ASYNCHRONOUS PROGRAMMING

WHAT IS ASYNC?

- REACTIVE/EVENT DRIVEN
- PROMISES & FUTURES
- EVENT LOOPS
- NONBLOCKING
- STREAMS

WHY?

VAPOR 2

SYNCHRONOUS EVENT LOOPS

VAPOR 2

```
// Creating a new user in a JSON API
drop.post("users") { request in
    // Extract form data
    guard
        let json = request.json,
        let email = json["email"]?.string,
        let password = json["password"]?.string
    else {
        throw Abort.badRequest
    }

    // Create a new user
    let user = try User(email: email, hasingPassword: password)

    // Success is dependent on successful insertion
    try user.save()

    return Response(status: .ok)
}
```

VAPOR 3

ASYNCHRONOUS EVENT LOOPS

VAPOR 3

```
/// Type safe registration form
struct RegistrationRequest: Decodable {
    var email: String
    var password: String
}

app.post("users") { request in
    // Extract form data
    let form = try JSONDecoder().decode(RegistrationRequest.self, from: request.body.data)

    // Create a new user
    let user = try User(email: form.email, hasingPassword: form.password)

    // Success is dependent on successful insertion
    return try user.save().map {
        // This transform will be ons successful save
        return Response(status: .ok)
    }
}
```

VAPOR 3

- LESS DEPENDENT ON EXTERNAL SOURCES
- SAME AMOUNT OF CODE (20 LINES)
- MUCH MORE PERFORMANT
- MUCH MORE SCALABLE

AWESOME!

HOW DO I USE IT?

3 MAIN CONCEPTS

- PROMISE + FUTURE
- STREAMS
- EVENT LOOP

PROMISE & FUTURE

THE OBJECTS

```
// Create a promise
// Promises are write-only, emitting an event
let promise = Promise<String>()

// A future can be extracted from a promise
// Futures are read-only, receiving the promised event
let future: Future<String> = promise.future
```

DELIVERING A PROMISE

```
let promise = Promise<String>()

// Complete the promise whenever it's ready
promise.complete("Hello world")

// Or fail the promise if something went wrong
promise.fail(error)
```

RECEIVING THE RESULT

```
let future: Future<String> = promise.future

// `then` will be called on success
future.then { string in
    print(string)

// `catch` will be called on error
}.catch { error in
    // Handle error
}
```

TRANSFORM RESULTS

```
struct UserSession: SessionCookie {
    var user: Reference<User>
}

app.get("profile") { request in
    let session = try request.getSessionCookie() as UserSession

    // Fetch the user
    return try session.user.resolve().map { user in
        // Map the user to a ResponseRepresentable
        // Views are ResponseRepresentable
        return try view.make("profile", context: user.profile, for: request)
    }
}
```

NESTED ASYNC OPERATIONS ❌

```
app.get("friends") { request in
    let session = try request.getSessionCookie() as UserSession

    let promise = Promise<View>()

    // Fetch the user
    try session.user.resolve().then { user in
        // Returns all the user's friends
        try user.friends.resolve().then { friends in
            return try view.make("friends", context: friends, for: request).then {  renderedView in
                promise.complete(renderedView)
            }.catch(promise.fail)
        }.catch(promise.fail)
    }.catch(promise.fail)

    return promise.future
}
```

NESTED ASYNC OPERATIONS ✅

```
app.get("friends") { request in
    let session = try request.getSessionCookie() as UserSession

    // Fetch the user
    return try session.user.resolve().flatten { user in
        // Returns all the user's friends
        return try user.friends.resolve()
    }.map { friends in
        // Flatten replaced this future with
        return try view.make("friends", context: friends, for: request)
    }
}
```

YOU CAN ADD MANY CALLBACKS

```
let future: Future<String> = generateLogMessage()

future.then(print)
future.then(log.error)

future.map { message in
    try LogMessage(message).save()
}.catch { saveFailure in
    log.fatal(saveFailure)
}
```

SYNCHRONOUS APIS ARE STILL USABLE

WHEN WORKING WITH SYNCHRONOUS APIS, YOU CAN BLOCK THE ASYNC OPERATION

```
// Throws an error if the promise failed
// Returns the expected result by blocking the thread until completion
let result = try future.blockingAwait(timeout: .seconds(5))
```

STREAM

AN ASYNCHRONOUS SEQUENCE OF EVENTS
WITH 2 PRIMARY TYPES

`InputStream`
&
`OutputStream`

... AND COMPLETELY PROTOCOL ORIENTED

OutputStream

EMITS EVENTS OF A SPECIFIC TYPE

```
// TCP Socket outputting data
socket.flatMap { byteBuffer in
    // returns a `String?`
    // `flatMap` will filter out invalid strings
    return String(bytes: byteBuffer, encoding: .utf8)
// Prints all valid strings
// `print` will print the String and return `Void`, creating a Void stream
// The stream without data will still be called for each event (such as errors)
}.map(print).catch { error in
    // handle errors
}
```

InputStream

RECEIVES EMITTED EVENTS OF A SPECIFIC TYPE

```
class PrintStream: InputStream {
    /// Used to chain errors from stream to stream
    public var errorStream: ErrorHandler?

    func inputStream(_ input: String) {
        print(input)
    }

    init() {}
}

let printStream = PrintStream()

// Takes received bytes from the sockets
socket.flatMap {
    // Tries to turn the bytes into a String
    return String(bytes: byteBuffer, encoding: .utf8)
// Drains it into a PrintStream's inputStream
}.drain(into: printStream)
```

USE CASE - THE VAPOR HTTP SERVER

```
let server = try TCPServer(port: 8080, worker: Worker(queue: myQueue))

// Servers are a stream of accepted web client connections
// Clients are an input and output stream of bytes
server.drain { client in
    let parser = RequestParser()
let router = try yourApplication.make(Router.se
    let serialiser = ResponseSerializer()

    // Parses client-sent bytes into the RequestParser
    let requestStream = client.stream(to: parser)

    // Parses requests to the Vapor router, creating a response
    let responseStream = requestStream.stream(to: router)

    // Serializes the responses, creating a byte stream
    let serializedResponseStream = responseStream.stream(to: serializer)

    // Drains the serialized responses back into the client socket
    serializedResponseStream.drain(into: client)
}
```

```
let client = try TCPClient(worker: Worker(queue: myQueue))

try client.connect(hostname: "example.com", port: 80)

try client.send(data)

let data = try client.read()
```

```
let client = try TCPClient(worker: Worker(queue: myQueue))

try client.connect(hostname: "example.com", port: 80)

try client.writable(queue: myQueue).then {
    try client.send(data)

    let data = try client.read()
}

try client.start()
```

EVENT LOOPS

A THREAD/DISPATCHQUEUE OF TASKS
TASKS CAN BE ADDED FOR AN EVENT

WORKERS PROVIDE CONTEXT

- SHARE 'GLOBALS' SUCH AS A DATABASE DRIVER
  - GUARANTEED TO BE THE SAME THREAD
    - REQUIRE NO THREAD-SAFETY

USING A GLOBAL DATABASE DRIVER ❌

USING THE CONTEXT'S DATABASE DRIVER ✅

RESULT

- PERFORMANCE
- STABILITY
- EASIER APIS

THE END