

# A Promise for a Better Future

*A tour of asynchronous constructs in Swift*

# Ben Scheirman

@subdigital



NS *Screen*cast

# Synchronous Code

```
func fetchUser(id: Int) -> User
```

```
let user = fetchUser(id: 5)
```

# Asynchronous Code

```
func fetchUser(id: Int, completion: @escaping (User?, Error?) -> Void)

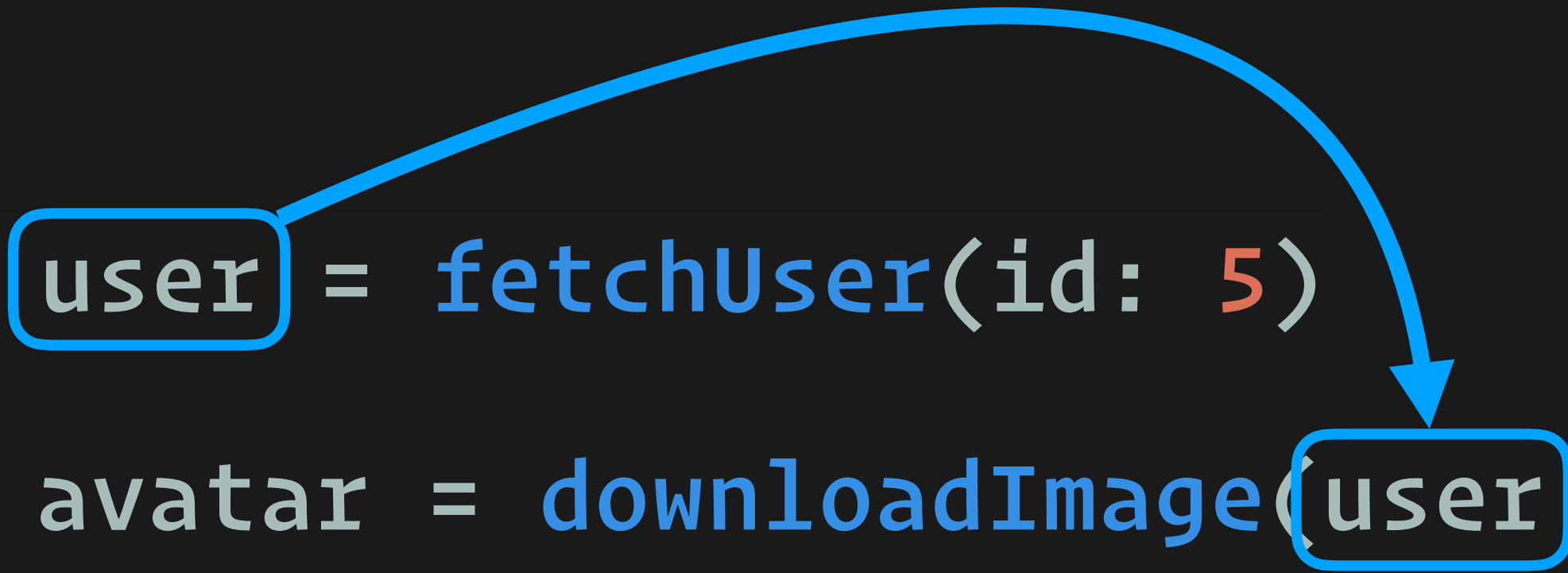
fetchUser(id: 5) { user, error in
    guard error != nil else {
        print("ERROR: \(error)")
        return
    }

    guard let user = user else {
        return
    }

    // user!
}
```

# Chaining Synchronous Code

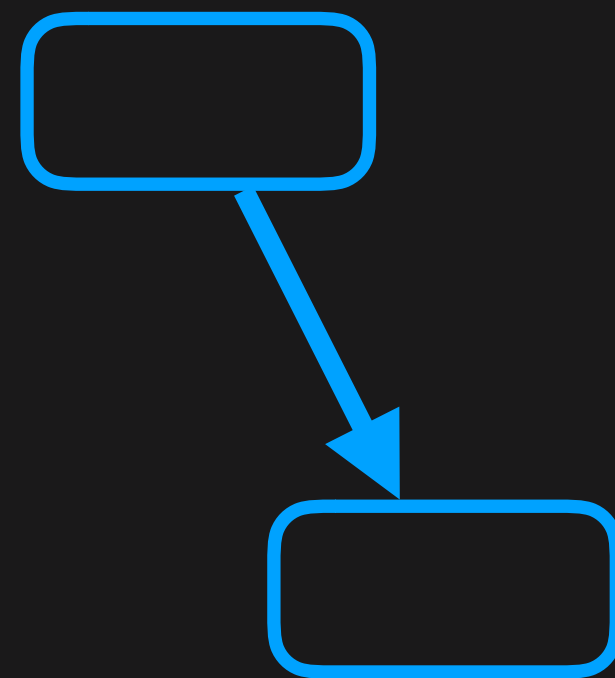
```
let user = fetchUser(id: 5)  
let avatar = downloadImage(user.avatar)  
imageView.image = avatar
```



The diagram illustrates the flow of data in the provided code. A blue box highlights the variable `user` in the first line. A blue curved arrow originates from this box and points to another blue box highlighting the `user` parameter in the second line, demonstrating how the value of `user` is passed to the `downloadImage` function.

