Before Starting

- Turn Bluetooth off
- Enable Don't Disturb Mode
- Start lazycoder: `lazycoder start Presentation/Snippets.markdown`
- Start ngrok: `ngrok http --subdomain=striker-api 8080`
- Prepare TablePlus
- Prepare Paw
- Prepare iTerm and its tabs
- Adjust Xcode font size (also for console)

Vapor Toolbox

First demo. After having the vapor toolbox installed, run:

```
vapor new tuesday
```

Choose the different options the toolbox lets you choose from.

Open the project with:

```
xed tuesday
```

- Show first the **Package.swift** file.
- Then, when package resolution finishes, start building.
- While it builds, show configure and routes.
  *Go over Models and Controllers. Explain what they do, and that the controllers here differ from UIKit view controllers.

Open the **routes.swift** file, and modify the `hello` handler to include a custom greeting.
After that, run the server - emphasize I can run either via command line, or with Xcode. For convenience, I'll use Xcode.
Once it's up, call via command line:

```
curl http://localhost:8080/hello
```

Striker

Explain we'll move to another demo. Now, open the Server - Starter directory and launch Xcode.

First, show the **Package.swift** file, with emphasis to the Shared package.

First Part: UserController methods

Open the web app in the browser, and explain we will implement an API for creating users. Show the QR code, and ask people to try to register. Once they see the 404 error, explain we need to create the controller that will handle this request first.

Create a new file named UserController. Add to the top:

```
import Fluent
import Vapor
import Shared
```

Now, define the controller. Because it handles more than one request, we want it to implement the `RouteCollection` protocol:

```
struct UserController: RouteCollection {

}
```

This protocol consists of only one method, the `boot(routes: RoutesBuilder)` method:

```
        func boot(routes: RoutesBuilder) throws {

        }
```

We will soon be able to fill our routes in the controller. Let's first add the methods themselves. First, we want to allow creating users. Explain what it means:

```
        private func create(req: Request) async throws -> UserResponse {

    }
```

Then, implement the create method. First, decode the user from the JSON body. Then, save it. Finally, convert it to the public struct define in shared, and wrap it in the response:

```
        let user = try req.content.decode(User.self)
                try await user.save(on: req.db)
                return UserResponse(user: try user.asPublic)
```

Let's run the server and try that again.

Good. Now we want another method, to list all the users. First, declare it:

```
    private func allUsers(req: Request) async throws -> UsersResponse {

    }
```

Now, perform the query for all users, and return them in the response.
Obviously, this is not secure in any way, and you should never expose a list of users publicly and without authorization. We're also not worrying here about pagination.

```
        let users = try await User.query(on: req.db).all()
        return UsersResponse(users: try users.asPublic)
```

We have our methods, but no one is calling them. Let's add'em to our controller:

```
        // example.com/users/...
            let users = routes.grouped("users")


                // GET example.com/users
                users.get(use: allUsers)


                // POST example.com/users/new
                users.post("new", use: create)
```

We have these errors, because Vapor routing expects the response content to be `Content`. The structs from the `Shared` package are only `Codable`, but not `Content`. Let's fix this:

```
extension Shared.User: Content {}
extension UserResponse: Content {}
extension UsersResponse: Content {}
```

Let's build and run, and try again. Now, who can guess what we are missing? We still haven't "connected" the app to use our new shiny controller.

Open **routes.swift**, and add:

```
    try app.register(collection: UserController())
```

Open now TablePlus, show the database and the `users` table.

Try running the `/users/all` call. From Terminal, and from Paw.

Second Part: Increment Goal Count

Let's implement now the `/goals/increment` method.

First, get the `userId` from the request. This is an extension I created, that extracts the userId from a request header. This is not secure or doesn't contain an authorization token - it is just for the purpose of the example.

```
let userId = try req.userId
```

Then, query the user from the DB based on the `userId`. Notice how Fluent uses both property wrappers, keypaths, and operator overloads to help build your query. If the user is not found, throw a bad request error (400)

```
guard let user = try await User.query(on: req.db)
        .filter(\.$id == userId)
        .first() else {
        throw Abort(.badRequest)
}
```

We're almost there. I previously implemented the `get(key:)` function, that receives a redis key, and gets the count for it. I also added a parameter that allows me to increment the count. We'll get two counts here: one for the current user, and another one for all users. Notice how I use async let, so the first method doesn't block the second.

```
async let currentCount = try await get(key: "goals-count:user:\(userId)", on: req, increment: t
async let todayCount = try await get(key: redisTodayKey, on: req, increment: true)
```

The next step is to update the live socket - as the iOS app will be listening to it. For every websocket connected, I keep a reference to, and now I update it with a JSON string:

```
let summary = try await GoalsSummary(today: todayCount).asJSONString
liveSockets.send(summary)
```

Now, finalize with the response for the http call:

```
return try await UserGoals(user: try user.asPublic, count: currentCount)
```