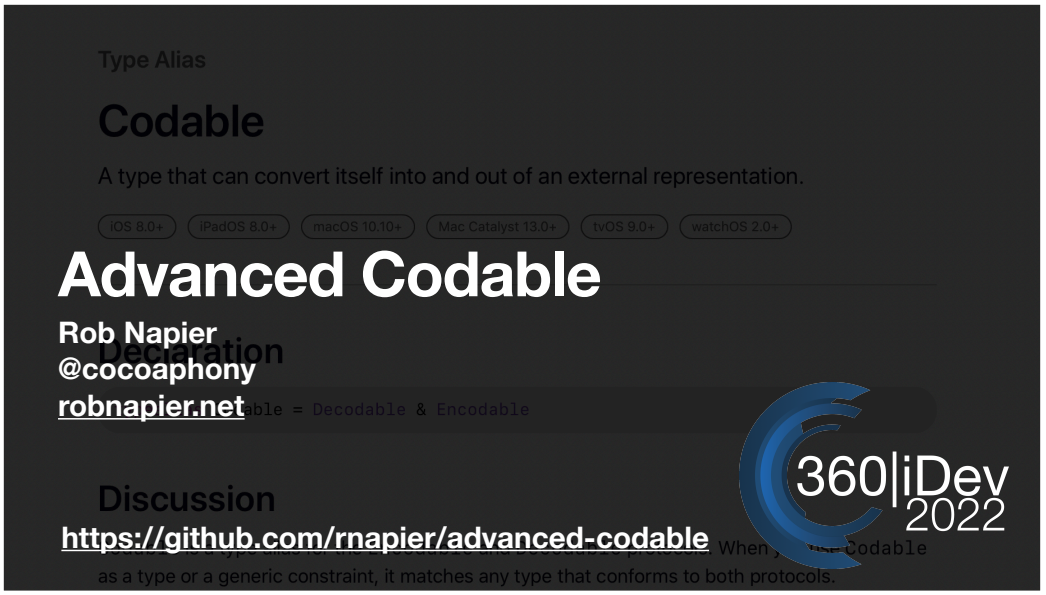
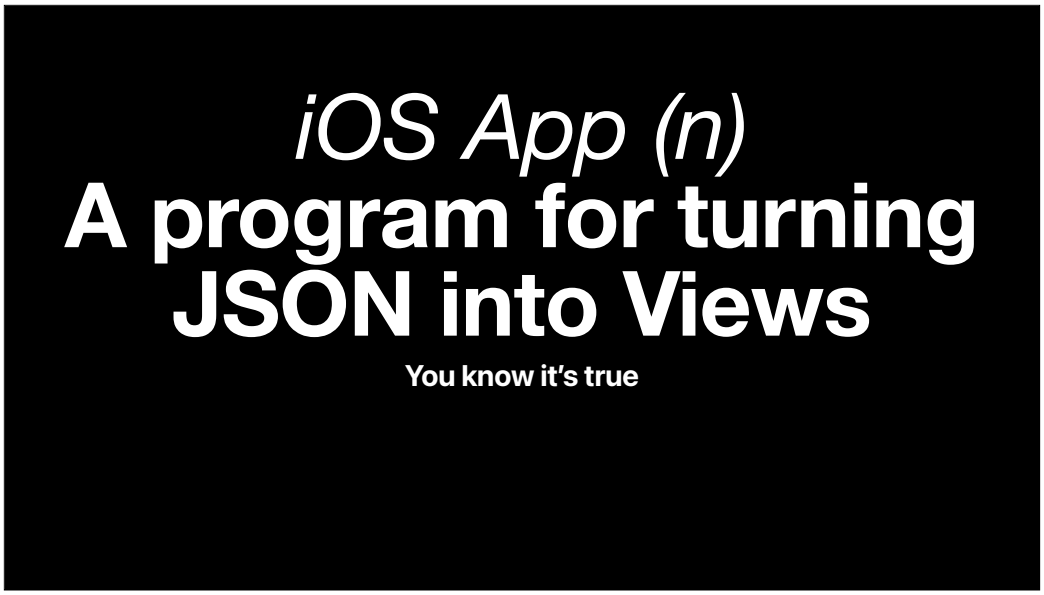

Color Check
Tiny, Dim Code
Tiny Code
Tiny, Highlighted Code
Tiny, Red-Highlighted Code
Small, Dim Code
Small Code
Dim Code
Code
Highlighted Code

You're all in the right session, right? Oh.

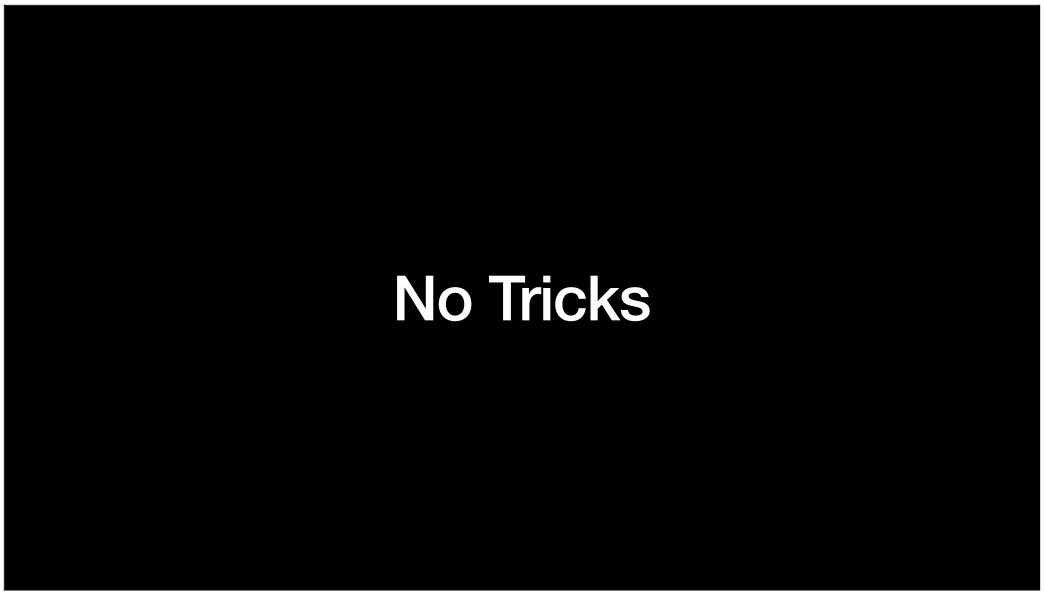


I never can spell. There we go. this is "Advanced Codable."

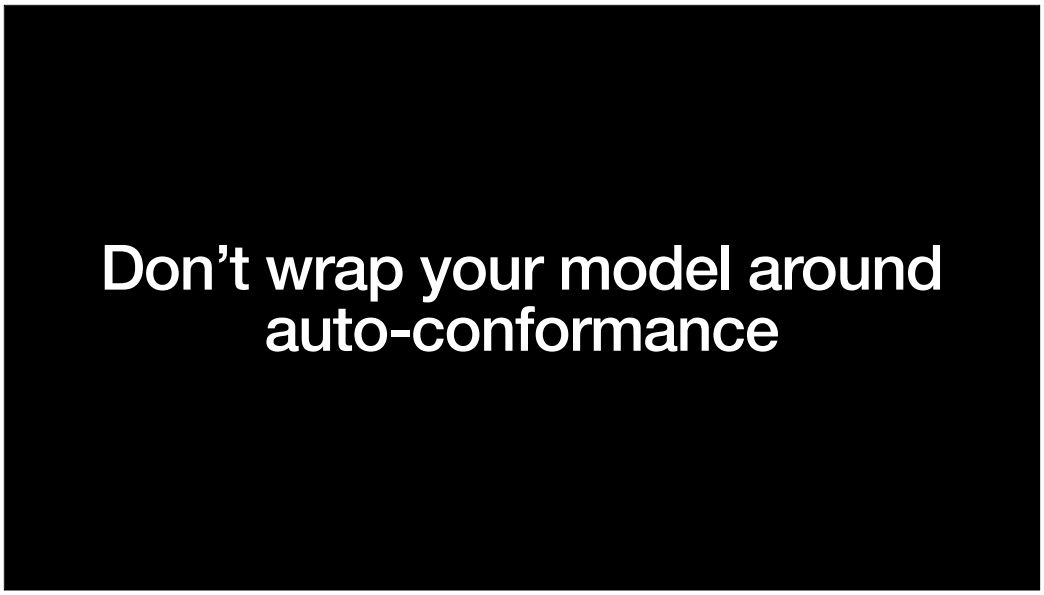
What is an iOS app?



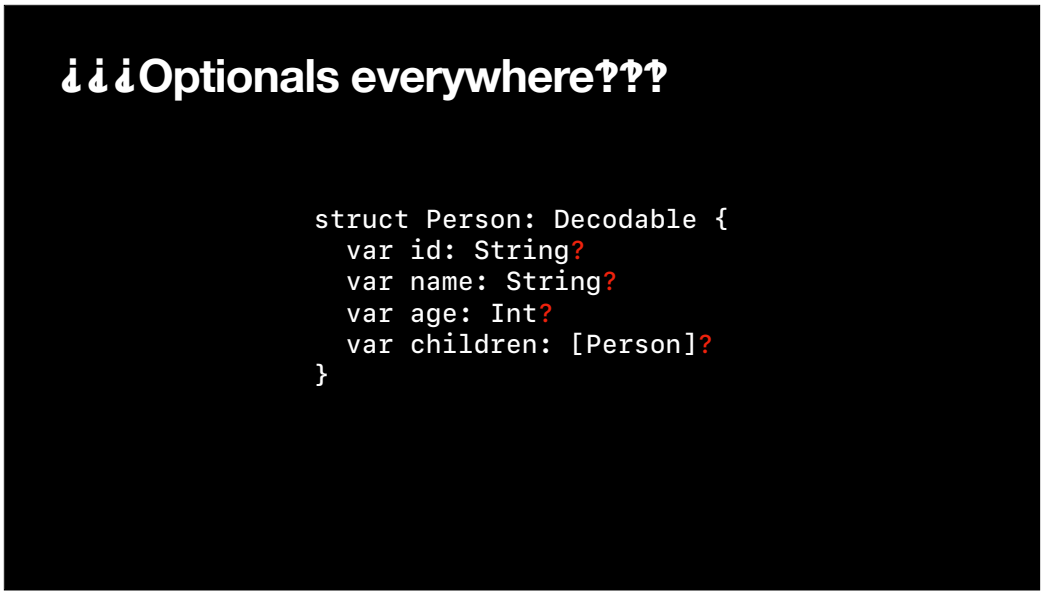
It's a program for turning JSON into Views. That's what we do all day. And the tool we use for that is Codable and specifically JSONDecoder. I expect most of you know the basics. Automatic conformances. The basics of writing a manual conformance. container(keyedBy:), decode(forKey:). Basic CodingKeys. So today I'm going to focus on some more complicated coding problems.



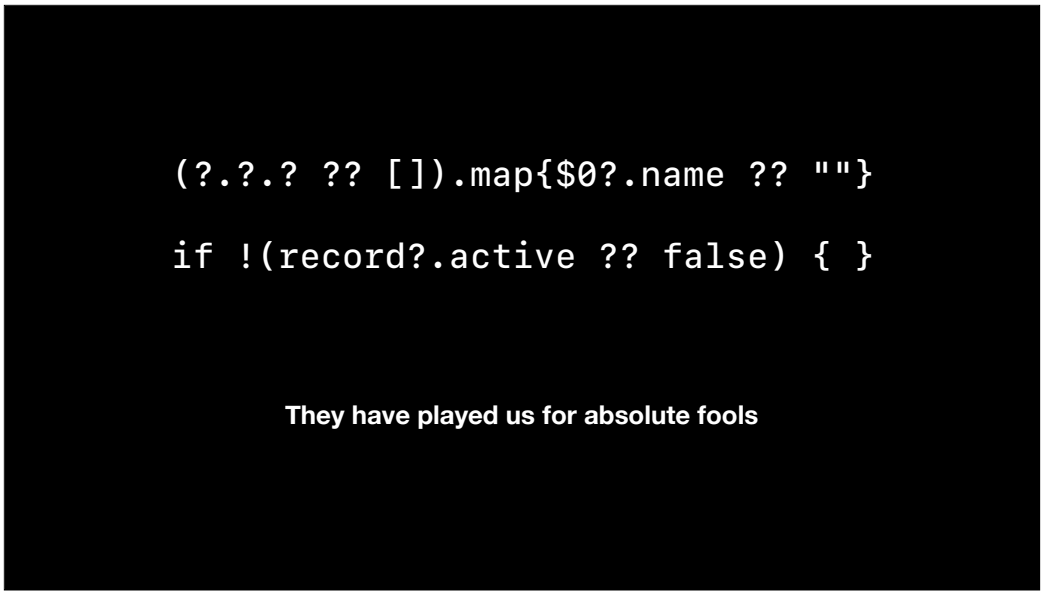
You should think of this talk as more “how to think about Codable” than a list of tips and tricks. I’m going to show a lot of code, a **lot** of code, and I’ll post it all for you, along with the whole text of this talk. I don’t want you to drown in the code examples. The point is to help you write your own code when you need it.