# Chaining Asynchronous Code

```
fetchUser(id: 5) { user, error in
```



```
}
```

# It gets more complex

All I have is a User ID (i.e “5”)

What I need is the weather in that user’s hometown....

Authenticated with a token

Int → User → Hometown → Zip → Coordinates → Weather

```
authenticate { token
    self.persistToken(token)
    self.fetchUser(id: 5) { user, error in
        guard let user = user else {
            self.handleError(error); return
        }
        let hometown = user.hometown
        self.fetchCoordinates(hometown.zip) { result in
            switch result {
                case let .failed(error): self.handleError(error)
                case let .success(lat, long):
                    self.fetchWeather(CLLocationCoordinate2d(lat, long)) {
                        weatherResult in
                            switch weatherResult {
                                case let .failed(error): self.handleError(error)
                                case let .success(weather):
                                    self.displayWeather(weather)
                            }
                        }
                    }
            }
        }
    }
}
```

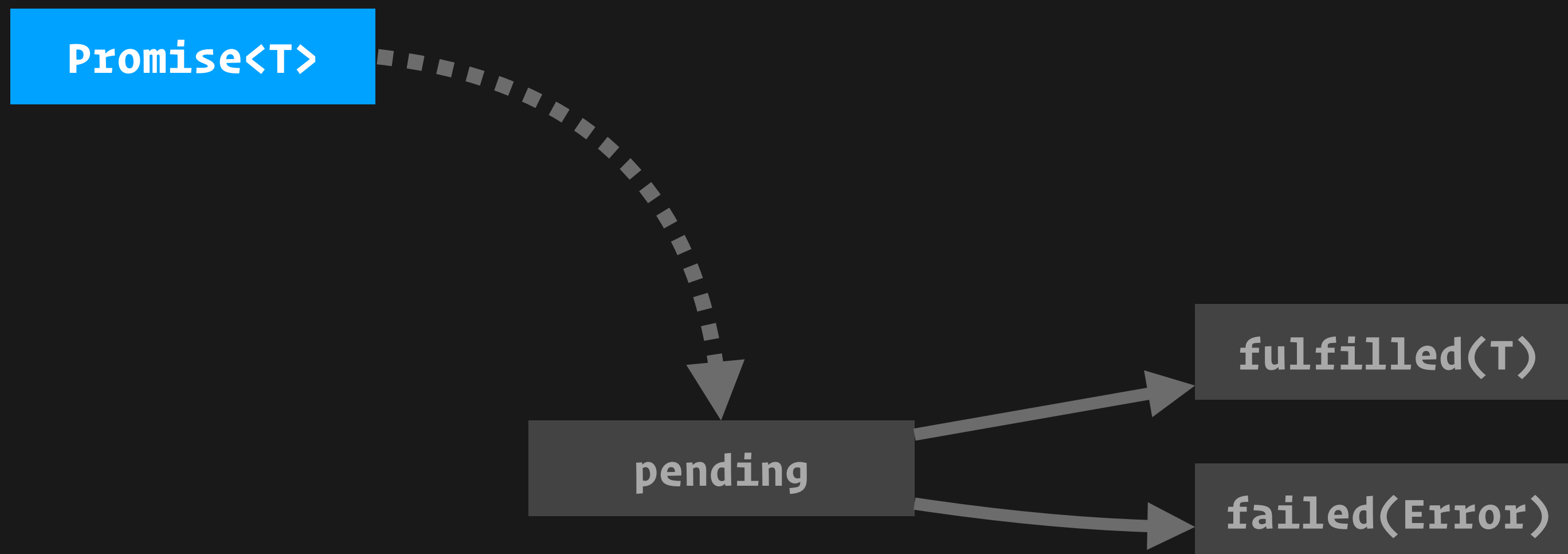
```
authenticate { token
  self.persistToken(token)
  self.fetchUser(id: 5) { user, error in
    guard let user = user else {
      self.handleError(error); return
    }
    let hometown = user.hometown
    self.fetchCoordinates(hometown.zip
      switch result {
        case let .failed(error): self
        case let .success(lat, long)
          self.fetchWeather(CLLocal
            weatherResult in
              switch weatherResult
                case let .failed
                case let .success
                  self.display
              }
            }
          }
        }
      }
```



Is there a better way?

- Promises \*PromiseKit
- Futures \*Vapor / SwiftNIO
- Async / Await \* Swift 7? (or 8??)

