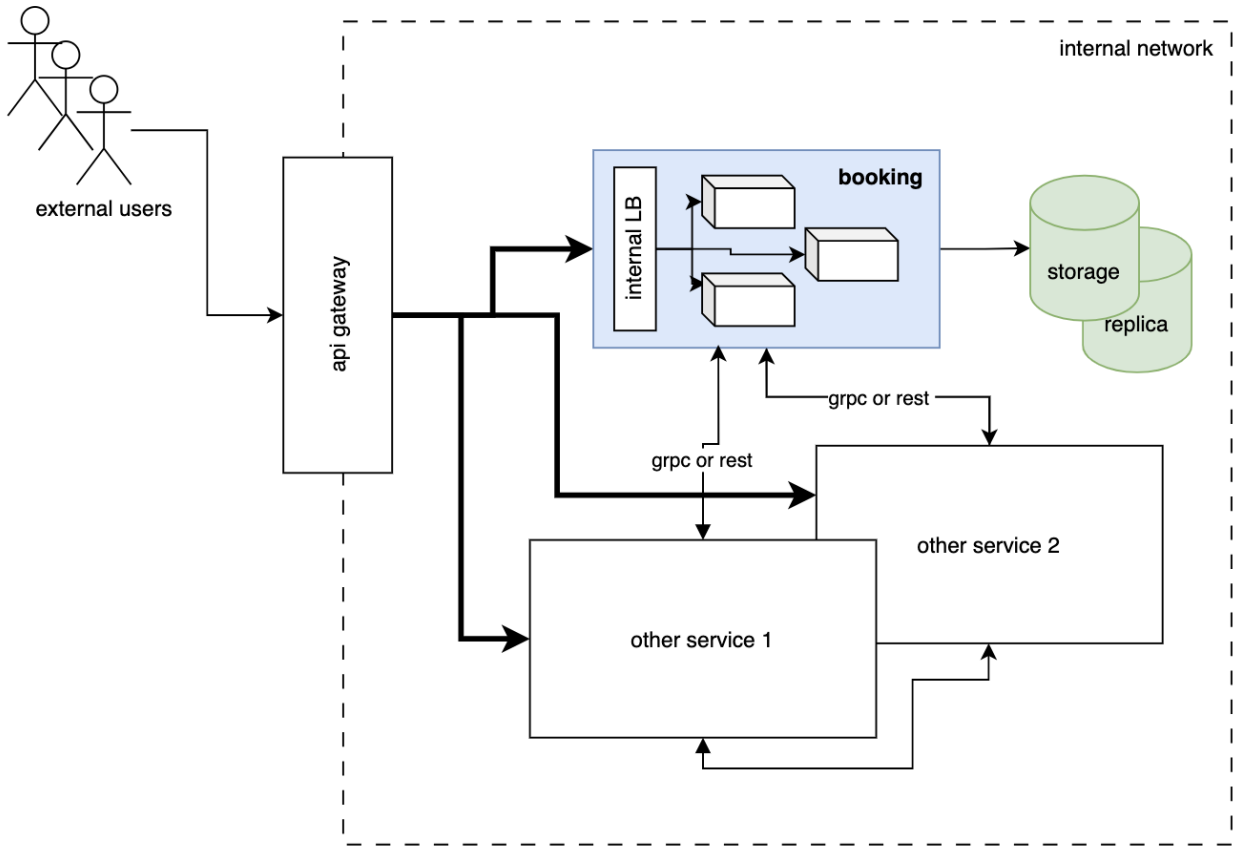
Feature Toggle API and App

Testing Strategy

System design (as understood on the requirements)



The above diagram illustrates several decisions that were considered to improve the reliability, functionality and security of the booker service as detailed below:

