Project Instructions

This document contains instructions for the project work in the course *Web Development – Advanced Concepts*.

Contents

Introduction

As project work, you should work in pair to create a platform per the instructions in this document. The platform will consist of:

- A relational database that stores the resources created on the platform.
- A none-relational database to store temporary data (sessions) on the platform.
- A web application that:
  - Web browsers can use to fetch webpages from (i.e. function as a website).
  - Exposes a REST API clients (e.g. smartphones) can use to work with the resources on the platform.
- A SPA application through which users can work with the resources on the platform.

You have a lot of freedom when it comes to the design and functionality of the platform, but the following requirement exists:

- There should exist at least three different type of resources on the platform:
  - One of the resources must represent *Accounts* users can create and login to.
  - The other resources should somehow have a relation to accounts (direct or indirect), i.e. a created resource should be owned by/belong to an account.

Use whichever type of resources you want in addition to accounts, but they could be:

- *Threads* logged in users can create and that other logged in users can write *Posts* to.
- *BlogPosts* logged in users can create and that other logged in users can write *Comments* on.
- *TodoLists* logged in users can create and add *TodoItems* to.
- ...

Implement the platform using the technologies taught in this course.

In addition to implementing the web application, you should also write a report describing how the platform works and how it has been implemented. The file `project-report-template.docx` on Ping Pong contains a template with further instructions about this. Your report will be a living document throughout the course (meaning that you will continuously improve/add content to it).

The file `project-grading-guidelines.pdf` on Ping Pong contains guidelines describing how your project work will be graded. You are recommended to read through it every now and then.

To help you, we have divided the project work into smaller parts that should be completed in order. The rest of this document contains descriptions of these smaller parts. Before you start working on the first part you are recommended to read through all the parts, which will give you a good overview of the work that lies ahead.

Good luck!

Installing required software

If you work on the computers in E2404, E2432 or E2433, all required software should already be installed for you, so you should not need to install anything yourself.

If you work on a school computer in any other room, you need to manually install the required software through the *Software Center* application (unless someone already has done that on the specific computer you sit at).

If you work on your own private computer, you need to download and install all required software yourself. In this course, we use only free software available for both Windows, Mac and Linux.

The required software is:

- **Docker**, https://docs.docker.com/install/
  - A program used for containerization, i.e. a program used for running other programs in an isolated and configured environment.

  Note about Windows:
  - Does your computer run (*Windows 10 Pro* or *Windows 10 Education*)? Then install the newer version of Docker: *Docker for Windows*.
    Note: The new version of Docker makes use of virtualization through Hyper-V. If you take the course *Android Development* you can speed up Android emulators by using virtualization either through Hyper-V or through Intel's HAXM. Both can't be used at the same time, so you avoid problems if you just stick to Hyper-V.
  - Does your computer run *Windows 10 Home* or *an older version of Windows*? Then install the older version of Docker: *Docker Toolbox on Windows*.
    Note: The old version of Docker does not completely support port forwarding, so you can't access containers through `localhost`, but must use the IP address of the Docker Machine instead. More information about that later.
- **Node.js**, https://nodejs.org/en/
  - Use whichever version you want, but most likely the newer the better. IF you don't want to gamble, go with the LTS version.
- **npm**, https://www.npmjs.com/
  - This one is installed along with Node.js, so you do not need to install it separately. However, you might want to update it to the latest version:
    - https://docs.npmjs.com/troubleshooting/try-the-latest-stable-version-of-npm

The following software tool is not required (use whichever IDE you want), but recommended:

- **Visual Studio Code**, https://code.visualstudio.com/
  - IDE with good support for writing, running and debugging Node.js and JavaScript code.

Part 1: Joining a project group

The project work should be carried out in pairs. Let the examiner know who you are working with by joining one of the project groups on Ping Pong.

If you can't find someone to work with, send the course coordinator an email at Peter.Larsson-Green@ju.se with the following information:

- Your name
- Your JU email
- The grade you are aiming for in this course
- The grade you got in the following pre-requites courses (or similar):
    - Introduction to Programming
    - Data Structures and Algorithms
    - Object-Oriented Programming
    - Network Programming
    - Web Development Fundamentals

The course coordinator will then try to match you up with students with similar ambitions and skills as you.

Part 2: Platform idea

Before you start working on this part you must:

- Complete and be approved on the laboratory work.