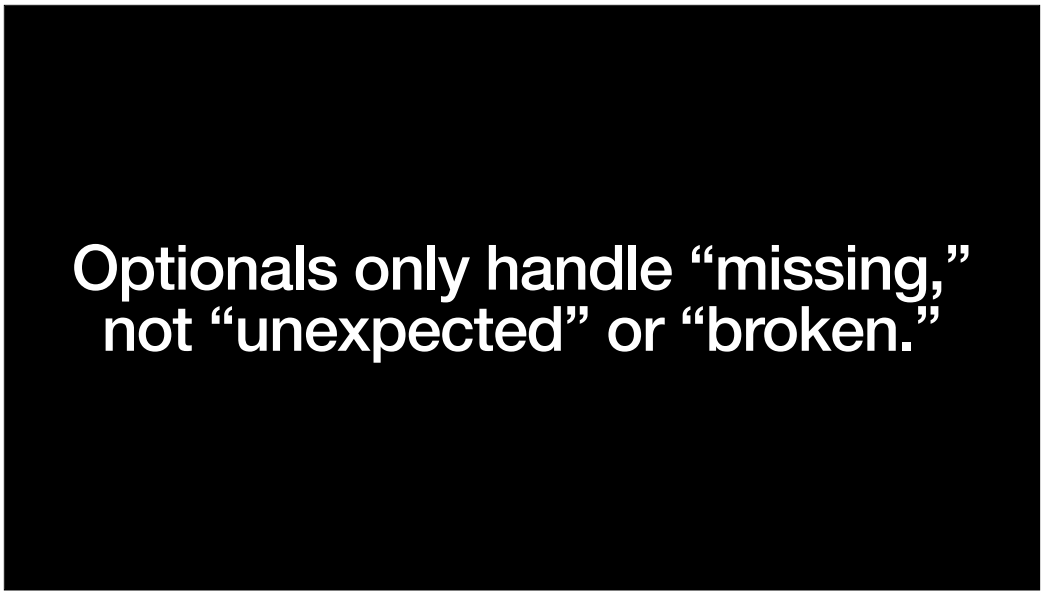
Swift can auto-conform types to Codable for us, and for some projects, that’s fine. But unfortunately, the moment you need anything it doesn’t offer, you lose **all** the automatic conformance and have to write everything by hand. That’s frustrating, but if you take one thing from this talk, it’s this: Don’t wrap yourself around auto-conformance. Writing conformances by hand is usually not very hard. It’s just a little tedious. And the things we do to avoid writing that simple, fast, easy to understand code, often come back to bite us.



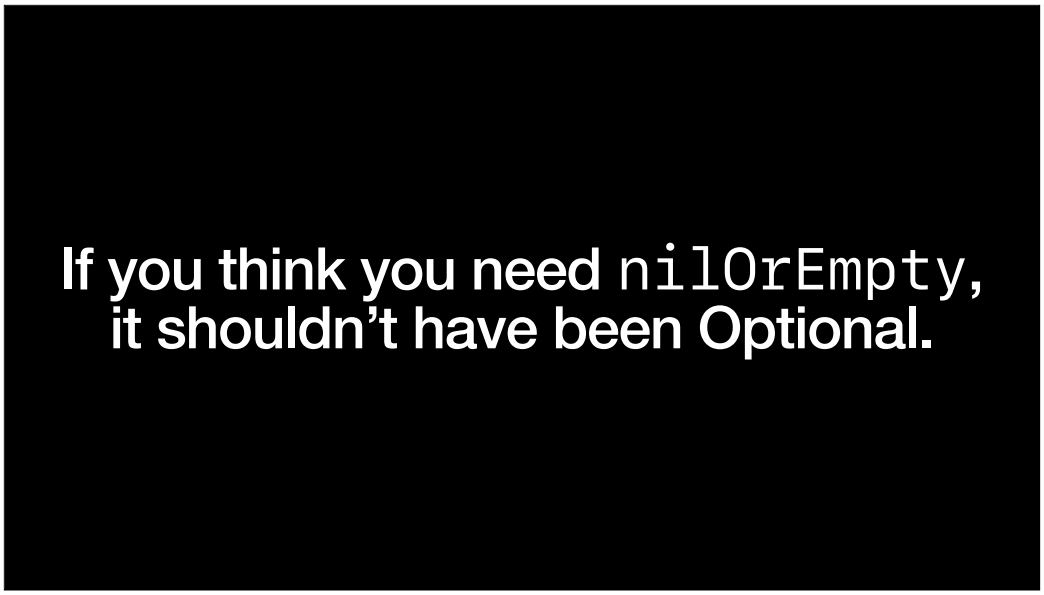
The most common trick I see people use is to mark every property as Optional. This has a bunch of problems. First, are all the properties **really** optional? I mean, is the ID optional? Can you really proceed without knowing **any** of the values? Wouldn't it be better to stop early rather than having corrupted records play havoc with the rest of your program?



Which is the second problem. Making everything Optional makes your UI code very complicated and overflowing with question marks and compactMaps and the like. It’s better to figure out your error handling in one place, validate the data, fix it if you must, and then let the rest of the program work with simple, clean data.

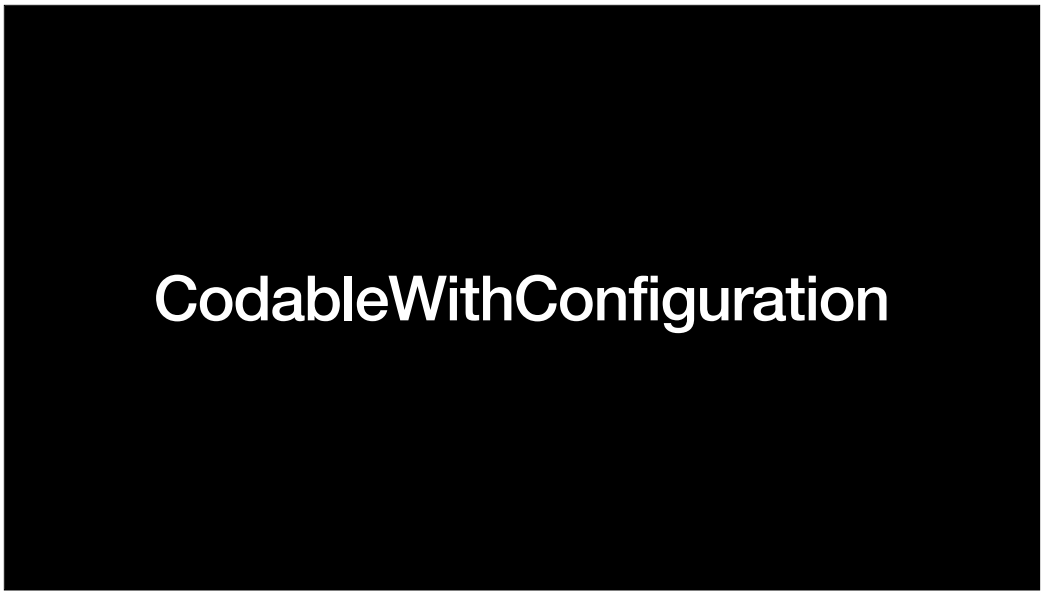


Marking everything Optional doesn’t really protect you that much anyway. If an enum is an unexpected value, or something you thought would always be an Int is a String sometimes, the whole parser will still throw, and you still need to deal with that. Optional just means it’s ok for the value to be missing.
Even in that case, the missing value should generally be replaced with a default. And we’ll talk about how to do that. That default should only be nil if “missing” means something different than “empty.”

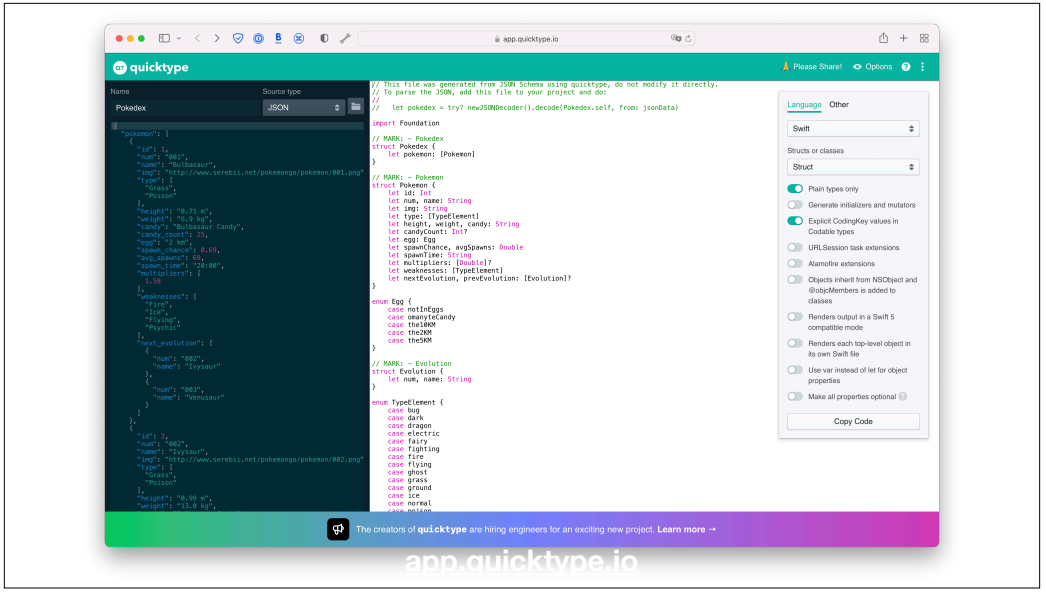


If you find yourself using nil-coalescing a lot, or you find yourself wanting to make a “nilOrEmpty” extension, then it probably shouldn’t have been Optional in the first place.

The other trick I see people try is property wrappers. They seem so promising, and when we finally get more general purpose code annotations, I expect they will be how Codable is implemented. But today, I find they have little corner cases that can get you stuck. Some crash if you don't write your initializers correctly. Some don't work with enums. They all force you to wrap your model types around encoding concerns, which is the opposite of what I want. And I don't know of any that handle errors well. If they work for you, that's great. There's nothing wrong with going down that road, but I won't be talking about that today.

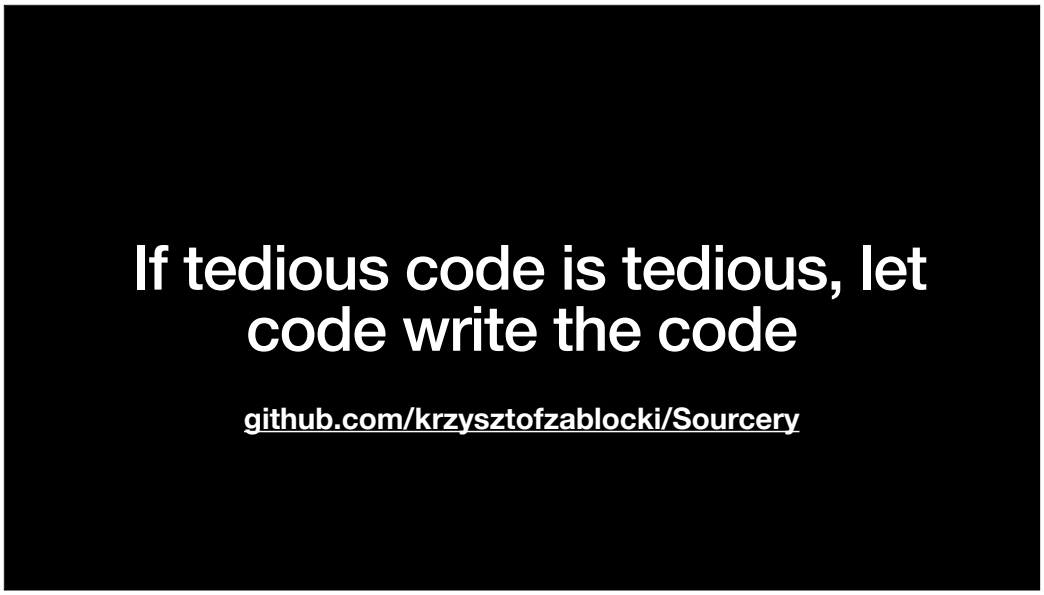


I haven’t seen many people try to use it, but the other trick I won’t be discussing today is CodableWithConfiguration. If you haven’t heard about it, it was added in iOS 15 and is used for encoding AttributedStrings. I haven’t been able to figure out many other uses for it. So, thing you might want to dig into, but I don’t know.



If you have a lot of very large objects and really don't want to maintain hand-written code for them, there are a couple of tools I recommend.

Your first stop for basic Codable problems should be quicktype. This is a free service and will take arbitrary JSON you paste in, and it will spit out Swift code, Or about 20 other languages. This is a ridiculously powerful tool if you have a reasonably well-behaved format and just need basic functionality.

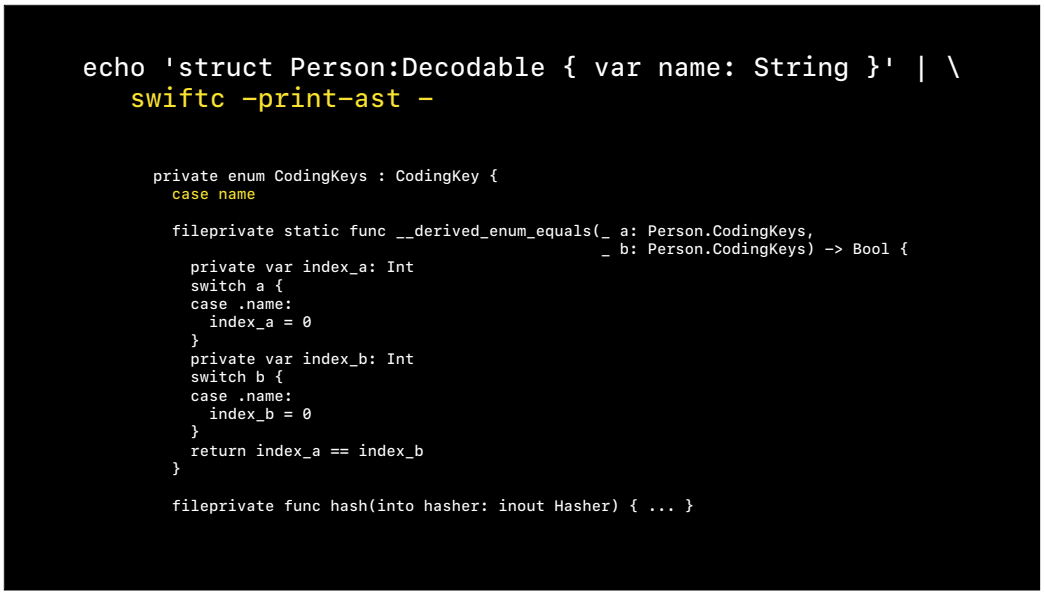


If that isn’t enough, and writing the conformances is still too much, my recommendation is code generation. Sourcery has a bunch of templates, but more importantly you can adapt it to your needs. If writing the code is the headache, let something write the code for you. But don’t bend your model around what auto-conformance offers.