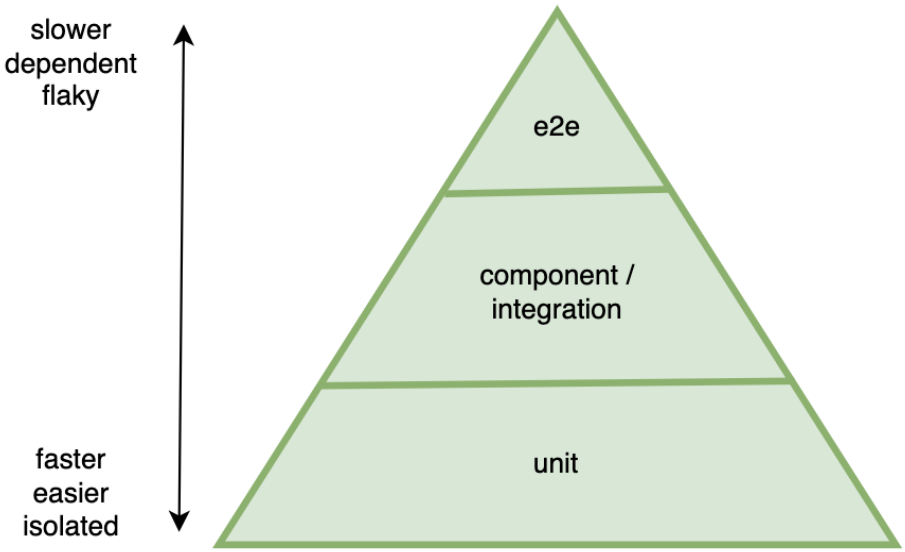
- An api gateway will provide access to external services. In this way we can perform rate limiting, security checks (such as blacklisting or whitelisting) and overall control of the access by third party services.
- The booker service will provide an internal load balancer (such as a K8s service) so that the service can scale up and down individually. This will allow it to be resilient and cost effective at the same time.

- To ensure high availability of the data the booker service storage database will consist of at least 2 instances. A primary instance and a replica instance with failover setup. This split would also allow the service to make efficient usage of write/read operations in cases where the replica could become highly accessible for analytic operations without affecting the performance of the app.
- The communication between internal services can be done via gRPC or Rest endpoints. In both cases the communication can be done directly and the recommendation is to use gRPC due to its improved performance capabilities.

Given the above details about the implementation, there are many details that we will consider during the definition of the strategy plan. For example, but not limited to:

- Security testing: This ensures that only users with the right privileges and permissions can access the corresponding apps. It's important to verify that the system is secure and protected against unauthorized access.
- Performance testing: This ensures that the system under test can handle the requests from multiple consumers without experiencing performance issues. It's crucial to evaluate the system's performance under different load conditions to identify any bottlenecks or scalability concerns.
- Functional testing and proper API design: As depicted in the architecture design, the API not only serves internal services but also needs to communicate effectively with other internal consumers. It's essential to design the API in a way that promotes proper communication and functionality. Some advantages of a well-designed and standardized API include:
    - Predictable behavior: Consumers will be able to anticipate the response and error codes returned by the API, allowing for better error handling and integration.
    - Easy addition of new consumers: With a properly designed API, integrating new consumers into the system becomes straightforward. The standardized interface facilitates the onboarding process for new consumers.
    - Consistency of behavior: A well-designed API ensures consistency in behavior across different consumers and endpoints. This consistency simplifies maintenance and reduces the chances of compatibility issues when making changes or updates.

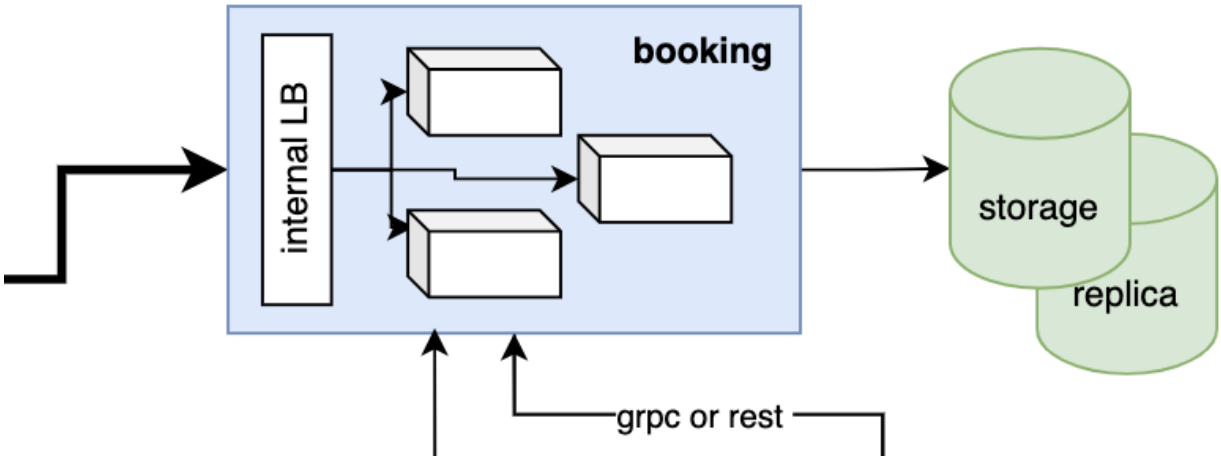
Testing approach



The "Testing pyramid" is a set of principles that offers valuable guidance on where to concentrate testing efforts. As shown in the illustration, the key objective is to prioritize testing at lower levels, as it brings greater benefits and shortens the testing feedback loop. By focusing on lower-level tests, such as unit tests, developers and engineers can quickly identify and address bugs before the code is executed. On the other hand, higher-level tests, like end-to-end or system tests, can be more prone to instability due to their reliance on proper system functioning and dependencies, such as data and integration with other systems. These complexities make higher-level tests less reliable and harder to maintain.

