MedTech
MEDITERRANEAN
INSTITUTE OF TECHNOLOGY
SOUTH MEDITERRANEAN UNIVERSITY

CS 453: Cloud and pervasive computing
Session: Fall 2021
Professor: Amina Mseddi

Lab 5: Platform As A Service

Objective

- Deploy an application on a platform as a service infrastructure

Tools presentation

These are the main tools we will need and the appropriate installation links for each.

- Git is used for version control. It will provide us with a convenient command line interface to save different versions of our code and commit them to GitHub.
- GitHub is a code hosting platform that hosts a Remote Repository of our code. We can use Git to push our code to GitHub and share it with the world.
- Postgres is the database system we will be using for this project.
- Heroku CLI is the command line interface for Heroku. This is going to allow us to push our git code to Heroku, and there are other useful functions for dealing with Heroku in there.
- Visual Studio Code or you can use your text editor of choice.
- Node.js is a JavaScript runtime to run JavaScript code outside of the browser. This will also allow us to run npm commands to install dependencies into our project.

Exercise 1

Deploy a Node.js web application that interacts with PostgreSQL on Heroku. Use git for your deployment.

Steps:

1. Create a project Lab5 on VS Code
2. Create empty files on that directory :
    a. .env:
    b. `index.js`
3. run `npm init -y` to create a new package.json file as well. This is where we will track the metadata of our application and can add npm scripts to shorten our command line syntax later

4. Add start and server scripts to the package.json so we will be able to run the application

```
"scripts": {
  "start": "node index.js",
  "server": "nodemon index.js",
  "test": "echo \"Error: no test specified\" && exit 1"
},
```

The command **`npm install`** installs packages for node.js . Use this command to install:

- nodemon as a **dev dependency**. (requires -D option) nodemon is a tool that helps develop node.js based applications by automatically restarting the node application when file changes in the directory are detected.
- cors, dotenv, express, knex, and pg as normal dependencies.

Take note of the newly added dependencies section and devDependencies section in your *package.json* file. Also take note of the *package-lock.json* and the *node_modules* folder.

Setting up the Server

We will create a basic server to make sure everything is running up.

1. Within the .env file lets create some environment variables for the project.

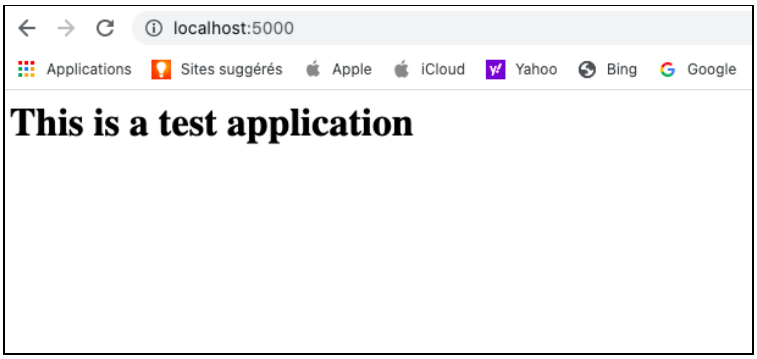
```
PORT=5000
NODE_ENV=development
```

2. Let's create the server with index.js file.

```
/* allows access to the 'environment variables' (.env file) */
require('dotenv').config()

/*  import express and cors [2] */
const express = require('express')
const cors = require('cors')

/*  create the server and define the port */
const server = express()
const port = process.env.PORT || 8000

/*  use cors and express.json (which a body-parser for incoming requests) */
server.use(cors())
server.use(express.json())

/* basic route to access the application in the browser */
server.get('/', (req, res) => {
   res.send('<h1>This is a test application</h1>')
})
```

```
/* Listen to the port in the variable and display a message */
server.listen(port, () => {
  console.log(`\n=== Server listening on port ${port} ===\n`)
})
```

- Run npm run server to start the server with nodemon.
- Now if you open your internet browser to *http://localhost:5000/* you should see this.



Uploading to GitHub

1. Create a new GitHub Repo
2. Run the command npx gitignore node.[1] This will create a .gitignore file that tells git what we don't want it to save in version control. This is very useful to store secrets like API keys.
3. Using the command line create a new repository for you project.

Q. Screenshot all the commands and your repository

Setting up the Dev Database

You should have ***Postgres*** and ***pgAdmin 4*** installed

1. Log in to PG Admin
2. Create a new server named *"YourName_svr"*. You should see a pop up in the middle of your screen with configuration options for your new database server. This might take some playing around with to get it working as it depends how you set up postgres on install.
3. Create a database "YourName_db" in your new created database server.
4. Update your environment variables in your .env file to match the information from your database configuration that you just went through.

Starting with knex

We need to start by running the command npx knex init. This command uses the knex library we installed earlier to create a default knexfile.js in the root of our project.

