Table of Contents

Breviary Project

Version 2020.06.26.01

TODO: Project description.

What This Repository Consists of

The repository consists of the following parts:

1. Code of Breviary including various modules - infrastructure deployment, backend service, client app.
2. Github actions for testing the code by performing the unit tests of the platform.
3. Technical overview of the framework. The documents and specification of Breviary, architecture and various usage examples.
4. This documentation.

Development

This document contains some basic information about project (Mild Blue) infrastructure, best practices and *must have* project parts.

Ask Marek or Lukas to give you all required accesses.

MildBlue Infrastructure

Project Repository

Source code is hosted in MildBlue GitHub.

Code Review

For code review of this project we are using MildBlue GitHub.

DevOps and Pipelines

For automatic tests, we are using GitHub Actions. Definition files are `.github`.

Task Management

Task management is provided via MildBlue Trello. Access can be given by some project administrator, but better is to ask anybody to add you into project group `breviary`.

Repository Management

For storing Docker images and Java modules we are using Docker Hub. Definition of all Docker images stored in Docker Hub must be stored in */docker_images* directory.

Password Management

For storing password and certificates we are using Bitwarden.

Application Logs

TODO LINKS

Development Environments

There exist multiple environments for development/test purposes:

Dev

This environment is used for development. It is automatically redeployed with every successful push and pipeline run into `master` branch.

TODO LINKS

Test

This environment is used for presentations. It must be redeployed manually.

TODO LINKS

Best Practices

Git

1. Branch name **should** start with task prefix, e.g., "`BREW-123` -add-teams-functionality".
2. Commit message **must** start with task prefix, e.g., "`BREW-123` New DB model added.".
3. Use full sentences for commit description in a form *Update of SOMETHING because of THIS.*. Try to describe it to be intelligible for others.
4. When doing code review, beware of single words such as *no, why, ?*. It is useless. Rather, write full sentences: *I think it should be done THIS way because of THIS.*. *Please, do it this way because of THIS.*, etc.
5. Try to make git tree well arranged, i.e., beware of many merging, etc.

Source Code

We use many tools to achieve a good code, such as *code review*, *tests*, *code analysis*, *code linting*, *automation*.

1. If there is something important/not cleared, make a comment.
2. If it is too complex, write it into *GitBook*.
3. Write tests for everything if possible.
4. Automate everything if possible.
    i. For common tasks such as build run, use *Makefile* tasks.
    ii. For more complex tasks create a script and put it into *scripts* directory.

Project Setup

To be able to develop and run project locally, some prerequisites must be satisfied:

- OpenJDK 11
- Docker
- Docker Compose

backend/src/main/resources/application.properties file

To be able to run `backend` application, you need to have set all required properties. You can create `application.propeties` file or use `.env.template` file.

- Copy `.env.template` to `.env` file.
- Go to `backend/src/main/resources`.
- Create symlink `ln -s ../../../../.env application.properties`.
- Set appropriate variables in `.env`. for local run, correct *DB* variables are enough. Example:

```
DB_USERNAME=postgres
DB_PASSWORD=breviary-db-pass
DB_URL=jdbc:postgresql://localhost:5433/postgres
```

Run Local DB

To run DB locally, there is prepared make task. To start DB in Docker running on port `5433` execute `make docker-start-local-db`. To stop DB, run `make docker-stop-local-db`.

Run Backend Locally

Just run app in your IDE. You have to run it twice on the first run.

Run Detekt

To run Detekt code style check, run `make detekt`.

Github Actions

Github Actions are defined in `.github`. All builds and tests run in `Docker` and uses `Docker compose`. For more details see this file.

With each PR, pipeline executing unit test is run. When pipeline of `master` branch successes, the code is pushed into GitHub.

Actions

There exist two pipelines:

TODO LINKS AND DESCRIPTION

Releases

TODO LINKS AND DESCRIPTION

Simple Action Workflow Diagram

```
@startuml

(*)  --> "Merge to master"
--> "Run Github action"
if "Action success" then
  -left-> [true] "Deploy to Staging"
else
  -> [false] (*)
endif

@enduml
```

Architecture

Base Overview

This section should describe base project structure and architecture.