It is crucial to recognize that the testing pyramid can be applied recursively, extending to various levels within the codebase. This recursive application implies that even the code itself can be divided into units, components, and end-to-end scenarios, especially in the context of a running server. Therefore, it's important to approach the testing pyramid as a guiding principle rather than a rigid framework. The specific levels and strategies within the testing pyramid should be determined on a project-by-project basis, considering the unique requirements and characteristics of the software being developed. Flexibility is key in tailoring the testing approach to best suit the project's needs and objectives.

Testing types, tooling and strategies



Automated testing
We will define at least 4 different ways of executing automated checks to verify the assess the quality checks of our system:

- **Unit tests**: Would be developed by the engineers as part of the product development of the feature. As a developer working in the context of "*feature A*", it is easier to understand how to properly test a particular function or unit of code. Along with the expertise of other engineers (verified through PRs), developers and test engineers can make sure that the new code is being tested adequately on this level.
  The advantages of unit tests are hugely appreciated when working with complex code bases, because they are a way to explain what the code is doing. They also ensure that refactoring is made easier and finally expose implementation bugs that are harder to spot on higher levels.
  Tooling:
  - API: Java (or Kotlin) provides a huge list of libraries with testing capabilities that would allow us to do the testing, for example Junit or TestNG as test frameworks along with Mockito to allow for dependency mocking.

- **Api testing**: *A coded testing framework (**Java + jUnit + REST-assured,**)*. When doing api testing we don't need to choose the same programming language as the one used to build the application, but it is always recommended to consider it so that we will have the technical support from the engineers. The advantage of using a coded custom framework is the ability to manipulate data and interaction with other services in an easier fashion, which allow for extensive testing and verify more complex scenarios.

  Also, since the api would be well documented using OpenApi, we can generate a client for the language of our code using a code generator tool, and then simplify the code that we will need to write. Also by using a code generator continuously we can identify breaking changes in the specifications that affect the test and would also affect other

consumers.

Security testing
One of the main concerns of our system would be security testing, we will need to ensure that the access to the api is restricted and monitored. The recommended way to test security on our application is through **integration/api** testing because this is where we want to add the limitation in our app.

In case of endpoints protected by a security token, we need to ensure that proper authorization testing is done effectively with corresponding standards. For example, making a clear distinction between 403 and 401 error codes.

Since we provide an api gateway to limit access not only by credentials, we need to test the proper cases for compliance with the gateway rules such as user agent details, localization and api rate limiting.

We would also need to evaluate the use of penetration tools, security scanners and event static analysis tools such as ***SonarQube*** to help us identify potential flaws in the system.

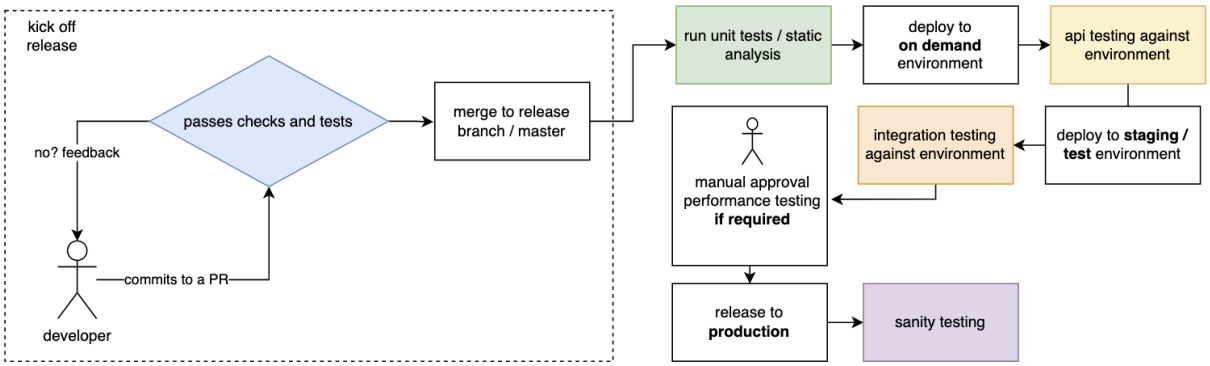
Performance and reliability testing
The *booker service* is a single point of failure in the higher picture of the architecture, therefore we need to make sure we can mitigate some risks by doing proper reliability tests such as:

