Microservices using Node.js

When your JavaScript application grows in size you start facing challenges with maintaining the code, fixing bugs, and implementing new features. Also, adding new developers to the project becomes complicated.

Applications are built from pieces, like packages and modules, but at some point those structures aren't enough to reduce the size and complexity of the application. The idea behind distributed systems is to break big, monolithic designs into small, independent programs which communicate with each other to exchange data and perform operations.

One of the many variants of distributed systems is the microservices architecture, which structures an application as a collection of loosely coupled services. Services are fine-grained and the communication protocols are lightweight (like the HTTP protocol).

There are few things worth emphasizing about the superiority of microservices, and distributed systems generally, over monolithic architecture:

- Modularity – responsibility for specific operations is assigned to separate pieces of the application
- Uniformity – microservices interfaces (API endpoints) consist of a base URI identifying a data object and standard HTTP methods (GET, POST, PUT, PATCH and DELETE) used to manipulate the object
- Robustness – component failures cause only the absence or reduction of a specific unit of functionality
- Maintainability – system components can be modified and deployed independently
- Scalability – instances of a service can be added or removed to respond to changes in demand.
- Availability – new features can be added to the system while maintaining 100% availability.
- Testability – new solutions can be tested directly in the "battlefield of production" by implementing them for restricted segments of users to see how they behave in real life.

In addition, every microservice can be written using the language, technique, or framework that's most appropriate to the tasks it will perform. The only feature that is necessary is the ability to publish RESTful APIs for communication with other services.

In addition, every microservice can be written using the language, technique, or framework that's most appropriate to the tasks it will perform. The only feature that is necessary is the ability to publish RESTful APIs for communication with other services.

To create microservices with Node.js you will need:

Node.js and npm

Create the heroes service

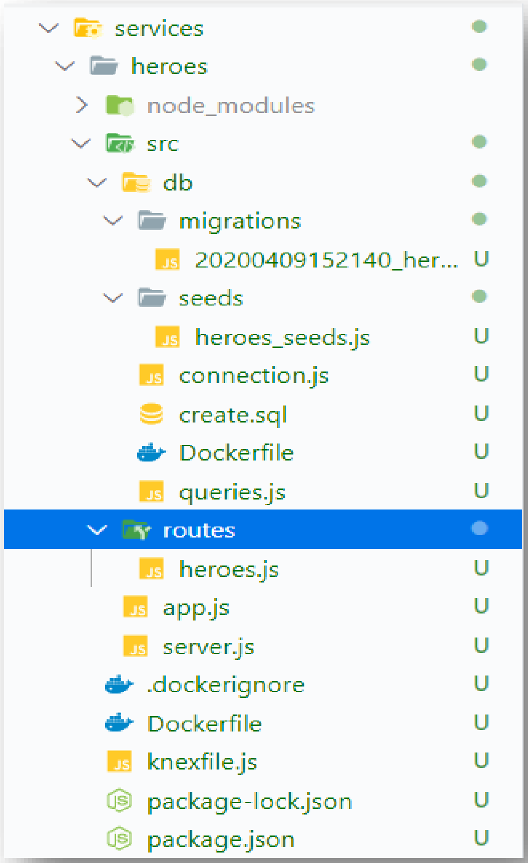
In the root directory create folder services. In folder services create folder heroes where we will be creatong the heroes service.

First step is to run **npm init –y** – initialize package.json file.

Next step is to run **npm install express body-parser** – install express and body parser.

The structure of heroes service:

- node_modules and package.json
- Dockerfile
- .dockerignore
- knexfile.js
- src directory



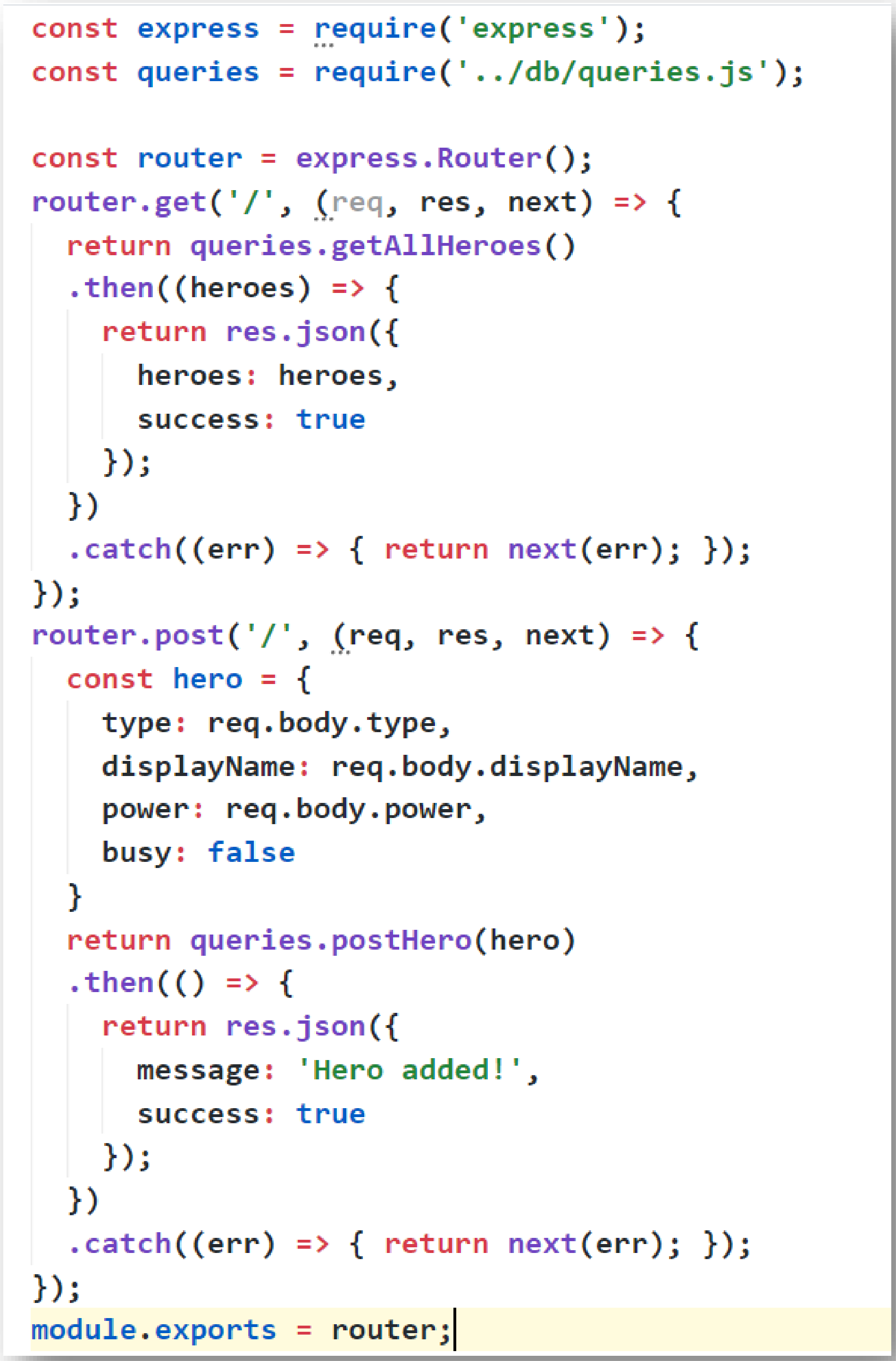
```
services                              ●
  heroes                              ●
    node_modules
    src                               ●
      db                              ●
        migrations                    ●
          20200409152140_her...       U
        seeds                         ●
          heroes_seeds.js             U
        connection.js                 U
        create.sql                    U
        Dockerfile                    U
        queries.js                    U
      routes                          ●
        heroes.js                     U
      app.js                          U
      server.js                       U
    .dockerignore                     U
    Dockerfile                        U
    knexfile.js                       U
    package-lock.json                 U
    package.json                      U
```

