# Purchase API Design

This document explains the design choices used in this project.

The design was influenced by several requirements:

1. Must be written in Java, showcasing job-relevant knowledge to the examiner.
2. The task involves setting up an API service to store and retrieve purchase transactions.
3. The application “should be runnable without installing additional software stack components, such as databases, web servers, or servlet containers (e.g., Jetty, Tomcat, etc.)”
4. Time constraints

**Spring Boot app**: as the job I’m applying for is looking for Spring Boot expertise and this framework is well-suited to small API services, I decided early on to develop a Spring Boot app solution.

For the rest of the tech stack, requirement 3) *“the app should be runnable without installing addition software”* limited options. At my current workplace, we use EBean ORM for persistence and I have significant experience in it. I prefer EBean as it abstracts away from EntityManager (making it more straightforward to write transactions. Ebean has no entity manager and no persist/merge/flush lifecycle methods and the persistence context is not involved in persisting.), has better control over generated SQL, and can generate Flyway/LiquiBase-esque DB migrations. For tests and running locally, EBean runs in a Docker container usually. As additional software is not allowed, I could not use EBean. The natural alternative was to use Spring’s default DB option of a H2DB in-memory database (to not require a database to be installed or additional software like Docker) and Hibernate ORM.

Given the use of H2DB, the tech stack chosen for this project was:

* **H2 in-memory SQL database and Hibernate ORM, Java 11, Spring Boot 2, Lombok, Maven + JUnit 5, Mockito for automated test suite**

The tech stack I’m using in my current role is:

* **Ebean ORM, Java 17, Spring Boot 3, Gradle** (much faster building than Maven)

Due to the limitation of using H2 DB, I felt it prudent to downgrade to Java 11 (and by extension Spring Boot 2) and Maven, as most tutorials/troubleshooting guides have not been updated to Java 17 and only use Maven. This would also avoid any needless complications with libraries that haven’t caught up with Java 17 (e.g., Jakarta changes) and getting stuck resolving those problems, when the main goal of this test is to work on the requirements.

**OpenAPI:** additionally, I define an api-sec.yaml file and use Swagger/OpenAPI to autogenerate API stubs for my code. I am a big fan of OpenApi, which along with **Lombok** saves a lot of developer boilerplate. More importantly, it makes the service more extensible for future API changes. The api-sec.yaml API definition is human-readable and easy to change. The autogenerated code can also be configured to automatically do a lot of validation - like checking that all requires parameters are present and in the right format (e.g., string length).