Software architecture document



Contents

[Introduction 1](#_Toc161051808)

[Non-FRs 1](#_Toc161051809)

[System context – C1 3](#_Toc161051810)

[Containers and technology – C2 3](#_Toc161051811)

[Why microservices 4](#_Toc161051812)

[Why event-driven? 5](#_Toc161051813)

[Security 5](#_Toc161051814)

[Analysis 5](#_Toc161051815)

[Design 6](#_Toc161051816)

[Implementation 7](#_Toc161051817)

[Validation 8](#_Toc161051818)

[CI/CD setup 8](#_Toc161051819)

# Introduction

This document provides a comprehensive architectural overview of the system, using a number of different views to depict different aspects of the system. It is intended to capture and convey the significant architectural decisions which have been made to implement the software solution, based on the FRs and Non-FRs. It is also incredibly useful as it gives an oversight of the entire system and exposes any potential pitfalls the application might experience.

# Non-FRs

|  |  |
| --- | --- |
| **Functional requirement** | **Description** |
| **SSD-5: Establish identity internal and external users** | The application determines the identity of external and internal users on the basis of a mechanism for identification and authentication, wherein the authentication data in a consolidated authentication facility are managed. |
| **SSD-7 Segregation of functions** | The authorizations of users (including administrators) within the application is arranged so that permissions can be assigned functionalities and separation of incompatible authorizations is possible. |
| **SSD-12B: Session termination** | The application terminates a session after a set period of inactivity by the user through automatic session termination. |
| **Stability** | Application must not contain any major bugs that degrade the user experience. A request to the client must not take longer than 2 seconds and the app can handle 100 000 concurrent requests/sec. |
| **Availability** | The application must be able to support many concurrent users (aiming for 1 000 000) at once. |
| **Downtime** | The application must experience minimal downtime in case of a technical issue |

# System context – C1

A screenshot of a computer

Description automatically generated

From the system context diagram it can be seen that the software is going to make use of an external image server, which will be used for uploading all of the media to a remote Amazon S3 bucket. This is the most robust solution that allows media to be stored in the most efficient way possible so that the database or the microservices have to deal with the sending of large files. This will overall decrease the processing time, contributing to the Non-FR of system response time (less than 1 second).

# Containers and technology – C2

A screenshot of a diagram

Description automatically generated

From the diagram it can be seen that the architecture of the application is a blend between the microservice approach and the event-driven approach.

Its main modules are composed by:

* **Authentication microservice**, deals with the authentication and management of users
* **Listing microservice**, implementing functionality of listings and comments
* **Bid microservice**, implementing functionality of placing bids on listings
* **Media microservice**, dedicated to managing media in an S3 bucket
* **Message broker** which implements a Pub/Sub pattern through which the microservices communicate

## Why microservices

Microservices are a very popular architecture choice in enterprise solutions. This stems from their robustness and independent scaling. Having different business cases implemented in separated microservices means that they can be independent. Some of the most important benefits of a microservice architecture that were considered in order to fulfill the non-functional requirements are the following:

* Robustness - Because microservices are deployed independently, it's easier to manage bug fixes and feature releases. In many traditional applications, if a bug is found in one part of the application, it can block the entire release process. New features might be held up waiting for a bug fix to be integrated, tested, and published
* Mix of technologies - Teams can pick the technology that best fits their service, using a mix of technology stacks as appropriate
* Fault isolation - If an individual microservice becomes unavailable, it won't disrupt the entire application
* Security – Since each microservice has a smaller code base, it’s a lot easier to implement security by design
* Scalability – each microservice can be scaled independently.
* Data isolation – it is much easier manage each individual database because there is only one small microservice connected to it

## Why event-driven?

Having microservices communicate asynchronously through a message broker has several benefits that help towards the non-functional requirements:

* Loose coupling – each microservice can be independent and function on its own even if the other microservice are out of order.
* GDPR/User privacy – having a pub/sub pattern makes it a lot easier to deal with distributed data in the case of GDPR regulations and user privacy
* Performance benefits – asynchronous messaging leaves operations running in the background and does not block the processing time of a request, thus decreasing response time

All of the abovementioned benefits contribute towards the non-functional requirements of this project, more specifically:

1. Performance
2. Scalability
3. Security
4. GDPR/Privacy

# Security

For the different phases of the SDLC, the following security activities are considered:

## Analysis

**Risk assessment**

In the analysis stage, risk assessment is an integral part of the project’s development. Here potential risks are identified such as leaking of personal information and theft of sensitive data like personal IDs and bank details.