Create src directory

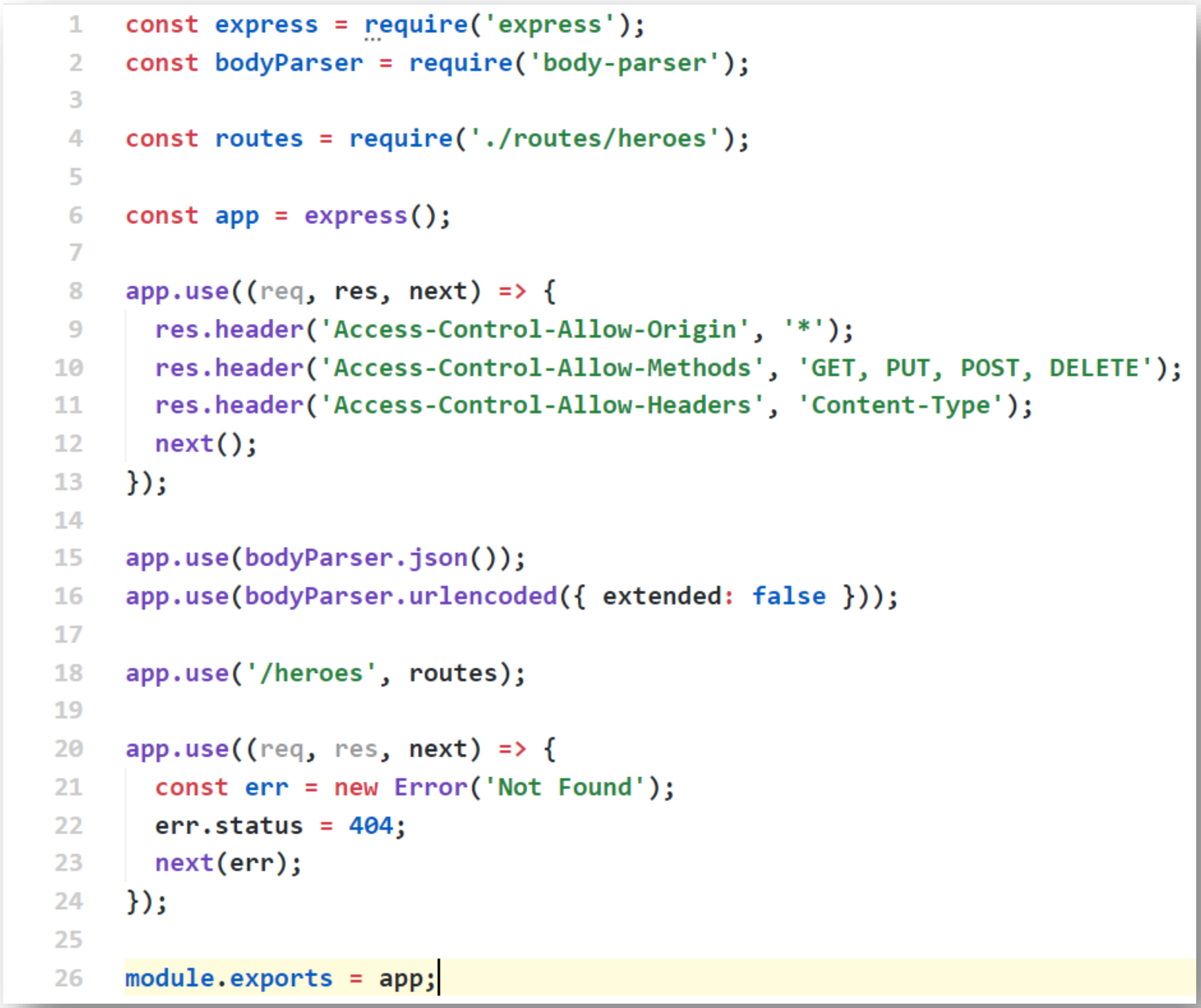
Inside src place folder db and routes and also files app.js and server.js.

Create heroes.js in routes, where will be stored all the routes for heroes.

In app.js set that those routes are at /heroes.

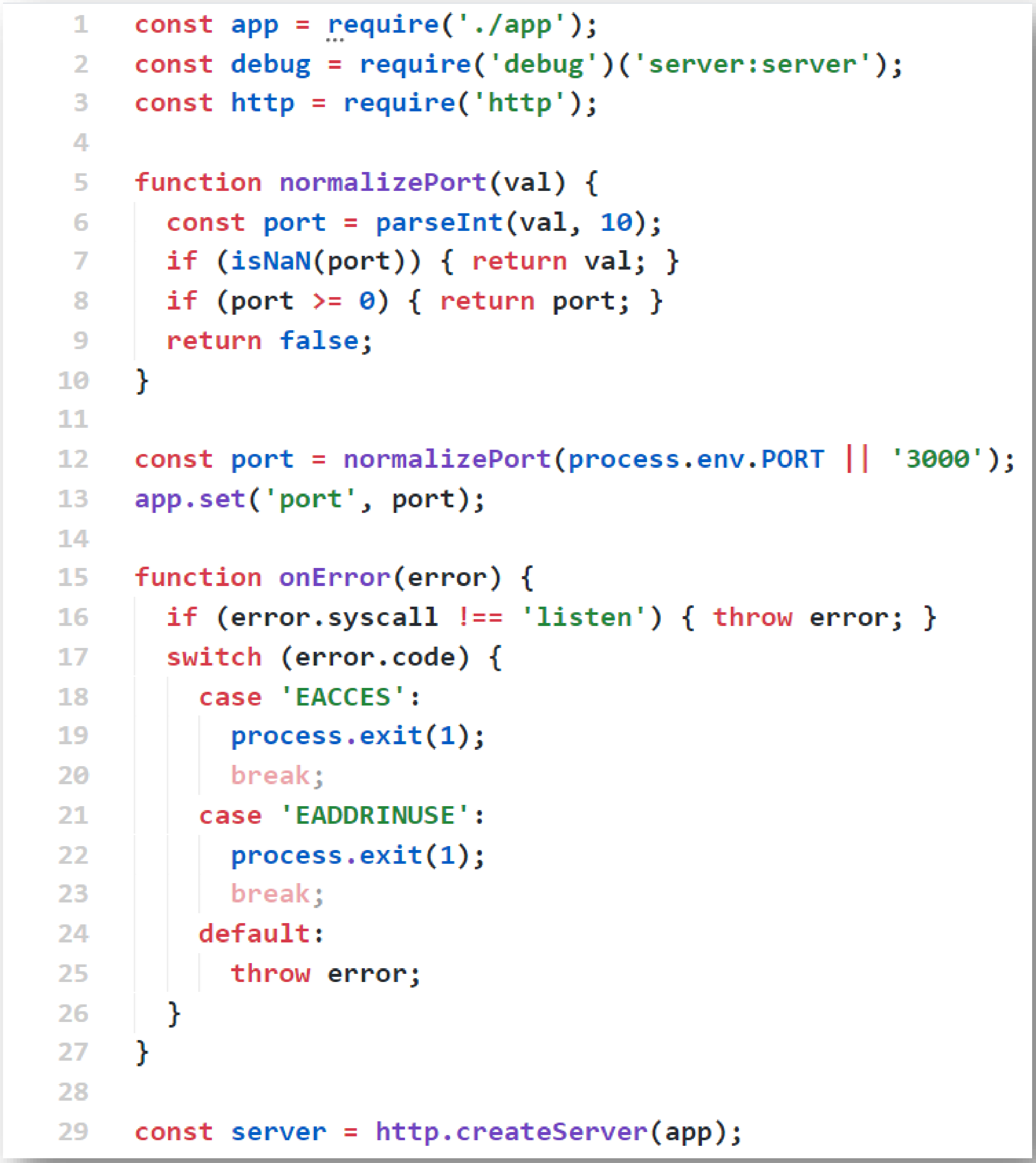


```
const express = require('express');
const queries = require('../db/queries.js');

const router = express.Router();
router.get('/', (req, res, next) => {
  return queries.getAllHeroes()
  .then((heroes) => {
    return res.json({
      heroes: heroes,
      success: true
    });
  })
  .catch((err) => { return next(err); });
});
router.post('/', (req, res, next) => {
  const hero = {
    type: req.body.type,
    displayName: req.body.displayName,
    power: req.body.power,
    busy: false
  }
  return queries.postHero(hero)
  .then(() => {
    return res.json({
      message: 'Hero added!',
      success: true
    });
  })
  .catch((err) => { return next(err); });
});
module.exports = router;
```

