SOFTWARE ARCHITECTURE DOCUMENT

Contents

Introduction

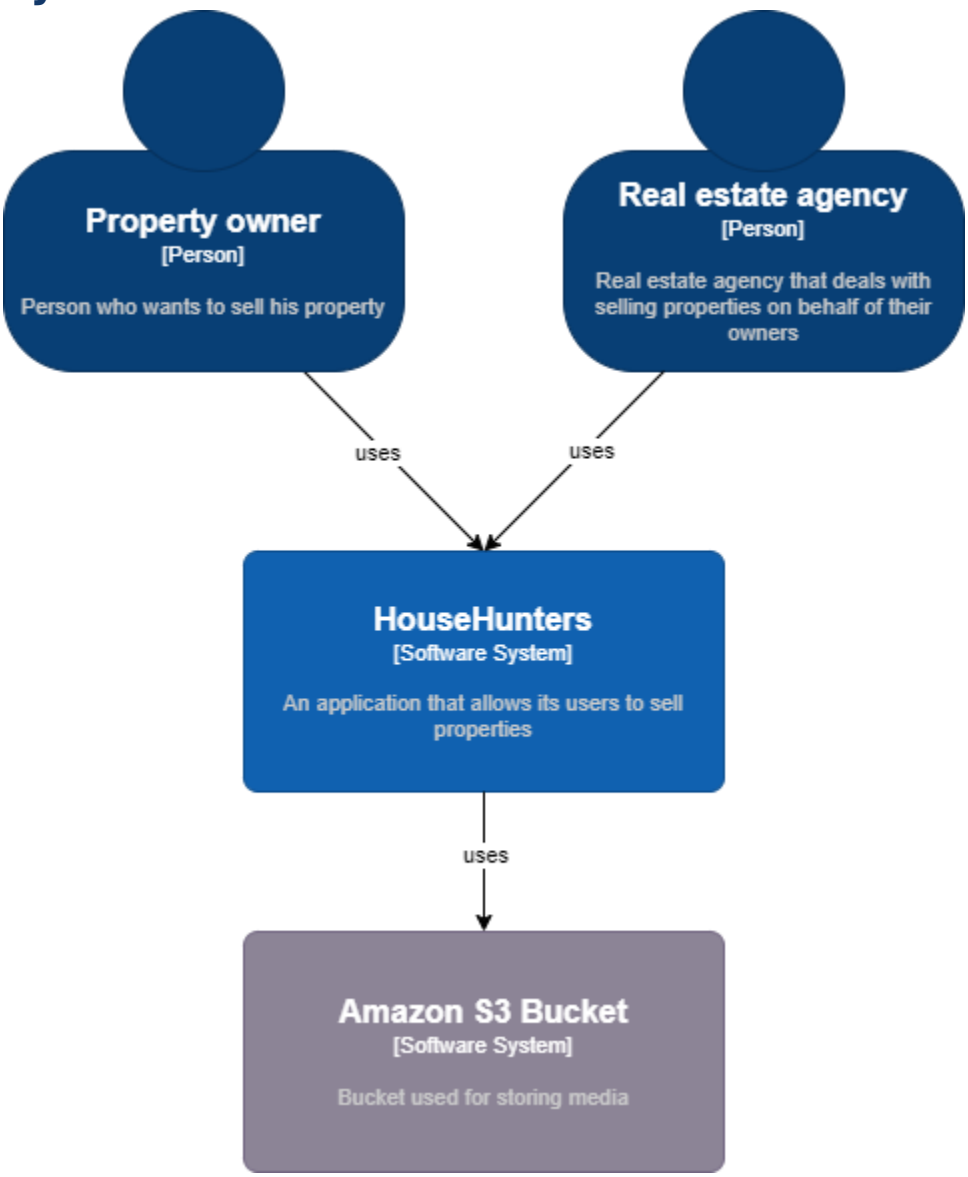
This document provides a comprehensive architectural overview of the system, using a number of different views to depict different aspects of the system. It is intended to capture and convey the significant architectural decisions which have been made to implement the software solution, based on the 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.

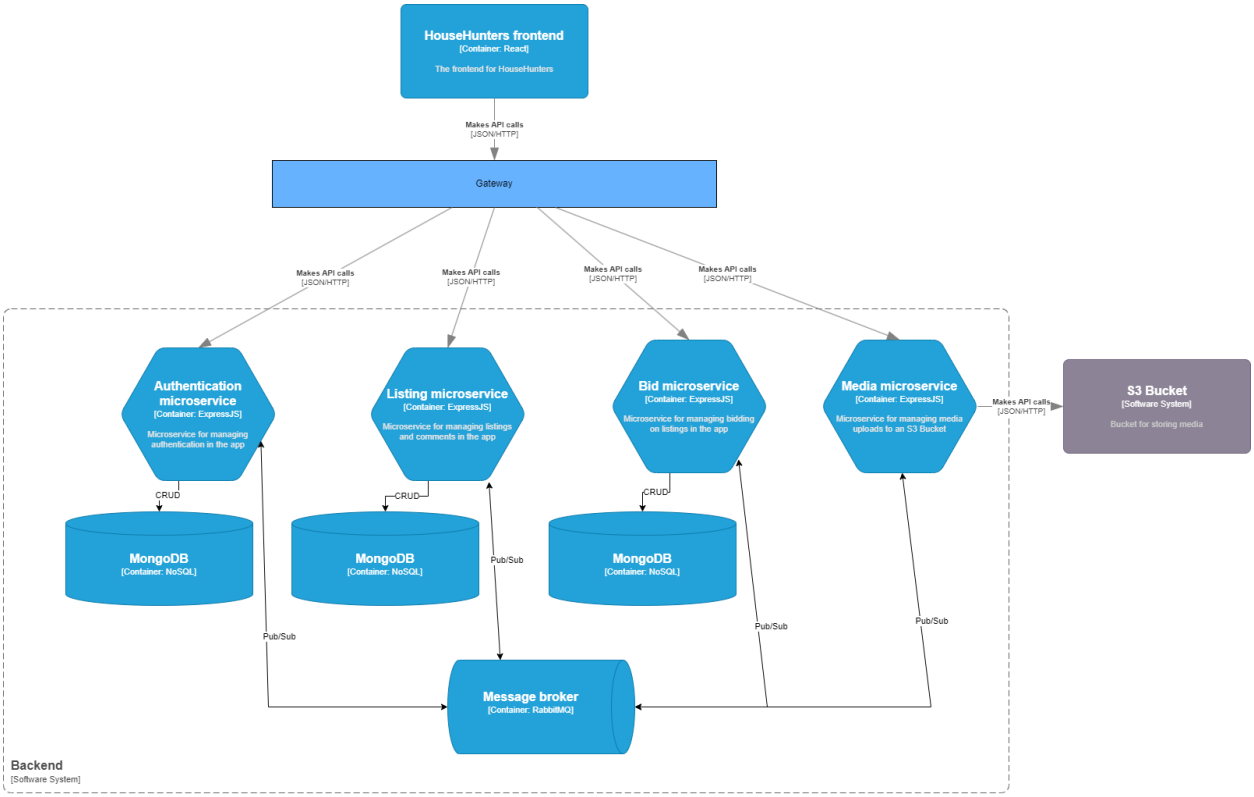
SSD-12B: Session termination	The application terminates a session after a set period of inactivity by the user through automatic session termination.
Stability	Application must not contain any major bugs that degrade the user experience. A request to the client must not take longer than 2 seconds and 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



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



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