# Promise Lifecycle



# Enter Promises

```
func fetchUser(id: Int) -> Promise<User>
```

# Enter Promises

```
func fetchUser(id: Int) -> Promise<User>
```

```
fetchUser(id: 5)  
    .done { user in  
        print("User: \(user.username)")  
    }
```

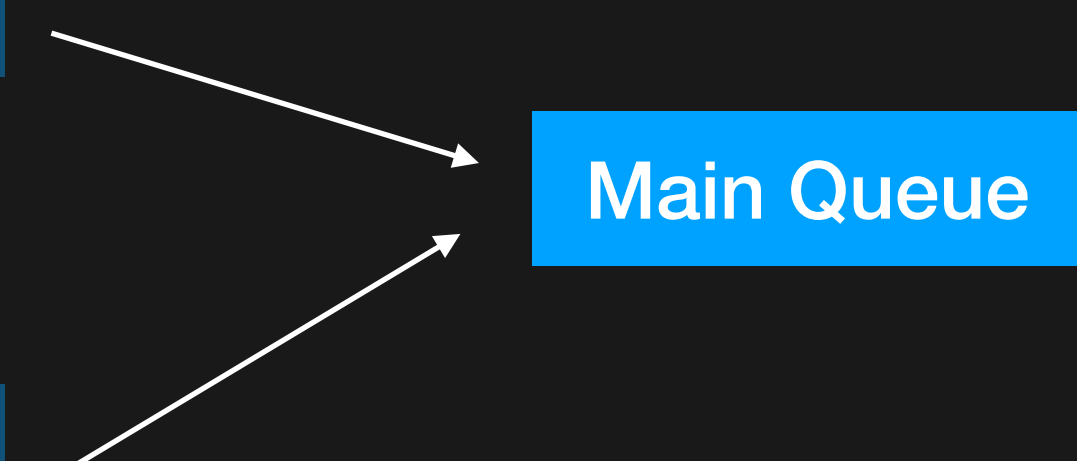
# Handling Errors

```
fetchUser(id: 5)
  .done { user in
    print("User: \(user.username)")
  }
  .catch { error in
    print("Uh-oh: \(error)")
  }
```

# DispatchQueue.main.async?

```
fetchUser(id: 5)
    .done { user in
        print("User: \(user.username)")
    }
    .catch { error in
        print("Uh-oh: \(error)")
    }

promise.done(on: .global()) {
    // on background queue
}
```



Main Queue

# Always run this code...

```
activityIndicator.startAnimating()
```

```
fetchUser(id: 5).done { user in  
    print("User: \(user.username)")
```

```
}
```

```
.catch { error in  
    print("Uh-oh: \(error)")
```

```
}
```

```
.ensure {  
    self.activityIndicator.stopAnimating()  
}
```

# Transform Promise Values

```
fetchUser(id: 5)

  .map { $0.hometown.zip }

  .done { zip in
    print("Hometown zip: \(zip)")
  }

  .catch { error in
    print("Uh-oh: \(error)")
  }
```

# Transform Promise Values

```
fetchUser(id: 5)

    .map { $0.hometown.zip }

    .done { zip in
        print("Hometown zip: \(zip)")
    }

    .catch { error in
        print("Uh-oh: \(error)")
    }
}
```

Promise<User>



Promise<String>

# Chaining Promises

```
func fetchWeather(zip: String) -> Promise<Weather>
```

# Chaining Promises

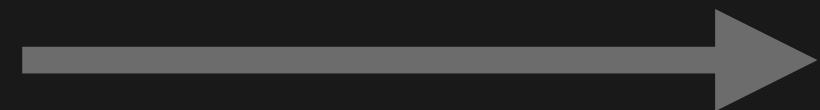
```
fetchUser(id: 5)
    .map { $0.hometown.zip }
    .then { fetchWeather(for: $0) }
    .done { weather in
        print("Weather: \(weather)")
    }
    .catch { error in
        print("Uh-oh: \(error)")
    }
```

..... Waits for new promise

A diagram illustrating the flow of promise chaining. A blue arrow points from the 'id: 5' argument of 'fetchUser' to the '\$0' parameter in the '.map' block. Another blue arrow points from the '\$0' parameter in the '.then' block to the 'weather' parameter in the '.done' block. A third blue arrow points from the 'fetchWeather' function call in the '.then' block to the 'weather' parameter in the '.done' block. A series of blue dots is positioned between the '.then' and '.done' blocks, with the text 'Waits for new promise' to its right.

