Web Development and API Design

Lesson 08:
RESTful APIs Practice

Prof. Andrea Arcuri

Goals

- Learn how to build a REST web service using NodeJS
- Learn how to write tests for a REST web service

Express

- In NodeJS, need to run an HTTP server
- **Express** is the most used/famous
  - but not necessarily the best
- Going to write "*HTTP Handlers*"
  - functions that are executed each time there is an incoming HTTP request for a given URI

Request/Response Objects

- We are not going to manipulate the HTTP messages directly
- Handler will take as input a JS object representing the *request*
  - it is *Express* that will create such object based on incoming HTTP
- Take as input as well a *response* object
  - we can modify it in the handler function
  - once handler function is completed, *Express* will create a HTTP response based on such response object

Static Assets

- As any HTTP server, can instruct *Express* to serve static assets
  - eg, all files under "*public*" folder
  - going to be a HTTP handler, like the others
- Can still use *WebPack* to create *bundle.js*, and put it under "*public*"

Application Entry Point

- Need one entry point JS file that starts *Express*
- We will use "*node*" command to run it
  - recall that JS is not compiled
- Eg in *package.json*, **"start": "node src/server/server.js"**
- But need to remember to build the *bundle.js* first, ie, **"build": "webpack --mode production"**

Development Mode

- During development, it is annoying to rebuild *bundle.js* and restart server at each code change
- *Hot Reload*: automatically detect if any change in the source code, and update server automatically

```
"dev": "concurrently \"yarn watch:client\" \"yarn watch:server\"",
"watch:client": "webpack --watch --mode development",
"watch:server": "nodemon src/server/server.js --watch src/server --watch public/bundle.js"
```

- In “*dev*”, run 2 processes in parallel, using the “*concurrently*” command
- 1) Run *WebPack* in “*watch*” mode, which rebuilds the *bundle.js* at each source code change
- 2) Run “*nodemon*”, which is equivalent to “*node*”, but can automatically restart if it detects any change in the files/folders specified with “*--watch*”

Testing a REST API

- Can of course write *Unit Tests*
- But also good to write "*System Tests*"
- In case of REST, we start the HTTP server, and from test cases, execute HTTP calls over TCP
  - and then write assertions on the returned HTTP responses

HTTP Library

- Going to use *SuperTest* library to make HTTP calls from the tests
  - also used to start server on an ephemeral port
- Challenge: server and tests are running on the **_same_** thread in *NodeJS*, so need to make proper use of *async/await*

Testing Frontend

- When writing tests for *React* components, they will fail when executing "*fetch()*"
- *fetch()* is a function in the browser, does not exist in *NodeJS*
- Can "*stub*" it away (similarly as we did with *alert()*)
  - ie, in the tests, create a custom function "*fetch()*" registered in the global scope
- Advanced option: start the server, and, in the stubbed *fetch()*, do call *SuperTest* to connect with the server