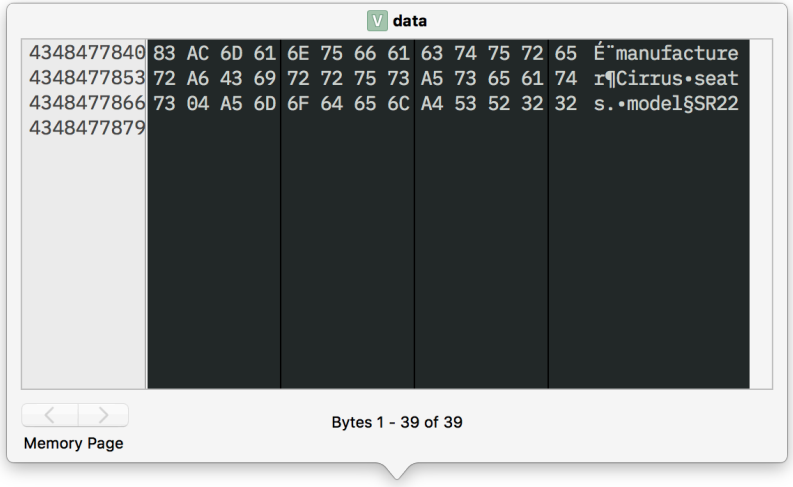
Let's bring back our `Plane` model from Chapter 1 for our maiden voyage with `MessagePackEncoder` — this time flying in a Cirrus SR22:

```
struct Plane: Codable {
    var manufacturer: String
    var model: String
    var seats: Int
}

let plane = Plane(manufacturer: "Cirrus", model: "SR22", seats:
4)

let encoder = MessagePackEncoder()
let data = try! encoder.encode(plane)
```

Printing a `Data` object isn't particularly illuminating, so let's instead set a breakpoint in Xcode and use Quick Look to view the `data` variable.

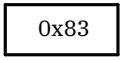


4. お疲れ様、古橋さん!

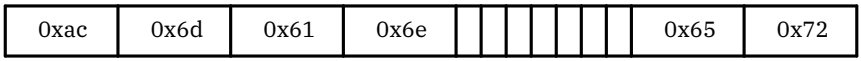
```
83 AC 6D 61  6E 75 66 61  63 74 75 72  65 72 A6 43
69 72 72 75  73 A5 73 65  61 74 73 04  A5 6D 6F 64
65 6C A4 53  52 32 32
```

You might balk at this wall of hexadecimal bytes, thinking it beyond human comprehension. But let's use what we've learned about the MessagePack format to break things down into individual parts to understand the whole.

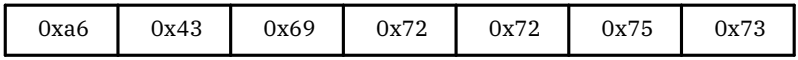
fixmap (3 key-value pairs)



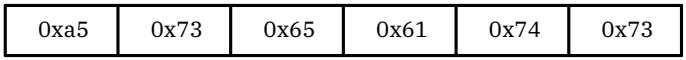
Key: fixstr (manufacturer)



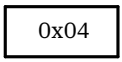
Value: fixstr (Cirrus)



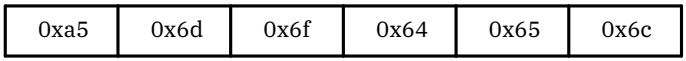
Key: fixstr (seats)



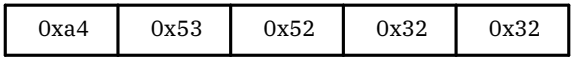
Value: fixint (4)



Key: fixstr (model)



Value: fixstr (SR22)



As impressive as it is for a human to decipher this output, it's much more practical to automate this by way of a `MessagePackDecoder`. If you're up for a challenge, see if you can implement this yourself applying what you learned about encoder (but in reverse).

To help get you started, here's a skeleton of what you'll implement:

```
public class MessagePackDecoder {
    public func decode<T>(_ type: T.Type,
                          from data: Data) throws -> T
                        where T : Decodable
}

class _MessagePackDecoder: Decoder {
    class SingleValueContainer: SingleValueDecodingContainer
    class UnkeyedContainer: UnkeyedDecodingContainer
    class KeyedContainer<Key>: KeyedContainer
            where Key: CodingKey
}
```

As mentioned at the start of this chapter, the world of data interchange is much bigger than JSON and HTTP. And we've demonstrated that `Codable` can be a great fit for at least one of those technologies. Just remember that the existence of a convenient client interface like `Codable` shouldn't be the deciding factor in how you architect your technology stack.

Recap

- The problem of data serialization and interchange is much larger than requesting JSON over HTTP.
- Binary serialization formats can offer improved performance and storage efficiency over text-based formats at the expense of human readability.
- Implementing a custom `Encoder` or `Decoder` requires a significant amount of implementation, but doing so allows you to work with other representation formats in the same, convenient way that you might with JSON or property lists.