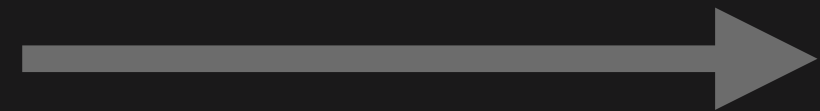
# Chaining Promises

```
fetchUser(id: 5)
```



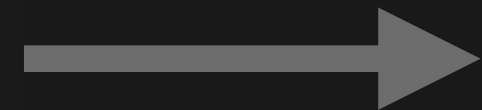
Promise<User>

```
.map { $0.hometown.zip }
```



Promise<String>

```
.then { fetchWeather(for: $0) }
```



Promise<Weather>

```
.done { weather in  
    print("Weather: \(weather)")  
}
```

```
.catch { error in  
    print("Uh-oh: \(error)")  
}
```

# Collecting Values

```
func heatWater() -> Promise<Water>
```

```
func grindBeans() -> Promise<Grounds>
```

```
func brew(water: Water, grounds: Grounds) -> Promise<☕>
```

# Collecting Values

```
heatWater()  
  .then { water in  
  
  }
```

# Collecting Values

```
heatWater()  
  .then { water in  
        grindBeans()  
      }  
  .then { grounds in  
      }
```

# Collecting Values

```
heatWater()  
  .then { water in  
    grindBeans()  
  }  
  .then { grounds in  
    ????  
  }
```

# Collecting Values

```
heatWater()  
  .then { water in  
          grindBeans()  
        }  
  .then { grounds in  
          ?????  
        }  
}
```

The diagram illustrates the flow of a value from a nested block to an outer block. A blue box highlights the variable `water` in the first `.then` block. A blue arrow points from this box to a second blue box in the second `.then` block, which is currently empty. A green question mark `?` is positioned above the arrow, indicating a question about how the value is passed or collected.

# Collecting Values

```
heatWater()  
  .then { water in  
    grindBeans().map { (water, $0) }  
  }  
  .then { water, grounds in  
  }
```

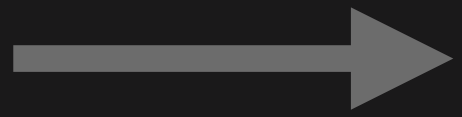
# Collecting Values

```
heatWater()  
  .then { water in  
    grindBeans().map { (water, $0) }  
  }  
  .then { water, grounds in  
    brewCoffee(water: water, grounds: grounds)  
  }
```

# Concurrent Promises

```
when(fulfilled: heatWater(), grindBeans())  
    .then { (water, grounds) in  
        brewCoffee(water: water, grounds: grounds)  
    }
```

# Timing

`after(.seconds(3))`  `Promise<Void>`

# Timing

```
let waitAtLeast = after(seconds: 0.3)
```

```
foo()
```

```
  .then { waitAtLeast }
```

```
  .done {  
    //...
```

```
  }
```

# Racing

```
race(promise1, promise2)  
  .done { result in  
    // first one wins!  
  }
```

# Racing

```
let timeout = after(seconds: 5)

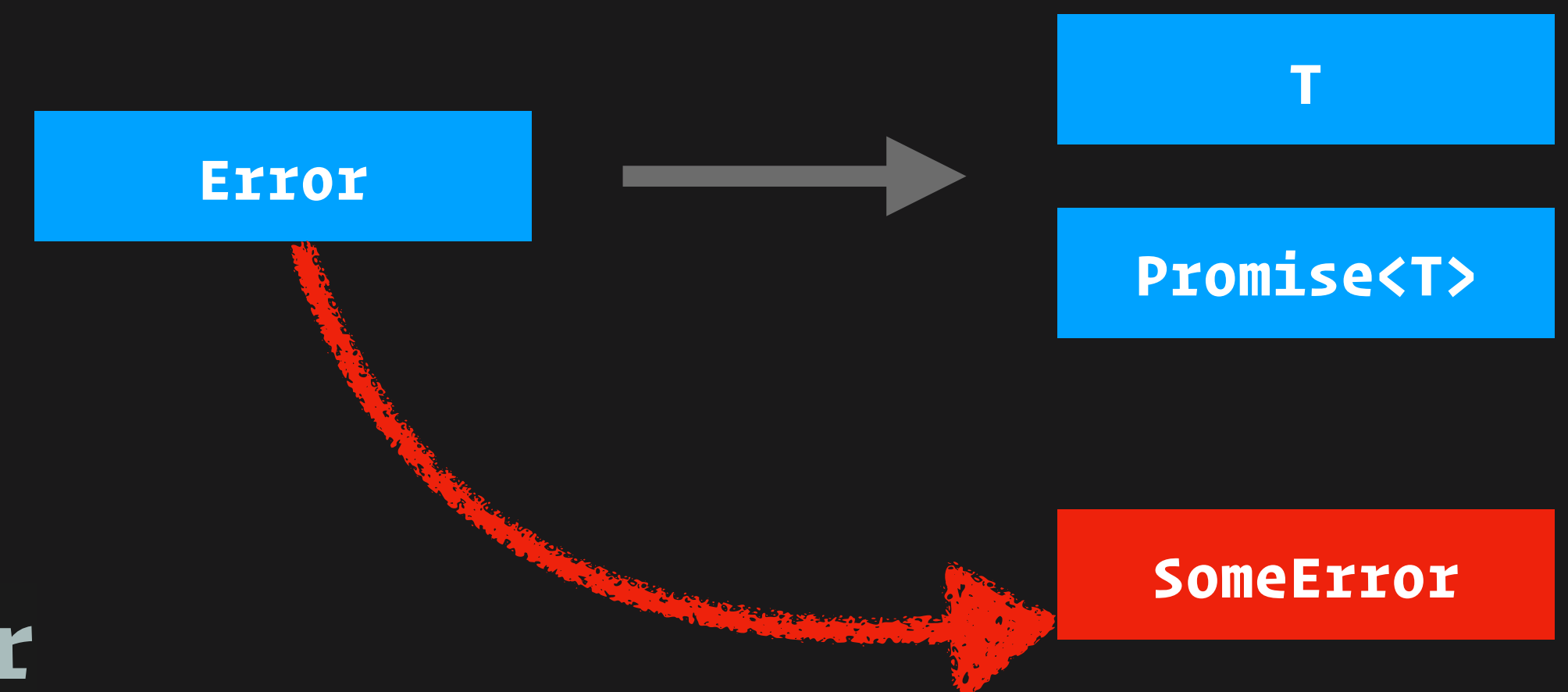
let promise1 = fetchData()

race(promise1.asVoid(), timeout)
    .done { _ in

}
```