Your first task is to come up with what the platform you will create should do. Try to be creative and create a platform that solves a real-world problem ordinary people are having. Feel free to ask your family and friends for problems they have that could be solved using a platform. Examples of real-world problems could be:

- I often forget meetings.
- I often make bets with my friends, they remember the bets I lose so I have to pay them, but I often forget the bets I win, so they never pay me.
- I like to keep track of which celebrities I've seen, but I rarely remember that.

The only important thing is that your platform can be implemented using at least 3 type of resources (accounts + at least two more), but the more useful it is the better.

Your platform should be described in the project report, so in this part of the project work, you will not do any programming, but only work on your report. You should be able to complete at least the Introduction chapter.

When you're done describing your idea in the report, discuss it with a teacher at one of the lab sessions.

Part 3: Setting up docker

Before you start working on this part, you are recommended to:

- View/Take the following videos/tests:
  - Docker Basics

The web application should be implemented in Node.js using the Express framework, and the resources on the platform should be stored in a MySQL database. We could create a Docker container that contains both, but when it comes to containerization it is better that a container just has a single responsibility, e.g. to just run the web application or to just run the database. Among other things, it is much easier to scale a container that has a single responsibility. Therefore, the web application should run in one container, and the database should run in another container.

The Web Application image

Start by creating a new folder to store all the source code for the entire platform. In this document we will refer to this folder as *the Docker folder*. Then create a sub-folder inside this one to store the source code for the web application. In this document we will refer to that sub-folder as *the Web Application folder*.

In the web application folder, run the following commands:

1. `npm init --yes`
2. `npm install express`

Then create the file `app.js` in the web application folder with the content shown in Figure 1 below.

```
const express = require('express')

const app = express()

app.get('/', function(request, response){
  response.send("Hello, World")
})

app.listen(8080, function(){
  console.log("Web application listening on port 8080.")
})
```

Figure 1, The content of the file app.js (hello world with Express).

Verify that the code you have written so far works by running the command `node app.js` in the web application folder and then visit http://localhost:8080 in a web browser and verify that you see the text *Hello, World*. If you do, then it works as it should, and you can stop running the web application.

You just ran the web application as an ordinary program on your own computer. Let's try to run it in a container using Docker instead. To do that we need to tell Docker which environment our code needs to run, and how it should be started. To do that we need to put our code in an image that contains all this information, and then ask Docker to start running that image in a new container.

To specify the environment in the image one usually inherit the environment from an existing image. The Docker Hub at https://hub.docker.com contains a collection of images we can use, and in our case we need an image with Node.js pre-installed. Node.js provides official images that comes with Node.js pre-installed, so try searching for *node* there and find a suitable image (e.g. `node:11.6.0`). Then create a file called `Dockerfile` in your web application folder that looks something like what shown in Figure 2 below.

```
FROM node:11.5.0

EXPOSE 8080

WORKDIR /web-application

COPY package*.json ./

RUN npm install

COPY app.js ./

CMD ["node", "app.js"]
```

Figure 2, The content of Dockerfile in the web application folder.

Docker can then use the information in this file to build a new image that contains your source code and Node.js to run it. To do that, run the command `docker build --tag=my-app:latest .` in your web application folder. This tells Docker to build the image and to add it to its own collection of images on your computer. At the same time, it will also download the `node:11.5.0` image (unless it has already downloaded it before).

To run the image in a new container, run the command `docker run -p 3000:8080 my-app:latest`. If this doesn't work, you might first need to start the Docker Machine by running the command `docker-machine start`. When the container is running you should be able to send HTTP requests to your web application by opening http://localhost:3000 in a web browser on your computer. The command used to start the container told Docker that all incoming traffic on port 3000 to your computer should be forwarded to the container's port 8080. This does unfortunately not work on all versions of Windows. If that is the case for you, then you first need to run the command `docker-machine ip` to get the IP address of the Docker Machine and use that one instead, e.g. visiting http://192.168.99.100:3000 (or whichever IP address your Docker Machine has).

On Mac and Linux you can stop the container from running by pressing down `[CTRL]` + `[C]`. On Windows you also need to run the command `docker container ls` to retrieve the name of the container and then run the command `docker container stop theContainerName`.

Now you know the basics about creating, running and stopping Docker containers. You probably also want to create a `.dockerignore` file as described at https://nodejs.org/en/docs/guides/nodejs-docker-webapp/, so you more conveniently can build an image with multiple source files. That page is good reading even if you don't want to create a `.dockerignore` file, so read it!