CodingKey

```
public protocol CodingKey : CustomDebugStringConvertible,
                            CustomStringConvertible,
                            Sendable {
 var stringValue: String { get }
  init?(stringValue: String)
  var intValue: Int? { get }
  init?(intValue: Int)
}
```

Enough setup. Let’s talk about CodingKey. It’s a protocol. It is not a magic enum thing. Coding keys do not have to be enums. There is some special compiler magic for when CodingKeys are enums, but it’s just a protocol.
It’s something that wraps a string value, that may also wrap an int value. That’s it.



You can see how the compiler generates automatic coding keys by running it through `swiftc -print-ast`. That will output the Abstract Syntax Tree as Swift code, after automatic conformances are injected.

It'll generate an enum, and a few dozen lines worth of conformances. It can easily be a couple of hundred lines of code. You'll notice that Equatable implementation is based on a switch statement and numeric values rather than the string comparison you might have expected. Comparing integers is a lot faster than strings.

```swift
private init?(stringValue: String) {
  switch stringValue {
  case "name":
    self = Person.CodingKeys.name
  default:
    return nil
  }
}

private init?(intValue: Int) {
  return nil
}
```

```swift
fileprivate var intValue: Int? {
  get { return nil }
}

fileprivate var stringValue: String {
  get {
    switch self {
    case .name:
      return "name"
    }
  }
}
```

But the important part is that it generates a very simple stringValue initializer and property. For int values, it just returns nil. It doesn't support int values. What if we made a struct that did the same thing.

```swift
public struct AnyCodingKey: CodingKey {
  public let stringValue: String

  public var intValue: Int?

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

  public init?(intValue: Int) { return nil }
}
```

Glad you asked. This is my absolute favorite custom type and I use it all the time. This is its most minimal form, and the way you'll see it pretty often in the wild. Lots of people have invented this under different names.
The form I use is a little more complicated. It supports Int keys, and most importantly it conforms to ExpressibleByStringLiteral so that I can use quoted strings as keys. But at its heart, it's just this, a coding key that can wrap any String and so can be the key of any JSON object.

```swift
struct Person: Decodable {
  var name: String
  var age: Int
  var children: [Person]?
}
```

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

  name     = try c.decode(String.self, forKey: "name")
  age      = try c.decode(Int.self, forKey: "age")
  children = try c.decodeIfPresent([Person].self, forKey: "children")
}
```

Why do I love this struct so much? Well, for one, it gets rid of the need for defining CodingKeys. The hard-coded string literals may cause you to freak out a little. But here's the thing. If you're only implementing Decodable or only implementing Encodable, that string will occur in exactly one place which is exactly the place you use it. That's better than creating a hand-written constant somewhere else. And I generally recommend that you only implement Encodable or Decodable unless you need them both. Unneeded conformances just add headaches and overhead.

```swift
extension Decoder {
  public func anyKeyedContainer() throws ->
  KeyedDecodingContainer<AnyCodingKey> {

    try container(keyedBy: AnyCodingKey.self)

  }
}




init(from decoder: Decoder) throws {
  let c = try decoder.anyKeyedContainer()

  name     = try c.decode(String.self, forKey: "name")
  age      = try c.decode(Int.self, forKey: "age")
  children = try c.decodeIfPresent([Person].self, forKey: "children")
}
```

I admit is a bit of code to write for every conformance. How can we make tighter?

First, I use AnyCodingKey a lot, so an `anyKeyedContainer` extension is nice.

```swift
extension KeyedDecodingContainer {
  public func decode<T: Decodable>(_ key: Key) throws -> T {
    try self.decode(T.self, forKey: key)
  }

  public func decodeIfPresent<T: Decodable>(_ key: Key) throws -> T? {
    try self.decodeIfPresent(T.self, forKey: key)
  }
}



init(from decoder: Decoder) throws {
  let c = try decoder.anyKeyedContainer()

  name     = try c.decode("name")
  age      = try c.decode("age")
  children = try c.decodeIfPresent("children")
}
```

And while I don’t like generic methods to rely exclusively on return values for type inference, it is very nice to make it an option, and it gets rid a lot of noise.

```
extension KeyedDecodingContainer {
  public subscript<T: Decodable>(key: Key) -> T {
    get throws { try self.decode(T.self, forKey: key) }
  }

  public subscript<T: Decodable>(ifPresent key: Key) -> T? {
      get throws { try self.decodeIfPresent(T.self, forKey: key) }
  }
}
```

```
init(from decoder: Decoder) throws {
  let c = try decoder.anyKeyedContainer()

  name     = try c["name"]
  age      = try c["age"]
  children = try c[ifPresent: "children"]
}
```

And if we're getting rid of noise, we can go further and use subscripts to make this very compact and focused.

```swift
extension KeyedDecodingContainer {
  public subscript<T: Decodable>(key: Key, default value: T) -> T {
    get throws { try self.decodeIfPresent(T.self, forKey: key) ?? value }
  }
}
```

```swift
init(from decoder: Decoder) throws {
  let c = try decoder.anyKeyedContainer()

  name     = try c["name"]
  age      = try c["age"]
  children = try c["children", default: []]
}
```

But why stop there? What about default values? Now there's no need for an Optional.

```
extension KeyedDecodingContainer {
  public subscript<T>(orEmpty key: Key) -> T
  where T: Decodable & RangeReplaceableCollection {
      get throws { try self.decodeIfPresent(T.self, forKey: key) ?? .init()}
  }
}
```

```
init(from decoder: Decoder) throws {
  let c = try decoder.anyKeyedContainer()

  name     = try c["name"]
  age      = try c["age"]
  children = try c[orEmpty: "children"]
}
```

Of course, “missing should be empty” is really common for collections, so if you want, you could add an `orEmpty` version. This is an appropriate place to check for `nilOrEmpty` rather than everywhere else in the program.

```
extension KeyedDecodingContainer {
  public subscript<T: Decodable>(key: Key, failWith value: T) -> T {
    get {
      do { return try decode(T.self, forKey: key) }
      catch {
        assertionFailure("Could not decode \(key): \(error)")
        return value
      }
    }
  }
}


init(from decoder: Decoder) throws {
  let c = try decoder.anyKeyedContainer()

  name     = c["name", failWith: "UNKNOWN"]
  age      = try c["age"]
  children = try c[orEmpty: "children"]
}
```

What about errors? We understandably want to recover gracefully from server-side surprises when we can, but being too aggressive in hiding errors makes debugging very difficult.
This `failWith:` subscript asserts in debug if there's any problem decoding `name`, but in release it will fall back with "UNKNOWN." It'll still throw in release if there's a problem decoding `age`. So you can mix-and-match your error handling.

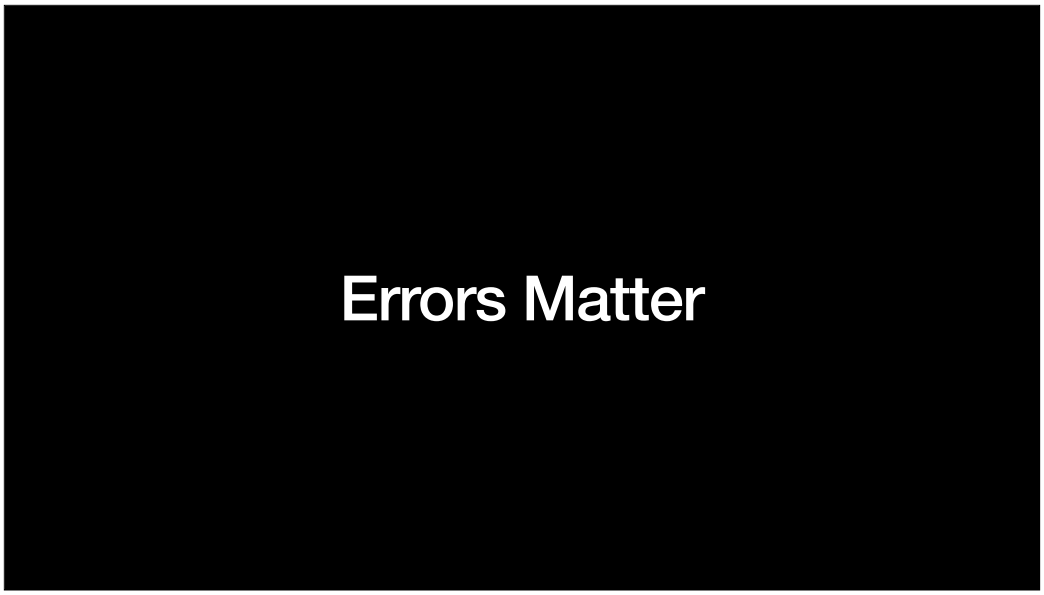
```swift
extension UnkeyedDecodingContainer {
  mutating func decodeCompactingErrors<C>(_: C.Type = C.self) -> C
  where C: RangeReplaceableCollection, C.Element: Decodable {

    var result: C = .init()
    while !isAtEnd {
      do { result.append(try self.decode(C.Element.self)) }
      catch { assertionFailure("...") }
    }
    return result
  }
}

extension KeyedDecodingContainer {
  public subscript<C>(compactingErrors key: Key) -> C
  where C : RangeReplaceableCollection, C.Element: Decodable {
    get {
      guard contains(key) else { return .init() }
      do {
        var container = try nestedUnkeyedContainer(forKey: key)
        return container.decodeCompactingErrors()
      } catch {
        assertionFailure("Could not decode \(key): \(error)")
        return .init()
      }
    }
  }
}
```

What about the `children` property? I can add extensions on keyed and unkeyed containers that assert in debug, to make it easier to detect bad assumptions about the format, but in release just filters out errors. You could add logging here even in release to help you debug field errors.

```
init(from decoder: Decoder) throws {
  let c = try decoder.anyKeyedContainer()

  name     = c["name", failWith: "UNKNOWN"]
  age      = try c["age"]
  children = c[compactingErrors: "children"]
}
```

With those together, the decoding logic looks like this. Of course you could also add extensions to handle date formatters, or base-64 decoders. There’s no need to configure this stuff globally in the Decoder. You can express it directly in the types where it belongs, but the code doesn’t have to be long.

The point isn’t some particular set of extensions. This isn't a library. It's a pattern. What makes sense depends on the kind of data you’re working with.

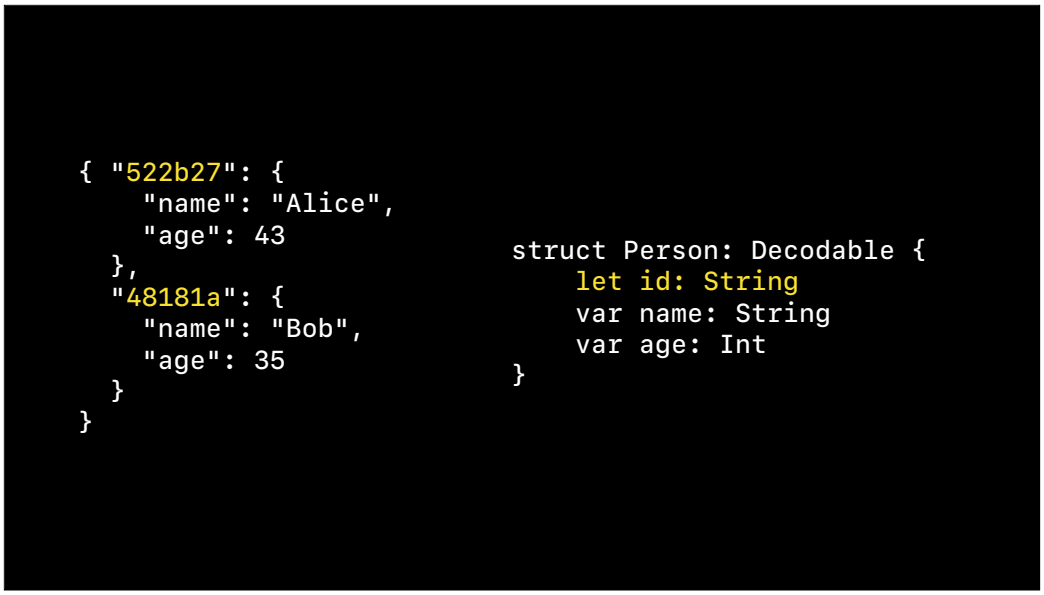


Even for a single API, different responses may need different error handling. If there's a corrupted record in your list of product recommendations, it's probably best to just throw away that one record and show the rest. But if there's a corrupted record in a data syncing response, you probably don't want to trust any of it. You certainly wouldn't want to quietly delete local records. There's no one-size-fits-all to error handling, and you really need to test how your app behaves when you get unexpected data.

Yes, this forces you to think about every property. I'm saying you need to think about every property and decide what to do about errors. And if that becomes too much code to write, I'm suggesting Sourcery to write it for you rather than trying to avoid it.

AnyCodingKey is really nice, and I use it a lot, but another powerful tool for handling arbitrary keys is a simple String-keyed dictionary.

Say you have JSON where the keys aren't fixed. They're some kind of identifier. Now, maybe you don't care about the identifiers. Then this is really simple. It's just a Dictionary of String to Person. Don't overthink this stuff. Sometimes people do.



But ok, what if the ID is part of Person object? How can you get an array of Person out of this JSON object? You could create a “PartialPerson” type and decode this as a Dictionary of those, and then glue things together into a Person, but…yuck. That’s not what we’re looking for.