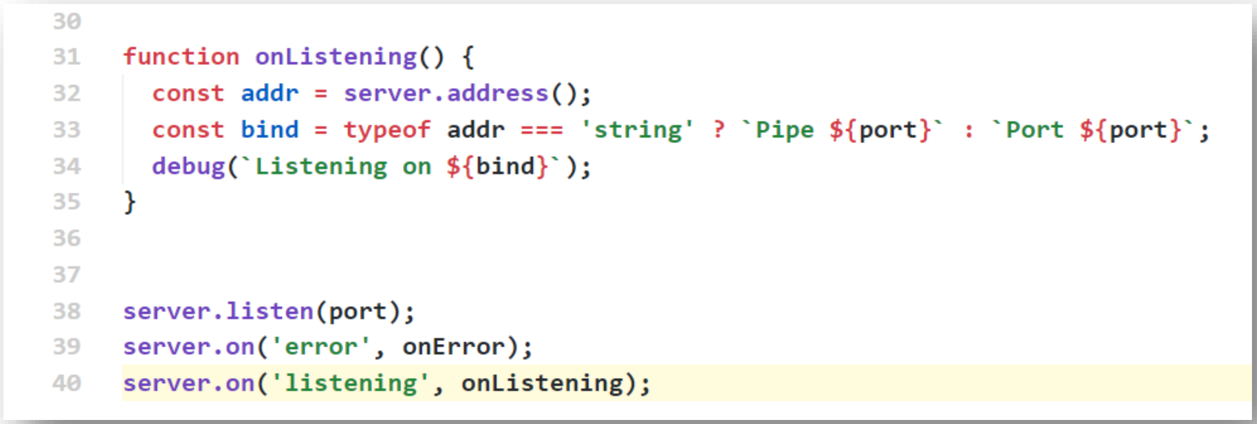


```
const express = require('express');
const bodyParser = require('body-parser');

const routes = require('./routes/heroes');

const app = express();

app.use((req, res, next) => {
  res.header('Access-Control-Allow-Origin', '*');
  res.header('Access-Control-Allow-Methods', 'GET, PUT, POST, DELETE');
  res.header('Access-Control-Allow-Headers', 'Content-Type');
  next();
});

app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));

app.use('/heroes', routes);

app.use((req, res, next) => {
  const err = new Error('Not Found');
  err.status = 404;
  next(err);
});

module.exports = app;
```

Server.js



```javascript
const app = require('./app');
const debug = require('debug')('server:server');
const http = require('http');

function normalizePort(val) {
  const port = parseInt(val, 10);
  if (isNaN(port)) { return val; }
  if (port >= 0) { return port; }
  return false;
}

const port = normalizePort(process.env.PORT || '3000');
app.set('port', port);

function onError(error) {
  if (error.syscall !== 'listen') { throw error; }
  switch (error.code) {
    case 'EACCES':
      process.exit(1);
      break;
    case 'EADDRINUSE':
      process.exit(1);
      break;
    default:
      throw error;
  }
}

const server = http.createServer(app);
```

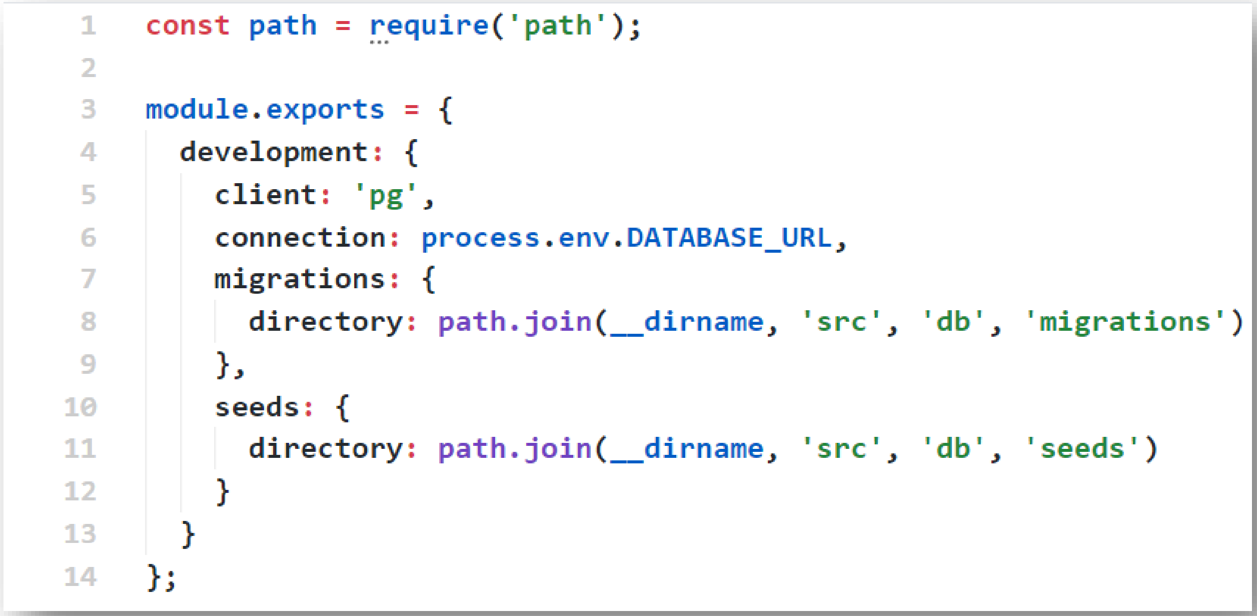


```
function onListening() {
  const addr = server.address();
  const bind = typeof addr === 'string' ? `Pipe ${port}` : `Port ${port}`;
  debug(`Listening on ${bind}`);
}


server.listen(port);
server.on('error', onError);
server.on('listening', onListening);
```

Add database - PostgreSQL

To connect with PostgreSQL *knex* is used. Knex.js is a "batteries included" SQL query builder for Postgres, MSSQL, MySQL, MariaDB, SQLite3, Oracle, and Amazon Redshift designed to be flexible, portable, and fun to use. Knex needs to be installed using command **npm install knex --save**.

In the src directory there is **knexfile.js** as config for knex.



