Software Architecture Document

(SAD)

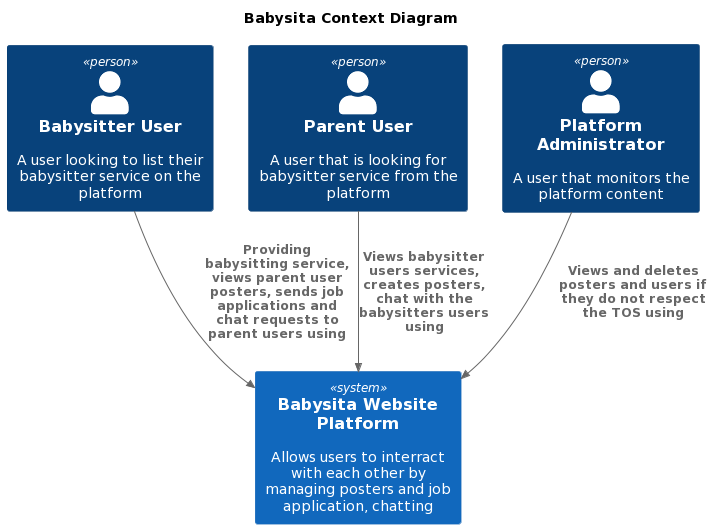
**Table of Contents**

1. Introduction
2. System Context (C1)
3. Containers and tech choices (C2)
4. Components (C3)
5. UML Class diagram
6. Sequence diagram
7. CI/CD pipeline diagram

**Introduction**

This document aims to describe the software architecture, delving into its functioning and the rationale behind the design decisions. It provides a detailed walkthrough of the C4 model, offering insights into each segment. The document ends with a Sequence diagram and CI/CD pipeline diagram.

**System Context (C1)**

****

**Containers and tech choices (C2)**

**A diagram of a babysitter

Description automatically generated**

The Babysita Website platform is built upon **three primary tiers** – Front End (FE), Back End (BE), and Database (DB). **Here are a few reasons for that**:

1. It gives you the ability to update the technology stack of one tier, without impacting other areas of the application. For example, if I decide to change from React to Angular the Backend and the Database will still work for the new frontend.
2. You can scale the application up and out. A separate back-end tier, for example, allows you to deploy to a variety of databases instead of being locked into one particular technology. It also allows you to scale up by adding multiple web servers.
3. It adds reliability and more independence of the underlying servers or services.
4. It provides an ease of maintenance of the code base, managing presentation code and business logic separately, so that a change to business logic, for example, does not impact the presentation layer.

For the FE, I opted for React. This decision was influenced by React's component-based code organization, which enhances readability and simplifies the process when dealing with numerous files. Additionally, the vast selection of libraries available and the supportive community further enrich its development capabilities. React's beginner-friendly attributes also played a role in this choice.

For the backend, I chose Java for my API due to its longstanding reputation of over twenty years, demonstrating both stability and maturity. Java-based applications, APIs included, can be used across multiple platforms without any changes. Moreover, Java provides a comprehensive collection of frameworks and libraries which facilitate API development (Spring Boot).

For the database, I selected MySQL. One of its strong suits simplifying integration with Java-based applications. MySQL is reputed for its swift performance, particularly when it comes to read-heavy tasks. It's tailored for web applications and can manage a significant volume of concurrent connections. Furthermore, its compatibility with multiple platforms such as Windows, Linux, and macOS provides versatility in deployment.

**Components (C3)**

**A diagram of a company

Description automatically generated with medium confidence**

For the BE software architecture of my project, I've chosen to follow the Semester 2 method, which emphasizes the use of service classes over use cases. This decision was primarily based on my comfort and familiarity with this strategy, as I perceive the use cases approach to be somewhat excessive for the context of my project.

The Spring Boot framework oversees the codebase. The choice for Spring Boot is grounded in its efficient dependency management, accelerated build process, automation capabilities, reduction in code requirements, and its built-in modules. At the business layer, Rest Controllers manage the sending and receiving of requests/responses. These are then relayed to the service classes responsible for the business logic, which, in turn, interact with the database (SOLID principles -> code separation and dependency inversion).

**UML Class diagram**

This is the full image of the source code of the BE without the relations due to enormous picture (**below key features are explained**)

If you want to check the full view of the diagram -> [Screenshot by Lightshot (prnt.sc)](https://prnt.sc/CLsu22jNJX4L)

**A screenshot of a computer

Description automatically generated**

1. **User functionality:**

The key features of the “Babysita” application make use of UserResponse/Request and UserEntity classes because of:

* Separation of Concerns -> UserEntity class is related to the database. UserResponse is converted UserEntity that is passed from the service class to the controller and the UserRequest is used for creating a new user.
* By using UserResponse, I can control the data exposed to external clients or users by excluding sensitive information like password.
* Validation: UserRequest includes validation rules and constraints specific to the data being transferred. This helps ensure that the data sent to the backend is valid and conforms to the expected format (less code in the service classes).