```
struct Person: Decodable {
  let id: String
  var name: String
  var age: Int

  enum CodingKeys: CodingKey { case name, age }

  init(from decoder: Decoder) throws {
    guard let id = decoder.codingPath.last?.stringValue else {
      throw DecodingError.dataCorrupted(.init(codingPath: decoder.codingPath,
                                              debugDescription: "..."))
      }

      self.id = id

      let c     = try decoder.container(keyedBy: CodingKeys.self)
      self.name = try c[.name]
      self.age  = try c[.age]
  }
}


try JSONDecoder().decode([String: Person].self, from: json).values
```

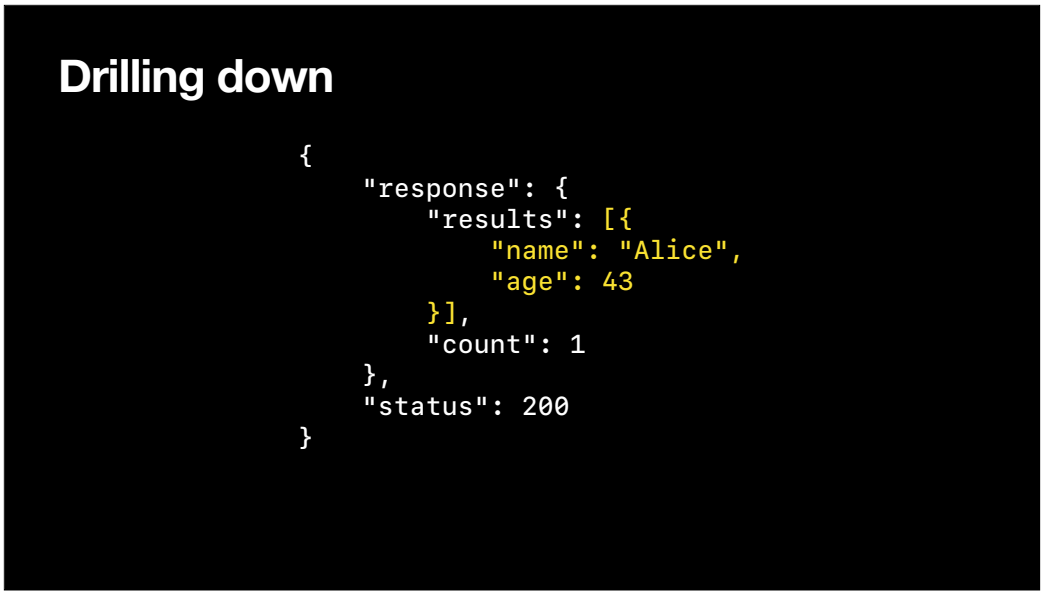
Here’s a better way.

```
struct Person: Decodable {
  let id: String
  var name: String
  var age: Int

  enum CodingKeys: CodingKey { case name, age }

  init(from decoder: Decoder) throws {
    guard let id = decoder.codingPath.last?.stringValue else {
      throw DecodingError.dataCorrupted(.init(codingPath: decoder.codingPath,
                                              debugDescription: "..."))
      }

      self.id = id

      let c     = try decoder.container(keyedBy: CodingKeys.self)
      self.name = try c[.name]
      self.age  = try c[.age]
  }
}


try JSONDecoder().decode([String: Person].self, from: json).values
```

The Decoder's coding path keeps track of the keys that got you to this point. Coding keys are just wrappers around strings, which is what we need. Decode a dictionary of String to Person, and while decoding each Person, take the last element of the coding path. That's the ID.

```
struct Person: Decodable {
  let id: String
  var name: String
  var age: Int

  enum CodingKeys: CodingKey { case name, age }

  init(from decoder: Decoder) throws {
    guard let id = decoder.codingPath.last?.stringValue else {
      throw DecodingError.dataCorrupted(.init(codingPath: decoder.codingPath,
                                              debugDescription: "..."))
      }

      self.id = id

      let c     = try decoder.container(keyedBy: CodingKeys.self)
      self.name = try c[.name]
      self.age  = try c[.age]
  }
}


try JSONDecoder().decode([String: Person].self, from: json).values
```

Decode the rest, and at the end, take the values of the dictionary.



Now sometimes a simple Dictionary doesn’t quite work out. For example, sometimes the values are different types. Like here, all you want is the results, and you want to throw away the rest.

You can’t decode a `[String: [String: [Person]]`, here because the `status` and `count` keys are in the way.

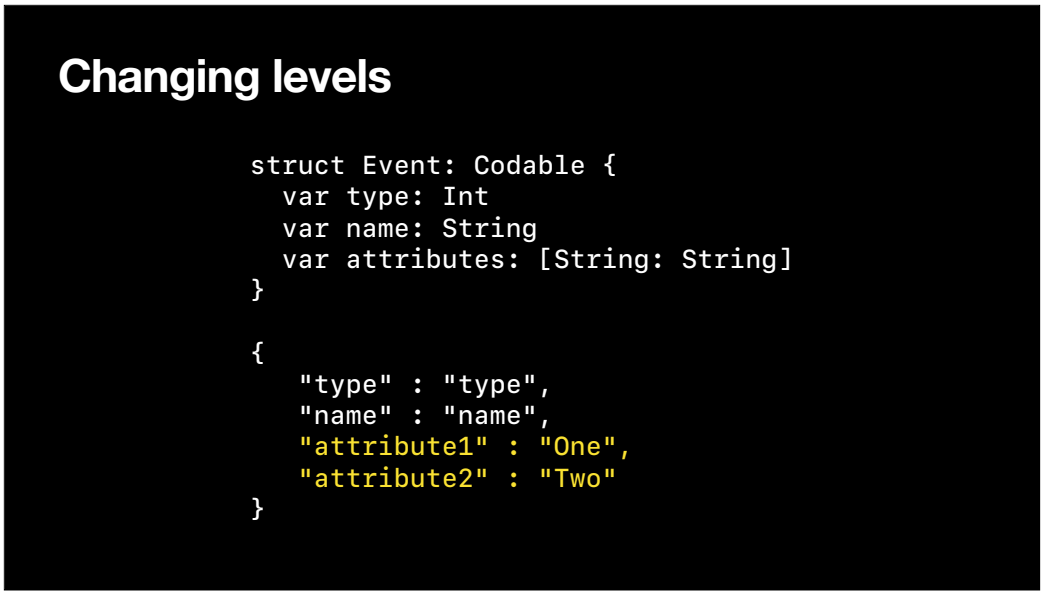
```
struct Person: Decodable {
    var name: String
    var age: Int
}

struct PersonResponse: Decodable {
  var results: [Person]
  init(from decoder: Decoder) throws {
    results = try decoder.anyKeyedContainer()
        .nestedAnyContainer("response")
        .decode("results")
  }
}

try JSONDecoder().decode(PersonResponse.self,
                         from: json).results
```

Create a PersonResponse wrapper to handle the response, and then drill down to the level you want using `.nestedContainer`, and decode it. If you've ever built layer upon layer of wrapper structs just to get to the level you want, this is an easier way.

```
extension Array: Decodable where Element == Person {
  init(from decoder: Decoder) throws {
    // ...
  }
}
```

```
Conformance of 'Array<Element>' to protocol 'Decodable'
conflicts with that stated in the type's module 'Swift' and
will be ignored; there cannot be more than one conformance,
even with different conditional bounds
```

Since the final output we want is an array of Person, it's really tempting to try to write an extension like this to have special handling for that case. But it just doesn't work. You'll get a warning…(build)
…and it's not kidding. You can't specialize conformances this way. Array conforms to Codable when its Element conforms to Codable. You can't create any other conformance to Codable, not even by adding extra `where` clauses. You're going to need a wrapper type.



Another really common question I run into is how to deal with JSON that is structured differently than your model objects. Say you have a this event type with a bunch of arbitrary properties all at the same level as some fixed properties. How do we handle this?

```
enum CodingKeys: CodingKey {
  case type, name
}

func encode(to encoder: Encoder) throws {
  var c = encoder.container(keyedBy: CodingKeys.self)
  try c.encode(type, forKey: .type)
  try c.encode(name, forKey: .name)

  try attributes.encode(to: encoder)
}
```

Encoding is very straight-forward. Encode the explicit parameters, and then let the Dictionary encode itself into the same encoder, at the same level.

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

  self.type = try explicitContainer[.type]
  self.name = try explicitContainer[.name]

  let attributeContainer = try decoder.anyKeyedContainer()

  var attributes: [String: String] = [:]
  for key in attributeContainer.allKeys
          where CodingKeys(stringValue: key.stringValue) == nil {
    let value = try attributeContainer.decode(String.self, forKey: key)
    attributes[key.stringValue] = value
  }
  self.attributes = attributes
}
```

Decoding is a little more work since you need to filter out the explicit keys, but it’s still pretty straightforward.

```
enum CodingKeys: CodingKey, CaseIterable {
  case type, name
}

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

  self.type = try explicitContainer[.type]
  self.name = try explicitContainer[.name]

  let attributeContainer = try decoder.anyKeyedContainer()

  var attributes: [String: String] = [:]
  for key in attributeContainer.allKeys
          where CodingKeys(stringValue: key.stringValue) == nil {
    let value = try attributeContainer.decode(String.self, forKey: key)
    attributes[key.stringValue] = value
  }
  self.attributes = attributes
}
```

Make a container for the explicit keys, and decode them.

```
enum CodingKeys: CodingKey, CaseIterable {
  case type, name
}

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

  self.type = try explicitContainer[.type]
  self.name = try explicitContainer[.name]

  let attributeContainer = try decoder.anyKeyedContainer()

  var attributes: [String: String] = [:]
  for key in attributeContainer.allKeys
          where CodingKeys(stringValue: key.stringValue) == nil {
    let value = try attributeContainer.decode(String.self, forKey: key)
    attributes[key.stringValue] = value
  }
  self.attributes = attributes
}
```

Make an any-keyed container.

```
enum CodingKeys: CodingKey, CaseIterable {
  case type, name
}

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

  self.type = try explicitContainer[.type]
  self.name = try explicitContainer[.name]

  let attributeContainer = try decoder.anyKeyedContainer()

  var attributes: [String: String] = [:]
  for key in attributeContainer.allKeys
          where CodingKeys(stringValue: key.stringValue) == nil {
    let value = try attributeContainer.decode(String.self, forKey: key)
    attributes[key.stringValue] = value
  }
  self.attributes = attributes
}
```

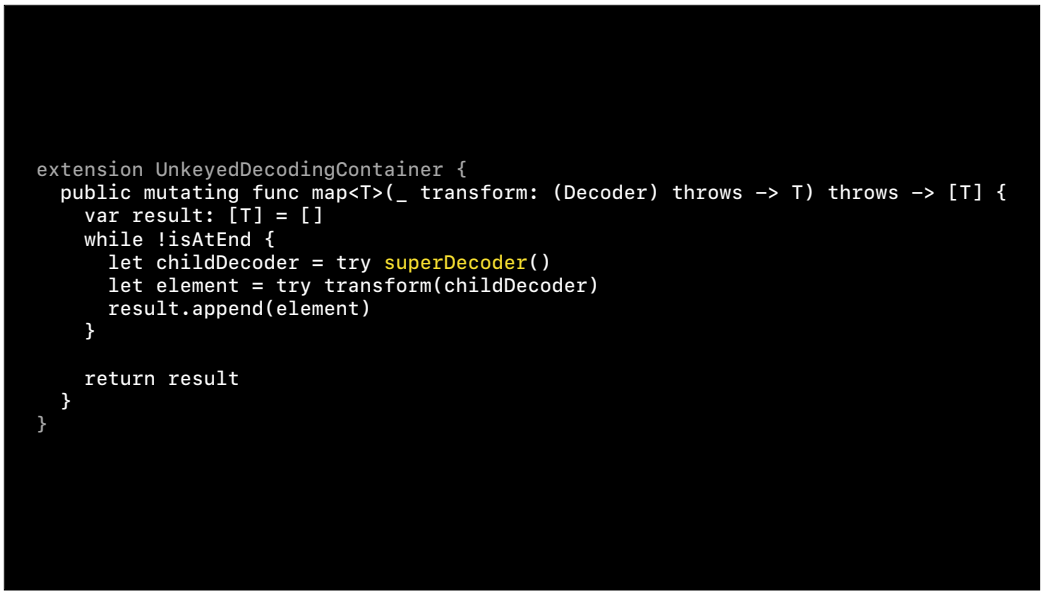
and then decode any keys that aren't in your CodingKeys. Containers are just views into a decoder’s storage. You’re free to make different ones with different keys over the same decoder. Totally supported.

```
enum CodingKeys: CodingKey {
  case type, name
}

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

  self.type = try explicitContainer[.type]
  self.name = try explicitContainer[.name]

  let attributeContainer = try decoder.anyKeyedContainer()

  var attributes: [String: String] = [:]
  for key in attributeContainer.allKeys
          where CodingKeys(stringValue: key.stringValue) == nil {
    let value = try attributeContainer.decode(String.self, forKey: key)
    attributes[key.stringValue] = value
  }
  self.attributes = attributes
}
```

Notice that you can always get string values out of any kind of CodingKey and you can always create a CodingKey from a supported string; that's built into CodingKeys. So there's no need to make your CodingKey enums be RawRepresentable as Strings. And I kind of recommend you don't. It's just extra overhead from automatic conformances.