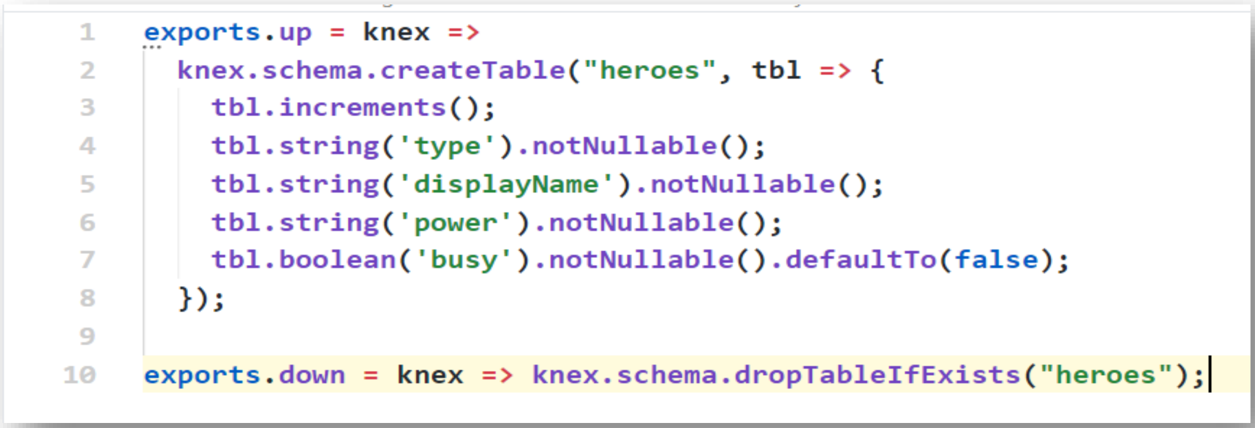
```
const path = require('path');

module.exports = {
  development: {
    client: 'pg',
    connection: process.env.DATABASE_URL,
    migrations: {
      directory: path.join(__dirname, 'src', 'db', 'migrations')
    },
    seeds: {
      directory: path.join(__dirname, 'src', 'db', 'seeds')
    }
  }
};
```

It says that the client is pg (node.js module for postgres), and sets location for migrations and seeds. Migrations are used to create tables for the database and seeds are used to initialize database.

Templates for migrations and seeds can be created using **knex migrate:make table_name** and **knex seed:make seeder_name**. Next, we can update the placeholder data to create our own table or seeder.

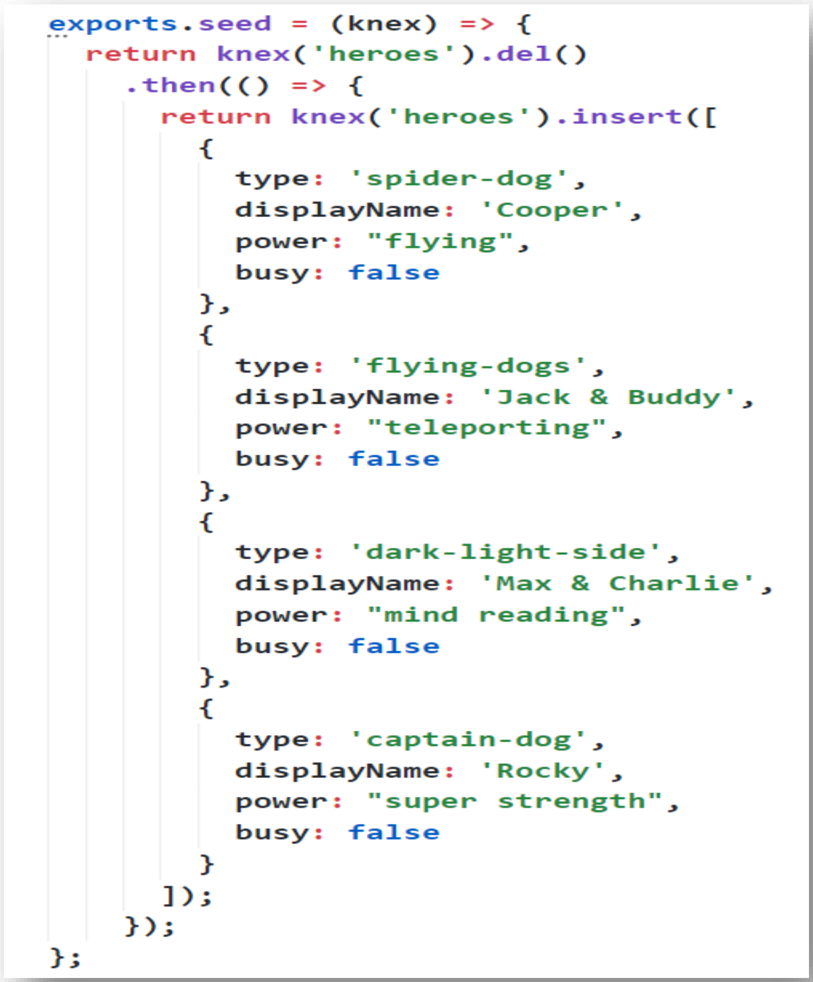
The db folder contains a folder for migrations, a folder for seeders and the files connection.js, queries.js, create.sql and Dockerfile.

Migrations – heroes.js



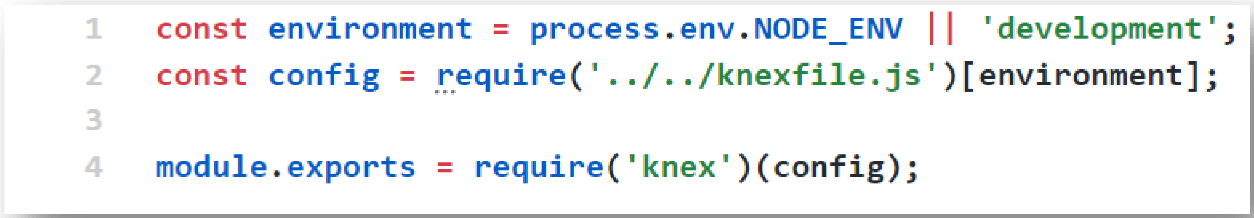
```
exports.up = knex =>
  knex.schema.createTable("heroes", tbl => {
    tbl.increments();
    tbl.string('type').notNullable();
    tbl.string('displayName').notNullable();
    tbl.string('power').notNullable();
    tbl.boolean('busy').notNullable().defaultTo(false);
  });

exports.down = knex => knex.schema.dropTableIfExists("heroes");
```

Seeders – heroes_seeds.js



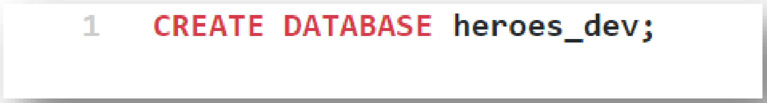
```
exports.seed = (knex) => {
  return knex('heroes').del()
    .then(() => {
      return knex('heroes').insert([
        {
          type: 'spider-dog',
          displayName: 'Cooper',
          power: "flying",
          busy: false
        },
        {
          type: 'flying-dogs',
          displayName: 'Jack & Buddy',
          power: "teleporting",
          busy: false
        },
        {
          type: 'dark-light-side',
          displayName: 'Max & Charlie',
          power: "mind reading",
          busy: false
        },
        {
          type: 'captain-dog',
          displayName: 'Rocky',
          power: "super strength",
          busy: false
        }
      ]);
    });
};
```

Connection.js – using knexfile to configure knex with appropriate database url



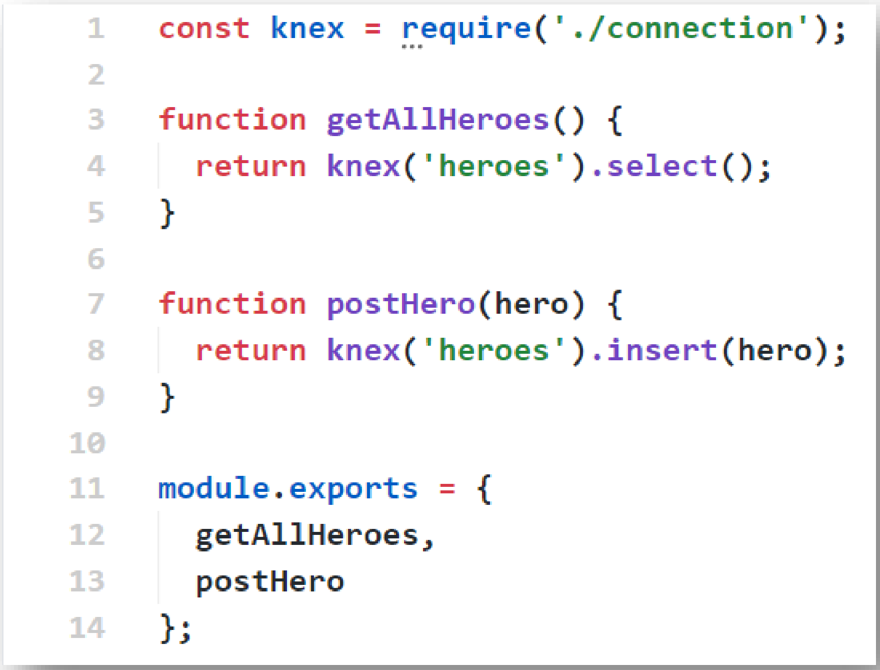
```
const environment = process.env.NODE_ENV || 'development';
const config = require('../../knexfile.js')[environment];

module.exports = require('knex')(config);
```