Your workflow will kind of be:

1. Make changes to your source code files on your host computer.
2. Build a new image.
3. Run the new image in a new container.
4. Test your web application in a web browser.
5. Stop the container from running.
6. Restart on #1.

To manually carry out these steps each time you want to test changes you make to the source code is quite time consuming and boring. Docker has a solution to this problem: volumes. With volumes you can synch folders on the host computer with folders in a running container. When you change a file in one of these folders on the host computer, Docker will automatically send the new version of the file to the corresponding folder in the container that is running, so the container always has the latest version of the file.

Using a volume is quite easy. When you start a container with the `docker run` command you also specify which folders that should be synched using the `-v` option, like (on Windows) `docker run -v //c/folder/on/host/computer:/folder/in/container` (plus the `-p`). You don't want to synch your entire Web Application folder (the `node_modules` folder in it and some other files/folder should not be synched), so it might be a good idea to put your source code files in a new folder, for example called `src`, and use volumes to synch this subfolder instead.

However, synching files is not enough to keep our web application in the container up to date. Each time a JavaScript file changes the web application needs to be restarted. To make that happen we can use an npm package called `nodemon`. Simply:

1. Install `nodemon` by running the command `npm install nodemon` in the web application folder.
2. Change your `package.json` file to contain a script called `start` that executes `nodemon src/app.js -L`.
   Note: The `-L` flag is probably only necessary if you run Docker Toolbox for Windows. Read more about it at https://stackoverflow.com/q/39239686/2104665. This flag will increase the CPU usage, so remove it if possible.
3. Change your `Dockerfile` to start the web application with the command `npm run start`.

When you have built and run this new image and used volumes to synch folders the web application in the container should restart as soon as you change any JavaScript file the web application is dependent on. Your new workflow will simply be:

1. Build and run the image using volumes.
2. Make changes to JavaScript code on the host computer.
   (Docker will automatically synch the files to the container, and `nodemon` in the container will restart the web application when this happens)
3. See the changes in your web browser.
4. Restart on #2.
   Note: When you install a new npm package you need to restart on #1.

If you want to be able to debug your Node.js application from Visual Studio Code, you need to tell Visual Studio Code how to connect to the Node.js application running in the container. This tutorial shows you how you can do that.

You have now successfully setup a project suitable for implementing a web application in Node.js with Docker containers. Good job!

The Database image

Since the MySQL database should run in a separate container, let us do something similar for the database.

Start by creating a new folder inside the Docker folder to store information about the database image. In this document we will refer to that sub-folder as *the Database folder*. Create a new file in this sub-folder called `Dockerfile` with the content similar to the one in Figure 3 below. That Dockerfile describes an image inheriting from `mysql:5.7.24`, which is an image that runs a MySQL server.

```
FROM mysql:5.7.24

COPY initialize-database.sql /docker-entrypoint-initdb.d/
```

Figure 3, The content of Dockerfile in the database folder.

Then create a new file in the Database folder called `initialize-database.sql`. The SQL code you write in that file will be executed by the MySQL server when it starts, so in this file you can write SQL code that creates the tables in the database your platform needs.

The database image can be started and stopped in the same way as your web application image. After you have started it you should be able to connect to it from the host computer using http://localhost:3306 (or the Docker Machine's IP address). If you have a MySQL Database Management Tool (such as MySQL Workbench), try it out!

Docker has a tool called *Docker Compose* that greatly simplifies for us to start and stop multiple containers at the same time, as well as handle the connection between them (which we need, since the web application needs to communicate with the database). To use it, simply create a file in the Docker folder called `docker-compose.yml` with the content shown in Figure 4 below.

```
version: '3'
services:
  "web-application":
    build: "./web-application"
    ports:
      - "3000:8080"
      - "9229:9229"
    volumes:
      - "./web-application/src:/web-application/src"
    depends_on:
      - database
  database:
    build: ./database
    ports:
      - "3306:3306"
    environment:
      - MYSQL_ROOT_PASSWORD=theRootPassword
      - MYSQL_DATABASE=webAppDatabase
```

Figure 4, The content of docker-compose.yml in the docker folder.

Your images can now be built and started in containers using the `docker-compose up` command in the Docker folder, and the web application can now use the domain name `database` to connect to the MySQL database that runs in another container. However, if you make changes which requires your images to be re-built, you need to re-build them manually yourself, or you can start docker compose with the `--build` flag. Then docker will re-build the images each time before it starts them.