Modify the knexfile.js with this code

```
/* allow the code to access our .env file to get our environment variables. */
require('dotenv').config()

/* imports our pg library into the code */
const pg = require('pg');

/* This is standard and required by the production database. We will set it up later. */
if (process.env.DATABASE_URL) {
  pg.defaults.ssl = { rejectUnauthorized: false }
}

/* Set the client as pg. Set migrations and seeds directories. The sharedConfig variable is
shared between our development and production environments. */

const sharedConfig = {
  client: 'pg',
  migrations: {directory: './data/migrations'},
  seeds: {directory: './data/seeds'},
}

/* sets the connection configuration settings for the development and production
environment. */
module.exports = {
  development: {
    ...sharedConfig,
    connection: {
      host: process.env.DB_HOST,
      user: process.env.DB_USER,
      password: process.env.DB_PASS,
      database: process.env.DB_NAME
    }
  },
  production: {
    ...sharedConfig,
    connection: process.env.DATABASE_URL,
    pool: { min: 2, max: 10 },
  },
};
```

Run the commands to create new migration and seed files.

```
npx knex migrate:make first-migration
npx knex seed:make 001_first-seed
```

New files are created in the data folder.

We modify the migration file to add a new table to our database, giving an auto incrementing profile_id column as well as a name column.

```
exports.up = (knex) => {
   return knex.schema
      .createTable('profiles', function (table) {
         table.increments('profile_id');
         table.string('name', 128).notNullable();
      });
   };

exports.down = (knex) => {
   return knex.schema.dropTableIfExists('profiles');
};
```

We can also add our seed code in the seed created file

```
const profiles = [
 {
   name: 'Imen'
 },
 {
   name: 'Aymen'
 },
 {
   name: 'Feres'
 },
 {
   name: 'Nadia'
 },
 {
   name: 'Mohamed'
 }
];

exports.seed = function (knex) {
 return knex('profiles').del()
 .then(() => {
   return knex('profiles').insert(profiles)
 })
};
```

Now we have our migration file written and seed file written we can actually run the migration and seed commands to populate the database.

```
npx knex migrate:latest
npx knex seed:run
```

These commands will take your migration file and create the table. Then it will take your seed file and pre-populate the table with data. You should be able to see the tables and data in pgAdmin.

Deploying to Heroku

1. re commit our code to GitHub
2. Create an account and log into Heroku
3. Create a new application

Q. Use command line to deploy your application on Heroku with git.

Adding the Heroku Postgres Database

- In Heroku UI, add Heroku postgres add-on in ressources.
- You can now click on the added database link and see it created a database within Heroku.
- You can go to your settings tab instead of resources. Click reveal config vars to see that Heroku auto generated a DATABASE_URL for you. Then add production as a value with NODE_ENV as a key to the config variables as well so your code base knows that this is the production environment
- click more in the top right, and click run console to open a console command box.
- run knex migrate:latest and knex seed:run commands. This will migrate and seed your Heroku database.
- You now have a database on Heroku

Connecting with Code

We have to build out models and routes to communicate with our database using code. To communicate with the database we need to create a database configuration file. In the data directory create a file called db-config.js and paste this code.

```
require('dotenv').config();

const knex = require('knex');
const dbEnvironment = process.env.NODE_ENV || 'development';
const configs = require('../../knexfile')[dbEnvironment]

module.exports = knex(configs)
```

- This code figures out what database environment we are using based on our NODE_ENV environment variable and sets it to dbEnvironment and then uses our configuration from our knexfile.js for that environment and sets it to a configs variable. We are using either development or production in this project. Then it exposes that database configuration so that our code can query the correct database and preform basic operations within our app.
- Once our db config file is set up we can create a model.js and route.js file at the root of the project. Normally I would put model and route files within an API folder but for

the sake of speed and simplicity I will put them in the main project folder. Open the newly created model.js and add this code.

```
const db = require('./data/db-config');

const findAll = () => {
  return db('profiles')
}

module.exports = {
  findAll
}
```

First we are taking our database configuration, and then creating a model function findAll that returns all entries of the profiles table we created. Lastly we export the model function so our routes can use it.

- Then we open our route.js file and add a basic route to return this database information.

```
const router = require('express').Router()
const Profiles = require('./model')

router.get('/', async (req, res) => {
  const profiles = await Profiles.findAll()
  res.json(profiles)
})

module.exports = router
```

This code uses express to create a router, and pulls our model function from the model function file. Then it creates a route that uses a GET request that responds with our profile information from the database. Your model and route file should look like this if you are following this guide exactly.

- Once you have your db-config.js, model.js, and route.js all set up, the last thing to do is connect it to your index.js file and test it.
- add a few lines to your index.js to make this work.

```
/* import routes from route.js */
const testRouter = require('./route')

/* tell the server to use that router and allows us to use it using '/test' route */
server.use('/test', testRouter)
```

Test it

Open up your internet browser and go to your site http://localhost:5000. To access the data using the new code we just wrote, add your /test route at the end of the URL. http://localhost:5000/test.

Wrapping Up

Push all your code to Github and Heroku to make sure everything is up to date.

Q. Screenshot your production site on Heroku.

Notes

[1] npx lets you run code built with Node.js and published through the npm registry. Previously, Node.js developers used to publish most of the executable commands as global packages, in order for them to be in the path and executable immediately.

This was a pain because you could not really install different versions of the same command.

Running npx commandname automatically finds the correct reference of the command inside the node_modules folder of a project, without needing to know the exact path, and without requiring the package to be installed globally and in the user's path.

[2] Express.js is a web framework that will assist us in creating our HTTP server. Cors stands for Cross-Origin-Resource-Sharing and, at a basic level, allows servers from different origins to request information from each other.