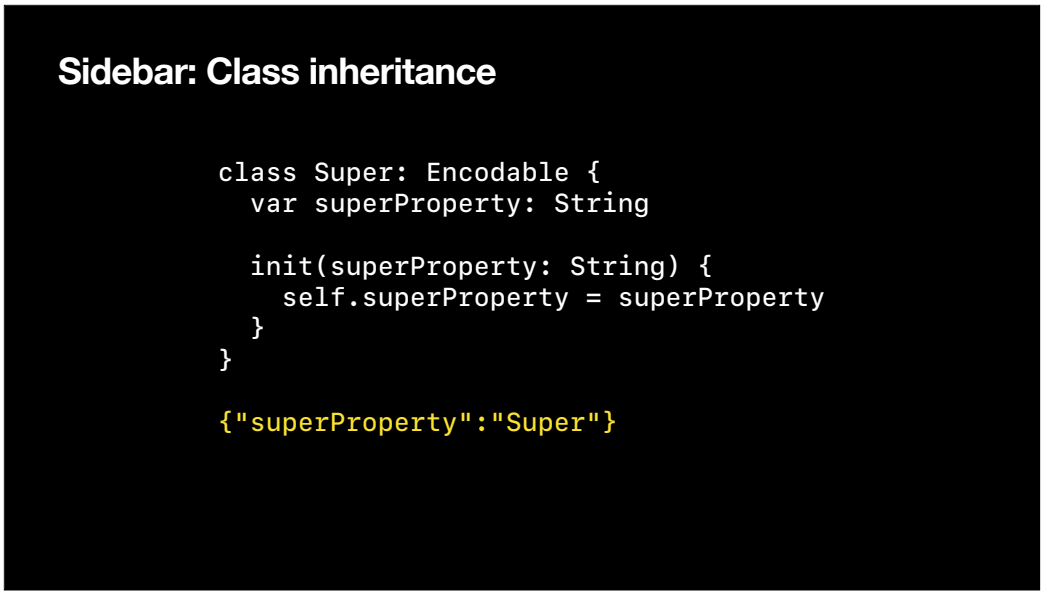
```
extension UnkeyedDecodingContainer {
  public mutating func map<T>(_ transform: (Decoder) throws -> T) throws -> [T] {
    var result: [T] = []
    while !isAtEnd {
      let childDecoder = try superDecoder()
      let element = try transform(childDecoder)
      result.append(element)
    }

    return result
  }
}
```

For the next example, it’s going to be really helpful to have a `map` extension on unkeyed containers, which handle arrays. This method accepts a transform function that takes a decoder and returns an element of some type. Unkeyed containers can support mixed types, but this map method is for the case where all the elements can be converted to the same type.



```
extension UnkeyedDecodingContainer {
  public mutating func map<T>(_ transform: (Decoder) throws -> T) throws -> [T] {
    var result: [T] = []
    while !isAtEnd {
      let childDecoder = try superDecoder()
      let element = try transform(childDecoder)
      result.append(element)
    }

    return result
  }
}
```

Decoders handle a specific level, a specific coding path. When descending, we need a new coding path. For containers, those are called “nested containers.” But for decoders, they’re called “super decoders,” for...reasons



Say you have some class, we'll call Super. It'll auto-generate an Encodable conformance that outputs JSON like this. No problem.

```
class Sub: Super {
  var subProperty: String

  init(superProperty: String, subProperty: String) {
    self.subProperty = subProperty
    super.init(superProperty: superProperty)
  }
}

{"superProperty":"Super"}  // ?!?!?!
```

Now consider the subclass. It already has an `encode(to:)` method from its superclass, so there won't be an auto-generated Encodable. It will just encode the superclass's properties.

```
override func encode(to encoder: Encoder) throws {
  var container = encoder.container(keyedBy: CodingKeys.self)

  try container.encode(self.subProperty, forKey: .subProperty)

    ... how to encode super? ...
}
```

So how should you encode the super class?

```
override func encode(to encoder: Encoder) throws {
  var container = encoder.container(keyedBy: CodingKeys.self)

  try container.encode(self.subProperty, forKey: .subProperty)

  try super.encode(to: encoder)
}
```

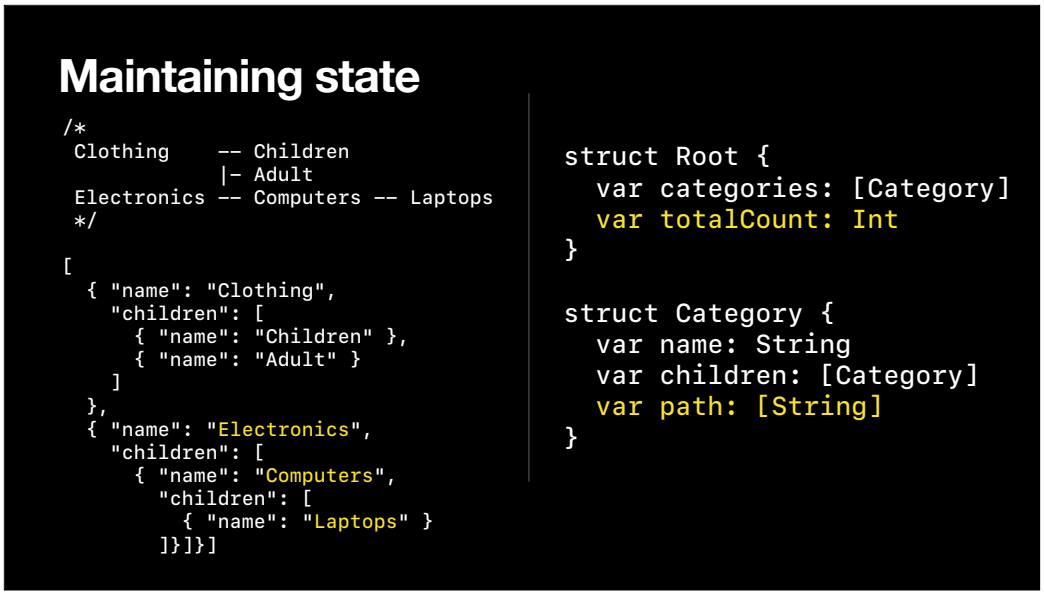
```
{"superProperty":"Super", "subProperty":"Sub"}
```

The obvious answer is to just call `super encode`, which is going to do what you're probably expect.

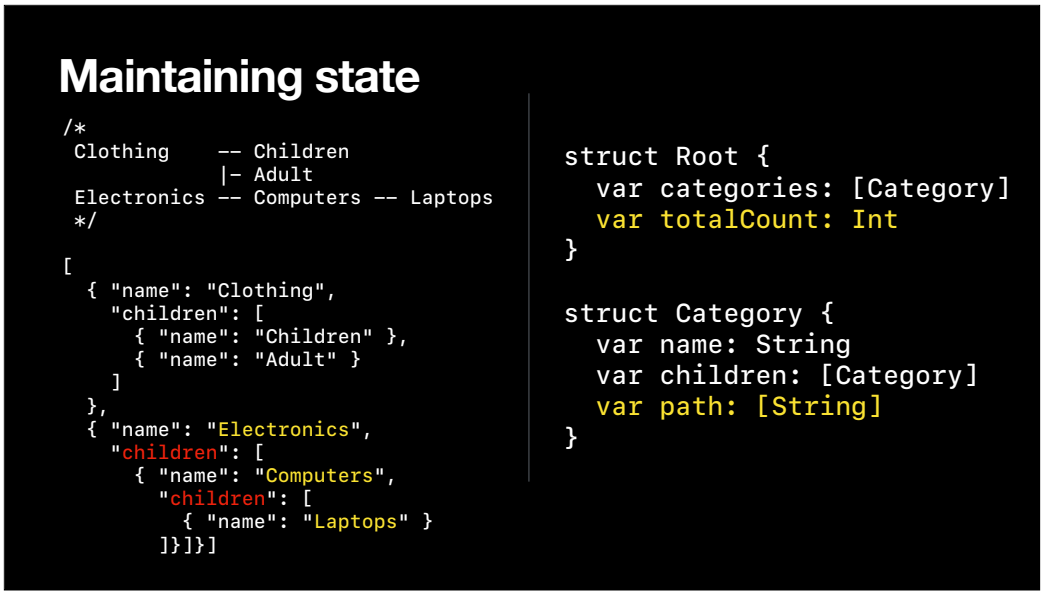
```
override func encode(to encoder: Encoder) throws {
  var container = encoder.container(keyedBy: CodingKeys.self)

  try container.encode(self.subProperty, forKey: .subProperty)

  try super.encode(to: encoder.superEncoder())
}
```

```
{"subProperty":"Sub","super":{"superProperty":"Super"}}
```

But another way is to to encode the super class in its own nested key using a “super” encoder. This is safer, since the superclass may encode in an incompatible way. It may have conflicting keys, or it may use a different kind of container. `superEncoder()` is just a shortcut for `superEncoder(forKey: "super")`. It really just means “nested encoder.”



What if you want to maintain some state while decoding? For example, may be useful for each category in this tree to know its full path. So Laptops would have a path of Electronics, Computers. How can we do that during the decoding process, without having to walk the tree a second time to update the paths?
And just to make things a little more interesting, we also want to keep a count of how many nodes in the tree. So some of the state information is stored in each node, but we also want to pass some state all the way back up to the root node.



Coding path doesn't help, because the path for Laptops would be "Item 1" (which is the key for array indexes), "children," "children," because those are the keys that identify this location. So that's not very helpful.

```
extension Root: Decodable {
  init(from decoder: Decoder) throws {
    var totalCount = 0

    var container = try decoder.unkeyedContainer()
    self.categories = try container.map {
      try Category(from: $0,
                   path: [],
                   totalCount: &totalCount)
    }
    self.totalCount = totalCount
  }
}
```

Root needs to capture the total count. It also needs to pass a starting path to the Category decoder.

```
extension Root: Decodable {
  init(from decoder: Decoder) throws {
    var totalCount = 0

    var container = try decoder.unkeyedContainer()
    self.categories = try container.map {
      try Category(from: $0,
                   path: [],
                   totalCount: &totalCount)
    }
    self.totalCount = totalCount
  }
}
```

For each element in the container, decode a category. This is not the normal Decodable initializer. In fact, Category doesn't even need to be Decodable for this. This is just a custom init at accepts a Decoder, and other parameters. totalCount is an inout variable so that the initializer can update it.

```
extension Category {
  enum CodingKeys: CodingKey { case name, children }

  init(from decoder: Decoder, path: [String], totalCount: inout Int) throws {
    self.path = path

    let container = try decoder.container(keyedBy: CodingKeys.self)
    self.name = try container[.name]

    var children: [Self] = []
    if container.contains(.children) {
      var childPath = path
      childPath.append(self.name)

      var childrenContainer = try container.nestedUnkeyedContainer(forKey: .children)
      children = try childrenContainer.map {
        try Category(from: $0, path: childPath, totalCount: &totalCount)
      }
    }
    self.children = children

    totalCount += 1
  }
}
```

And here’s the initializer for Category.

```
extension Category {
  enum CodingKeys: CodingKey { case name, children }

  init(from decoder: Decoder, path: [String], totalCount: inout Int) throws {
    self.path = path

    let container = try decoder.container(keyedBy: CodingKeys.self)
    self.name = try container[.name]

    var children: [Self] = []
    if container.contains(.children) {
      var childPath = path
      childPath.append(self.name)

      var childrenContainer = try container.nestedUnkeyedContainer(forKey: .children)
      children = try childrenContainer.map {
        try Category(from: $0, path: childPath, totalCount: &totalCount)
      }
    }
    self.children = children

    totalCount += 1
  }
}
```

It saves off the current path so far.

```
extension Category {
  enum CodingKeys: CodingKey { case name, children }

  init(from decoder: Decoder, path: [String], totalCount: inout Int) throws {
    self.path = path

    let container = try decoder.container(keyedBy: CodingKeys.self)
    self.name = try container[.name]

    var children: [Self] = []
    if container.contains(.children) {
      var childPath = path
      childPath.append(self.name)

      var childrenContainer = try container.nestedUnkeyedContainer(forKey: .children)
      children = try childrenContainer.map {
        try Category(from: $0, path: childPath, totalCount: &totalCount)
      }
    }
    self.children = children

    totalCount += 1
  }
}
```

Decodes the “normal” parameters. This is my keyed container subscript working with a regular CodingKey.

```
extension Category {
  enum CodingKeys: CodingKey { case name, children }

  init(from decoder: Decoder, path: [String], totalCount: inout Int) throws {
    self.path = path

    let container = try decoder.container(keyedBy: CodingKeys.self)
    self.name = try container[.name]

    var children: [Self] = []
    if container.contains(.children) {
      var childPath = path
      childPath.append(self.name)

      var childrenContainer = try container.nestedUnkeyedContainer(forKey: .children)
      children = try childrenContainer.map {
        try Category(from: $0, path: childPath, totalCount: &totalCount)
      }
    }
    self.children = children

    totalCount += 1
  }
}
```

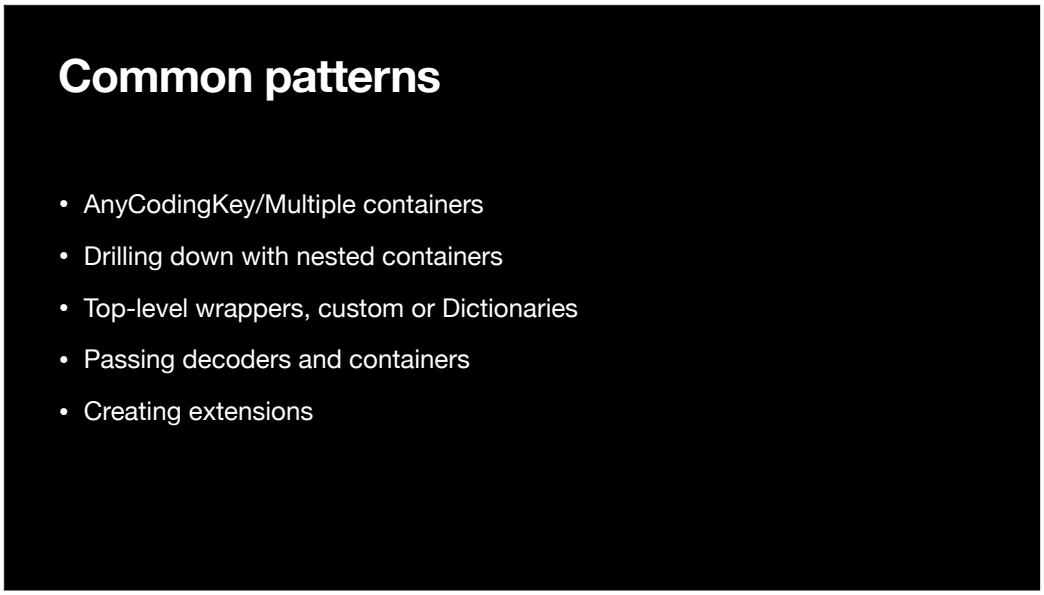
Then, if there are children, extend the path with this node's name.

```
extension Category {
  enum CodingKeys: CodingKey { case name, children }

  init(from decoder: Decoder, path: [String], totalCount: inout Int) throws {
    self.path = path

    let container = try decoder.container(keyedBy: CodingKeys.self)
    self.name = try container[.name]

    var children: [Self] = []
    if container.contains(.children) {
      var childPath = path
      childPath.append(self.name)

      var childrenContainer = try container.nestedUnkeyedContainer(forKey: .children)
      children = try childrenContainer.map {
        try Category(from: $0, path: childPath, totalCount: &totalCount)
      }
    }
    self.children = children

    totalCount += 1
  }
}
```

