Vapor

Updating

```
vapor self udate
```

New Project

```
vapor new <insert-project-name-here>
```

Vapor creates a folder structure that consists of the following folders+files:

```
1. Sources - this contains source files
2. Public
3. Resources - contains the views
4. Package.swift
```

Sources contains a file named `main.swift`

Droplet

Inside the `main.swift` file is the following line:

```swift
let drop  = Droplet()
```

Note: A `Droplet` is responsible for **registering** routes, **starting** the server, **appening** middleware, and more.

Routing

After the creation of `drop`, add the following snippet:

```swift
drop.get("hello") { request in
    return "Hello, world!"
}
```

`.get("hello")` will create a new `route` on `drop` that will match all `GET` requests to `/hello`.

All route closures are passed an instance of **Request** that contains information such as the URI requested and data the data sent to the given endpoint.

Running

At the bottom of `main.swift` you have to make sure you serve your `Droplet`.

```swift
drop.run()
```

Compiling

Go to the root directory of the project and run

```
vapor build
```

Run

Boot up the server by running the following command:

```
vapor run serve
```

After the server is running, you should be able to visit `http://localhost:8080/hello` in the browser.

Xcode

Generate Project

Vapor Toolbox

To generate a new Xcode project for a project, use:

```
vapor xcode
```

after creating the project

Droplet

Initialization

```Swift
import Vapor

let drop = Droplet()

// where the magic happens

drop.run()
```

Environment

The `environment` property contains the current enviornment your application is running in. These environments usually consist of

```
1. Development
2. Testing
3. Production
```

```Swift
if drop.envionment == .production {
    //run production code here
}
```

The environment is `development` by default. To change it, pass the `--env=` flag as an argument

```
vapor run serve --env=production

vapor run serve --env=development
```

Working Directory

The `workDir` property contains a path to the current working directoy of the application relative to where it (the Droplet) started. This property assumes you started the `Droplet` from its root directory.

```
drop.workDir // "var/www/my-project/"
```

Folder Structure

Minimum Folder Structure

Recommended: Put your Swift code inside the `App/` folder. This will allow you to create subfolders in `App/` to organize your models and resources.

This is how packages should be structured:

```
| App
    * main.swift
| Public
* Package.swift
```

The `Public` folder is where all the publicly accessible files should go such as CSS, images, and JavaScript files.

Models

The `Models` folder is where you put your database and other model files.

```
| App
    | Models
        * User.swift
```

Controllers

The `Controllers` folder is where you put your route controllers.

```
| App
    | Controllers
        * UserController.swift
```

Views

The `Views` folder in `Resources` is where Vapor will look when you render views for your endpoints.

```
| App
| Resources
    | Views
        * user.html
```

The following code wwould load the `user.html` file:

Swift

```
drop.view.make("user.html")
```

Config

```
| App
    | Config
        * app.json          // default app.json
            | development
                * app.json  //overrides app.json when in development environment
            | production
                * app.json  //overrides app.json when in production environment
            | secrets
                * app.json  //overrides app.json in all environments, ignored by g
```

JSON

Request

JSON is automatically available in `request.data` alongside **form-urlencoded data** and **query data**. So if you want to access JSON sent through a request then call the property on the `request` object sent through the closure. Same for queries from GET requests.

```Swift
drop.get("hello") { request in
    guard let name = request.data["name"]?.string else {
        throw Abort.badRequest
    }
    return "Hello, \(name)!"
}
```

JSON Only

To **specifically** target JSON from the request, use the `request.json` property

```Swift
drop.post("json" { request in
    guard let name = request.json?["name"]?.string else {
        throw Abort.badRequest
    }

    return "Hello, \(name)!"
}
```

This will **only** work if the request is sent with JSON data.

Response

To respond *with* JSON simply wrap your data structure with `JSON(node: )`

```Swift
drop.get("version") { request in
    return try JSON(node: [
        "version": "1."
    ])
}
```

Views

Views Directory

Views return HTML data from your application. They can be created from either pure HTML docs or passed through renderers such as **Mustache** or **Stencil**.

HTML

```Swift
drop.get("html") { request in
    return try drop.view.make("index.html")
}
```

Templating

Templated documents like *Leaf*, *Mustache*, or *Stencil* can take a `Context`. A `Context` can pass data into the template for use in the view.

```Swift
drop.get("template") { request in
    return try drop.view.make("welcome", [
        "message": "Hello, world!"
    ])
}
```

Leaf

A simple templating language that can make generating views easier.

Syntax

Structure

Leaf tags are made up of 4 elements:

- Token: `#` is the Token

- Name: A `string` that identifies the tag
- Parameter List: `()` May accept 0 or more arguments.
- Body(optional): `{}` Must be separated from the Parameter List by a space

Examples:

- `#()`
- `#(variable)`
- `#import("template")`
- `#export("link") { <a href="#()"></a> }`
- `#index(friends, "0")`
- `#loop(friends, "friend") { <li>#(friend.name)</li> }`
- `#raw() { <a href="#raw"> Anything goes! </a> }`

Chaining

The double token `##` indicates a chian. It can be applied to any standard tag

`#if(hasFriends) ##embed("getFriends")`

Custom Tag

`import Leaf`

A custom tag that takes two arguments, an array, and an index to access

```Swift
class Index: BasicTag {
    let name = "index"

    func run(arguments: [Argument]) throws -> Node? {
        guard
            arguments.count == 2,   //make sure 2 arguments were passed through
            let array = arguments[0].value?.nodeArray, //first argument should be
            let index = arguments[1].value?.int,
            index < array.count
        else { return nil }

        return array[index]
    }
}
```

And register the tag in our `main.swift` file with:

```swift
if let leaf = drop.view as? LeafRenderer {
    leaf.steam.register(Index())
}
```

Controllers

`import HTTP`

Basic

```swift
final class HelloController {
    func sayHello(_ req: Request) throws -> ResponseRepresentable {
        guard let name = req.data["name"] else {
            throw Abort.badRequest
        }

        return "Hello, \(name)"
    }
}
```

Registering

Required The signature of each method in the controller must follow a certain structure.

