

---

## Why Freddy?

Parsing JSON elegantly and safely can be hard, but Freddy is here to help. Freddy is a reusable framework for parsing JSON in Swift. It has three principal benefits.

First, Freddy provides a type safe solution to parsing JSON in Swift. This means that the compiler helps you work with sending and receiving JSON in a way that helps to prevent runtime crashes.

Second, Freddy provides an idiomatic solution to JSON parsing that takes advantage of Swift's generics, enumerations, and functional features. This is all provided without the pain of having to memorize our documentation to understand our magical custom operators. Freddy does not have any of those. If you feel comfortable writing Swift (using extensions, protocols, initializers, etc.), then you will not only understand how Freddy is organized, but you will also feel comfortable using Freddy.

Third, Freddy provides great error information for mistakes that commonly occur while parsing JSON. If you subscript the JSON object with a key that is not present, you get an informative error. If your desired index is out of bounds, you get an informative error. If you try to convert a JSON value to the wrong type, you get a good error here too.

So, Freddy vs. JSON, who wins? We think it is Freddy.

## Usage

This section describes Freddy's basic usage. You can find more examples on parsing data, dealing with errors, serializing `JSON` instances into `NSData`, and more in the Wiki. You can read the documentation to see the full API.

### Deserialization: Parsing Raw Data

**Basic Usage** Consider some example JSON data:

```
1  {
2      "success": true,
3      "people": [
4          {
5              "name": "Matt Mathias",
6              "age": 32,
7              "spouse": true
8          },
9          {
10             "name": "Sergeant Pepper",
11             "age": 25,
12             "spouse": false
13         }
14     ]
15 }
```

```

14     ],
15     "jobs": [
16         "teacher",
17         "judge"
18     ],
19     "states": {
20         "Georgia": [
21             30301,
22             30302,
23             30303
24         ],
25         "Wisconsin": [
26             53000,
27             53001
28         ]
29     }
30 }

```

Here is a quick example on how to parse this data using Freddy:

```

1 let data = getSomeData()
2 do {
3     let json = try JSON(data: data)
4     let success = try json.getBool(at: "success")
5     // do something with `success`
6 } catch {
7     // do something with the error
8 }

```

After we load in the data, we create an instance of `JSON`, the workhorse of this framework. This allows us to access the values from the JSON data. We `try` because the `data` may be malformed and the parsing could generate an error. Next, we access the `"success"` key by calling the `getBool(at:)` method on `JSON`. We `try` here as well because accessing the `json` for the key `"success"` could fail - e.g., if we had passed an unknown key. This method takes two parameters, both of which are used to define a path into the `JSON` instance to find a Boolean value of interest. If a `Bool` is found at the path described by `"success"`, then `getBool(at:)` returns a `Bool`. If the path does not lead to a `Bool`, then an appropriate error is thrown.

**Use Paths to Access Nested Data with Subscripting** With Freddy, it is possible to use a path to access elements deeper in the json structure. For example:

```

1 let data = getSomeData()
2 do {
3     let json = try JSON(data: data)
4     let georgiaZipCodes = try json.getArray(at: "states", "Georgia")
5     let firstPersonName = try json.getString(at: "people", 0, "name")
6 } catch {

```

---

```
7 // do something with the error
8 }
```

In the code `json.getJSONArray(at: "states", "Georgia")`, the keys `"states"` and `"Georgia"` describe a path to the Georgia zip codes within `json`. Freddy's parlance calls this process "subscripting" the JSON. What is typed between the parentheses of, for example, `getArray(at:)` is a comma-separated list of keys and indices that describe the path to a value of interest.

There can be any number of subscripts, and each subscript can be either a `String` indicating a named element in the JSON, or an `Int` that represents an element in an array. If there is something invalid in the path such as an index that doesn't exist in the JSON, an error will be thrown.

More on Subscripting

**JSONDecodable: Deserializing Models Directly** Now, let's look an example that parses the data into a model class:

```
1 let data = getSomeData()
2 do {
3     let json = try JSON(data: data)
4     let people = try json.getJSONArray(at: "people").map(Person.init)
5     // do something with `people`
6 } catch {
7     // do something with the error
8 }
```

Here, we are instead loading the values from the key `"people"` as an array using the method `getArray(at:)`. This method works a lot like the `getBool(at:)` method you saw above. It uses the path provided to the method to find an array. If the path is good, the method will return an `Array` of `JSON`. If the path is bad, then an appropriate error is thrown.

