Software Engineering 2 - Prof. Di Nitto Elisabetta
Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano

CLup - Customer Line-up

ITD
Implementation and Testing Delieverable

Andrea Franchini(10560276)
Ian Di Dio Lavore (10580652)
Luigi Fusco(10601210)

07-02-2020

Contents

1 Introduction

1.1 Scope

This document describes the implementation and testing process of a working prototype of the service described in the “Requirement Analysis and Specification Document” and ”Design Document”. This document is intended to be a reference for the developer team and explains the choices regarding used software, frameworks, programming languages. It also provide input on how to perform integration testing between the implemented components.

1.2 List of definitions and abbreviations

1.3 Abbreviations

- **RASD** - Requirement Analysis and Specification Document
- **DD** - Design Document
- **JS** - JavaScript
- **R*n*** - Requirement *n*

1.4 Definitions

- **Client** - In this document client means a web app.

1.5 Reference documents

- Requirement Analysis and Specification Document (rasd.pdf)
- Design Document (dd.pdf)

1.6 Document structure

- Chapter 1 presents the requirements that have been implemented in the prototype.
- Chapter 2 presents the adopted programming languages and frameworks, justifying each choice.
- Chapter 3 covers the structure of the source code.
- Chapter 4 explains the testing process in greater detail.
- Chapter 5 provides explanations on how to run, test and build the prototype.

2 Implemented Requirements

Requirements have the same nomenclature present in the RASD and DD.

We anticipate that the prototype implementation of the *Totem* is an exemplification of what described in the RASD and DD, but of course lacks the extra hardware, such a scanner. The main features are present, for example validating a ticket or joining the queue.

Admin services have not been implemented, as we deemed them not important for a user functionality centered demo as, apart from the initial configuration of the store and of the timeslots, they mostly concern the long run management of the system. All of the Admin functionalities can be easily simulated with simple queries to the database.

2.1 Implemented

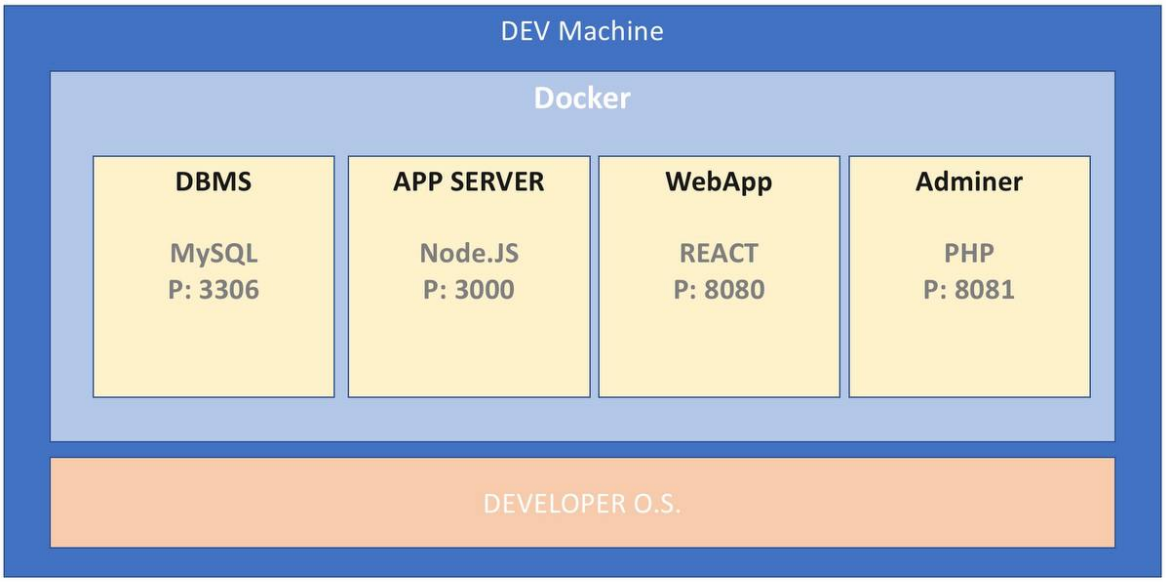
- R1 - Allow a User to sign up for an Account after providing a mobile phone number.
- R2 - Allow a Registered User to find Stores nearby ~~a specified~~ *the user or an hardcoded* location. ***Note:*** *This is because the Google Maps API that converts addresses to coordinates is paid.*
- R4 - Allow a Registered User to get in the virtual line at a specified store.
- R5 - Allow a Totem User to get in the virtual line of the store where the totem is installed. ***Note:*** *For the purposes of the demo, the user interface of the totem contains fields to specify the auth token and the store id. In the final systems these would be hard-coded and dependent on where the totem is installed.*
- R6 - Allow a Registered User to preview an estimate of the queue time.
- R8 - Allow a Registered User to cancel their reservation.
- R9 - Allow a Registered User to leave the virtual queue.
- R10 - Allow a Registered User and a Totem User to retrieve a scannable QR Code/Barcode that they must present in order to be granted access to a store.
- R12 - The System cancels User reservations in case of a major delay.
- R13 - The System enforces the limits on the allowed number of concurrent Customers inside a store by restricting the access at the entry points (for example, automatic doors or turnstile). ***Note:*** *Although the physical barrier is not present, the functionality is implemented in software and a text message is shown indicating if access is granted or not.*
- R14 - The System grants a User with a reservation access only within ashort time (set by the manager) after the User's time of reservation.