# Recover

```
featchWeather(for: ...)  
  .recover { error in  
    return sunnyWeather  
  }  
  .done { ... }
```



# Firstly

```
doThis()  
  .then {  
    doThat()  
  }  
  .done {  
    updateUI()  
  }
```



# Firstly

```
firstly {  
    doThis()  
}.then {  
    doThat()  
}  
.done {  
    updateUI()  
}
```

# Creating Promises

```
func fetchUser(id: 5) -> Promise<User> {
```

```
}
```

# Creating Promises

```
func fetchUser(id: 5) -> Promise<User> {  
    return Promise { seal in
```

```
}
```

```
}
```



# Creating Promises

```
func fetchUser(id: 5) -> Promise<User> {  
    return Promise { seal in  
        userDB.loaduserWithID(5) { user, error in
```

```
        }  
    }  
}
```

# Creating Promises

```
func fetchUser(id: 5) -> Promise<User> {  
    return Promise { seal in  
        userDB.loaduserWithID(5) { user, error in  
            if let e = error {  
                seal.reject(e)  
            }  
        }  
    }  
}
```

# Creating Promises

```
func fetchUser(id: 5) -> Promise<User> {  
    return Promise { seal in  
        userDB.loaduserWithID(5) { user, error in  
            if let e = error {  
                seal.reject(e)  
            } else if let u = user {  
                seal.fulfill(u)  
            }  
        }  
    }  
}
```

# Retry

```
func attempt<T>(maximumRetryCount: Int = 3,  
    delayBeforeRetry: DispatchTimeInterval = .seconds(2),  
    _ body: @escaping () -> Promise<T>  
    ) -> Promise<T> {
```

}

# Retry

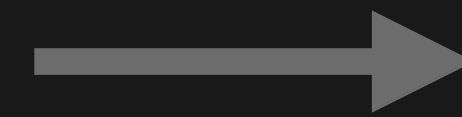
```
func attempt<T>(maximumRetryCount: Int = 3,  
               delayBeforeRetry: DispatchTimeInterval = .seconds(2),  
               _ body: @escaping () -> Promise<T>  
               ) -> Promise<T> {  
  
    var attempts = 0  
    func attempt() -> Promise<T> {  
        attempts += 1  
        return body()  
            .recover { error -> Promise<T> in  
                guard attempts < maximumRetryCount else { throw error }  
                return after(delayBeforeRetry)  
                    .then(on: nil, attempt)  
            }  
    }  
    return attempt()  
}
```

# Retry

```
attempt(maximumRetryCount: 3) {  
    uploadFile(path: pathToFile)  
}  
.then { remoteURL in  
    //...  
}  
.catch { _ in  
    // we attempted three times but still failed  
}
```

