Integration and Testing Document

De Martini Luca - Duico Alessandro

Source code

Application Server source code

Client Web Application source code

Introduction

Clup is a three-tier system, consisting of a Database (Postgres), the Clup binary (`ITD/backend`) and the Clup Webapp (`ITD/frontend`). A middleware (Redis) is employed for session storage.

Scope

This document gives a concrete description of the Implementation and Testing procedure.
In particular, it covers:

- the reasons for choosing those requirements that are implemented;
- the adopted development technologies: languages, frameworks and API standards;
- the structure of the source code;
- how testing is performed;
- the prerequisites and instructions for building and installing.

Implemented features

Being a Proof-of-Concept, only the core features for generating tickets to line-up at a Shop were implemented, plus the monitoring functions available to the staff.

Referring to the DD, the following requirements are satisfied:

- **[R1]** The system shall keep track of the list of Customers waiting to visit each Shop
- **[R2]** The system shall allow customers to request the right to visit a shop as soon as possible
- **[R3]** The system shall give Customers a Token associated with their position in the waiting line
- **[R5]** The Staff shall be able to scan Customer generated Tokens using a textual input
- **[R6]** Given a Token, the Staff application shall be able to verify its validity
- **[R7]** Given a Token, the Staff application shall be able to verify its position in the waiting line
- **[R8]** Given a Token, the Staff application shall be able to mark it as used and update the list of Customers currently inside the Shop
- **[R12]** The system shall ask Customers to specify the approximate duration of their visit
- **[R13]** The system shall automatically infer an estimate visit duration for returning customers
- **[R14]** The system shall give Customers an estimate of the waiting time remaining before it's their turn
- **[R16]** The system shall be able to connect to a Maps service to show information about travel time
- **[R17]** Customers shall be able to specify the categories of items they intend to buy
- **[R18]** The system shall keep track of the number of customers visiting the Shop on a Department basis

Adopted frameworks and languages

Rust language

The Rust programming language was chosen for its performance, reliability and productivity. It's fast and memory-efficient and allows building highly concurrent applications without fear of data races and most other synchronization problems. Rust's highly praised documentation, tooling and compiler make it a productive language empowering everyone to build reliable and efficient software.

Actix

Actix is a A powerful, pragmatic, and extremely fast web framework for Rust.

It was chosen for the implementation for three main reasons: Actor model concurrency, performance and safety.

PostgreSQL

PostgreSQL is The World's Most Advanced Open Source Relational Database

PostgreSQL is a powerful, open source object-relational database system with over 30 years of active development that has earned it a strong reputation for reliability, feature robustness, and performance.

Redis

Redis is an open source, in-memory data structure store, used as a database, cache, and message broker

It provides a fast memory data structure that can be used for authentication and session storage without putting load on the relational database.

Docker

Docker simplifies and accelerates the development workflow, while giving developers the freedom to innovate with their choice of tools, application stacks, and deployment environments for each project.

Docker is the most popular container solution. Containers are a standardized unit of software that allows developers to isolate their app from its environment.

Vue.js

Vue.js is an open-source MVVM (model-view-viewmodel) front end JavaScript framework for building user interfaces and single-page applications.

Its main features are:

- reusability of Components, defined with the use of an HTML-based syntax that allows binding the rendered DOM to the underlying Vue instance's data;
- reactivity, that means each component keeps track of its dependencies during its render, so the system knows precisely when to re-render, and which components to re-render;
- support of single-page applications. Instead of the default method of the browser loading entire new pages, it interacts with the user by dynamically rewriting the current web page

with new data. The consistency of the navigation history is kept by the "vue-router" package, which provides an API to update the application's URL.

Source code structure

Backend

The code for the application server is contained in the `backend` directory.

- In the root the `Cargo.toml` file lists dependencies and the packages' configuration
- The `migrations` directory contains the SQL files for the database migrations
- The `tests` directory contains the integration tests for the backend
- The `src` directory contains the Rust source files for the application server. This is the module structure:
  - `bin` contains the executable
  - `api` contains the endpoints for the API (MVC Controller)
  - `models` contains the models and associated Data Access Objects (MVC Model)
  - `utils` contains utils functions for session state, url encoding and tests

Unit tests can be found in a submodule in the same source file as the component they are testing (as for Rust guidelines).

Frontend

The code for the application server is contained in the `frontend` directory.

- `package.json` lists the main info about the Node.js package, provides release and debug dependencies and useful scripts that can be executed with `yarn run <script>`
- `App.vue` is the main Vue.js component, which contains every other page
- The `public` directory hosts static files that are served as they are by the webserver
- The `src` directory contains the source code for the reactive part of the Web Application:
  - `assets` - any assets that are imported into your components
  - `components`- all the components of the projects that are not inside `views`
  - `router` - scripts that handle the routing
  - `store` - scripts related to the Vuex store
  - `translations` - locales files, not present
  - `views` - the components that are routed. They represent the pages of traditional HTML.