Part 4: Implementing the Web Application

Before you start working on this part, you are recommended to:

- View/Take the following videos/tests:
  - Layered Architecture in Node.js

Implement the website using a three-layered architecture:

- **The Presentation Layer** should only receive HTTP requests and generate and send back HTTP responses. This is the only layer that should use HTTP, Sessions, Cookies, HTML and CSS (and client-side JavaScript, if you want), so it is only in this layer we will be using Express. It will carry out the HTTP request by telling the Business Logic Layer what to do.
- **The Business Logic Layer** should only do computations. This includes applying business logic rules such as validation as authorization. If the Business Logic Layer needs to store/retrieve/update/delete any resource, it will simply ask the Data Access Layer to do that for it.
- **The Data Access Layer** should only be responsible for persistence of the resources, in this case stored in a MySQL database. It should provide operations one can use to store/retrieve/update/delete resources in the database. This is the only layer where you should use SQL.

Figure 5 below is a visualization of a three-layered architecture.

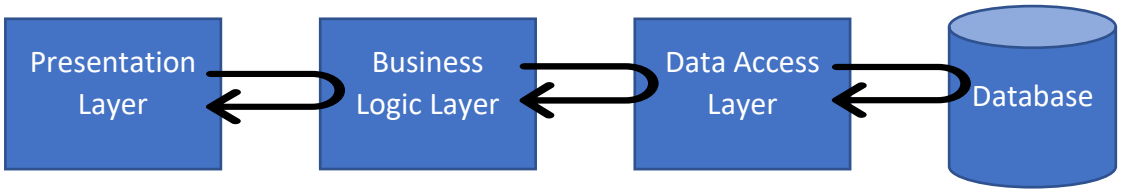

Figure 5, A three-layered architecture. Each rectangle represents a layer, and arrows shows what each layer makes use of. The Presentation Layer is the one receiving HTTP requests from clients.

There exists different ways one can separate the code into these three layers. The simplest way is probably to just create three different folders, one for each layer, and then put all code belonging to a layer in the corresponding folder.

To help you get started you can use the skeleton code available in the ZIP file `the-community.zip` on Ping Pong.

On your website, users should at least be able to create new resources and to browse and view existing resources. You do not need to implement update and delete functionality; that is an extra task you can complete at the end of the course if you have time for it. You neither need to implement authorization/login functionality now. That is the next part of the project work.

Part 5: Adding Authorization

Before you start working on this part, you are recommended to:

- View/Take the following videos/tests:
  - Scaling Web Applications
  - Scaling Databases

When the web application receives a successful sign in request from a client it needs to remember to which account that client signed in to, so it knows that when it receives requests from that client in the future. To do that one usually uses sessions. A session contains information about a client stored on the server-side. Where the sessions are stored on the server-side has a major impact on how the web application can be scaled.

The most commonly used approach to scaling web applications is by running multiple instances of them and use a load balancer to distribute the load (the HTTP requests) between them, as seen in Figure 6 below. Each instance can for example run in its own container or on its own server. When the load increases, we just launch more instances to handle it, and when it decreases, we just remove some of the instances (so we don't need to pay for having servers up and running that don't do any work).

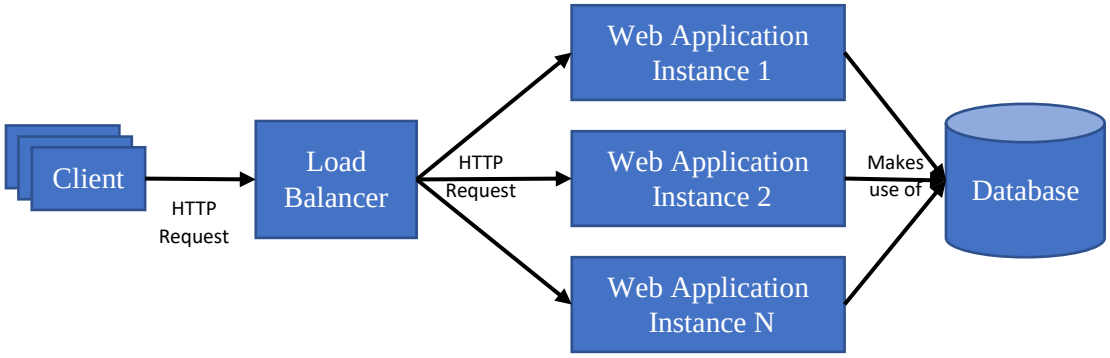

Figure 6, A commonly used architecture for scaling.
The Load Balancer receives HTTP requests from Clients and then forwards them to (one of) the Web Application Instances.