The "UserMapper" class serves as the bridge for converting between UserResponse, UserRequest and UserEntity representations, fulfilling a crucial role in the business logic by connecting the Controllers and the Repositories.

I have chosen to implement inheritance (all User classes) and an abstract parent class for the UserEntity and this decision is underpinned by compelling reasons:

* Futureproofing: By employing inheritance and an abstract parent class, ensure that the UserEntity class remains adaptable to accommodate potential future roles. This approach allows to seamlessly extend functionality without the need for extensive code modifications.
* Eliminating Code Redundancy: The utilization of inheritance helps us eradicate code duplication. We can define common attributes and behavior in the abstract parent class, preventing redundant code in each specific user role class.
* Abstract UserEntity Class: The UserEntity class is marked as abstract because it represents a generic role that doesn't exist as an independent entity. It serves as a foundational template for deriving more specialized user roles.

Key difference in the User feature is the fact that the parent UserEntity class is abstract while the UserResponse and UserRequest classes are not. This is since to create a new user the UserRequest class is created and later transformed based on the “role” to a Babysitter/Parent/AdminRequest. As for the UserResponse -> it is first fetched from the database and later converted to the child class for the chosen role (babysitter/admin/parent). However, UserEntity is abstract because it is being used only to save types of users in one table so direct implementation of the class is not needed.A screenshot of a computer

Description automatically generatedA screenshot of a computer program

Description automatically generatedA screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

1. **Authentication**

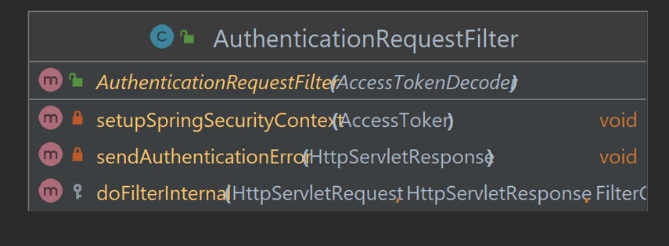
This represents one of the application's core features, leveraging JWT token-based authentication and authorization (OAuth is out of scope for now). Within the JWT token payload, I have chosen to store three crucial pieces of information: the user's ID, their role, and the token's expiration date. The rationale behind this decision is: simplifies frontend management for both web performance and security considerations, and it ensures that private data remains undisclosed.

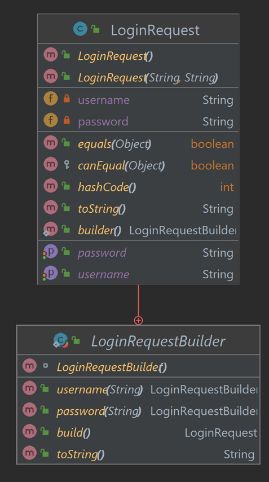
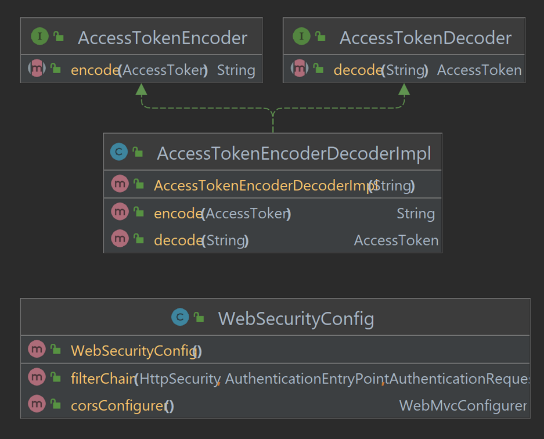
The authentication process involves two primary classes: "LoginRequest" and "LoginResponse." The "LoginRequest" class encapsulates user credentials, which are transmitted as a request from the client side. These credentials are then validated on the server side within the "AuthenticationRequestFilter" class. This filter verifies whether the provided credentials correspond to a defined user and compares the stored hashed password with the provided unhashed password after hashing it. If all validation steps succeed, an AccessToken is generated and sent as a "LoginResponse."

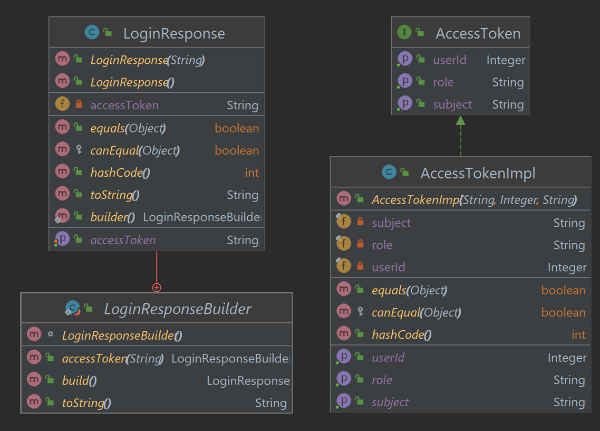
The responsibility for token management and preventing unauthorized access to the platform lies with the "AccessTokenEncoderDecoder." This component implements two interfaces. The reason for this design choice is the interdependency of their features, making it unnecessary to split them into two separate classes. This consolidation enhances code organization and maintains the integrity of the token-related functionality.

