Adding auth into your app

Ways to auth:

- easy way: https://auth0.com/docs/quickstart/backend/nodejs
  - pricey. Not free after 7k users
- passport (advanced). very versatile. not necessarily easy.

PQs

- Postman
- Sequelize in your app
- create a <appname>_<envname> postgres database

- User model definition
- Sync User table in app script (at least add the user table to your database)
- Ensure you can connect to sequelize

Goal 1: Signing up

Be able to make a POST request to **`localhost:3000/user/signup`** with a request body:

```
{
    "email": "test1@gmail.com",
    "password": "12345",
    "firstName": "myFirst",
    "lastName": "myLast",
}
```

and have our server

1. create a user with the credentials
2. get back a user record from our database as the response.

- The passwords should be encrypted in our database

Goal 2: Logging in

Be able to make a POST request to **`/user/login`**

- Validate credentials

-- Skip to logging in first (easier to understand first)

Goal 3: Give logged in users access to authenticated routes via tokens

- prevent non-authenticated users from accessing these routes
- npm package: jsonwebtoken

passportjs.org/docs

See "Username & Password" section

Passport
* allows you to authenticate using various strategies
 * username & password: the "local" strategy
 * strategies are packaged as individual npm modules (passport-<strategyname>)

passport-local
* see http://passportjs.org/docs/username-password
* lets you authenticate using a username and password in your Node.js applications

```
passport.use(new LocalStrategy({
    usernameField: 'email',
    passwordField: 'passwd',
    session: false
  },
  function(username, password, done) {
    // ...
  }
));
```

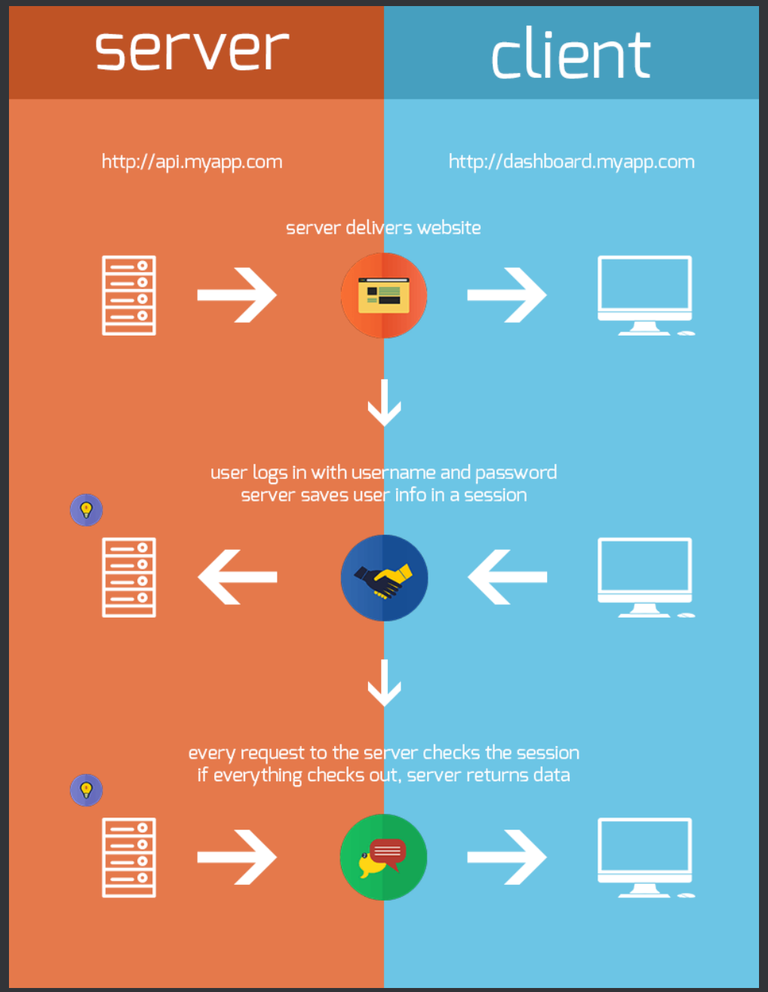
- See passport docs: Verify Callback section

```
import Strategy as LocalStrategy from'passport-local';


passport.use('local-signup', new LocalStrategy({
    usernameField: 'email',
    passwordField: 'password',
    passReqToCallback: true
}, processSignupCallback));   // <<-- more on this to come
```

- Since the HTTP protocol is stateless, this means that if we authenticate a user with a username and password, then on the next request, our application won’t know who we are. We would have to authenticate again.