With this architecture, if the sessions are stored locally in the web application instances, then each clients' HTTP request must be forwarded to the same instance all the times, because it is only that instance that have access to the client's session. This can be achieved with a technique called *sticky sessions* (the load balancer tells the client to create a cookie specifying which web application instance it should forward the requests from that particular client to), but a solution that is easier to implement is to simply store the sessions in the database. This works since all web application instances are connected to the same database, so they store the sessions at the same place. However, fetching the session from the database for each request will make it take a little bit longer to carry out the requests.

The sessions could be stored in the MySQL database we already have, but sessions are just temporary data. They don't really have anything to do with the data we store in our MySQL database, so why use the same database? According to the single responsibility principle it is much better to store it in another database, ones whose only responsibility it to store sessions. Since sessions don't consist of any relational data, it is better to use a NoSQL database instead of a relational database since NoSQL databases often are faster and easier to scale than relational databases. The type of NoSQL database

that makes most sense for this use-case is a key-value database (the session id is used as the key, and the session itself as the value).

Add another container to your solution that runs a Redis database. Then connect to it from your web application and store your sessions in it. In your web application you will:

- Use the npm package `express-session` to add session support to Express.
- Use the npm package `connect-redis` to store the sessions in a Redis database.

Then add the login feature to your web application and add authorization. Remember that sessions are something we use to make HTTP stateful (hence we only use them in the presentation layer) and authorization is something we should implement in the business logic layer (which users that are authorized to do what should be independent of how they use our application).

Part 6: Using Dependency Injection

Before you start working on this part, you are recommended to:

- View/Take the following videos/tests:
  - Dependency Injection in Node.js

The three layers the web application consists of have some dependencies:

- The presentation layer has some dependencies on the business logic layer.
- The business logic layer has some dependencies on the data access layer.

These dependencies are hardcoded at multiple places, as shown in Figure 7 and Figure 8 below (Figure 9 below doesn't contain any hardcoded dependencies, but it shown for completeness of the three layers).

```
const express = require('express')
// Here we have hardcoded a dependency on the account manager.
const accountManager = require('../business-logic-layer/account-manager')

const router = express.Router()

router.get("/", function(request, response){
  accountManager.getAllAccount(function(errors, accounts){
    const model = {
      errors: errors,
      accounts: accounts
    }
    response.render("accounts-list-all.hbs", model)
  })
})

module.exports = router
```

Figure 7, account-router.js in the Presentation Layer.

```
// Here we have hardcoded a dependency on the account repository.
const accountRepository = require('../data-access-layer/account-repository')

exports.getAllAccounts = function(callback){
  accountRepository.getAllAccounts(function(errors, accounts){
    callback(errors, accounts)
  })
}
```

Figure 8, account-manager.js in the Business Logic Layer.

```
const mysql = require('mysql')
const connection = mysql.connect(...)

exports.getAllAccounts = function(callback){
  const query = "SELECT * FROM accounts ORDER BY username"
  connection.query(query, function(error, accounts){
    if(error){
      callback(["Database error"], null)
    }else{
      callback([], accounts)
    }
  })
}
```

Figure 9, account-repository.js in the Data Access Layer.

Hardcoding dependencies like this makes our web application less flexible. For example, each time we use the account manager, the account manager will always use the account repository that works with the data in the MySQL databases. You might think this is not such a big deal, because that's what we want to happen when our web application runs, right? Yes, if we want to use the code to run the web application, then yes, that is what we want to happen. But what if we write tests and just want to run the code inside the account manager to test if that works? Then we don't want the account manager to make use of the account repository that communicates with the MySQL database, because if the test fails when we run it, then the problem could just as well be in the account repository (e.g. broken connection with the database).

When we test the code in the account manager through tests we have written, we don't want it to use the account repository that makes use of the MySQL database. Instead, we can provide a mockup of the account repository, as the one shown in Figure 10 below.

```
const allAccounts = [] // All accounts are stored in this array.

exports.getAllAccounts = function(callback){
  callback([], allAccounts)
}
```

Figure 10, Mockup of the account repository in the Data Access Layer.

For our tests, we want the account manager to make use of the mockup version of the account repository, and when we run the web application for real we want the account manager to make use of the account repository that makes use of the MySQL database. Because of this, we can't hardcode the dependencies we have.