Create a nested unkeyed container, and recurse into this initializer for each elements to create all the children.

```
extension Category {
  enum CodingKeys: CodingKey { case name, children }

  init(from decoder: Decoder, path: [String], totalCount: inout Int) throws {
    self.path = path

    let container = try decoder.container(keyedBy: CodingKeys.self)
    self.name = try container[.name]

    var children: [Self] = []
    if container.contains(.children) {
      var childPath = path
      childPath.append(self.name)

      var childrenContainer = try container.nestedUnkeyedContainer(forKey: .children)
      children = try childrenContainer.map {
        try Category(from: $0, path: childPath, totalCount: &totalCount)
      }
    }
    self.children = children

    totalCount += 1
  }
}
```

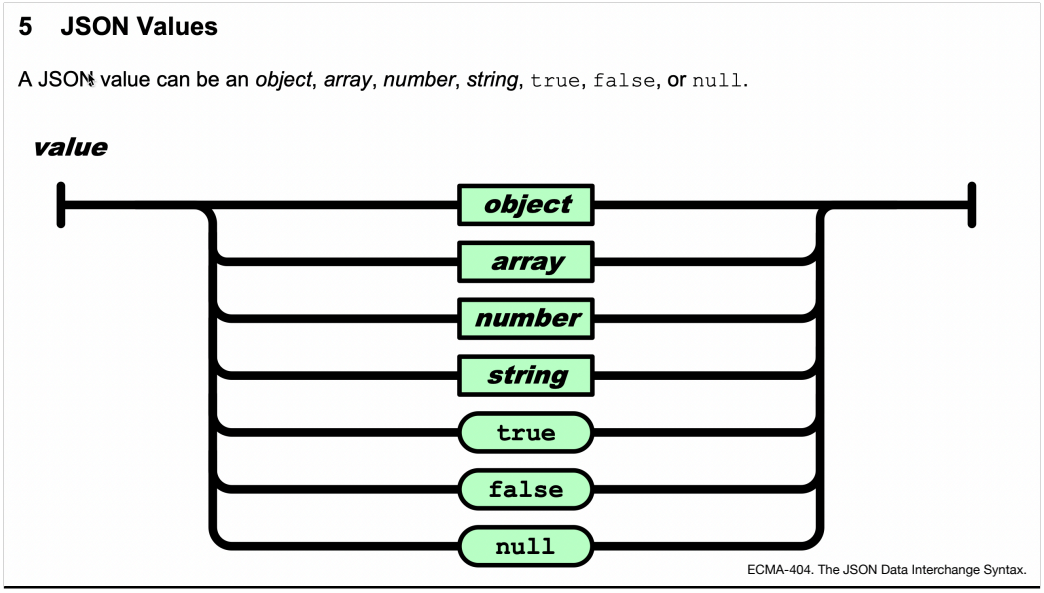
And at the end, increment totalCount. The important thing here is that Decoders can be passed to arbitrary methods. They're not valid after the top-level Decodable initializer completes. You can't store them in properties or the like. But you're free to pass them around and use them without everything having to conform to Decodable. And that means you can pass state down the tree, and even use inout parameters to pass state up the tree.



By now, I hope you're seeing some of the patterns in the solutions to problems I see come up all the time. (build)AnyCodingKey helps for lots of problems.
(build)You can use nested containers to drill down without creating intermediate types.
(build)You'll often need top-level "Response" wrappers to handle nested Arrays, but you shouldn't forget about good old String-to-Something Dictionaries. They solve a lot of problems without needing a new type.
(build)Coders and containers are just normal objects and you can pass them around and create extensions on them as long as they don't outlive the top-level init.
(build)And extensions are really good. You can make custom coders very streamlined without losing any flexibility.

And those are great tools for turning JSON into Model objects. But what if you don’t know what the JSON will be? What if you need to deal with it as arbitrary JSON. Or if writing all the Codable conformances is more trouble than it’s worth?

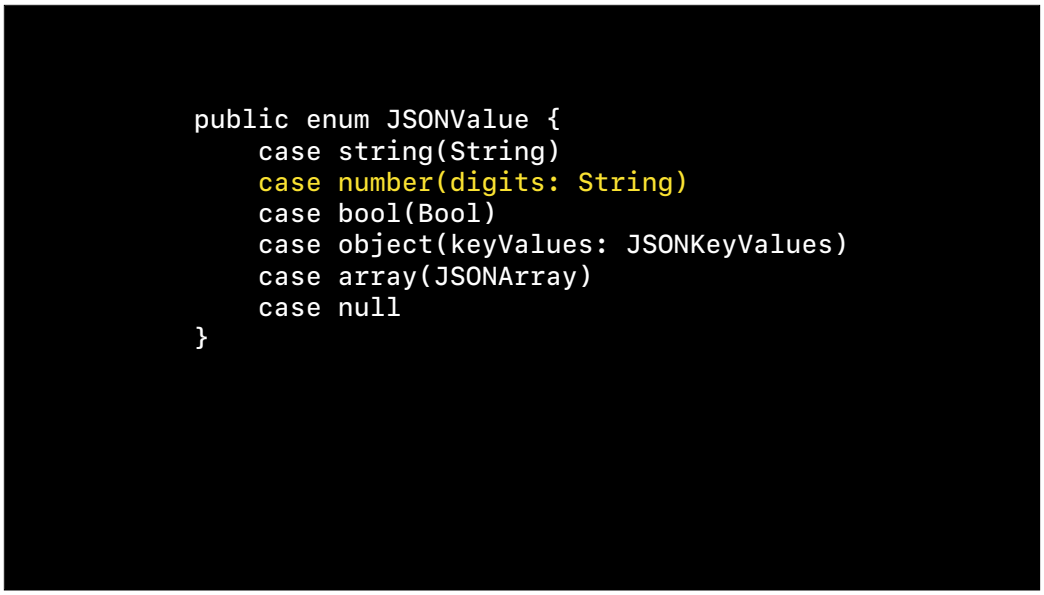
You may have done this with JSONSerialization. Please stop. JSONSerialization is horrible. It spits back Any. JSON values are not “Any.” They can’t be UIViewControllers. They can’t be CBPeripherals. They can be one of a pretty short list of things.



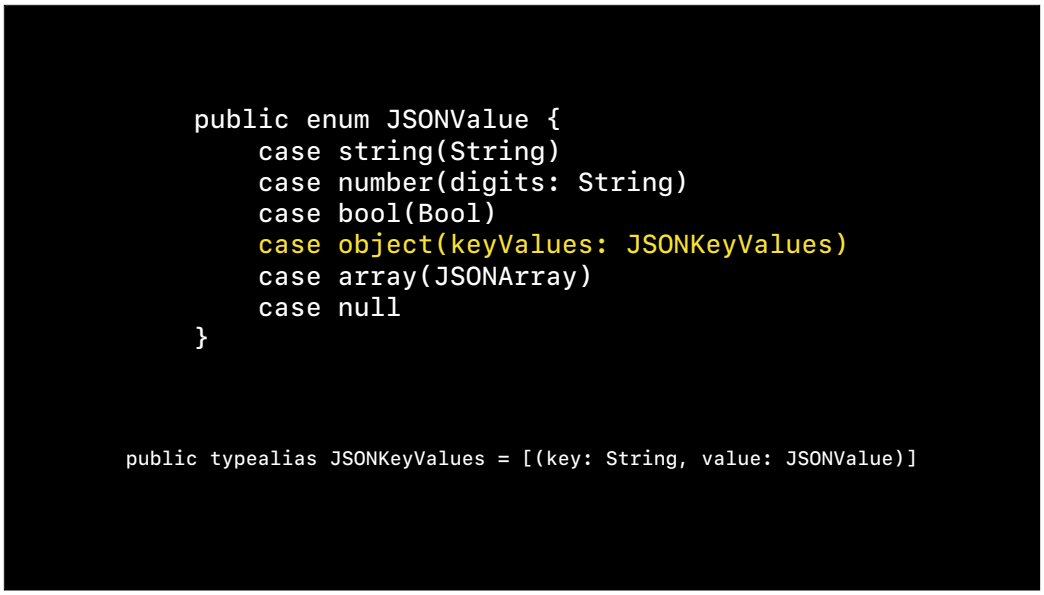
Object, array, number, string, true, false, null. That's it. And that means that a raw JSON decoder should return an enum.

```
public enum JSONValue {
    case string(String)
    case number(digits: String)
    case bool(Bool)
    case object(keyValues: JSONKeyValues)
    case array(JSONArray)
    case null
}
```

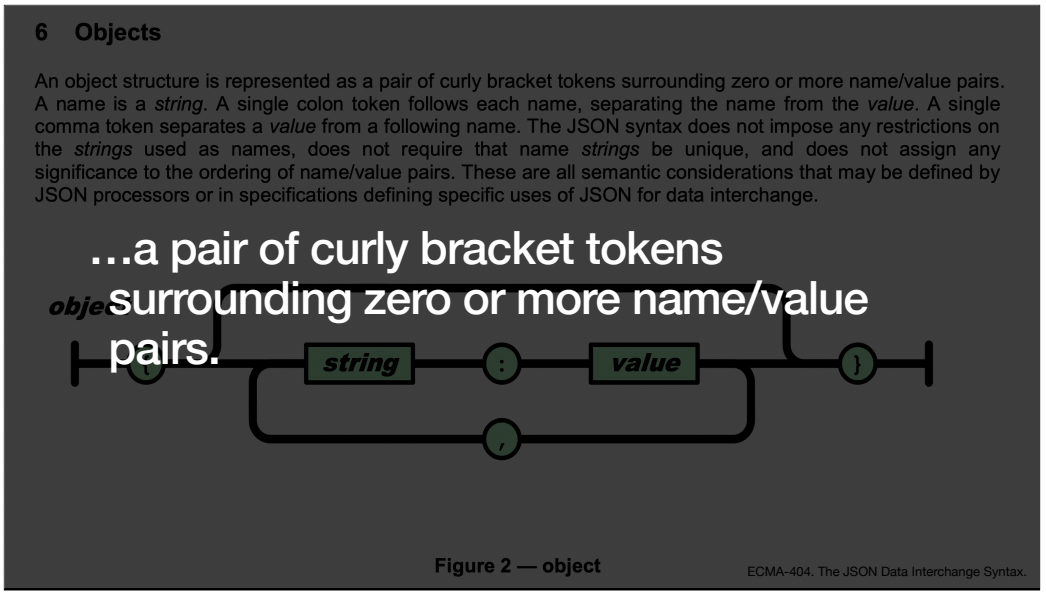
Something like this. But before I talk about how to use this type, I want to point out a couple of oddities in its definition.



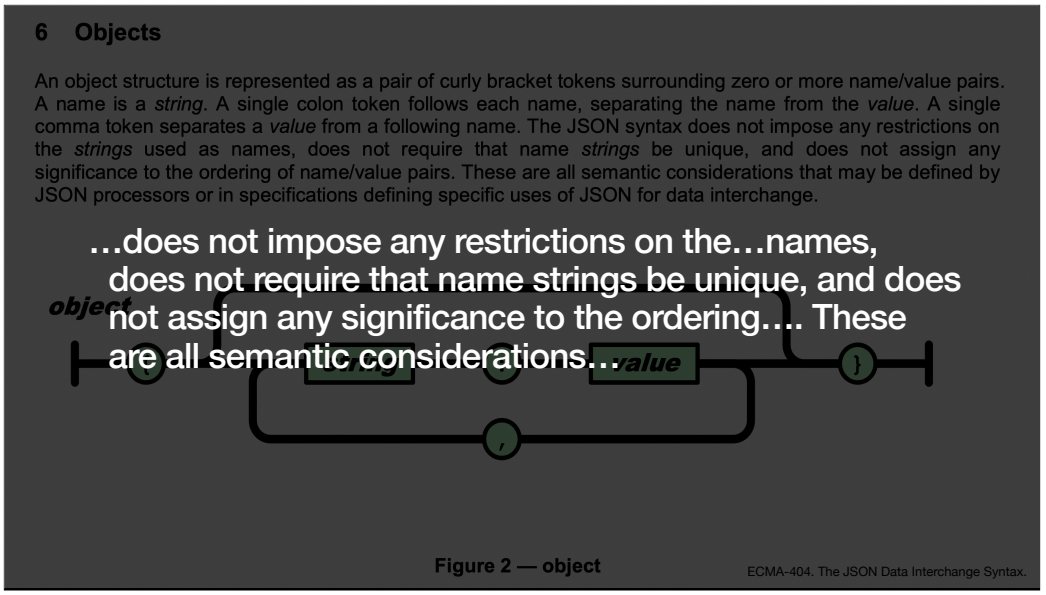
Why store numbers as strings? Because JSON defines numbers as a sequence of decimal digits, plus possibly a sign, decimal point, and base ten exponent. But the key point is that JSON encodes decimal numbers of arbitrary length and precision. Nothing in JSON limits numbers to the range or values of Double. And JSON numbers are absolutely base 10, not base 2. There's no numeric type in stdlib that can fully express a JSON number. Decimal is pretty close, but Double is completely wrong. 1/10 is a repeating fraction in binary, so Double notoriously has rounding problems when dealing with currencies.
Now, stdlib handles this well. As long as you encode and decode numbers as Decimal, it'll avoid rounding errors. But there's no way to encode or decode things outside the range of Decimal. If you have a BigInt type, there's no way to encode it as a JSON number using stdlib. But it's still legal JSON.



How about this one? So what's JSONKeyValues?(build)
Huh, weird. Why not a Dictionary? Dictionaries are unordered collections. JSON objects are…



"ECMA-404. The JSON Data Interchange Syntax. Section 6, Objects. An object structure is represented as a pair of curly bracket tokens surrounding zero or more name/value pairs."
Sure, but isn't that just another way to say "dictionary?"

