Authentication

Authentication in the context of a user accessing an application tells an application who the current user is and whether or not they're present. A full authentication protocol will probably also tell you a number of attributes about this user, such as a unique identifier, an email address, and what to call them when the application says "Good Morning". Authentication is all about the user and their presence with the application, and an internet-scale authentication protocol needs to be able to do this across network and security boundaries.

The most basic one… Username and Password

Local strategy that accepts Username and Password to authenticate a user was one of the initial strategies people used in their applications.

- Takes in a username/password, then check the database to see if those credentials are valid or not.
- If valid, let user perform few set of operations
- Not very secure

Bearer Strategy

- Bearer tokens are typically used to protect API endpoints, and are often issued using OAuth 2.0.
- These are simple encrypted user information which can be decrypted by the server to figure out what user is requesting any resource

OAuth2.0

- With the rise of social networking, single sign-on using an OAuth provider such as Facebook or Twitter has become a popular authentication method.
- The OAuth 2.0 authorization framework enables a third-party application to obtain limited access to an HTTP service, either on behalf of a resource owner by orchestrating an approval interaction between the resource owner and the HTTP service, or by allowing the third-party application to obtain access on its own behalf.
- OAuth defines four roles:
  - Resource Owner
  - Client
  - Resource Server
  - Authorization Server

OAuth2.0

- A **`user`** calls an API from his `application server`, for ex. `/auth/google` to login via google
- The app server now calls the provider(`google`) and asks for an authentication handshake
- It also redirects the client browser to point to the providers auth screen (*i.e. select account to login, google's page*)
- When the user allows or denies the authentication request, the provider (`google` *in this case*) calls a predefined hook(*An API residing on our application server*) with some details about the user's authentication status.
- The hook receives the data about the user and can detect with own business logic whether to login or signup the said user.

Passport Module

- Passport is authentication middleware for Node.
- For basic Username and passport authentication : https://github.com/jaredhanson/passport-local
- For HTTP bearer strategy : https://github.com/jaredhanson/passport-http-bearer
- For different providers that use OAuth :
  - Facebook : https://github.com/jaredhanson/passport-facebook
  - Google : https://github.com/jaredhanson/passport-google-oauth
  - Twitter : https://github.com/jaredhanson/passport-twitter
- `npm install passport --save`
- `npm install passport-<strategy-name> --save` (can be either LocalStrategy, GoogleStrategy etc.)

Useful links:

- https://www.digitalocean.com/community/tutorials/an-introduction-to-oauth-2
- http://passportjs.org/

Exercises:

1). Create a signup API and form(using ejs template) for user to signup [Mandatory data: name, email, password]. Use mongodb to store user information like name, email and password(encrypt the password using bcrypt[https://www.npmjs.com/package/bcrypt]).

2). Update the passport local authentication strategy taught in the session to use the user information stored in mongodb.

3). Setup dummy facebook app(https://developers.facebook.com/) and use it along with passport-facebook(http://www.passportjs.org/docs/facebook/) to login. To successfully use facebook login for testing purpose, the dummy app has to be in development mode.

Thank You!