How to write a good backend

if you have technical questions, feel free to open a issue in biletado/biletado.gitlab.io

API v1

- select your API to start with, either `assets` or `reservation`
- each API has at least one request to the other one
- each API needs to check for authentication on some methods
- authorization will not be checked
- feel free when choosing programming languages or frameworks
- you may check the authentication via middleware, this includes maybe traefik
- you are not forced to use the equipped postgres-database as long as the API reacts like demanded
  - include your db-engine in the compose-environment

what to pay attention

containers

> 15%

- Pack your software inside a container so others can easily start your software in their own environments.
- adjust your copy of `biletado/compose/compose.yaml` by your needs.
- keep your containers simple, stupid. The smaller your runtime container is the smaller is the amount of potential security issues (i. e. use a build-container to create the runtime-image)
- choose good base-images
  - good choices are e. g. official docker-images, the ones built by bitnami or official images of your softwaredeveloper like bazelbuild for Bazel

script everything

> 5%

- create scripts for automated creation of your images, best is to use CI-integration like gitlab-ci or GitHub Actions

- `docker build` is not always the way to go, some frameworks like quarkus have their own way to build container-images

choose your framework wisely

> 15%

- use established web-frameworks which are designed for creating APIs
- probably some good ones are
  - Symfony (PHP)
  - Laminas (PHP)
  - Quarkus (Java/Kotlin)
  - Spring Boot (Java/Kotlin)
  - Golang stdlib (Go)
  - Gin Web Framework (Go)
  - Rust-Frameworks
  - django (Python)
  - Ruby on Rails (Ruby)
  - Dart-Frameworks
  - Typescript-Frameworks
- not so good ones are
  - PHP StdLib
  - Perl oneliners
  - javascript in general (IMHO)

secure your Queries

> 5%

- use always prepared statements to prevent SQL-Injection
- use an ORM or ODM for faster development

clean code

> 5%

- you are writing the legacy code of tomorrow, write it careful

- in half a year your own code is as far away from yourself like to any co-worker
- write self-documenting code
  - small classes and methods
  - good method- and variable-names
- use consistent code-style
- this means sadly, you should not use assambler, Brainfuck, 2k16 (😞) or most other esoteric languages

validate the JWT

> 10%

- load the public-key from the id-provider and check if the JWT is applicable and has a valid signature

tracing

> 5%

- respect the tracing-header and reuse it for further requests so that the spans can be followed when requesting through traefik
- have a look at how to implement tracing inside your application

configurability

> 5%

- make your software and container-images configurable so that they can run in different environments
  - this can be by environment-variables or config-files which can be mounted into the container
  - e. g. the database-connection parameters or the HTTP-endpoints where requests are getting send to

licensing

> 1% (bonus)

- choose a appropriate license for your software to make it easy for others to reuse your code
  - e. g. MIT or BSD-2-clause

version-control

> 1% (bonus)

- use the version-control-system of your choice to make it easy to collaborate
- if you choose the right one, you get a container-registry and CI/CD for free

write documentation

> 5%

- document your project, especially your configuration-parametes and how to use your container
- a readme-file inside the code-repository is always a good choice

testautomation

> 5%

- include some tests inside your CI/CD-process, e. g. unit-tests, linting and/or automated API-Tests
- test your project before or after building and don't deliver it if the tests fail

logging

> 5%

- write useful logs to stdout and stderr with different, configurable loglevels to help yourself and other developers when debugging
- worth to have a look at: Elastic Common Schema

ah, and implement the API

> 20%

- CRUD-Operations
- Do good input-validation *before* communicate with other APIs or Databases
- use always descriptive return values, i. e. choose the right HTTP-Status and return meaningful errormessages if necessary