Faculty of Science University of Lisbon

Instant Messaging App
Software Architecture Document (SAD)

CONTENT OWNER: Rodrigo Bray nº66105, Mário Carvalho nº66129, Francisco Encarnação nº66131

DOCUMENT NUMBER:	RELEASE/REVISION:	RELEASE/REVISION DATE:
• 1	• 1.0	• 16 December, 2025
•	•	•
•	•	•
•	•	•
•	•	•
•	•	•

Table of Contents

List of Figures

No table of figures entries found.

List of Tables

1 Documentation Roadmap

The Documentation Roadmap should be the first place a new reader of the SAD begins. But for new and returning readers, it is intended to describe how the SAD is organized so that a reader with specific interests who does not wish to read the SAD cover-to-cover can find de-sired information quickly and directly.

Sub-sections of Section 1 include the following.

- Section 1.1 (“Document Management and Configuration Control Information”) explains revision history. This tells you if you’re looking at the correct version of the SAD.

- Section 1.2 (“Purpose and Scope of the SAD”) explains the purpose and scope of the SAD, and indicates what information is and is not included. This tells you if the information you’re seeking is likely to be in this document.

- Section 1.3 (“How the SAD Is Organized”) explains the information that is found in each section of the SAD. This tells you what section(s) in this SAD are most likely to contain the information you seek.

- Section 1.4 (“Stakeholder Representation”) explains the stakeholders for which the SAD has been particularly aimed. This tells you how you might use the SAD to do your job.

- Section 1.5 (“Viewpoint Definitions”) explains the *viewpoints* (as defined by IEEE Standard 1471-2000) used in this SAD. For each viewpoint defined in Section 1.5, there is a corresponding view defined in Section 3 (“Views”). This tells you how the architectural information has been partitioned, and what views are most likely to contain the information you seek.

- Section 1.6 (“How a View is Documented”) explains the standard organization used to document architectural views in this SAD. This tells you what section within a view you should read in order to find the information you seek.

1.1 Document Management and Configuration Control Information

- Revision Number: 1.0
- Revision Release Date: 16/12/2025
- Purpose of Revision: Creation of the document for the Software Desing course project with the first version of the Instant Messaging App documentation.

- Scope of Revision: Initial version completed with all sections completed.

1.2 Purpose and Scope of the SAD

This SAD specifies the software architecture for an instant messaging app**.** All information regarding the software architecture may be found in this document, although much information is incorporated by reference to other documents.

What is software architecture? The software architecture for a system[1] is the structure or structures of that system, which comprise software elements, the externally-visible properties of those elements, and the relationships among them [Bass 2003]. "Externally visible" properties refers to those assumptions other elements can make of an element, such as its provided services, performance characteristics, fault handling, shared resource usage, and so on. This definition provides the basic litmus test for what information is included in this SAD, and what information is relegated to downstream documentation.

Elements and relationships. The software architecture first and foremost embodies information about how the elements relate to each other. This means that architecture specifically omits certain information about elements that does not pertain to their interaction. Thus, a software architecture is an *abstraction* of a system that suppresses details of elements that do not affect how they use, are used by, relate to, or interact with other elements. Elements interact with each other by means of interfaces that partition details about an element into public and private parts. Software architecture is concerned with the public side of this division, and that will be documented in this SAD accordingly. On the other hand, private details of elements—details having to do solely with internal implementation—are not architectural and will not be documented in a SAD.

Multiple structures. The definition of software architecture makes it clear that systems can and do comprise more than one structure and that no one structure holds the irrefutable claim to being the architecture. The neurologist, the orthopedist, the hematologist, and the dermatologist all take a different perspective on the structure of a human body. Ophthalmologists, cardiologists, and podiatrists concentrate on subsystems. And the kinesiologist and psychiatrist are concerned with different aspects of the entire arrangement's behavior. Although these perspectives are pictured differently and have very different properties, all are inherently related; together they describe the architecture of the human body. So it is with software. Modern systems are more than complex enough to make it difficult to grasp them all at once. Instead, we restrict our attention at any one moment to one (or a small number) of the software system's structures. To communicate meaningfully about an architecture, we must make clear which structure or structures we are discussing at the moment—which *view* we are taking of the architecture. Thus, this SAD follows the principle that documenting a software architecture is a matter of documenting the relevant views and then documenting information that applies to more than one view.

[1] Here, a system may refer to a system of systems.

For example, all non-trivial software systems are partitioned into implementation units; these units are given specific responsibilities, and are the basis of work assignments for programming teams. This kind of element will comprise programs and data that software in other implementation units can call or access, and programs and data that are private. In large projects, the elements will almost certainly be subdivided for assignment to sub-teams. This is one kind of structure often used to describe a system. It is a very static structure, in that it focuses on the way the system's functionality is divided up and assigned to implementation teams.

Other structures are much more focused on the way the elements interact with each other at runtime to carry out the system's function. Suppose the system is to be built as a set of parallel processes. The set of processes that will exist at runtime, the programs in the various implementation units described previously that are strung together sequentially to form each process, and the synchronization relations among the processes form another kind of structure often used to describe a system.