Create.sql – sql instruction to create database heroes_dev



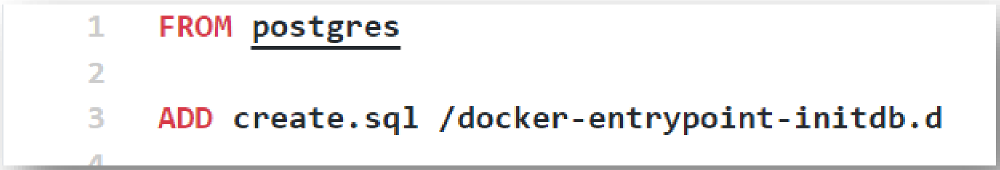
```sql
CREATE DATABASE heroes_dev;
```

Queries.js – list of all queries we can do on our database (used in heroes routes)



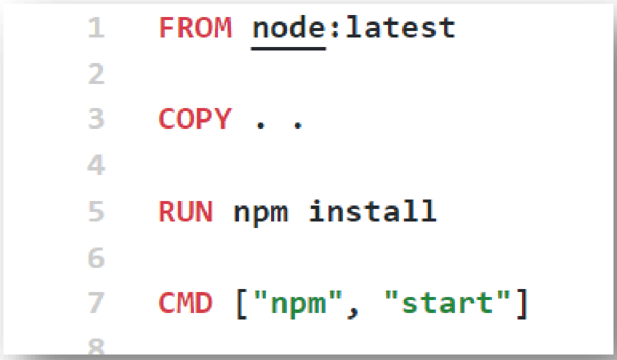
```javascript
const knex = require('./connection');

function getAllHeroes() {
  return knex('heroes').select();
}

function postHero(hero) {
  return knex('heroes').insert(hero);
}

module.exports = {
  getAllHeroes,
  postHero
};
```

Dockerfile for database – base image is postgres and db init instructions are in create.sql file



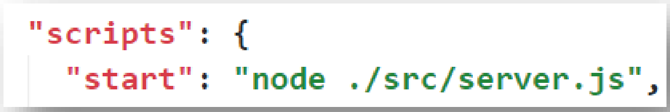
```dockerfile
FROM postgres

ADD create.sql /docker-entrypoint-initdb.d
```

Dockerfile for entire microservice – base image is node:latest, we first copy our entire src directory, run npm install to install necessary dependecies and then set instruction for starting. If we want to use npm start, package.json must contain start script.



```
FROM node:latest

COPY . .

RUN npm install

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

Package.json – on npm start run server.js file with node – our server starts listening on port 3000.

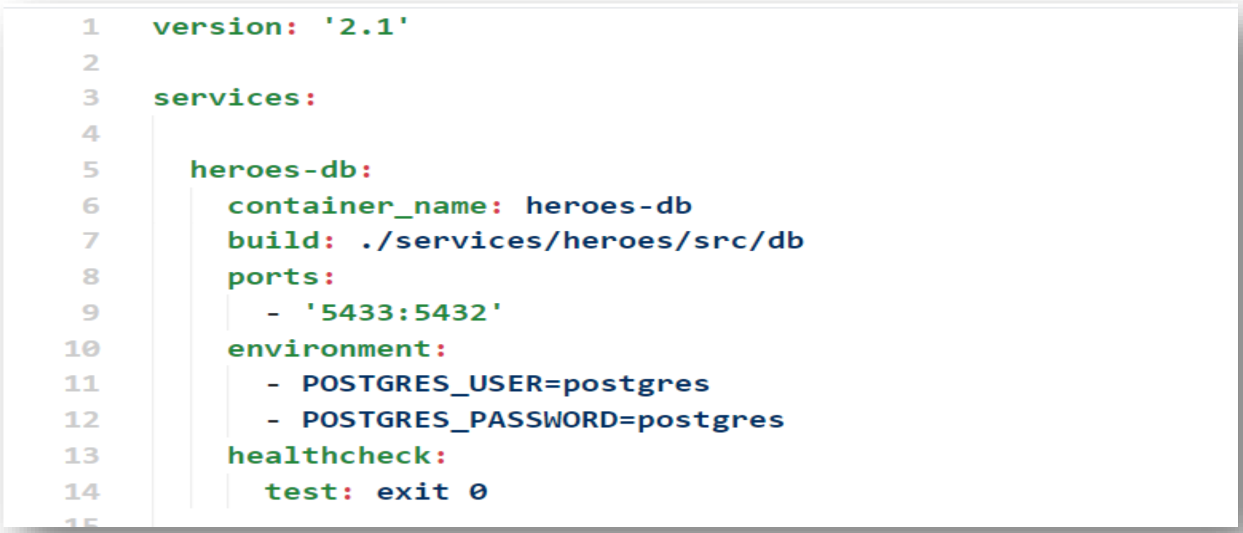