2.2 Not implemented

- R3 - Allow a Registered User to filter out stores based on available timeframes, days and distance. ***Explanation:*** *while the implementation isn't difficult, implementing this feature would have slowed down the development and testing process.*
- R11 - The System notifies the Users affected by delay. ***Explanation:*** *it requires external paid service that isn't needed for a prototype, especially a web based one. Had we implemented a native mobile app, this feature would have been implemented.*
- R15 - Allow System Managers to set the division of the maximum number of people allowed between the normal queue, the priority queue for people with special needs and the book a visit slot capacity. ***Explanation:*** *While the division between reservation and queue is implemented, we deemed the priority queue of secondary importance for a demo.*
- R16 - The System calculates the average shopping time by recording every time a user enters and exits the store.
- R17 - Allow System Managers to set a limit to the people allowed into the store at a time.
- R18 - Allow System Managers to choose the frequency and size of thetime slots.
- R19 - Allow System Managers to know the average time spent in the store.
- R20 - Allow System Managers to know the current and past number of people in the store.
- R21 - Allow System Managers to check the current status of the queue and of the time slots.

3 Adopted Technologies

The chosen frameworks and technologies have been chosen with regards the Component View (Section 2.2) of the Design Document. Components not critical for a prototype such as a reverse proxy, a CDN, or firewalls have been disregarded in this implementation.



3.1 Frameworks and programming languages

3.1.1 Back-end

We decided to implement the *Application Server* with JavaScript and NodeJS, because it offers great horizontal scalability and generally good performance in handling multiple simultaneous requests. In order to ease development, we chose the industry standard server framework ExpressJS, which allows us to easily implement a REST API for the clients.

The chosen DBMS is MySQL, because it's a time-tested, fast and secure platform. The connection to the database is implemented with a singleton. This way all requests generated by a single instance of the server go through the same connection.

3.1.2 Front-end

In order to develop a working prototype in a short timeframe, we decided to implement only the web application client, which offers the same functionalities of a native application, exception made for push notifications. The web app can be also easily ported into a Progressive Web App (PWA), which allows users to use it offline.

Of course, being the client a web application, the obvious choice regarding suitable programming languages is JavaScript, allowing us to have a codebase written in the same language. JavaScript is a solid language that, although its quirks, has great documentation and a vast user knowledgebase. TypeScript, a superset of JavaScript that enforces explicit types, has not been used because it would have slowed down the development process. In a real-world use case, it should definitely be considered, as it provides by itself a way to better document code without effort.

We decided to use an industry-standard framework such as React to better organize the structure of the web app, while maintaing high extendability, modularity and tidiness of the code. React is a JavaScript framework that allow to develop a web application through JavaScript, and utilizes a special syntax very similar to HTML to describe the components. React also support functional programming, meaning that each component of the web application defined by the developer (e.g.: a view, a button...) is represented by a function. By combining these components it's easy to maintaing a clean and reusable codebase.

To make requests against the REST API, we used the new standard `fetch` API.

The totem is implemented as a barebone web interface sending requests to the backend. It contains all the functionalities of the physical totem, with the added flexibility of specifying the identification token and the associated store for demo purposes.

The functionality of detecting an exit from the store is implemented as a button in a dedicated front-end. In the real system the event would be triggered based on the needs of the store by an existing system, like the creation of a receipt or a receipt scanner at the exit of the store.

3.2 Additional software

In this part other notable tools are briefly explained.