# PromiseKit ❤️ CoreLocation

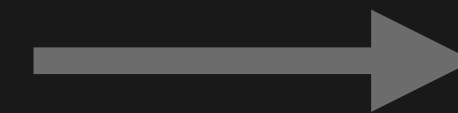
`CLLocationManager.promise()`



`Promise<CLLocationCoordinate2D>`

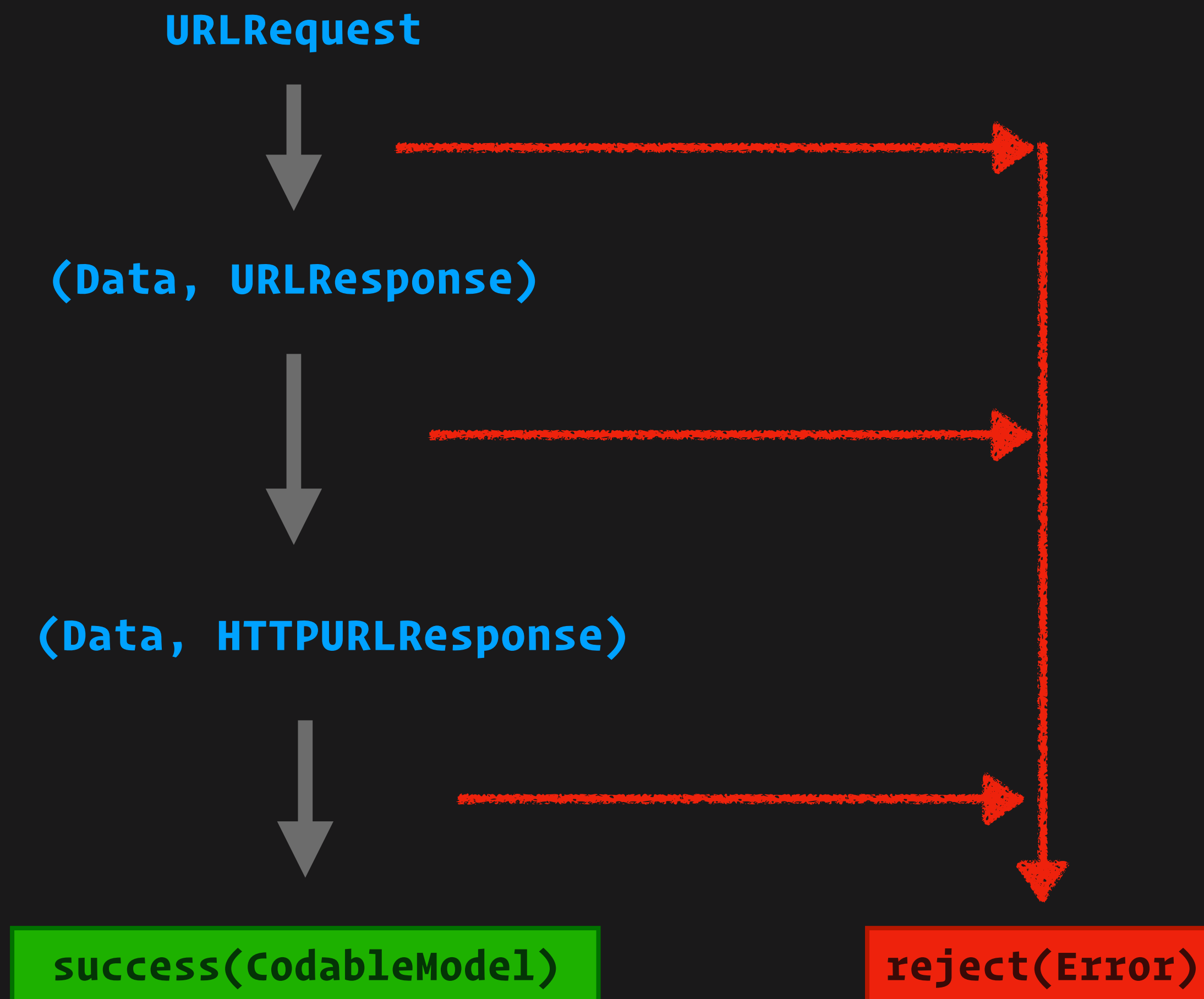
# PromiseKit ❤️ URLSession

`URLSession.shared.dataTask(.promise, with: req)`



`Promise<(Data, URLResponse)>`

# PromiseKit ❤️ URLSession



# PromiseKit ❤️ URLSession

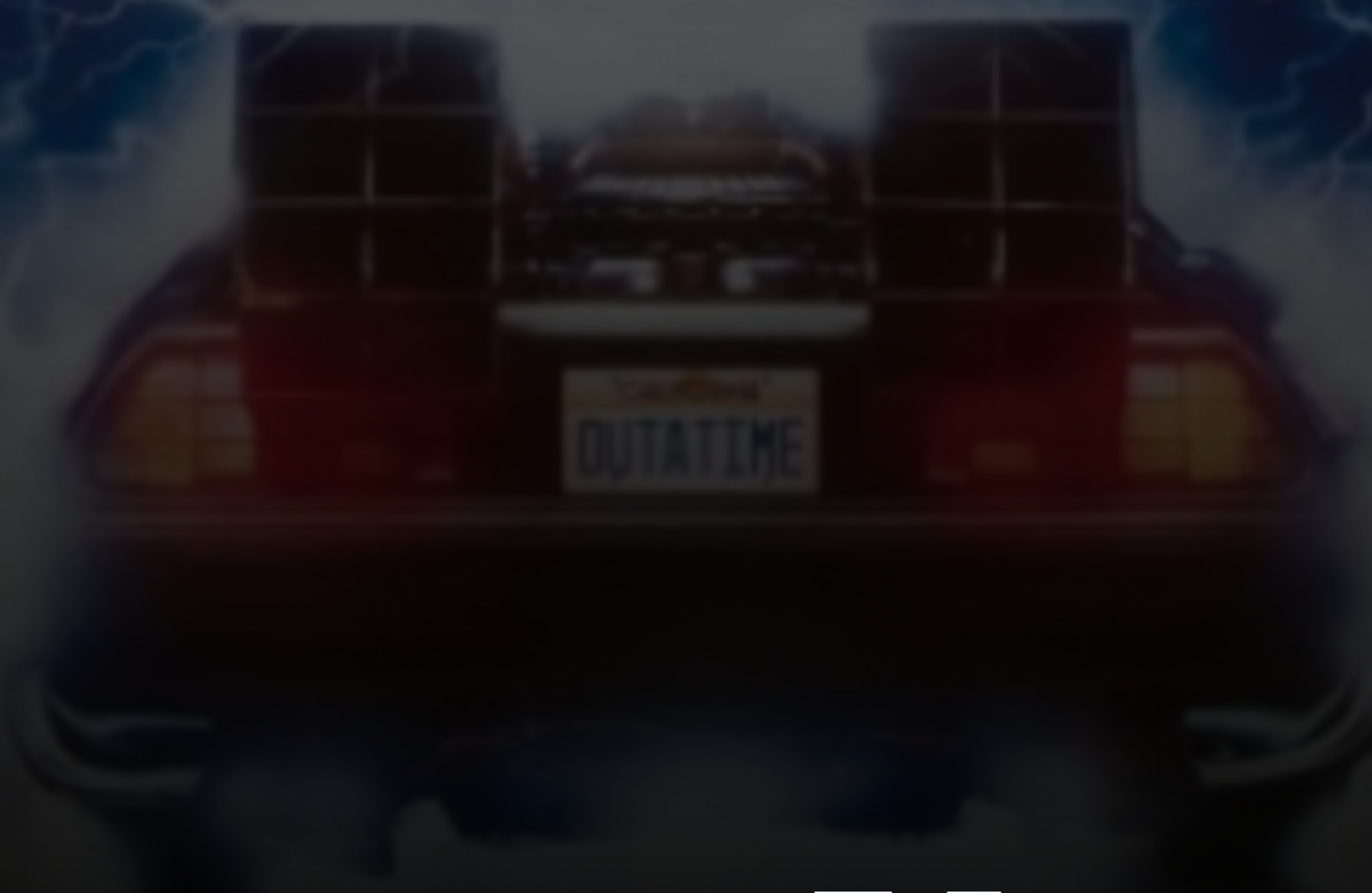
```
firstly {  
    URLSession.shared.dataTask(.promise, with: req)  
}  
.compactMap { result in  
    guard let http = result.response as? HTTPURLResponse else { return nil }  
    return (data: result.data, http: http)  
}  
.map { (result: (data: Data, http: HTTPURLResponse)) -> Joke in  
    guard result.http.statusCode == 200 else {  
        throw APIError.requestFailed(result.http, result.data)  
    }  
    return try JSONDecoder().decode(Joke.self, from: result.data)  
}
```

*“I admit I’m quite afraid  
of elevators, so I’m taking  
steps to avoid them”*

# PromiseKit ❤️ URLSession

DEMO

# BACK TO THE FUTURE



*Futures*

in

Vapor / SwiftNIO

# Vapor Futures are SwiftNIO Futures

```
typealias Future = EventLoopFuture
```

# Both also have Promises

```
typealias Promise = EventLoopPromise
```

**What's the difference?**

A **Future** is a *read-only*  
container for a value that  
hasn't arrived yet.

A **Promise** is writable  
( usually only once).

Your code will create a  