Testing

Testing has been done following the Design Documents guidelines. Unit tests are written alongside the code for the model using a bottom up approach.
Integration tests are written in the `ITD/backend/tests` directory and test the API via simulated requests. They are commented describing the sequence of operation being tested.

- `account_api_test.rs` simulates customer account registration and login.
- `get_ticket_test.rs` simulates the process of acquiring tickets and retrieving them.
- `full_enter_exit_test.rs` simulates entrances and exits of a shop with one staff member and three customers, covering both out of order access and full occupancy.

Continuous integration

GitLab CI is used as a Continuous Integration platform. Every commit triggers the execution of the workflow specified in the `.github/rust-ci.yml` file. The workflow starts the full backend environment in a containerized Ubuntu environment, together with a Redis and a PostgreSQL container. It runs the database migrations then builds the executable and runs all tests. If any test fails or there are errors during the build process, the commit will be marked as broken signaling that it should not be merged into main and that it should be fixed.

Installation instructions

This section provides two different approaches to build and install the CLup system.

The first one uses Docker and docker-compose to automate all the steps from development environment setup and requirement installation to deployment. It provides a way to automate all the build process while allowing deployment and behaviour consistency on different platforms since, even on different operating systems, the CLup system will run in a container with the environment it was built for. This is the recommended way to build and install CLup for deployment and testing.

The alternative way to build and install CLup involves setting up the build environment and configuring services manually. This method is recommended for CLup developers since it's significantly faster for consecutive builds, however the environment setup can be more cumbersome since manual installation limits automation possibilities.

Using docker-compose (recommended)

- Install docker: https://docs.docker.com/get-docker/ (all commands will assume the user is memeber of the `docker` group, refer to the guide for instructions)
- Install docker-compose: https://docs.docker.com/compose/install/
- Change to the ITD directory `cd ITD/`
- Build `docker-compose build`
- Run `docker-compose up`
- The clup service will be accessible at `0.0.0.0:8080` (can be changed by editing the `docker-compose.yaml`)
- The API can be directly accessed at `127.0.0.1:5000` if needed

Installation only

The `DeliveryFolder` includes prebuilt docker images for both the backend and the frontend. (For UNIX systems the `run.sh` in the `DeliveryFolder/ITD/artifacts` directory can be used for a one command execution)

- Change to the artifacts directory `cd DeliveryFolder/ITD/artifacts`
- Load the prebuilt images `docker image load -i clup-frontend.tgz; docker image load -i clup-backend.tgz`
- Run `docker-compose up`
- The clup service will be accessible at `0.0.0.0:8080` (can be changed by editing the `docker-compose.yaml`)
- The API can be directly accessed at `127.0.0.1:5000` if needed

> Note: for both build and install and intstallation only, the `SESSION_KEY` and `ENCODING_KEY` should be set or they will use (insecure) default values

Building and installing (manual) - Backend

Requirements

- **Rust** https://www.rust-lang.org/tools/install
- **Sqlx CLI (Optional)** `cargo install sqlx-cli --no-default-features --features postgres`
- **PostgreSQL** https://www.postgresql.org/
- **Redis** https://redis.io/

Environment variables

Configuration for the main binary is supplied through environment variables. Example:

```
DATABASE_URL="postgresql://user:pass@localhost:5432/clup_sqlx"
REDIS_URL="127.0.0.1:6379"
SESSION_KEY="0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f"
ENCODING_KEY="0f0f0f0f"
API_URL="0.0.0.0:5000" # defaults to "0.0.0.0:5000"
```

When running the application server binary environment variable configurations will be read from a `.env` file, if present.

Working directory

```
cd ITD/backend
```

Build

Note: PostgreSQL must be running and the `DATABASE_URL` variable must contain a valid url with correct user and password

```
cargo build
```

Test

Note: Most test require a PostgreSQL connection and Integration tests require a Redis connection (`REDIS_URL` environment variable correctly set)

```
cargo test
```

Install

Install the clup executable to `~/.cargo/bin` so it can be run by typing `clup` in the terminal (with `~/.cargo/bin` with `PATH`)

```
cargo install
```

Run

Build if necessary and run `clup`

```
cargo run
```

Building and installing - Frontend

Requirements

- **Node.js** https://nodejs.org/en/download/
- **Yarn** (or any other package manager for Node.js) https://yarnpkg.com/getting-started/install

Environment variables

```
VUE_APP_API_BASE_URL="http://localhost:5000" # should point to API_URL
```

Working directory

```
cd ITD/frontend
```

Install

```
yarn install
```

Run

```
yarn run serve
```

This command will start a development web server that is not suited for production.

Alternatively, you can run:

```
yarn run build
```

This will generate the static files for the web application that can then be served using a web server (such as Nginx or Lighttpd). The static files are generated and placed in the `dist` folder. The `docker-compose` method uses **Nginx** to serve the static pages and routes all `/api/` endpoints to the backend api.