Agenda

- Exercise 2
- Tooling (npm, Docker)

Exercise 2

In the second exercise, you will develop a simple CRUD application with a relational database back-end that implements Blockstarter 1.0.

Your task is to build an application back-end that fulfills the requirements of implementing an API which can be sucessfully tested with the Postman collection in the *api_tests* subdirectory.

You only need to implement part of the data model and logic from the user stories for this exercise.

Blockstarter 1.0 API

Projects

- `POST /projects` creates a new Project
- `GET /projects/:id` retrieves one Project
- `GET /projects` retrieves all Projects
- `PUT /projects` updates one Project
- `DELETE /projects` deletes one Project

Create a similar CRUD API for **Creators** and **Backers**.

Backers

- `PUT /backers/addbalance` to add an amount X to the Backer's account balance

Exercise 2: Setup

Please use the following development setup (fragments of which are already given to you):

- A Dockerized Node.js application that implements the desired business logic and uses the express framework for implementing an API
- A Dockerized relational database (given is Postgresql, but you may exchange it with another relational database of your choice)

The multi-container application setup should be instantly launched and must be fully operational by simply running a `docker-compose up` command.

Tooling

- Build automation with npm
- Code generators
- Build automation with Docker

Build automation with npm

The most popular package managers in the JavaScript universe are bower and npm.

- Front-end JavaScript developers traditionally use bower.
- Node.js (back-end) developers use npm.

However, there is a trend to use npm as a universal package management tool for front- and back-end development, and even as a replacement of more complex scripting and task automation tools.

package.json

You can turn your directory into a node package/project by simply adding a *package.json* file.

With a *package.json* file, you can document the dependencies of your project and make the build and other tasks reproducible for other developers.

Generate an initial *package.json* file: `$ npm init -y`

```
{
 "name": "test-app",
 "version": "1.0.0",
 "description": "",
 "main": "index.js",
 ...
}
```

Install dependencies (1/2)

You can add node modules as dependencies to your package/project via the npm command line with `npm install` (or short `npm i`). For example, install the "commander" module like this:

```
npm i commander
```

Output:

```
test-app@1.0.0 /path/to/test-app
└─┬ commander@2.9.0
  └── graceful-readlink@1.0.1
```

Install dependencies (2/2)

You can add the `--save` parameter to your installation command, thereby automatically adding the dependency property to your *package.json* file.

```
{
 ... as before ...
 "dependencies": {
   "commander": "^2.9.0"
 }
}
```

For distinguishing between node modules that you use in development vs. production, you can add the parameter `--save-dev` instead of `--save` to your command.

npm scripts

The *package.json* "scripts" property enables you to define custom npm scripts, i.e., commands that can be executed via `npm run`. For example, add the following "foo" dummy property to your *package.json* file, which prints the string "bar" when you execute it:

```
{
 ... other properties ...
  "scripts": {
       "foo": "echo \"bar\""
   },
 ... other properties ...
}
```

Run the "foo" script as follows: `npm run foo`.

More *package.json* options

Find more *package.json* options for automating build processes, such as

- how to add executables to your PATH,
- use of configurable environment parameters,
- customize your directory structure, and more

at https://docs.npmjs.com/files/package.json

Code generators

For bootstrapping your application, you can use code generators, such as

- yeoman
- express-generator

If you use a code generator, please make sure that you remove unnecessary code and configuration artifacts from your exercise code submission.

Docker setup

Install Docker on your laptop (recommended: install the Docker Toolbox which includes useful tools like Docker Compose and Kitematic)
- https://docs.docker.com/toolbox/overview/

Build automation with Docker

You can build new Docker images using a *Dockerfile* that contains instructions for building the image. You can build upon existing base images and extend them with additional read-only file system layers. Please refer to the Dockerfile reference for an overview of all currently supported Dockerfile instructions.

To build your Docker image, open a shell at the location of your Dockerfile and run `docker build`.

Dockerize a Node.js app

Let's build an image from a Dockerfile for a simple node express app.

First, we create an *index.js* file with the express dependency inside an *app/src* subdirectory. The app listens on port 8080 and replies "Hello world" to GET requests at "/".

app/src/index.js

```javascript
// app/index.js
'use strict';
const express = require('express');
// Constants
const PORT = 8080;
// App
const app = express();
app.get('/', function(req, res) {
   res.send('Hello world\n');
});
app.listen(PORT);
console.log('Running on http://localhost:' + PORT);
```

app/package.json

The following *package.json* file in the *app* subdirectory specifies the start script and our only dependency, the express framework.

```json
{
    "name": "docker_web_app",
    "version": "1.0.0",
    "description": "Node.js on Docker",
    "author": "First Last <first.last@example.com>",
    "main": "server.js",
    "scripts": {
        "start": "node index.js"
    },
    "dependencies": {
        "express": "^4.13.3"
    }
}
```

Dockerfile

Next, we add a *Dockerfile* to the directory on the same level as the *app* subdirectory:

```
FROM node:7.1-slim

# Create app directory
RUN mkdir -p /myapp/src
WORKDIR /myapp

# Install app dependencies
ADD package.json /myapp/
RUN npm install

# Add app source
ADD src /myapp/src

EXPOSE 8080
CMD [ "npm", "start" ]
```

Dockerfile instructions

- The first line (`FROM`) specifies the base image which our own image is based on, i.e., "node:7.1-slim".
- Next, we create the directory */myapp/src* and make */myapp* our current working directory.
- Then we `ADD` the *package.json* file into the /myapp directory and install the dependencies by executing `npm` (which is pre-installed via the node:7.1-slim base image) via a `RUN` instruction.
- Next, the source code (in this case, only the *index.js* file is copied into the container image)
- Furthermore, we `EXPOSE` port 8080 which is the port that our node app binds to (see *index.js*).
- Finally, a command is defined via `CMD` for starting the node app when the container instance is started.

Docker compose

Docker compose allows us to easily launch and shut down multi-container applications. For example:

- Create a *docker-compose.yml* file and open your shell at that directory.
- Run the command `docker-compose up --build`

Alternatively, you can bring the services up in the background using detached mode with `up`, see what's going on with `ps` and remove the containers entirely via `down --volumes`:

- `docker-compose up -d --build`
- `docker-compose ps`
- `docker-compose down --volumes`

Example *docker-compose.yml*

```yaml
version: '2'
services:
  node-app:
    build: app
    ports:
      - "4000:8080"
    volumes:
      - ./app/src:/myapp/src
    depends_on:
      - db
    environment:
      - PG_HOST=db
      - PG_PORT=5432
      - PG_USER=postgres
      - PG_DATABASE=app

  db:
    build: db
```