This signature must be of `(_ varName: Request) throws -> ResponseRepresentable`. Both `Request` and `ResponseRepresentable` are made available by importing the `HTTP` module.

To register the controller:

```swift
let helloController = HelloController()
drop.get("hello", handler: helloController.sayHello)
```

Resources

Controllers that conform to `ResourceRepresentable` can be registered into a router as a RESTful resource.

```swift
final class UserController {
    func index(_ request: Request) throws -> ResponseRepresentable {
        return try User.all().makeNode().converted(to: JSON.self)
    }

    func show(_ request: Request, _ user: User) -> ResponseRepresentable {
        return user
    }
}
```

These are typical `index` and `show` routes. Indexing returns a JSON list of all users and showing returns a JSON representation of a single user.

Having *UserController* **extend** `ResourceRepresentable` makes the standard RESTful structure easier since we won't have to register each individual route.

```swift
extension UserController: ResourceRepresentable {
    func makeResource() -> Resource<User> {
        return Resource(
            index; index,
            show: show
        )
    }
}
```

Conforming *UserController* to `ResourceRepresentable` requires that the signatures of the `index` and `show` methods match what the `Resource<User>` is expecting.

Then to register the *UserController*

```swift
let users = UserController()
drop.resource("users", users)
```

Middleware

Middleware allows you to modify requests and responses as they pass between the client and the server.

Basic

```swift
final class VersionMiddleware: Middleware {
    func respond(to request: Request, chainingTo next: Responder) throws -> Respor

        /* Immediately ask the next middleware in the chain to respond to the requ
         This happens until the request eventually reaches the Droplet (our server
        let response = try next.respond(to: request)

        /* Modify the response to contain a version header */
        response.headers["Version"] = "API v1.0"

        /* return the response. This follows a chain until it reaches the client *
        return response
    }
}
```

Then supply this middleware to our `Droplet`

```swift
let drop = Droplet()
drop.middleware.append(VersionMiddleware())
```

Request

The middleware can also modify or interact with the **request** being sent to the server

```swift
/* This example doesn't modify the response send to the client. It only chains res
    Droplet */
func respond(to request: Request, chainingTo next: Responder) throws -> Response {

    /* Check to see if the request's cookies has a key called "token" and if this
        our "secret" */
    guard request.cookes["token"] == "secret" else {
        throw Abort.badRequest
    }

    return try next.respond(to: request)
}
```

Validation

For validating data coming into your application

Common Usage

Several useful convenience validators are included by default.

You can use these as is or combine them to create your own custom validators.

```Swift
class Employee {
    var email: Valid<Email>
    var name: Valid<Name>

    init(request: Request) throws {
        name = try request.data["name"].validated()
        email = try request.data["email"].validated()
    }
}
```

By declaring both `email` and `name` properties of type `Valid<T>` you ensure that these properties can only ever contain valid data.

To store something in `Valid<T>` property you must use the `.validated()` method. This is available for any data returned by `request.data`

Validators

- `Valid<OnlyAlphanumeric>`
- `Valid<Email>`
- `Valid<Unique<T>>`
- `Valid<Matches<T>>`
- `Valid<In<T>>`
- `Valid<Contains<T>>`
- `Valid<Count<T>>`

Validators vs ValidationSuites

Validators like `Count` or `Contains` can have multiple configurations.

```Swift
let name: Valid<Count<String>> = try "Vapor".validated(by: Count.max(5))
```

We are validating that the string is at **most** 5 characters long.

Custom Validator

```Swift
class Name:ValidationSuite {
    static func validate(input value: String) throws {
        let evaluation = OnlyAlphanumeric.self
            && Count.min(5)
            && Count.max(20)

        try evaluation.validate(input: value)
    }
}
```

The only method you have to implement is `validate(value:Type) throws`

Sessions

`import Sessions`

Sessions help you store information about a user between requests.

Middleware

Enable sessions on your `Droplet` by adding an instance of `SessionMiddleware`

```Swift
import Sessions

let memory = MemorySessions()
let sessions = SessionsMiddleware(sessions: memory)
```

Then add to the `Droplet`

```Swift
let drop = Droplet()
drop.middleware.append(sessions)
```

Request

After `SessionMiddleware` has been enabled, you can access the `req.sessions()` method to get the access to session data.

```swift
let data = try req.session().data
```

Example - Remembering the user's name

Store

```swift
drop.post("remember") { req in
    guard let name = req.data["name"]?.string else {
        throw Abort.badRequest
    }

    // This stores the name into the session
    //Must wrap the name around a Node
    try req.session().data["name"] = Node.string(name)

    return "Remembered name."
}
```

Fetch

on `GET /remember` fetch the name from the session data and return it

```swift
drop.get("remember") { req in
    guard let name = try req.session().data["name"]?.string else {
        return throw Abort.custom(status: .badRequest, message: "Please POST the r
    }

    return name
}
```

Hash

Example

To hash a string, use `hash` on `Droplet`

```swift
let hashed = drop.hash.make("vapor")
```

SHA2Hasher

By default, Vapor uses a SHA2Hasher with 256