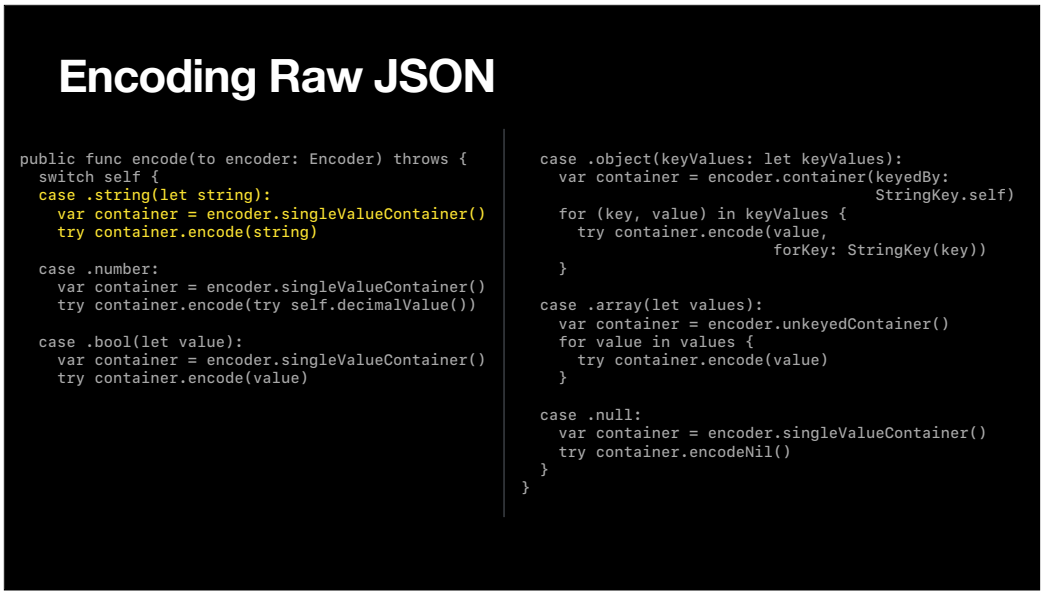


"The JSON syntax does not impose any restrictions on the strings used as names, does not require that name strings be unique, and does not assign any significance to the ordering of name/value pairs. These are all semantic considerations that may be defined by JSON processors…" Don't misread that "does not assign any significance to the order." That doesn't mean they're unordered. It means that whether they're ordered or unordered is up to the parser. JSON totally supports ordered and duplicate keys. But unfortunately, stdlib doesn't. JSONEncoder and -Decoder internally use dictionaries, which squash duplicates and scramble the order. The only way to fix that is to write your own JSON parser. Which is not as hard as it sounds.

Encoding Raw JSON

```
public func encode(to encoder: Encoder) throws {
  switch self {
  case .string(let string):
    var container = encoder.singleValueContainer()
    try container.encode(string)

  case .number:
    var container = encoder.singleValueContainer()
    try container.encode(try self.decimalValue())

  case .bool(let value):
    var container = encoder.singleValueContainer()
    try container.encode(value)

  case .object(keyValues: let keyValues):
    var container = encoder.container(keyedBy:
                                      StringKey.self)
    for (key, value) in keyValues {
      try container.encode(value,
                           forKey: StringKey(key))
    }

  case .array(let values):
    var container = encoder.unkeyedContainer()
    for value in values {
      try container.encode(value)
    }

  case .null:
    var container = encoder.singleValueContainer()
    try container.encodeNil()
  }
}
```

Anyway, once you have this JSONValue type, encoding is pretty simple. For each type, create container, encode the thing.



For simple types, the container is a single value container, which you may not have run into before. It’s just what it sounds like. A container that holds one thing. They’re not that common, but they’re useful for this kind of problem.

Encoding Raw JSON

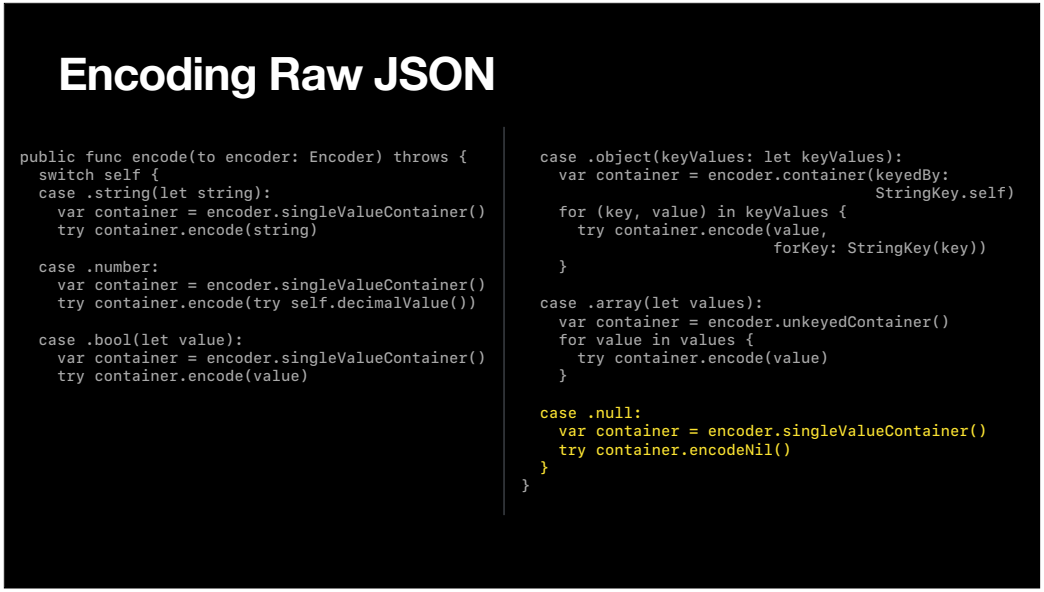
```
public func encode(to encoder: Encoder) throws {
  switch self {
  case .string(let string):
    var container = encoder.singleValueContainer()
    try container.encode(string)

  case .number:
    var container = encoder.singleValueContainer()
    try container.encode(try self.decimalValue())

  case .bool(let value):
    var container = encoder.singleValueContainer()
    try container.encode(value)

  case .object(keyValues: let keyValues):
    var container = encoder.container(keyedBy:
                                      StringKey.self)
    for (key, value) in keyValues {
      try container.encode(value,
                           forKey: StringKey(key))
    }

  case .array(let values):
    var container = encoder.unkeyedContainer()
    for value in values {
      try container.encode(value)
    }

  case .null:
    var container = encoder.singleValueContainer()
    try container.encodeNil()
  }
}
```

For objects, encode each key value.

Encoding Raw JSON

```
public func encode(to encoder: Encoder) throws {
  switch self {
  case .string(let string):
    var container = encoder.singleValueContainer()
    try container.encode(string)

  case .number:
    var container = encoder.singleValueContainer()
    try container.encode(try self.decimalValue())

  case .bool(let value):
    var container = encoder.singleValueContainer()
    try container.encode(value)

  case .object(keyValues: let keyValues):
    var container = encoder.container(keyedBy:
                                      StringKey.self)
    for (key, value) in keyValues {
      try container.encode(value,
                           forKey: StringKey(key))
    }

  case .array(let values):
    var container = encoder.unkeyedContainer()
    for value in values {
      try container.encode(value)
    }

  case .null:
    var container = encoder.singleValueContainer()
    try container.encodeNil()
  }
}
```

For arrays, encode each value.



And for null, encode nil into a single value container. That’s it.

```
extension JSONValue: ExpressibleByStringLiteral { ... }
extension JSONValue: ExpressibleByUnicodeScalarLiteral { ... }
extension JSONValue: ExpressibleByExtendedGraphemeClusterLiteral { ... }
extension JSONValue: ExpressibleByIntegerLiteral { ... }
extension JSONValue: ExpressibleByFloatLiteral { ... }
extension JSONValue: ExpressibleByNilLiteral { ... }
extension JSONValue: ExpressibleByBooleanLiteral { ... }
extension JSONValue: ExpressibleByArrayLiteral { ... }
extension JSONValue: ExpressibleByDictionaryLiteral { ... }
```

Let’s go a step further, and slap on a bunch of ExpressibleBy… conformances.

```
let person: JSONValue = [
  "name": "Alice",
  "age": 43,
  "active": true,
  "addresses": [
    [
      "street": "123 Main St.",
      "city": "New York",
      "state": "NY"
    ],
    [
      "street": "987 South East Blvd.",
      "city": "Los Angeles",
      "state": "CA"
    ],
  ],
]
```

```
try JSONEncoder().encode(person)

{"age":43,
 "active":true,
 "addresses":[
   {"street":"123 Main St.",
    "city":"New York",
    "state":"NY"},
   {"street":"987 South East Blvd.",
    "city":"Los Angeles",
    "state":"CA"}
 ],
 "name":"Alice"}
```

Then you can create JSON by hand rather beautifully in code. I think this is really nice. Much nicer than a `[String: Any]`. If you want arbitrary JSON, make a type for it. Take a second and appreciate what ExpressibleBy...Literal can let you do. ExpressibleByDictionaryLiteral even maintains key order and supports duplicate keys, so you can do everything JSONValue can do in a simple, Swifty syntax. I love Swift.

```
extension JSONValue: Decodable {
  public init(from decoder: Decoder) throws {
    let matchers = [decodeNil, decodeString, decodeNumber,
                    decodeBool, decodeObject, decodeArray]

    for matcher in matchers {
      do {
        self = try matcher(decoder)
        return
      }
      catch DecodingError.typeMismatch { continue }
    }

    throw DecodingError.typeMismatch(JSONValue.self,
                                     .init(codingPath: decoder.codingPath,
                                           debugDescription: "Unknown JSON type"))
  }
}
```

Decoding is a little more complicated, but not bad, and it’s a very good example of how to decode things that might be one of a variety of types.

```
extension JSONValue: Decodable {
  public init(from decoder: Decoder) throws {
    let matchers = [decodeNil, decodeString, decodeNumber,
                    decodeBool, decodeObject, decodeArray]

    for matcher in matchers {
      do {
        self = try matcher(decoder)
        return
      }
      catch DecodingError.typeMismatch { continue }
    }

    throw DecodingError.typeMismatch(JSONValue.self,
                                     .init(codingPath: decoder.codingPath,
                                           debugDescription: "Unknown JSON type"))
  }
}
```

I have a sequence of small functions that try to decode as each kind of value.

```
extension JSONValue: Decodable {
  public init(from decoder: Decoder) throws {
    let matchers = [decodeNil, decodeString, decodeNumber,
                    decodeBool, decodeObject, decodeArray]

    for matcher in matchers {
      do {
        self = try matcher(decoder)
        return
      }
      catch DecodingError.typeMismatch { continue }
    }

    throw DecodingError.typeMismatch(JSONValue.self,
                                     .init(codingPath: decoder.codingPath,
                                           debugDescription: "Unknown JSON type"))
  }
}
```

If they throw `typeMismatch`, then try the next matcher.
Notice that there's no `try?` here. It's do/catch, and only catches `typeMismatch`. This means that other errors, like `dataCorrupted` deep inside a nested object, give you useful errors that point to where the problem is. The typical code that uses `try?` throws away all that information, and the only error you get is "something went wrong somewhere."
This is all you really need for a Codable JSONValue. It's a couple of hundred lines of code, but it opens up so many options.

```
private func decodeString(decoder: Decoder) throws -> JSONValue {
  try .string(decoder.singleValueContainer().decode(String.self))
}

private func decodeNumber(decoder: Decoder) throws -> JSONValue {
  try .number(digits:
        decoder.singleValueContainer().decode(Decimal.self).description)
}

private func decodeBool(decoder: Decoder) throws -> JSONValue {
  try .bool(decoder.singleValueContainer().decode(Bool.self))
}
...
```

The decoding functions are as simple as you'd think they are. Try to decode a type. Throw if it fails.

```
{
  "abilities": [{
    "ability": {
      "name": "limber",
      "url": "https://pokeapi.co/api/v2/ability/7/"
    },
    "is_hidden": false,
    "slot": 1
  }, {
    "ability": {
      "name": "imposter",
      "url": "https://pokeapi.co/api/v2/ability/150/"
    },
    "is_hidden": true,
    "slot": 3
  }],
  ...
```


For example, I love the the PokéDex API. It's so huge and complicated. Makes a great test case.

```
{
  "abilities": [{
    "ability": {
      "name": "limber",
      "url": "https://pokeapi.co/api/v2/ability/7/"
    },
    "is_hidden": false,
    "slot": 1
  }, {
    "ability": {
      "name": "imposter",
      "url": "https://pokeapi.co/api/v2/ability/150/"
    },
    "is_hidden": true,
    "slot": 3
  }],
  ...
```

```
let ditto = try JSONDecoder().decode(JSONValue.self,
                                     from: json)
try ditto
    .value(for: "abilities")
    .value(at: 1)
    .value(for: "ability")
    .value(for: "name")
    .stringValue()           // "imposter"
```

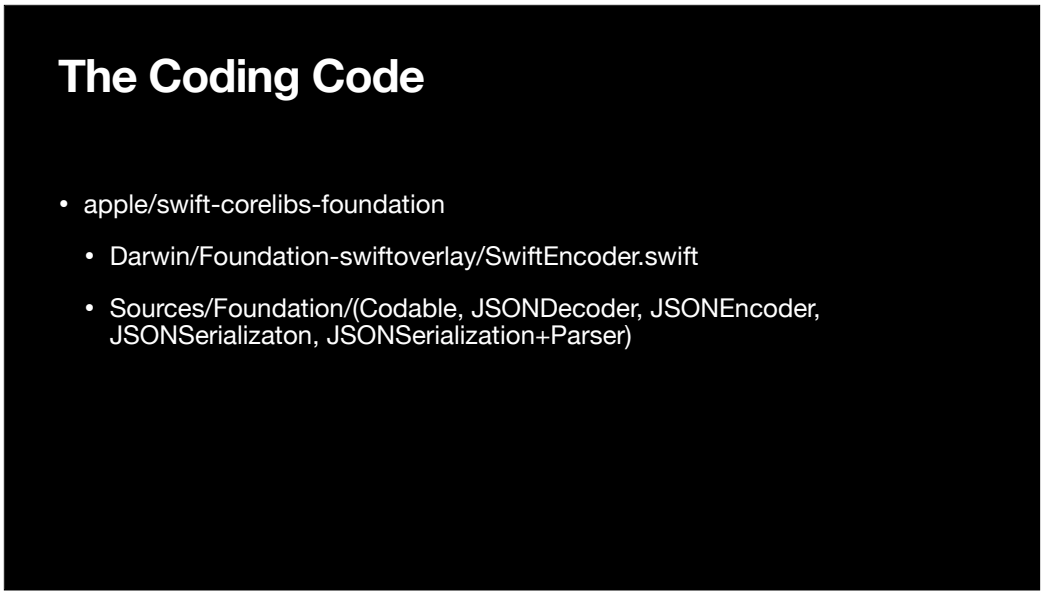
By writing about 100 lines worth of convenience functions on JSONValue, I can drill into this data structure without creating any new structs. Decode Ditto. Abilities, index 1, ability, name, turn it into a string.

```swift
try ditto["abilities"][1]["ability"]["name"].stringValue()
// "imposter"
```

```swift
let dd = ditto.dynamic

dd.abilities[1].ability.name  // "imposter"
dd.game_indices[0].version.name // "red"
dd.game_indices[0].version.wrongName // .null
```

Or add a few subscripts to make it even nicer.(build)
Or add `@dynamicMemberLookup` if you like. So that's nice. I'll give you a link at the end where you can download all the code if you're interested. It's pretty straightforward stuff once you have a type that holds a JSONValue.