Project Structure

Project can be separated into following parts (modules):

BuildSrc

Base Gradle module.

Backend

Spring Boot 2 REST API application.

Gradle

Gradle wrapper binaries.

Book

GitBook documentation.

Backend

Overview

Backend service is written in Kotlin and uses Spring Boot 2 as a framework.

Project Structure

Project structure follows common patterns for such application type, i.e., IoC, ORM, etc.

Gradle

Application uses Gradle as a build tool. All tasks such as `buidl`, `test`, `buildDistZip`, etc., are managed via Gradle.

Configuration

The main configuration is defined in `resources/application.yml`. This configuration uses environment properties for particular configuration. For this purpose, the exists `.env` file as mentioned in Development.

DB Migration

As a DB migration tool is used `Flyway` library. Migrations are stored in `resources/application.yml`. Name of the migration must follow name pattern `V[MIGRATION_NUMBER]__[MIGRATION_DESCRITPION_UNDESCORED].sql`. Migration is executed during each application start. So in case of invalid migration, application can fail to start.

Tests

As a test library is used `Kotlin tests`. Name of the test should describe what it is testing as a sentence, e.g., `Should test that data is deleted properly`.

Swagger REST API Specification Generation

Application generates *Swagger REST API Specification* to be able to implement (generate) client app with predefined types and route specification. This is done by special unit test `generateSwagger`. There is also make task `generate-swagger-json-to-client` that generates a JSON file with specification and copy in into `/client` folder.

Application Run

To run the application, do the following steps:

1. Create `.env` file based on template `.env.template` and set proper variables.
2. Execute `make docker-start-local-db` in root directory to start the database in Docker.
3. Run the backend application in Idea (`BackendApplication.kt`) or other IDE. (You have to run it twice on the first run.)

Deployment

This section should contain deployment of services to various servers/clouds, etc.

Infrastructure Deployment

Infrastructure deployment is provided via TODO scripts. See more details in TODO README.

Documentation of Documentation

Documentation is managed via gitbook.

We write markdown documentation as we feel necessary. Document each directory using `README.md` files and if possible, link them together via `SUMMARY.md` in the root folder. There may be multiple `README-*.md` files which can be linked from each other. However, each of the files must be mentioned (linked to) in `SUMMARY.md`.

Document also the main implementation details, aspects, which required the change of implementation along with the analysis/reasons for the new solution to avoid similar problems in the future.

Conventions

If you refer to a file (or pathname), mark it up like `file.extension`. You will not encounter *Markdown* problems with names like `__init__.py` which contain nested *Markdown* markup. Even if there is no special character in the name, stay to the convention to keep the documentation uniform.

If you refer to a module (or project) name, mark it up like this: *Module*

Creating the Documentation

To run the *gitbook server* from the root project directory use `make book-serve`. This `Makefile` goal just runs gitbook in docker. To automatically reload as you type, use `serve.sh` from `book` directory. To stop serving, run `make-kill`.

Generating PDF

To generate the documentation in a `pdf` form run the following command from the root directory:

```
make book-pdf
```

Running Natively

Install `node` and `npm` and just use `serve.sh`. Then navigate to http://localhost:4003 to see the document.

Although the documentation root is in `book`, the `serve.sh` script ensures the live reloading of the book even if the files outside of the `book` directory change. To do so, it uses `inoticoming` file watcher for relevant files and notifies gitbook to update, if a file has changed. Live reloading is an optional feature which does not necessarily work correctly in all cases (graphs/UMLs update). In case it did not run correctly, kill the `serve.sh` script and run it again.

Plugins

Katex and *PlantUML*.

Inline Math

$\int_{-\infty}^{\infty} g(x)dx$

Upon adding a new plugin, please run `rm -rf book/_book book/node_modules`.

Versioning

While updating documentation, raise its version with update description in `book/versions.md` and in `book/package.json`.

Operation Manual

TODO

Versions

- *2020.06.26.01* - Local run.