None of these structures alone is *the* architecture, although they all convey architectural information. The architecture consists of these structures as well as many others. This example shows that since architecture can comprise more than one kind of structure, there is more than one kind of element (e.g., implementation unit and processes), more than one kind of interaction among elements (e.g., subdivision and synchronization), and even more than one context (e.g., development time versus runtime). By intention, the definition does not specify what the architectural elements and relationships are. Is a software element an object? A process? A library? A database? A commercial product? It can be any of these things and more.

These structures will be represented in the views of the software architecture that are provided in Section 3.

Behavior. Although software architecture tends to focus on structural information, *behavior of each element is part of the software architecture* insofar as that behavior can be observed or discerned from the point of view of another element. This behavior is what allows elements to interact with each other, which is clearly part of the software architecture and will be documented in the SAD as such. Behavior is documented in the element catalog of each view.

1.3 How the SAD Is Organized

This SAD is organized into the following sections:

- **Section 1 ("Documentation Roadmap") provides information about this document and its intended audience**. It provides the roadmap and document overview. Every reader who wishes to find information relevant to the software architecture described in this document should begin by reading Section 1, which describes how the document is organized, which stakeholder viewpoints are represented, how stakeholders are expected to use it, and where

information may be found. Section 1 also provides information about the views that are used by this SAD to communicate the software architecture.

- **Section 2 ("Architecture Background") explains why the architecture is what it is.** It provides a system overview, establishing the context and goals for the development. It describes the background and rationale for the software architecture. It explains the constraints and influences that led to the current architecture, and it describes the major architectural approaches that have been utilized in the architecture. It includes information about evaluation or validation performed on the architecture to provide assurance it meets its goals.
- **Section 3 (Views") and Section 4 ("Relations Among Views") specify the software architecture**. Views specify elements of software and the relationships between them. A view corresponds to a viewpoint (see Section 1.5), and is a representation of one or more structures present in the software (see Section 1.2).
- **Sections 5 ("Referenced Materials") and 6 ("Directory") provide reference information for the reader.** Section 5 provides look-up information for documents that are cited elsewhere in this SAD. Section 6 is a *directory*, which is an index of architectural elements and relations telling where each one is defined and used in this SAD. The section also includes a glossary and acronym list.

1.4 Stakeholder Representation

This section provides a list of the stakeholder roles considered in the development of the architecture described by this SAD. For each, the section lists the concerns that the stakeholder has that can be addressed by the information in this SAD.

Each stakeholder of a software system—customer, user, project manager, coder, analyst, tester, and so on—is concerned with different characteristics of the system that are affected by its software architecture. For example, the user is concerned that the system is reliable and available when needed; the customer is concerned that the architecture can be implemented on schedule and to budget; the manager is worried (in addition to cost and schedule) that the architecture will allow teams to work largely independently, interacting in disciplined and controlled ways. The developer is worried about strategies to achieve all of those goals. The security analyst is concerned that the system will meet its information assurance requirements, and the performance analyst is similarly concerned with it satisfying real-time deadlines.

This information is represented as a matrix, where the rows list stakeholder roles, the columns list concerns, and a cell in the matrix contains an indication of how serious the concern is to a stakeholder in that role. This information is used to motivate the choice of viewpoints chosen in Section 1.5.

Stakeholder	Concerns

Users	Availability, message latency, and responsiveness
Developers	Project structure and easiness for adding new features, code quality
Maintainers	Database structure, project structure, and deployment.
Evaluators (Professor)	Code quality, functional requirements, and non-functional requirements

1.5 Viewpoint Definitions

The SAD employs a stakeholder-focused, multiple view approach to architecture documentation, as required by ANSI/IEEE 1471-2000, the recommended best practice for documenting the architecture of software-intensive systems [IEEE 1471].

As described in Section 1.2, a software architecture comprises more than one software structure, each of which provides an engineering handle on different system qualities. A *view* is the specification of one or more of these structures, and documenting a software architecture, then, is a matter of documenting the relevant views and then documenting information that applies to more than one view [Clements 2002].

ANSI/IEEE 1471-2000 provides guidance for choosing the best set of views to document, by bringing stakeholder interests to bear. It prescribes defining a set of viewpoints to satisfy the stakeholder community. A viewpoint identifies the set of concerns to be addressed, and identifies the modeling techniques, evaluation techniques, consistency checking techniques, etc., used by any conforming view. A view, then, is a viewpoint applied to a system. It is a representation of a set of software elements, their properties, and the relationships among them that conform to a defining viewpoint. Together, the chosen set of views show the entire architecture and all of its relevant properties. A SAD contains the viewpoints, relevant views, and information that applies to more than one view to give a holistic description of the system.

The remainder of Section 1.5 defines the viewpoints used in this SAD. The following table summarizes the stakeholders in this project and the viewpoints that have been included to address their concerns.