```
"scripts": {
  "start": "node ./src/server.js",
```

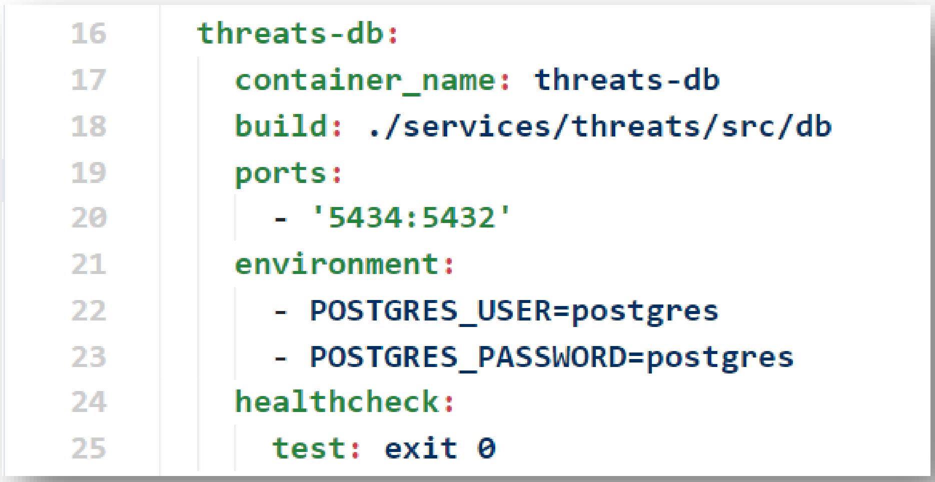
This project should contain two services, heroes and threats.

Threats service is created almost the same as heroes, and code can be found on: https://github.com/teodorislava/nodejs-microservices

We have Dockerfile for heroes database, threats database, heroes service and threats service. If we want to run this as multi-container app we need **docker-compose.yml** file. It is placed in the root of our project.



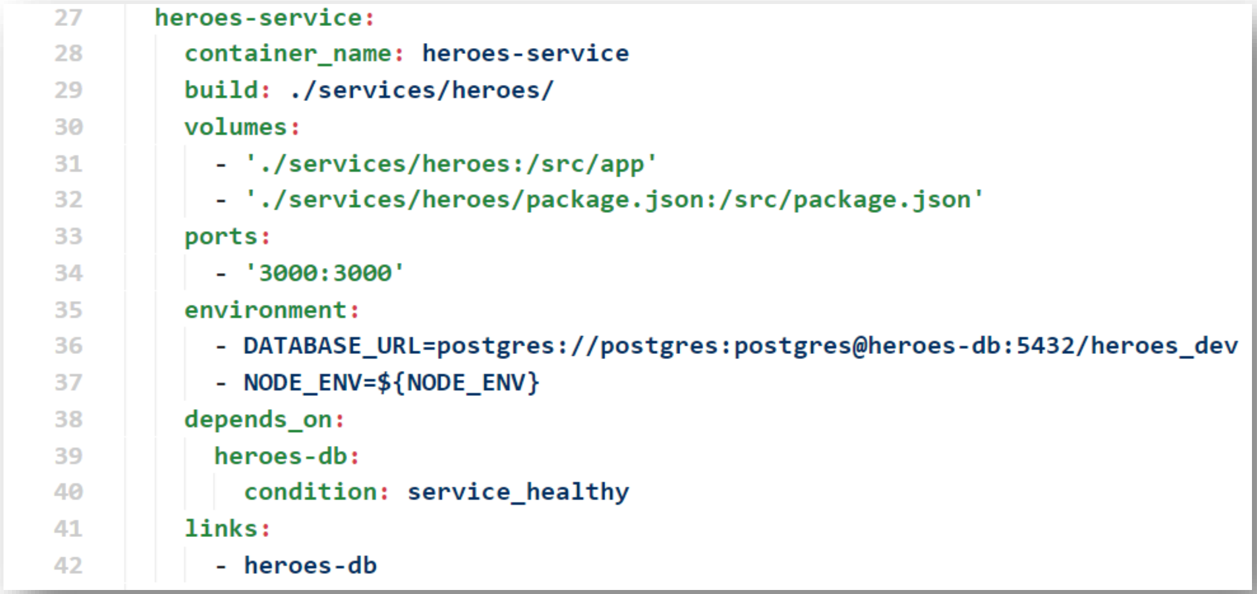
```
version: '2.1'

services:

  heroes-db:
    container_name: heroes-db
    build: ./services/heroes/src/db
    ports:
      - '5433:5432'
    environment:
      - POSTGRES_USER=postgres
      - POSTGRES_PASSWORD=postgres
    healthcheck:
      test: exit 0
```



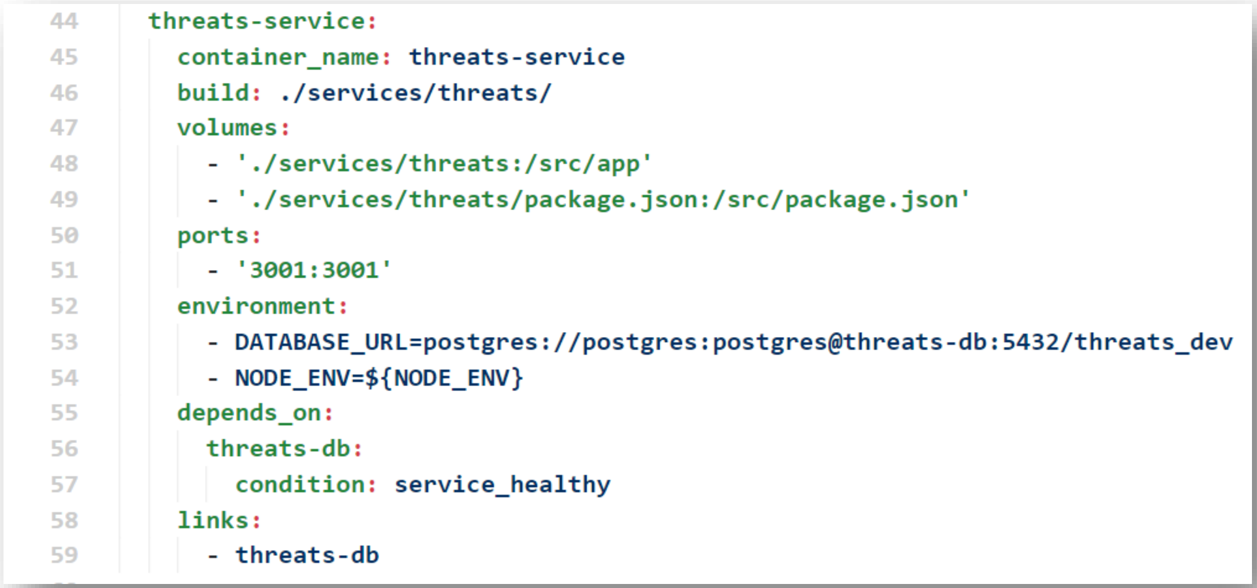
```
  threats-db:
    container_name: threats-db
    build: ./services/threats/src/db
    ports:
      - '5434:5432'
    environment:
      - POSTGRES_USER=postgres
      - POSTGRES_PASSWORD=postgres
    healthcheck:
      test: exit 0
```

We must set POSTGRES_USER and POSTGRES_PASSWORD.

Then in DATABASE_URL we can set the connection string to this database. Heroes service depends on the heroes database and the condition is that it should be healthy. This means that we need to start the heroes database container first, wait to see whether it will become healthy and then start heroes service.



```yaml
heroes-service:
  container_name: heroes-service
  build: ./services/heroes/
  volumes:
    - './services/heroes:/src/app'
    - './services/heroes/package.json:/src/package.json'
  ports:
    - '3000:3000'
  environment:
    - DATABASE_URL=postgres://postgres:postgres@heroes-db:5432/heroes_dev
    - NODE_ENV=${NODE_ENV}
  depends_on:
    heroes-db:
      condition: service_healthy
  links:
    - heroes-db
```