- Load testing: Make sure the api can handle the proper load as requested by its consumers. Write and automate tests for:
    - Peak load testing.
    - Endurance testing.
    - Scaling testing.
- High availability: Ensure that proper fallback mechanisms are placed so that the system can become tolerant to partial failure.

The load test target numbers should be realistic to the expected load from the systems so that we can avoid the issue of overengineering our applications to handle an unexpected load. I would recommend using (**python + locust)**, as the load testing framework, because it provides enough tools to execute scalable testing while keeping the tests readable and extensible.

Test execution

We should take advantage of the CI/CD toolset to execute the testing. The priority is to run testing as part of the release pipelines and if possible define testing gates that would block the release if the automated test fails.



As described on the pipeline graph above each one of the types of testing can be executed as part of the release process. When the process is mature enough we can define a continuous delivery pipeline or can utilize a manual testing step where exploratory or other types of testing can be executed before releasing the changes to production.

Test suites

The diagram above describe at least 4 different test suites that we should maintain during the release pipeline:

- Unit tests suite/static tools: Once the code of a new feature is merged and ready for release the first step is to execute the full suite of unit tests to ensure that there were no merge conflicts.
- API tests: If possible spin a temporary environment that will allow for isolated testing of the service. At this point execute the service regression suite or integration tests to make sure the proper functionality of the running service.
- Regression integration tests: Considering the bigger picture where the '*booker service*' belongs to a bigger ecosystem it is important to execute E2E testing to guarantee integration cross service.
- Sanity tests: It is important to understand that environment configuration can affect the functionality of the application therefore it is always recommended to run a smaller suite of sanity checks in production to ensure liveness and functionality without affecting real life data.

Manual test execution

When collaborating with multiple stakeholders, such as Business Analysts, Project Managers, and other Engineers, it is essential to reevaluate the release process to ensure convenience for all parties involved. Several considerations should be taken into account:

- Performance / Security Testing**:** When introducing new code that could potentially impact performance or security, it becomes crucial to conduct the recommended tests. This ensures that any performance implications are identified and addressed before the release.
- Manual Testing for Third-Party Dependencies**:** Automation may not cover all aspects, especially when dealing with third-party dependencies that require specific data or

configurations. In such cases, manual testing plays a vital role in verifying the functionality of the system, providing comprehensive validation.

- Leveraging Acceptance Testing: If acceptance testing is a requirement from the business, the release window can be utilized to execute the necessary tests. This enables us to ensure that the software meets the specified business criteria and gains approval for deployment.

By considering these factors, we can enhance the efficiency and reliability of the release process, fostering better collaboration among stakeholders and delivering a robust and high-quality software product.

Observability

Once a feature is deployed to a production environment, it is vital to recognize that testing should not cease. The engineering team, comprising both QA and developers, bears the responsibility of incorporating essential observability features into the system. These features include metrics, reports, logging mechanisms, and real-time tracking graphs, which collectively ensure the ongoing and smooth operation of the system.

Definition of test artifacts

Test cases
Bug reports
Production issue reports

Note for the reviewers

The assessment document above primarily focuses on an overarching testing strategy for a specific application. While some assumptions about the technology and system design may differ from reality, it acknowledges the potential for strategy adjustments accordingly.

I did not want to write test cases about any specific details, but focused more on the interaction of the API with its consumers and quickly wrote a strategy that we could apply to ensure shift left testing, leveraging part of the effort on the development team and quality mindset within the team.

Additionally, there is a brief mention of the technologies that could be utilized for testing. It is acknowledged that selecting the most suitable testing tools depends on various factors such as budget, tech stack, and the team's preferences. Choosing the appropriate tools without comprehensive details would be unrealistic.

I would have been able to sit in front of this document and extend the strategy for hours, but I wanted to keep it as short but good as possible.

Finally I left out some things that might be important with more context, but can also be trivial such as:

- Objectives of the test strategy document
- Roles and responsibilities
- Test data management
- Test case design
- Defect management
- Risk assessment
- Metrics