Finishing up, I want to just give some inspiration for things you could do if you're willing to hack on the JSONCoder itself. It's not as straightforward as JSONValue, but it's not that bad. Again, I'll post links to the code at the end. This is just inspiration.



Your best friend for understanding JSON coding is swift-corelibs-foundation. In includes almost all the code you need to understand the system. There are two versions of the Codable system:
* The version used by all Apple platforms, known as Darwin, which is in Darwin/Foundation-swiftoverlay/SwiftEncoder.swift, and
* The version used by all non-Apple platforms, in Sources/Foundation.
In my opinion, the non-Darwin version is much easier to understand, and is definitely nicer to extend. I write a lot of specialized JSON parsers, and I almost always use the non-Darwin code as my starting point. It's closely based on Fabian Fett's swift-extras-json, which is not surprising since Fabian joined Apple a few years ago.

JSONSerialization+Parser.swift

```
internal struct JSONParser {
    mutating func parse() throws -> JSONValue {
        try reader.consumeWhitespace()
        let value = try self.parseValue()
        #if DEBUG
        defer {
            guard self.depth == 0 else {
                preconditionFailure("Expected to end parsing with a depth of 0")
            }
        }
        #endif

        // ensure only white space is remaining
        var whitespace = 0
        while let next = reader.peek(offset: whitespace) {
            switch next {
            case ._space, ._tab, ._return, ._newline:
                whitespace += 1
                continue
            default:
                throw JSONError.unexpectedCharacter(...)
            }
        }

        return value
    }
```

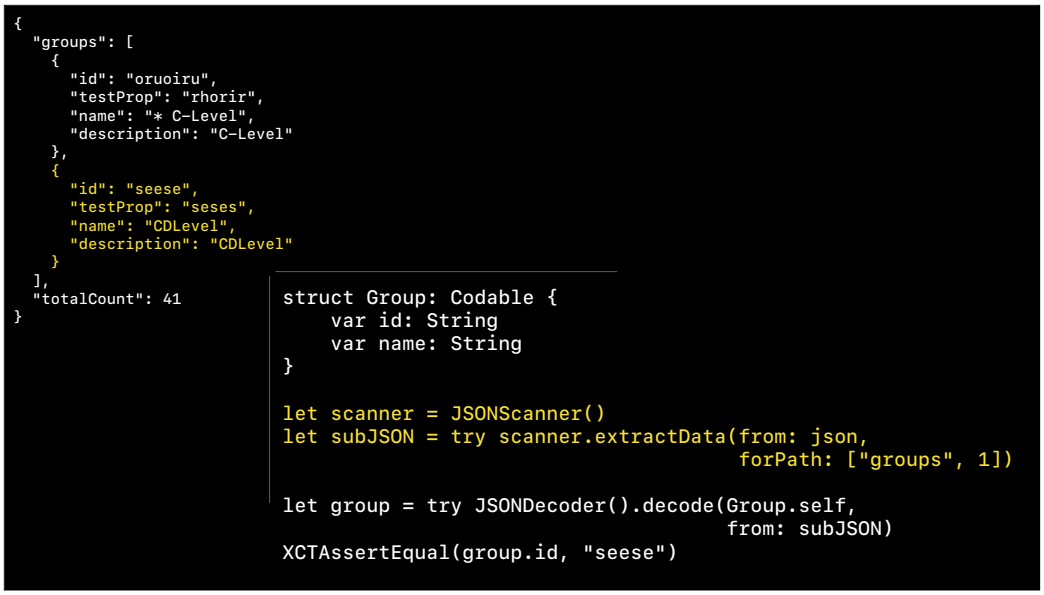
This is the parse function from the stdlib JSONParser. This is what actually takes data and turns it into a JSONValue. This JSONValue is a little bit different than mine, but very similar. The whole method is mostly a wrapper around this one call to parseValue, which parses the top-level value and returns it. The rest the method is just whitespace handling and error checking.

```
mutating func parseValue() throws -> JSONValue {
    var whitespace = 0
    while let byte = reader.peek(offset: whitespace) {
        switch byte {
        case UInt8(ascii: "\""):
            reader.moveReaderIndex(forwardBy: whitespace)
            return .string(try reader.readString())
        case ._openbrace:
            reader.moveReaderIndex(forwardBy: whitespace)
            let object = try parseObject()
            return .object(object)
        case ._openbracket:
            reader.moveReaderIndex(forwardBy: whitespace)
            let array = try parseArray()
            return .array(array)
        case UInt8(ascii: "f"), UInt8(ascii: "t"):
            reader.moveReaderIndex(forwardBy: whitespace)
            let bool = try reader.readBool()
            return .bool(bool)

        ...
```

The parseValue method is pretty straight-forward. It uses an internal helper called DocumentReader that moves a cursor through its data. If it encounters a double-quote, it consumes a string. If it encounters an open brace, it consumes an object. JSON is really very easy to parse. I has almost no ambiguities. You never have to backtrack or anything like that. This makes it very easy to modify JSONParser to do other things.

```
public struct JSONScanner {

    public func extractData<Source>(from data: Source,
                                    forPath path: [CodingKey]) throws
    -> Source.SubSequence
    where Source: BidirectionalCollection<UInt8>
    {
      var reader = DocumentReader(array: data)

      try reader.consumeWhitespace()

      // For each key in the coding path, advance to that location
      for key in path {
        if let index = key.intValue { try reader.consumeArray(toIndex: index) }
        else { try reader.consumeObject(key: key.stringValue) }
      }

      // Return subsequence between start and end of value
      let startIndex = reader.readerIndex
      try reader.consumeValue()
      return reader.array[startIndex..<reader.readerIndex]
  }
```

An interesting project is to copy the entire JSONParser file, remove all the parts that actually generate values, and turn it into a scanner that can find where a given coding path begins and ends. So rather than the normal parseArray that returns an Array, it has consumeArray that just advances the DocumentReader without creating or returning a value. This is just the top-level function, extractData, that replaces parse. But the rest of the changes aren't very complicated.

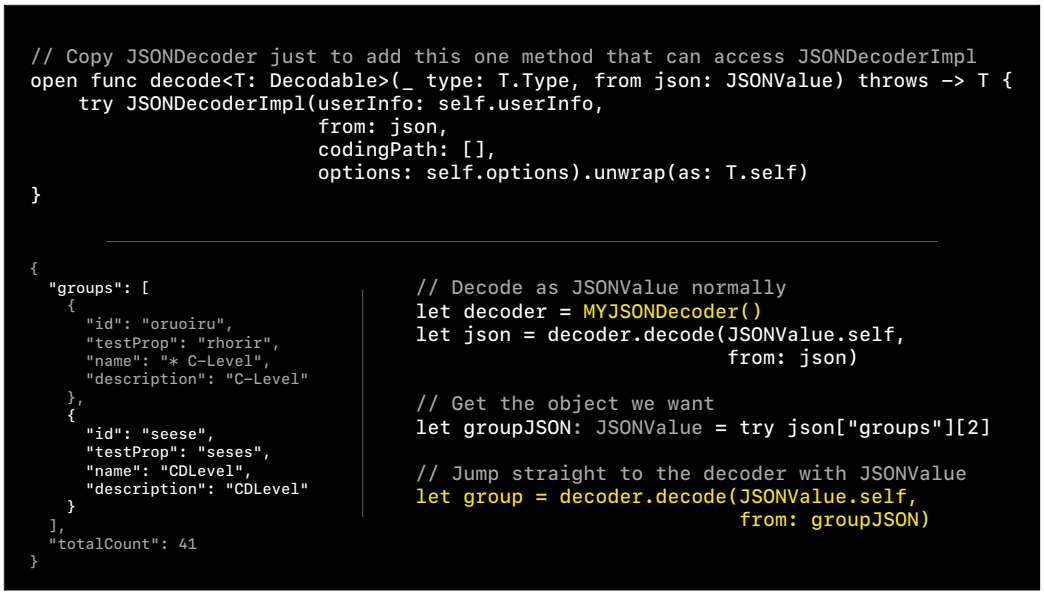


So say you have some JSON, and you don't really want to parse all of it. You could just scan the JSON for the part you want, like the second element of the groups key. It doesn't have to create any data structures, so it's fast and memory efficient. And when it finds the part you want, it can stop scanning.
That returns a new Data that can be parsed by JSONDecoder.

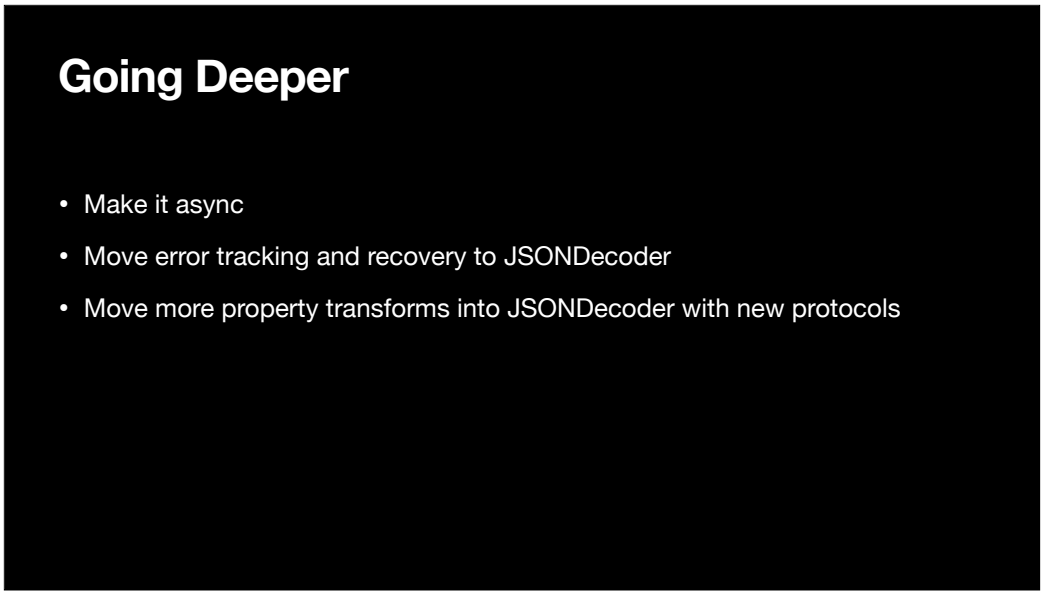
JSONDecoder.swift

```
open class JSONDecoder {
  open func decode<T: Decodable>(_ type: T.Type, from data: Data) throws -> T {
    do {
      var parser = JSONParser(bytes: Array(data))
      let json = try parser.parse()
      return try JSONDecoderImpl(userInfo: self.userInfo,
                                 from: json,
                                 codingPath: [],
                                 options: self.options).unwrap(as: T.self)
    } catch let error as JSONError {
      throw DecodingError.dataCorrupted(...)
    } catch {
      throw error
    }
  }
}
```

Moving up the stack, JSONParser is used by JSONDecoder. It's important to know that JSONDecoder is not a Decoder. It is not the thing passed to Decodable init methods. It is a wrapper that create a Decoder called JSONDecoderImpl. That's what gets passed to your init methods. Unfortunately, it's fileprivate, so if we subclass JSONDecoder, we can't create JSONDecoderImpl. Very frustrating. If you want to make changes to JSONDecoder itself, you pretty much have to cut and paste all of JSONDecoder.swift and rename the top-level type.
Note how decode works. It creates a JSONValue and then hands it to JSONDecoderImpl.



But what if we already had a JSONValue? Copy JSONDecoder to a new file and rename the top level type. Then, add new `decode` method that accepts a JSONValue. With that, use the JSONValue helpers to dig through all the data to just the part we want, and hand that to the decoder. So this is another way to do the same thing as the scanner. This isn't as fast, because JSONParser still parses the entire file, but it's very convenient.



And of course you could go further. I'm currently building an async version of this so I can start working on payloads while they're downloading, and abort the download if I don't need it all.
I'm exploring how to move error handling and fallbacks into JSONDecoder so that Decodable implementations can be simpler.
I'm exploring new protocols that would let Decodable objects do more configuration while still using default conformances.
I think there's a ton of ways that the entire Codable system could be improved, and I think a lot of them could be done without modifying stdlib or the compiler, so...watch this space.

Take-aways

- Just write the code. Don’t wrap yourself around auto-conformance.
- Make it easier with extensions and AnyCodingKey.
- Errors matter. Design for them and test your handling.
- Decoders and containers aren’t magical. They can be passed around.
- JSONValue is a powerful type for handling raw JSON.
- Hacking on (a copy of) JSONDecoder is quite doable.

So that’s it. As I said, this talk isn’t about this or that specific technique. It’s about getting you to write some code, and think about your errors. Build extensions that work for your problems. Use AnyCodingKey and JSONValue to deal with squirrelly formats. And don’t be afraid to replace JSONDecoder if you have to. It’s not magic. It’s just code. Have a great conference.