Note: Mockups are not necessarily only created for our own code when doing testing. Another example is making use of third-party services through APIs that costs money. When our web application runs in production for real, we want to make use of the real third-party API and are willing to pay for using it, but when we run it locally on our own computers for development/testing, we don't want to pay for using it, so we use a mockup version instead.

So, in our account manager, we can't hardcode which version of the account repository we should use. Instead, the account repository that should be used will be *injected* to the account manager. The account manager itself doesn't care about which account repository it uses as long as it has access to the one it should be using. We say that the account manager is dependent on the interface of the account repository, which means that all different account repositories we implement must expose the same set of functions/same interface the account manager can use.

When we start the program, we specify which account repository to use, and then when we use the account manager it will the account repository we specified. How we make all this happen depends on which dependency injection framework/container we use. The npm package awilix gives us this functionality. Figure 11, Figure 12 and Figure 13 below shows one way to use it.

```
module.exports = function({}){
  // Name all the dependencies in the curly brackets (none in this case).

  const allAccounts = []

  return {
    getAllAccounts: function(callback){
      callback([], allAccounts)
    }
    // Continue to list all other functions in account repository here.
  }

}
```

Figure 11, account-repository.js in the Data Access Layer.

```
module.exports = function({accountRepository}){
  // Name all the dependencies in the curly brackets.

  return {
    getAllAccounts: function(callback){
      accountRepository.getAllAccounts(function(errors, accounts){
        callback(errors, accounts)
      })
    }
    // Continue to list all other functions in account manager here.
  }
}
```

Figure 12, account-manager.js in the Business Logic Layer.

```
const express = require('express')

module.exports = function({accountManager}){
    // Name all the dependencies in the curly brackets.

  const router = express.Router()

  router.get("/", function(request, response){
    accountManager.getAllAccount(function(errors, accounts){
      const model = {
        errors: errors,
        accounts: accounts
      }
      response.render("accounts-list-all.hbs", model)
    })
  })

  return router

}
```

Figure 13, account-router.js in the Presentation Layer.

With `awilix` we can then in the main file specify which dependencies we want to use, as shown in Figure 14 below. If we want to change which account repository to use, we just need to change that at line 4 in that file.

```
const awilix = require('awilix')

// Import the ones we want to use (real or mockup), real in this case.
const accountRepository = require('data-access-layer/account-repository')
const accountManager = require('business-logic-layer/account-manager')
const accountRouter = require('presentation-layer/account-router')

// Create a container and add the dependencies we want to use.
const container = awilix.createContainer()
container.register("accountRepository", awilix.asFunction(accountRepository))
container.register("accountManager", awilix.asFunction(accountManager))
container.register("accountRouter", awilix.asFunction(accountRouter))

// Retrieve the router, which resolves all other dependencies.
const theAccountRouter = container.accountRouter
```

Figure 14, Creating an using an awilix container to resolve dependencies.

So, to use dependency injection with awilix like this you need to:

1. Change your JavaScript files to export a single function that receives an object (the container) containing all the dependencies that should be used.
2. In your main file, create a new awilix container and add all these functions to it.
3. Retrieve the values you need from the container.

If you want, feel free to use any other dependency injection method you prefer.

Part 7: Multiple Data Access Layers/ORM

Before you start working on this part, you are recommended to:

- View/Take the following videos/tests:
  - Using an ORM in Node.js

Now that your web application makes use of dependency injection, let us see if it works by implementing another data access layer. The new data access layer should have the same interface as the old one (must implement the same exposed functions) but should be implemented differently. Let us implement it using an Object-Relational Mapping framework, so you get some practice on using that.

Use whichever ORM you want, but Sequelize is quite simple and straight forward to use. Feel free to use any database you want. Using your existing MySQL database is OK, but to practice on using Docker we recommend you add another container running another database, e.g. PostgreSQL.

When you're done your architecture could look like the one shown in Figure 15 below, but when running the application only one of the data access layers will be used.