```yaml
threats-service:
  container_name: threats-service
  build: ./services/threats/
  volumes:
    - './services/threats:/src/app'
    - './services/threats/package.json:/src/package.json'
  ports:
    - '3001:3001'
  environment:
    - DATABASE_URL=postgres://postgres:postgres@threats-db:5432/threats_dev
    - NODE_ENV=${NODE_ENV}
  depends_on:
    threats-db:
      condition: service_healthy
  links:
    - threats-db
```

docker-compose up –build

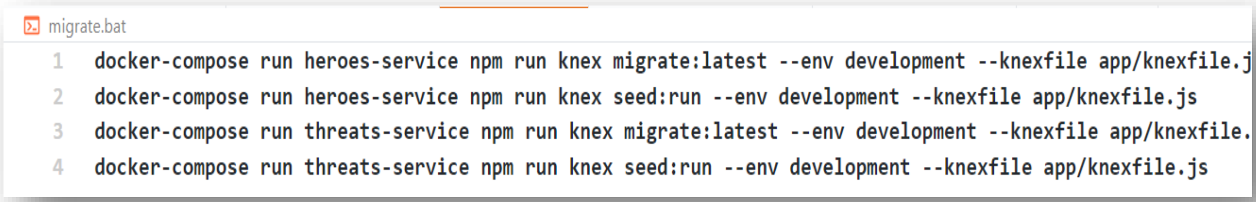
Docker-compose up - Builds, (re)creates, starts, and attaches to containers for a service.

--build creates images before starting containers.

-d option can start and run containers in the background.

When we start our containers, after that we need to start our migrations and seed our database. These set of commands can be placed in .bat or .sh file.

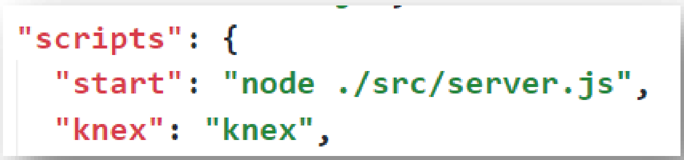
migrate.bat



migrate.bat

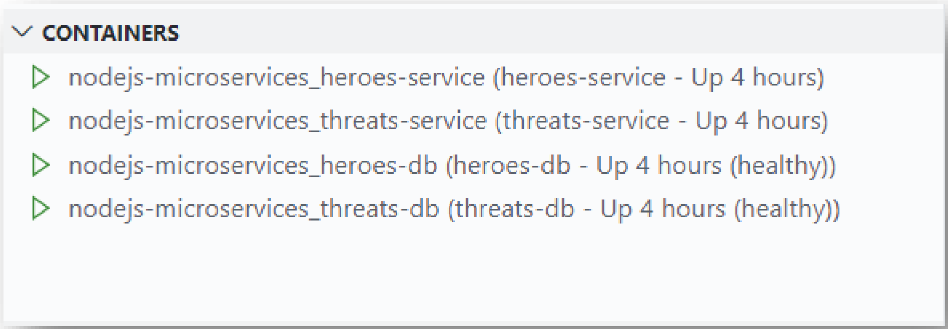
```
docker-compose run heroes-service npm run knex migrate:latest --env development --knexfile app/knexfile.j
docker-compose run heroes-service npm run knex seed:run --env development --knexfile app/knexfile.js
docker-compose run threats-service npm run knex migrate:latest --env development --knexfile app/knexfile.
docker-compose run threats-service npm run knex seed:run --env development --knexfile app/knexfile.js
```

In order to be able to run knex:migrate and knex:seeed instructions inside of the container, package.json should be updated.



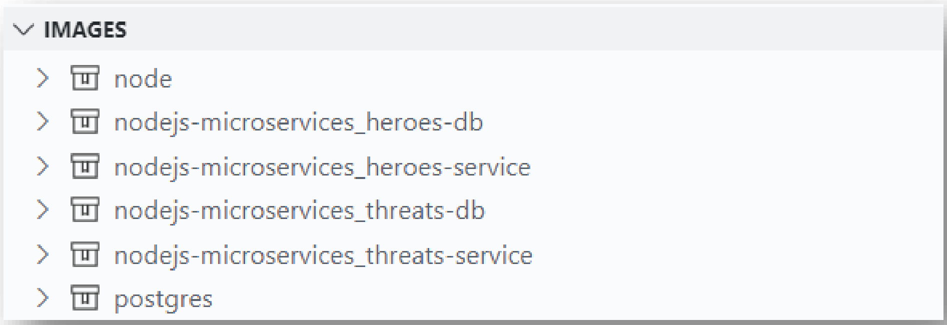
```
"scripts": {
  "start": "node ./src/server.js",
  "knex": "knex",
```

The results after docker-compose up --build



CONTAINERS

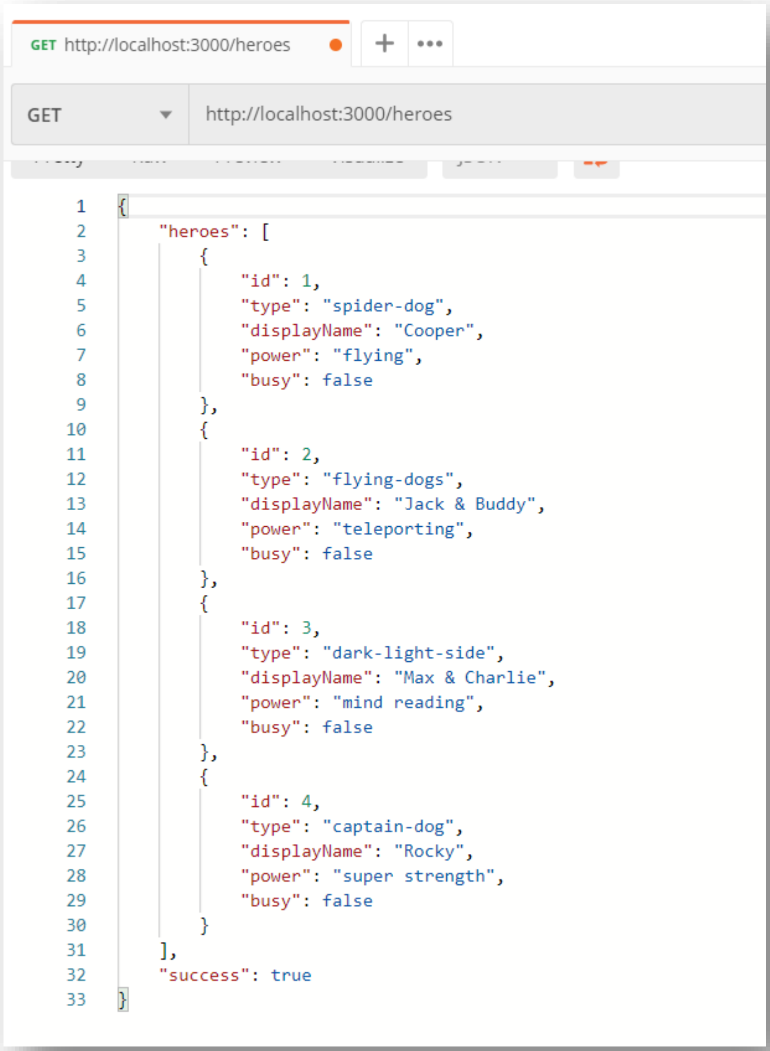
```
nodejs-microservices_heroes-service (heroes-service - Up 4 hours)
nodejs-microservices_threats-service (threats-service - Up 4 hours)
nodejs-microservices_heroes-db (heroes-db - Up 4 hours (healthy))
nodejs-microservices_threats-db (threats-db - Up 4 hours (healthy))
```



IMAGES

- node
- nodejs-microservices_heroes-db
- nodejs-microservices_heroes-service
- nodejs-microservices_threats-db
- nodejs-microservices_threats-service
- postgres

Testing heroes service on localhost:3000 using Postman - GET

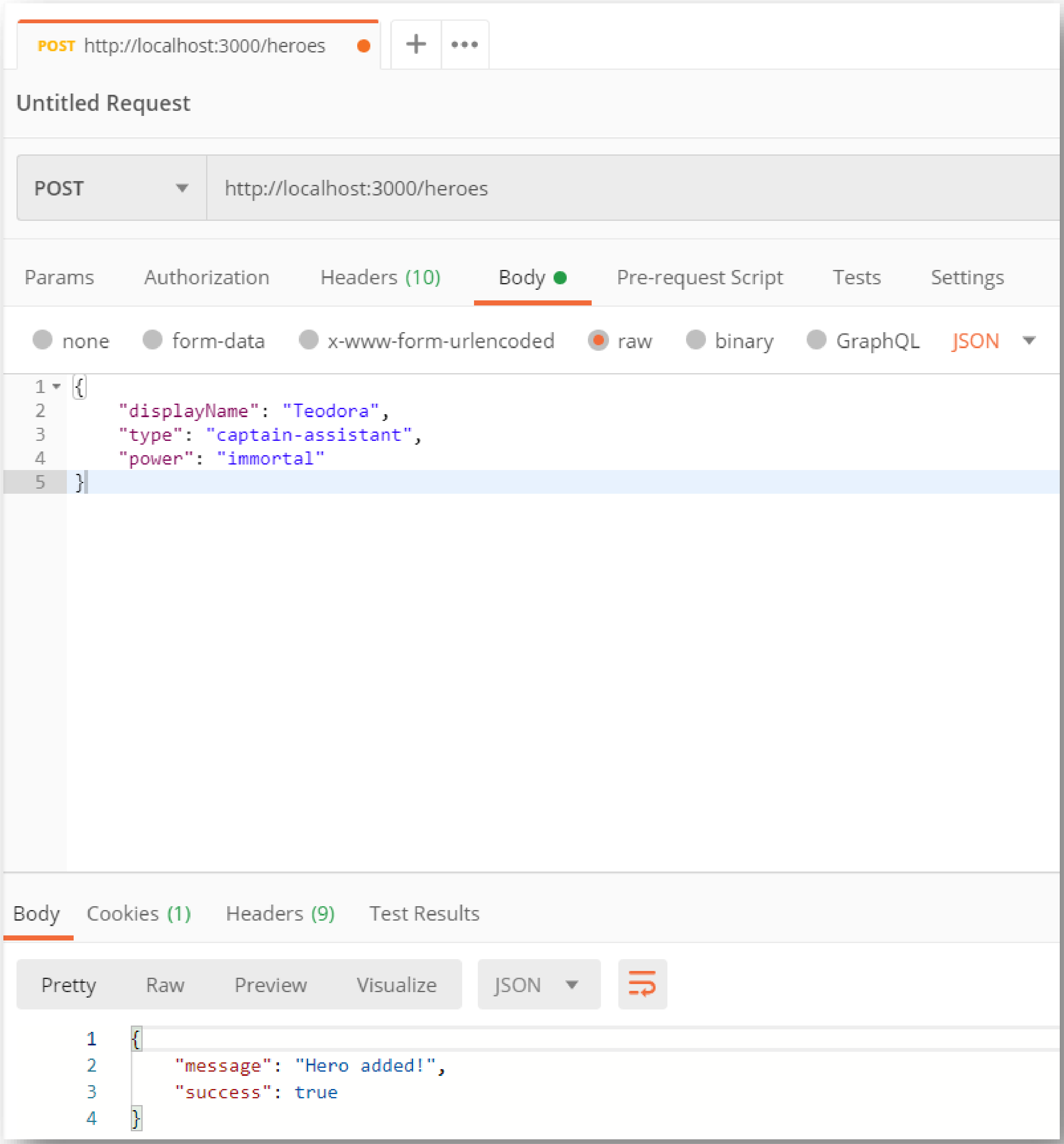


GET http://localhost:3000/heroes