Yarn In order to better organize the project dependencies, it's a wise choice to use a packet manager. By having a codebase written entirely in JS, we adopted Yarn, a fast and reliable packet manager for JavaScript. It also handles custom scripts to ease the development workflow.

Jest To perform unit and integration testing, we've chosen Jest, an industry-standard JavaScript testing framework, initally created for internal use at Facebook. It allows to easily and rapidly define tests associated to a particular functional unit by changing the JS test file extension to ".test.js". This way each functional unit can have its respective test in the same directory.

Docker We decided from the beginning that our architecture would need to be as scalable as possible in the long run. Containerizing our components was essential for this purpose and it provided a fast deployment option for the developers in out team. As described in the section 6, by using Docker we enabled the deployment of the whole infrastructure with just one command. To easily deploy a configured database for development, we utilized a MySQL image, an industry-standard container solution. To manipulate the DB, we used Adminer, a lightweight management system for SQL.

4 Source Code Structure

The whole project is contained in a single repository. In the root directory (while referring to the repository of this project, such directory is `/ITD/`). In the root folder are present many config files, notable entries are:

- `package.json` specifies the project dependencies and contains scripts to ease development.
- `docker-compose.yml` specifies the deploy configuration of this prototype and allow to setup easily the needed services.
- `.env` contains some system parameters. Albeit it should not be committed in a Git repository, we believe it's an acceptable exception for a prototype.

The relevant folders are the following:

- `__tests__` contains additional integrations tests.
- `database` contains configurations and dumps for the MySQL database.
- `src` contains the source code of the prototype.
  - `client` contains the client source code.
    - `components` contains React modules that implement one single function.
      - `App.js` is the root of the web application.
      - `Button.js` represent a simple button component.
      - `ErrorMsg.js` is a panel to display errors.
      - `Field.js` is a form field with options to limit input.
      - `PrivateRoute.js` is a custom routing component needed to route an unauthorized user to the login page.
      - `StoreMap.js` display a maps to display stores through the Google Maps API and draws markers on it.
      - `TicketCache.js` reroutes users that have a ticket to the ticket view.
    - `css` contains `styles.css`, a CSS file to tweak minor aspects of the web app. The overall style is handled through a library loaded from the dependencies.
    - `images` contains various icons for the web app.
    - `views` contains the various views that the web app can show. Views handles the fetching of data from servers and display it.
      - `LoginView.js` handles the login process (corresponds to RASD Fig. 5A, 5B).
      - `SettingsView.js` handles the logout of a user. Various, not-implemented settings would be configurable from here.
      - `StoreDetailView.js` shows the details of a single store, and allow a user to join the queue or go the timeslot view (corresponds to RASD Fig. 5E).

        - `StoreListView.js` shows the stores in the vicinity of a user (corresponds to RASD Fig. 5C)
        - `TicketListView.js` shows active tickets of a user (corresponds to RASD Fig. 5G, 5H)
        - `TimeslotsView.js` shows the available timeslots to a user, allowing them to make a reservation (corresponds to RASD Fig. 5A, 5F).
        - `WelcomeView.js` is the first view an new user sees.
    * `defaults.js` contains constants used in the web app.
    * `index.html` is the root of the web app. Mandatory since it's a website.
    * `index.js` is the entry point of the web app logic. It is loaded by `index.html` and loads `components/App.js`
- `server` contains the server source code.
    * `components` contains the various service components needed by the server. They correspond to the homonymous components presented in the DD. In particular `QueryManager` handles the connection to the DBMS service. Certain methods of these components have been slightly changed to simplify the implementation (for example, the id of a reservation is generated by the database instead of being send to it). The overall functionality of a component is kept unchanged.
        - `AccountManager`
        - `QueryManager`
        - `QueueManager`
        - `ReservationManager`
        - `SmsApi` (stub, instead of sending an SMS prints it to `stdout`)
        - `StoreSearch`
        - `TicketManager`
    * `errors` contains definitions of various errors that the server can throw.
    * `main.js` the entry point of the server. Contains the REST API endpoints. The REST API is developed according to the DD.
    * `utils.js` contains utility functions.
- `totem` contains an example of the totem interface and software.
- `store` contains a demo to trigger the detection of an exit from a specific store.

5 Testing

Testing using *Jest* was performed in a bottom-up fashion, following the schema described in the Design Document. The first module to be developed and tested was the QueryManager, with all other functionalities depending on it. Few tests target the QueryManager directly, as the single functionalities it offers are often extremely simple, and are better suited to be tested when interacting with eachother to execute higher level functionalities. In order to make the tests easily replicable, all tests are performed in a transaction which is always rolled back, and the tests are executed sequencially. This way, each test assumes an empty database, and leaves the database empty after execution. The *fakedate* module was used in order to alter the current time returned by the system and testing time related events, like the cancellation of a ticket or the correct execution of a reservation. For this reason, all dates are generated in the business tier instead of the database.