The central components responsible for implementing all the logic are the "AuthService" and "AuthController" classes. The decision to keep them separate from the User-related classes is driven by a commitment to maintaining single responsibility and enhancing code maintainability through a clear separation of functionalities.

Furthermore, this separation aligns with the principle of modularity, allowing each class to focus on its specific domain without becoming overly complex. In this context, authentication is distinct from user roles and functionalities, serving as the foundational key to unlocking these broader capabilities within the application. This architectural choice promotes clarity, ease of maintenance, and a more organized codebase. The communication between the FE and BE is controlled by the “WebSecurityConfig” class which manages the CORS configuration and the security of the HTTP by preventing unknown serves connection.

****

** **

**A screenshot of a computer

Description automatically generatedA screenshot of a computer

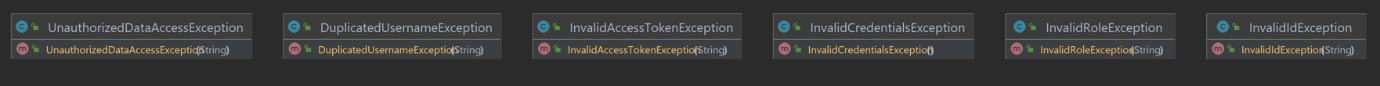
Description automatically generated**

1. **Exceptions and Exception Handler**

The "GlobalExceptionHandler" class is responsible for handling the custom exceptions, which are all derived from more general exceptions like "RuntimeException" and "ResponseStatusException." This implementation serves two primary purposes. Firstly, in the backend it assists with debugging. Instead of receiving generic exceptions that often lack a comprehensive explanation for the failure, it allows me to pinpoint the exact reason behind the issue. I specifically chose "RuntimeException" and "ResponseStatusException" as parent exceptions because of their functionality when dealing with status failures or runtime exceptions. Additionally, the exception messages are designed to be utilized in both the frontend and backend for user-friendliness. In the frontend, they help users understand why a particular functionality isn't working or if incorrect data has been provided.

A screenshot of a computer

Description automatically generated



1. **Job Application feature**

This represents the central feature of the application, serving as the crucial link between Parent and Babysitter users. Consequently, the code implementation includes services for Poster, Parent, and Babysitter.

During the creation process of a JobApplication, it is mandatory to provide three key identifiers: parentID, posterID, and babysitterID. These identifiers are essential because the JobApplication object is initiated by a specific individual (the babysitter) for a particular purpose (the poster), and it is intended for the attention of another individual (the parent). These identifiers establish the context and facilitate the seamless communication and interaction between the different users involved in the process. The use of DTO prevents issues with fetching data and enhances the application performance by reducing the amount of effort need to load a whole object (Poster/Parent/Babysitter Entity) with a simple integer (babysitter/parent/posterId).

A screenshot of a computer program

Description automatically generatedA screenshot of a computer program

Description automatically generated

**Sequence diagram**

A screenshot of a diagram

Description automatically generated

The sequence diagram illustrates the process flow of a babysitter applying for a job. It begins with the babysitter checking available posters, signifying the initial stage of seeking job opportunities. Upon finding a suitable poster, the babysitter decides to apply.

This action leads to an interaction where the babysitter applies to the poster and receives confirmation, marking the initiation of a formal job application process. Subsequently, the babysitter creates and sends a job application to the parent, with the status marked as "Pending." During this period, the babysitter retains the ability to view the poster.

An important aspect of the process is the communication between the babysitter and the parent, depicted as a chat loop, allowing for interaction before the job application's final status is determined.

The diagram then branches into two outcomes: if the parent finds the application satisfactory, they accept it, changing the application's status to "Approved." This approval grants the babysitter 10 points and results in the poster no longer being displayed on the posters page. Conversely, if the application is unsatisfactory, the parent rejects it, leading to the deletion of the application.

This diagram effectively captures the entire lifecycle of a babysitter’s job application, from the initial interest in a poster to the final outcome of the application process.

**CI/CD pipeline diagram**

**A screenshot of a computer

Description automatically generated**

**Developer:** Writes code and commits the changes to a version control system (GitLab).

**GitLab:** Provides a Git repository manager, a pipeline script (“.gitlab-ci.yml”) and triggers a GitLab Runner for the pipeline.

**GitLab Runner:** Runs jobs in the pipeline. This involves building the code, running tests + test report, building a docker image with the provided Dockerfile. After the image has been build, it is pushed to Docker Hub. The last job is to send the test report to SonarQube.

**Docker Hub:** Stores the image ([dankofontys/babysita-repo general | Docker Hub](https://hub.docker.com/repository/docker/dankofontys/babysita-repo/general)).

**SonarQube:** Reads the test report and creates a code review which displays information about the code quality and suggestions for improvements. Results are located at localhost:9000.