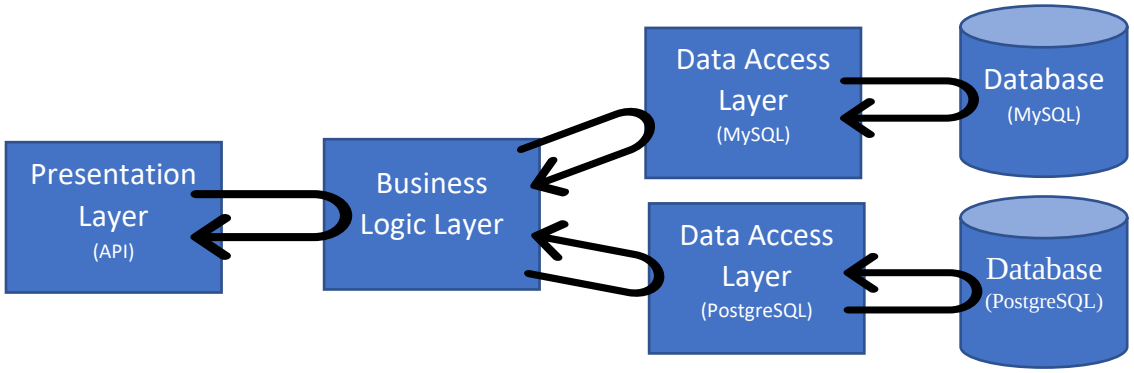

Figure 15, Current architecture.

Part 8: Adding a REST API

Before you start working on this part, you are recommended to:

- View/Take the following videos/tests:
  - REST API Basics
  - REST API in Express
  - REST API Authorization
  - JSON Web Tokens
  - Third-Party Authentication

Users can use the platform on their smartphones through the web browser on it, but the user experience can often be improved my implementing and let the users use a native smartphone application instead. This way, the GUI would consist of GUI components native applications have, which would appear much more appealing to the user compared to the GUI components web pages consists of. Furthermore, a native application would also be able to make use of native features on the smartphone which aren't accessible through client-side JavaScript code.

However, unlike the web application, a native smartphone application can't communicate directly with our database. One need to know the username and password to the database to use it, and we can't put that in a native smartphone application, because then any hacker would be able to retrieve it from the smartphone application after they have downloaded it. Then they could login to the database and change it however they please. Instead we will add a REST API to our web application, and the native application should be able to store/retrieve/update/delete data in the database through it. The platform will be as shown in Figure 16 below.

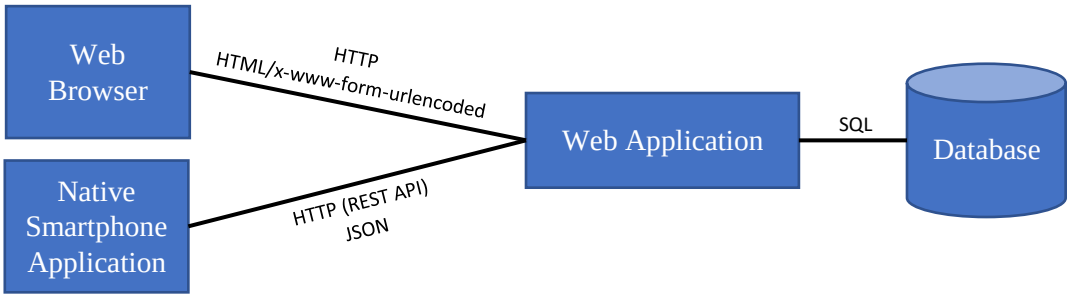

Figure 16, Overview of the platform.

The difference between the way the web browser and the native smartphone application communicates with the web application will primarily be the data format they use. Both will communicate with the web application using HTTP, but the web application will send data (submitted forms) in the data format application/x-www-form-urlencoded and receive back HTML code, while the native smartphone application will send and receive in the data format application/json (JSON), which is a much simpler data format.

The REST API can be implemented as an additional Presentation Layer in the web application, as shown in Figure 17 below. Since the users who use the platform through the native smartphone application should be able to do the same thing as users who use the platform through the web browser, we should not need to make that much changes to the code in the business logic layer nor the data access layer

(but we might need to add code). You should be able to re-use most of the functionalities from these layers.

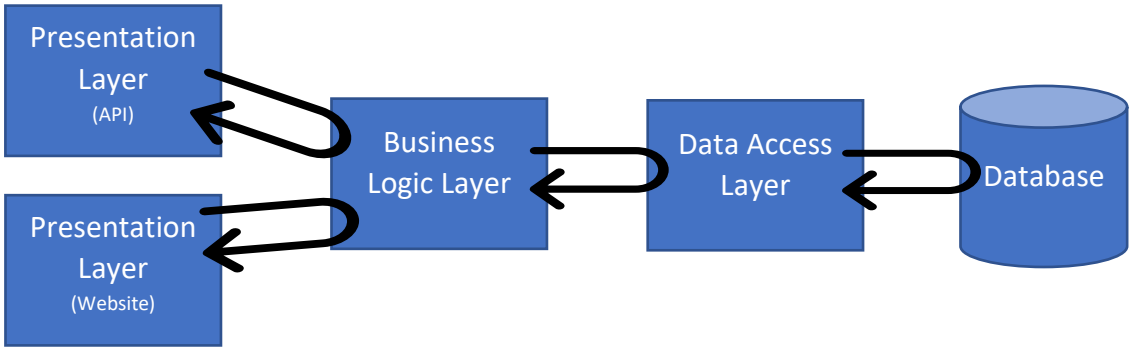

Figure 17, Current architecture (with one Data Access Layer).