Tests were written and executed in sync with the development of the associated functional unit, in order to test step by step the correctness of the implemented subfunctionalities, progressively moving towards more complex

and complete tests. For this reason our tests cover both the basic functionalities and the corner cases, with a particular focus on the enforcement of access constraints and on the automatic cancellation of tickets.

System testing was performed using the *supertest* framework in conjunction with *Jest*. *supertest* allows to generate an instance of the server and to simulate calls to the REST API directly from code. System testing was performed simulating a possible normal use of the system, while stimulating all possible functionalities and situations.

System testing was designed in order to check the correctness of the following scenarios:

- Creation and login of a user
- Creation of a queue ticket
- Automatic cancellation of a queue ticket
- Deletion of a queue ticket
- Ticket checking:
  - Successfull checking
  - Denied because not first
  - Denied because store is full
  - Denied because of reservation slots
- Creation of a user reservation
- Automatic cancellation of a reservation

6 Installation Instructions

Here we report the instructions to install all software dependencies needed to run the service. This instructions are also available in the GitHub repository in the ITD folder. The two proposed setup are mutually exclusive:

- If you run the "recommended way" to later run the "development setup" you have to remember stop the containers that are active or else you will end up with conflicts on the active PORTS

6.1 RECOMMENDED WAY

- Download the content of the ITD folder
- Install Docker and docker-compose
- Navigate to the main directory (the one with the docker-compose.yml file)
- Run: `docker-compose up -d`
- To see the internal logs of the NodeJS server: `docker logs --follow clup_server`

NOTE: if you are on Linux (and you haven't created a docker user) please add "sudo" to the preceding commands)

6.2 Development setup

6.2.1 Installing the toolchain

- Install NodeJS, possibly the LTS version (14.15.4).
  - If you use tools such `nvm`, please run `nvm install --lts` and `nvm use --lts`
- Open the terminal and run `npm install -g yarn` to install the packet manager we're using, Yarn

- If you are developing this project, installing Nodemon is recommended `npm install -g nodemon`. Please note that it's already included in the *developer dependencies*, so if you do not wish to install it globally, skip this step.
- Clone the repository using either one of the following commands
  - `git clone https://github.com/ian-ofgod/DiDioLavoreFuscoFranchini-CLup`
  - `gh repo clone ian-ofgod/DiDioLavoreFuscoFranchini-CLup`
- Change directory into the ITD folder `cd DiDioLavoreFuscoFranchini-CLup/ITD`
- Modify the .env file so that DB_ADDRESS="localhost"
- Run `yarn install` to install the dependencies

6.2.2 Setting up the database

You'll have to install Docker on your device, and make sure docker-compose is installed as well.

- Run `docker-compose -f docker-compose.dev.yml up -d` to spin up the images specified in the `docker-compose.dev.yml` file.

Visit http://localhost:8081 to see if Adminer is running, and log in with the credentials specified in the `.env` file (server: `db`, username: `clup_admin`, password: `clup`, database: `db_clup`).

Without Docker:

- Install MySQL server, then create a schema and a user with the credentials found in the `.env` file, otherwise create your own and edit the `.env` file.
- In case of errors about permissions, try running the following command, especially if you are not using Docker:

```
create user clup_admin@localhost identified with mysql_native_password by 'clup';
grant all privileges on * . * to clup_admin@localhost;
```

6.2.3 Verify the installation

- Run `yarn server-dev` to run the developer server
- Run `yarn client-dev` to run the client server
- Run `yarn client-build` to compile the client package (output is in `dist`)
- Run `yarn test` to run the full test suite with Jest
  - NOTE: Inside `.env` change `DB_ADDRESS="db"` to `DB_ADDRESS="localhost"`
  - NOTE: Run `yarn clear-db` before the tests to clear the DB tuples
  - NOTE: To restore the DB status to the DEMO one, run `yarn reset-db`

6.2.4 Using the system

The web application should be served at `localhost:8080`, while the REST API should be available at `localhost:3000`. Adminer can be used to configure and access the database easily at `localhost:8081`, the credentials are the one in the `.env` file.

To view the token example page, open the HTML file contained in `src/totem` in your browser.