**Promise**, but *return* a  
**Future**.

# Creating Futures (synchronously)

```
func greet(_ req: Request) -> Future<String> {  
    let eventLoop: EventLoop = req.eventLoop  
    return eventLoop.newSucceededFuture(result: "Hello... McFly!")  
}
```

# Creating Futures (asynchronously)

```
func greet(_ req: Request) -> Future<String> {  
    let promise = req.eventLoop.newPromise(of: String.self)  
    DispatchQueue.global().async {  
        sleep(1)  
        promise.succeed(result: "Great Scott!")  
    }  
    return promise.futureResult  
}
```

# Future.do

```
let decodeFuture: Future<Todo>  
  
decodeFuture = try req.content.decode(Todo.self)  
  
decodeFuture.do { todo in  
    print("Decoded Todo! \(todo)")  
}
```

PromiseKit: done

# Future.and

```
let future1: Future<Todo?> = try Todo.find(1, on: req)
let future2: Future<Todo?> = try Todo.find(2, on: req)

future1.and(future2)
  .do { todo1, todo2 in
    print("Results: \(todo1) and \(todo2)")
  }
```

PromiseKit: `when(fulfilled:)`

# Future.catch

```
someFuture.catch { error in  
    print("We found an error :(")  
}
```

PromiseKit: `catch`

# Future.map

```
func randomNumber(on worker: Worker) -> Future<Int> {  
    return worker.future(Int.random(in: 1...100))  
}  
  
// Request is a Worker  
randomNumber(on: req)  
    .map(to: String.self) { num in  
        return "\($num) jiggawatts"  
    }
```