```
GET    http://localhost:3000/heroes
```

```
{
    "heroes": [
        {
            "id": 1,
            "type": "spider-dog",
            "displayName": "Cooper",
            "power": "flying",
            "busy": false
        },
        {
            "id": 2,
            "type": "flying-dogs",
            "displayName": "Jack & Buddy",
            "power": "teleporting",
            "busy": false
        },
        {
            "id": 3,
            "type": "dark-light-side",
            "displayName": "Max & Charlie",
            "power": "mind reading",
            "busy": false
        },
        {
            "id": 4,
            "type": "captain-dog",
            "displayName": "Rocky",
            "power": "super strength",
            "busy": false
        }
    ],
    "success": true
}
```

Testing heroes service on localhost:3000 using Postman - POST



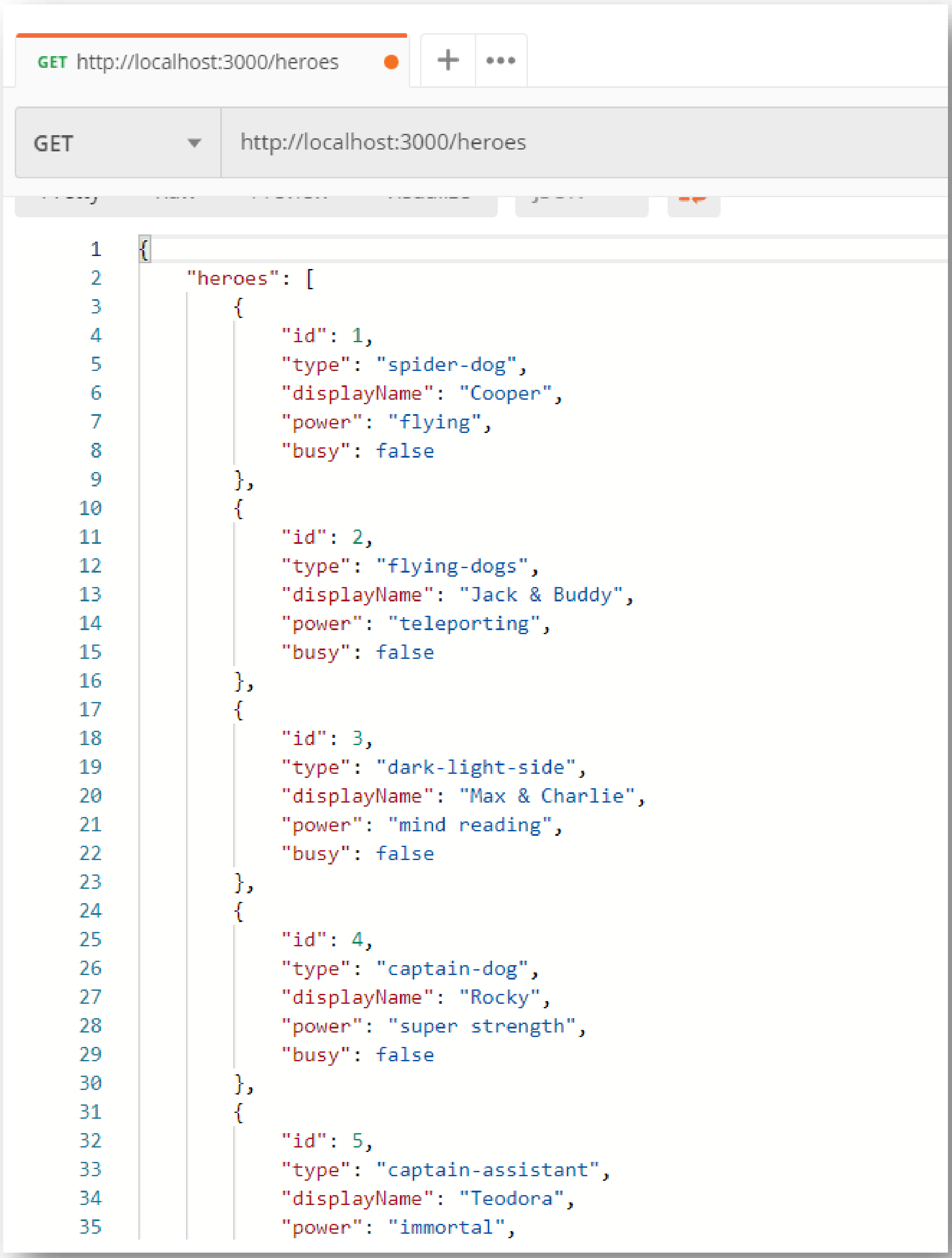
POST http://localhost:3000/heroes

```
{
    "displayName": "Teodora",
    "type": "captain-assistant",
    "power": "immortal"
}
```

Response Body:

```
{
    "message": "Hero added!",
    "success": true
}
```

Testing heroes service on localhost:3000 using Postman – GET after adding new hero


GET http://localhost:3000/heroes
GET
http://localhost:3000/heroes

```
{
    "heroes": [
        {
            "id": 1,
            "type": "spider-dog",
            "displayName": "Cooper",
            "power": "flying",
            "busy": false
        },
        {
            "id": 2,
            "type": "flying-dogs",
            "displayName": "Jack & Buddy",
            "power": "teleporting",
            "busy": false
        },
        {
            "id": 3,
            "type": "dark-light-side",
            "displayName": "Max & Charlie",
            "power": "mind reading",
            "busy": false
        },
        {
            "id": 4,
            "type": "captain-dog",
            "displayName": "Rocky",
            "power": "super strength",
            "busy": false
        },
        {
            "id": 5,
            "type": "captain-assistant",
            "displayName": "Teodora",
            "power": "immortal",
```