We can then call `map` on that `JSON` array. Since the `Person` type conforms to `JSONDecodable`, we can pass in the `Person` type's initializer. This call applies an initializer that takes an instance of `JSON` to each element in the array, producing an array of `Person` instances.

Here is what `JSONDecodable` looks like:

```
1 public protocol JSONDecodable {
2     init(json: JSON) throws
3 }
```

It is fairly simple protocol. All it requires is that conforming types implement an initializer that takes an instance of `JSON` as its sole parameter.

To tie it all together, here is what the `Person` type looks like:

---

```

1 public struct Person {
2     public let name: String
3     public let age: Int
4     public let spouse: Bool
5 }
6
7 extension Person: JSONDecodable {
8     public init(json value: JSON) throws {
9         name = try value.getString(at: "name")
10        age = try value.getInt(at: "age")
11        spouse = try value.getBool(at: "spouse")
12    }
13 }

```

`Person` just has a few properties. It conforms to `JSONDecodable` via an extension. In the extension, we implement a **throwing** initializer that takes an instance of `JSON` as its sole parameter. In the implementation, we **try** three functions: 1) `getString(at:)`, 2) `getInt(at:)`, and 3) `getBool(at:)`. Each of these works as you have seen before. The methods take in a path, which is used to find a value of a specific type within the `JSON` instance passed to the initializer. Since these paths could be bad, or the requested type may not match what is actually inside of the `JSON`, these methods may potentially throw an error.

## Serialization

Freddy's serialization support centers around the `JSON.serialize()` method.

**Basic Usage** The `JSON` enumeration supports conversion to `Data` directly:

```

1 let someJSON: JSON = ...
2 do {
3     let data: Data = try someJSON.serialize()
4 } catch {
5     // Handle error
6 }

```

**JSONEncodable: Serializing Other Objects** Most of your objects aren't `Freddy.JSON` objects, though. You can serialize them to `Data` by first converting them to a `Freddy.JSON` via `JSONEncodable.toJSON()`, the sole method of the `JSONEncodable` protocol, and then use `serialize()` to convert the `Freddy.JSON` to `Data`:

```

1 let myObject: JSONEncodable = ...
2
3 // Object -> JSON -> Data:

```

---

```
4 let objectAsJSON: JSON = myObject.toJSON()
5 let data: Data = try objectAsJSON.serialize()
6
7 // More concisely:
8 let dataOneLiner = try object.toJSON().serialize()
```

Freddy provides definitions for common Swift datatypes already. To make your own datatypes serializable, conform them to `JSONEncodable` and implement that protocol's `toJSON()` method:

```
1 extension Person: JSONEncodable {
2     public func toJSON() -> JSON {
3         return .dictionary([
4             "name": .string(name),
5             "age": .int(age),
6             "spouse": .bool(spouse)])
7     }
8 }
```

## Getting Started

Freddy requires iOS 8.0, Mac OS X 10.9, watchOS 2.0, or tvOS 9.0. Linux is not yet supported.

You have a few different options to install Freddy.

### Carthage

Add us to your `Cartfile`:

```
1 github "bignerdranch/Freddy" ~> 3.0
```

After running `carthage bootstrap`, add `Freddy.framework` to the “Linked Frameworks and Libraries” panel of your application target. Read more.

### CocoaPods

Add us to your `Podfile`:

```
1 source 'https://github.com/CocoaPods/Specs.git'
2 platform :ios, '8.0'
3 use_frameworks!
4
5 pod 'Freddy'
```

Then run `pod install`.

---

## Submodules

1. `git submodule add https://github.com/bignerdranch/Freddy.git Vendor/Freddy`
2. Drag `Freddy.xcodeproj` into your Xcode project.
3. Add `Freddy.framework` to the “Linked Frameworks and Libraries” panel of your application target.

Carthage can be used to check out dependencies and maintain Git submodule state as well.