PromiseKit: map

# Future.flatMap

```
Todo.find(1, on: req)
  .map(to: Todo.self) { maybeTodo in
    guard let todo = maybeTodo else {
      throw Abort(.notFound)
    }
    return todo
  }
  .flatMap { todo in
    return todo.delete(on: req)
  }
```

PromiseKit: then

# Future.flatMap

```
let future: Future<Future<String>>
```



flatMap



```
let future: Future<String>
```

**Vapor Futures are similar to  
traditional Promises**

They take **asynchronicity** and  
wrap it in a type

They blur the line between  
**asynchronous**  
and  
**synchronous**

**What if this was built-in to Swift?**

Hold my ~~beer~~ dog



# Swift Concurrency Manifesto



[bit.ly/concurrency-manifesto](https://bit.ly/concurrency-manifesto)

# Async / Await

SE -XXXX

[bit.ly/async-await-proposal](https://bit.ly/async-await-proposal)

```
func loadWebResource(_ path: String, completionBlock: (result: Resource) -> Void) { ... }
func decodeImage(_ r1: Resource, _ r2: Resource, completionBlock: (result: Image) -> Void)
func dewarpAndCleanupImage(_ i : Image, completionBlock: (result: Image) -> Void)

func processData1(completionBlock: (result: Image) -> Void) {
    loadWebResource("datapofile.txt") { dataResource in
        loadWebResource("imagedata.dat") { imageResource in
            decodeImage(dataResource, imageResource) { imageTmp in
                dewarpAndCleanupImage(imageTmp) { imageResult in
                    completionBlock(imageResult)
                }
            }
        }
    }
}
```

```
func loadWebResource(_ path: String) async -> Resource
func decodeImage(_ r1: Resource, _ r2: Resource) async -> Image
func dewarpAndCleanupImage(_ i : Image) async -> Image

func processImageData1() async -> Image {
    let dataResource = await loadWebResource("datapofile.txt")
    let imageResource = await loadWebResource("imagedata.dat")
    let imageTmp      = await decodeImage(dataResource, imageResource)
    let imageResult   = await dewarpAndCleanupImage(imageTmp)
    return imageResult
}
```

```
func loadWebResource(_ path: String) async -> Resource
func decodeImage(_ r1: Resource, _ r2: Resource) async -> Image
func dewarpAndCleanupImage(_ i : Image) async -> Image

func processImageData1() async -> Image {
    let dataResource = await loadWebResource("datapofile.txt")
    let imageResource = await loadWebResource("imagedata.dat")
    let imageTmp      = await decodeImage(dataResource, imageResource)
    let imageResult   = await dewarpAndCleanupImage(imageTmp)
    return imageResult
}
```

```
func loadWebResource(_ path: String) async -> Resource
```

```
let dataResource = await loadWebResource("datapofile.txt")
```

# Synchronous Code

```
func fetchUser(id: Int) -> User
```

```
let user = fetchUser(id: 5)
```

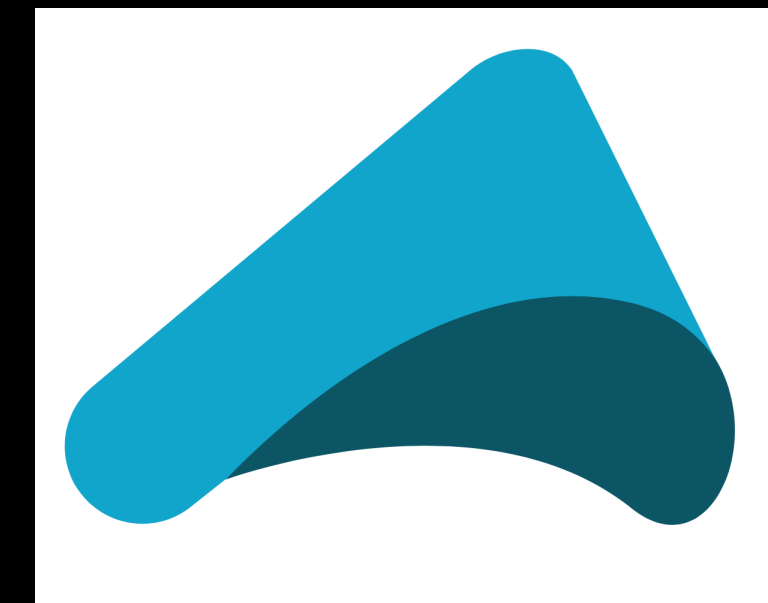
# Synchronous Code

```
func fetchUser(id: Int) async -> User
```

```
let user = await fetchUser(id: 5)
```



# Learning from other communities



**Learning Promises / Futures**  
**Today will help you tomorrow**

*Thank you!*

*Thank you!*

Ben Scheirman

@subdigital



*NS Screencast*