Now the REST API can run as its own Express application in one container, and the Website runs as an Express application in its own container, and these can scale independently of each other. If you instead prefer to run both presentation layers in the same container, that is OK as well. Then the presentation layers can export their Express Application objects which then is used by the fourth layer (which also is an Express application). See the documentation for `app.use()` to learn how one Express Application object can make use of other Express Application objects.

The REST API needs to support the following operations:

- Create a new account.
- Login to an existing account.
- When logged in, apply CRUD (Create, Read, Update, Delete) operation for at least one of the other two resources that belongs to the user.

One usually doesn't use sessions in REST APIs to remember that a user has logged in. Sessions relies on cookies and supporting all features of cookies is hard for clients. Instead, one usually uses tokens instead, which are much simpler.

Tokens can be implemented in different ways, but the simplest way is by using JSON Web Tokens. These can easily be created and verified in Node.js using the npm package `jsonwebtoken`.

Implement *Access Tokens* as described by the OAuth 2.0 Framework and implement ID Tokens as described by the OpenID Connect specification. Use the *Resource Owner Password Credentials Grant*, and make sure to get all the details right.

Part 9: Implementing a Single-Page Application

Before you start working on this part, you are recommended to:

- View/Take the following videos/tests:
  - AJAX

Since implementing a native smartphone application is not part of this course, you will instead implement a Single-Page Application (SPA). A SPA is a website that only consists of a single webpage and a lot of client-side JavaScript code to make it appear to work as an ordinary application. When users click on links or submit forms, the web browser doesn't submit these and load a new web page. Instead, client-side JavaScript codes listens for these events and handles them itself. This way, we can add loading indicators while fetching a new "page"/submitting a form and add transitions when going from one "page" to another.

So, your next task is to implement a SPA that makes use of the REST API on the platform. In the SPA users should be able to:

- Create a new account.
- Login to an existing account, and then:
  - Apply CRUD operation for at least one of the other two resources.

Since the SPA only consists of static HTML, CSS and JS files (and possibly images), we don't need to implement a fancy web application to serve these files to clients requesting them. Instead, we can use a program that serves files, such as nginx. Run nginx (or whichever similar program you prefer) in a new container to serve the static files for the SPA.

Since web browsers follow the same-origin policy, your REST API needs to support cross-origin resource sharing, so add that to it unless you already have done it.

Part 10: Optional tasks

Here are some optional tasks you must complete if you want to get a grade higher than 3. Remember that completing these extra tasks does not necessarily give you a higher grade, but you have to complete them to have a chance to get a higher grade. Also, do not forget to look through the file `project-grading-guidelines.pdf`.

A fancy website (required for grade 4 and 5)

Do also implement update and delete operations for your resources on the website (not necessary for the REST API).

A fancy SPA (required for grade 4 and 5)

In your single-page application, when you have sent an HTTP request to the REST API and wait for an HTTP response, show a loading indicator of some kind. When forms are submitted, users should not be able to submit the form again while you are waiting for the response.

To verify that it works as it should, you need simulate latency (so it takes a couple of seconds before you get back an HTTP response). This can be done using the `setTimeout()` function in your JavaScript code, or (even better) by telling the web browser to simulate it for you. Here is described how to do that in Google Chrome.

Supporting third-party authentication (required for grade 5)

Lets users create a new account on your platform by using their existing account on another platform supporting OpenID Connect instead creating yet another username and password. You can for example use Google.

This does only have to work either through the website or through the REST API. You choose which one.

Part 11: Demonstration

Demonstrate how your application works to the rest of your class. The reason for the demonstration is two-folded:

- You get some practice in presenting your work, which is a very important skill in your future professional career.
- You get to see other ways to implement similar functionality (hopefully not all websites will look the same).

Your presentation will not be graded; consider it as (mandatory) practice.

Part 12: Submitting your work

Submit your work on Ping Pong for grading.