## Swift Package Manager

Add us to your `Package.swift`:

```
1 import PackageDescription
2
3 let package = Package(
4     name: "My Nerdy App",
5     dependencies: [
6         .Package(url: "https://github.com/bignerdranch/Freddy.git",
7                   majorVersion: 3),
8     ]
9 )
```

## iOS 7

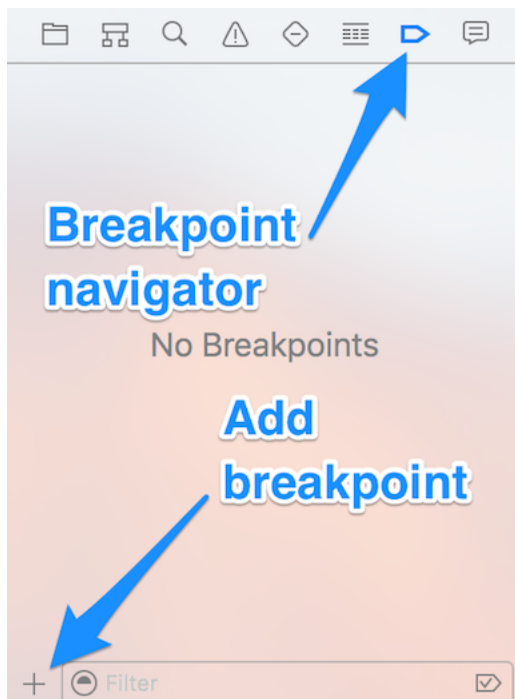
If you would like to use Freddy with iOS 7, then you will need to use a previous release of Freddy.

## Setting Breakpoint Errors

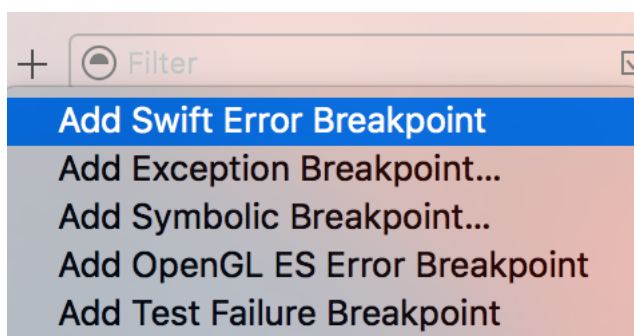
It can be helpful to set breakpoints for errors when you start working with a new set of JSON. This allows you to explore the structure of the JSON when you break. In particular, you will likely want to set a breakpoint for `Freddy's JSON.Error` so that you can inspect what went wrong.

Here is how you can set this sort of breakpoint:

- 1) Go to the Breakpoint navigator
- 2) Click the “+” button in the bottom left corner



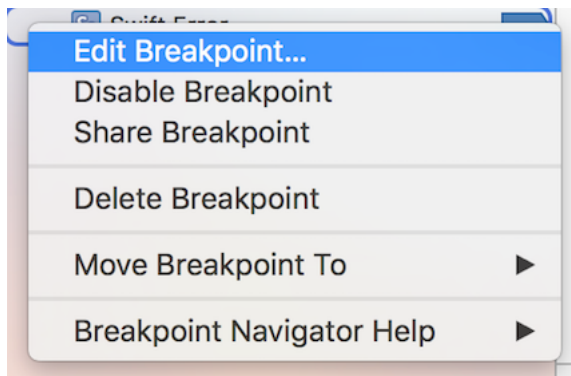
3) Select “Add Swift Error Breakpoint”



Now you have a breakpoint that will only trigger when a Swift error is generated. But your program will break whenever *any* Swift error is thrown. What if you only want to break for `Freddy's JSON.Error` error?

You can edit the breakpoint to add a filter:

- 1) Right-click your new error breakpoint
- 2) Select Edit Breakpoint...



- 3) A window will appear with a text box for "Type"
- 4) Enter `JSON.Error`

A screenshot of the "Swift Error" breakpoint configuration window. It has a title bar with a checkmark icon and the text "Swift Error". Below the title bar, there are four sections: "Type" with a text box containing "JSON.Error", "Ignore" with a numeric input field set to "0" and a label "times before stopping", "Action" with a button labeled "Add Action", and "Options" with a checkbox labeled "Automatically continue after evaluating actions".

And that is pretty much it! You now have an error breakpoint that will only trigger when errors of type `JSON.Error` are thrown. Take a look at the framework's tests for further examples of usage. The Wiki also have a lot of very useful information.