Traditionally, we remember a user's signed-in status with **session variables**



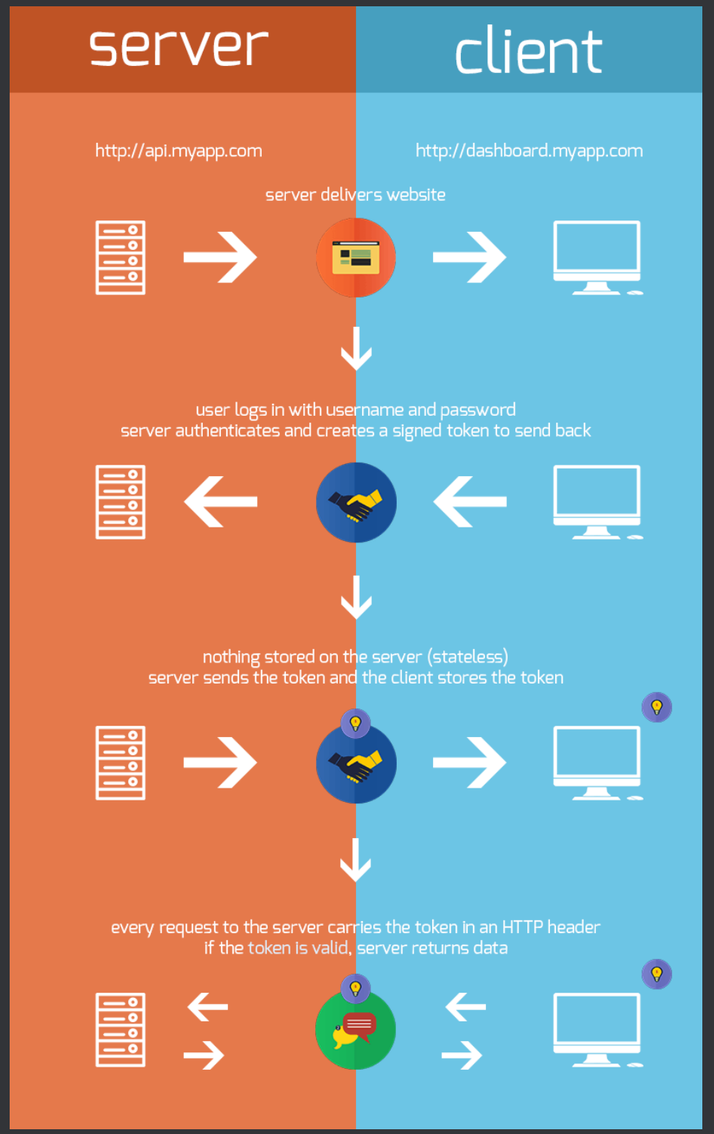
Server sessions

- require us to keep sending that user to the same server that they logged in at

Better strategy: Using Tokens (for greater scalability)

1. User Requests Access with Username / Password
2. Application validates credentials
3. Application provides a signed token to the client
4. Client stores that token and sends it along with every request
5. Server verifies token and responds with data

- Every single request will require the token. This token should be sent in the HTTP header
- token also expires after a set amount of time, so a user will be required to login once again


server
client
http://api.myapp.com
http://dashboard.myapp.com
server delivers website
user logs in with username and password
server authenticates and creates a signed token to send back
nothing stored on the server (stateless)
server sends the token and the client stores the token
every request to the server carries the token in an HTTP header
if the token is valid, server returns data

3rd-party apps

- how we provide selective permissions to third-party applications. We could even build our own API and hand out special permission tokens if our users wanted to give access to their data to another application

bcrypt

bcrypt is a password hashing functio

jsonwebtoken

https://www.npmjs.com/package/jsonwebtoken

Return json web token as a string, that encodes (is "signed with") the user credentials and expires, as well as a secret string (secret key) from the server

- JWT have all the claims in itself and is signed by the server as well
- server storage not required; client storage is required (stateless --> more scalable!)

passport-jwt

Verifies the jwt is valid, and when to validate the jwt (on what routes?)

jwt: authenticated routes

On every request to a restricted resource, the client sends the access token in the query string or Authorization header. The server then validates the token and, if it’s valid, returns the secure resource to the client.