For example, in this project, if the bidding was connected to an actual payment flow and was not just symbolic, it would pose a risk of credit card details getting compromised.

However, since the state of this project resembles a social media, the risks are a lot lower and associated with the leaking of personal details like addresses or names.

**Defining security requirements**

In order to mitigate the risks, security requirements need to be identified by following information providers like CIA and OWASP.

**Including requirements in the functional and non-functional requirements**

After the security requirements are identified, they are listed as functional or non-functional requirements. In this case the non-functional requirements of the project are related to being able to distinguish between internal and external users, while also having access level control of the functional requirements.

Moreover, sessions need to be handled properly and terminated once the user is not active.

## Design

**Deciding upon credentials and roles**

In the design stage, it’s essential to map out how users are going to identify themselves and what their roles are going to be. In the project’s case, internal users are marked as ‘Administrators’ and have elevated privileges, while external users are marked as normal users and have low privileges by default.

**Deciding upon secrets**

Since this project is a one-man operation, project sensitive secrets are all kept safely in environment variables on the developer’s machine. Sensitive secrets are never committed to source code and are securely kept in GitLab CI/CD variables. If there were more developers, they would all have privileges according to GitLab’s standards, so only authorized developers can access secrets.

**How to secure data**

Security of distributed data is important in the event of a data leak. To mitigate this, data inside the application is marked using the CIA’s classification of data. If a certain piece of data is marked as confidential or personal, it is encrypted so that it can be secure.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Availability | Integrity | Confidentiality | Privacy |
| User passwords | 3 | 3 | 2 | p |
| User related info (mobile numbers, emails) | 1 | 1 | 0 | 0 |
| Listing information | 2 | 1 | 0 | 0 |
| Comments | 0 | 1 | 0 | 0 |
| Bids | 2 | 1 | 0 | 0 |

From the data classification table it can be seen that the only private and confidential information is user’s passwords, which are securely hashed in the database.

Other information regarding their activity on the application is protected for modification only by the users owning it but is open for the public.

If the application would implement real bidding and payment transactions, bank details would be confidential and end-to-end encrypted.

**Secure communication between services**

During the design process, it needs to be ensured that services communicate securely with each other. In the case of this project, the microservices are open to HTTP communication. However, certain actions are restricted and require an authentication token. That token can only be acquired by logging in the system, which requires knowledge of the user’s password. Therefore, HTTP communication from the frontend to the microservices is secure.

As for between the microservices, the message broker is secured by requiring authentication via credentials that are known only to the microservices. A malicious user cannot tap into the message broker without the credentials. Therefore, the only way to publish a message to the broker is through HTTP and a microservice, which is an established secure flow.

## Implementation

**Code reviews**

During implementation it is essential to conduct code review. This way, the knowledge of several experienced developers can be combined into one. In the context of this semester, code reviews are always conducted between fellow classmates to ensure that code is up to quality and is secure.

**Input validation and output encoding**

Any data that is subject to being input in the system is validated. Output from the system is also encoded in a safe JSON format and rendered using JSX, which mitigates risks from bad actors like code injection and XSS attacks. The application also avoids displaying user-generated contents with HTML directly. The user input is always sanitized through the backend first.

**OWASP top 10 guidelines**

The OWASP top 10 vulnerabilities are always taken into consideration during the implementation stage of SDLC. This way the most common vulnerabilities experienced in software development can be avoided. Improvements made by following the OWASP top 10 can be seen in a separate document.

## Validation

**Vulnerability scanning**

During the validation stage of SDLC, vulnerability scanning is used in order to determine weaknesses of 3rd party libraries. In the project’s context, npm has this functionality built in. Here is what vulnerability scanning in npm looks like:

A screen shot of a computer

Description automatically generated

These vulnerabilities can be addressed by running the `npm audit` command. However, sometimes these vulnerabilities cannot be fixed automatically and manual intervention is required.

**Small-scale security tests (e2e and integration level)**

While tests like these are not strictly security tests, they are a great way of establishing authentication and authorization flaws, as well as data integrity and configuration errors. By these tests, exploits in the system can be identified earlier than a full security or penetration test.

**Automated security testing (OWASP ZAP)**

As a final step in the validation stage, the ZAP tool from OWASP is as an automated security scanner that runs security tests to make sure there are no critical vulnerabilities. The scanner is used both in local development and in the CI/CD pipeline.

Screenshot of ZAP results from a local run

A screenshot of a computer

Description automatically generated

# CI/CD setup