Table 1: Stakeholders and Relevant Viewpoints

Stakeholder	**Viewpoint(s) that apply to that class of stakeholder's concerns**
Users	Component-and-Connector Viewpoint
Developers	Module Viewpoint, Component-and-Connector Viewpoint
Maintainers	Allocation Viewpoint
Evaluators	All Viewpoints

1.5.1 Component-and-Connector Viewpoint Definition

1.5.1.1 Abstract

The Component-and-Connector Viewpoint describes the runtime behavior of the application. It's used to identify the executing components and how they interact via connectors (connections such as HTTP, JDBC) to achieve functionalities.

1.5.1.2 Stakeholders and Their Concerns Addressed

Stakeholders and their concerns addressed by this viewpoint include:

- Users: Concerned with system availability, responsiveness, and latency.
- Developers: Need to understand the data flow, communication protocols, and how the Frontend interacts with the backend.
- Evaluators: Are concerned with functional requirements and runtime qualities like performance.

1.5.1.3 Elements, Relations, Properties, and Constraints

Elements are runtime components (e.g. React App, Database). Relations are the mechanisms for interaction (e.g. HTTP/1.1 requests, JDBC connections). Properties include characteristics like

latency, throughput, and the protocol version in use. Constraints include statelessness with respect to HTTP requests and database persistence.

1.5.1.4 Language(s) to Model/Represent Conforming Views

Views using Component-and-Connector viewpoint are represented using UML Component Diagrams for structure and UML Sequence Diagrams for behavior.

1.5.1.5 Applicable Evaluation/Analysis Techniques and Consistency/Completeness Criteria

Completeness/consistency criteria include (a) all runtime interactions are modeled by a connector, (b) components interfaces match the connectors attached to them. Applicable Evaluation/Analysis techniques include performance testing, scenario-based analysis, and security analysis of the data in transit.

1.5.2 Module Viewpoint Definition

1.5.2.1 Abstract

The Module Viewpoint describes how the system is structured as a set of code units (packages, classes, and layers). Shows the static organization of the software highlighting the separation between Frontend and Backend as well as the layering inside.

1.5.2.2 Stakeholders and Their Concerns Addressed

Stakeholders and their concerns addressed by this viewpoint include:

- Developers: Need to understand the project structure to implement new features and fix bugs.
- Evaluators: Concerned with code quality, modularity, and separation of concerns.

1.5.2.3 Elements, Relations, Properties, and Constraints

Elements are packages, classes, interfaces, and modules. Relations are "depends-on", "implements", and "is-part-of". Properties of elements include their name, responsibility, and visibility. Constraints include dependencies that are not allowed and layering rules.

1.5.2.4 Language(s) to Model/Represent Conforming Views

Views using Component-and-Connector viewpoint are represented using UML Class Diagrams and UML Package Diagrams for static organization and dependencies.

1.5.2.5 Applicable Evaluation/Analysis Techniques and Consistency/Completeness Criteria

Completeness/consistency criteria include (a) every source code file map to a module, (b) no circular dependencies exist between architectural layers, (c) interfaces are clearly defined for each module. Applicable analysis techniques include static code analysis to verify layering rules and code quality standards.

1.5.3 Allocation Viewpoint Definition

1.5.3.1 Abstract

The Allocation Viewpoint describes how the software elements are mapped into the hardware where they run.

1.5.3.2 Stakeholders and Their Concerns Addressed

Stakeholders and their concerns addressed by this viewpoint include:

- Maintainers: Concerned with deployment procedures, environment configuration, and database persistence;
- Developers: Need to replicate the environment locally for testing.
- Evaluators: Verify if the system is correctly containerized.

1.5.3.3 Elements, Relations, Properties, and Constraints

Elements include software artifacts and environmental nodes such as Docker Containers and the host machine. Relations are primarily "allocated-to" or "hosted-on". Properties include hardware requirements (memory, CPU), image versions, and network port mappings.

1.5.3.4 Language(s) to Model/Represent Conforming Views

Views conforming to the Allocation viewpoint are represented using UML Deployment Diagrams

1.5.3.5 Applicable Evaluation/Analysis Techniques and Consistency/Completeness Criteria

Completeness criteria include (a) every software component is assigned to an execution environment; (b) communication paths between nodes are valid. Applicable analysis techniques include deployment testing (e.g., docker compose up) and resource usage monitoring.

1.6 How a View is Documented

Section 3 of this SAD contains one view for each viewpoint listed in Section 1.5.

Each view is documented as follows, where the letter *i* stands for the number of the view: 1, 2, etc.:

- Section 3.i: Name of view.
- Section 3.i.1: View description. This section describes the purpose and contents of the view. It should refer to (and match) the viewpoint description in Section 1.5 to which this view conforms.
- Section 3.i.2: Primary presentation. This section presents the elements and the relations among them that populate this view packet, using an appropriate language, languages, notation, or tool-based representation.
- Section 3.i.3: Element catalog. Whereas the primary presentation shows the important elements and relations of the view packet, this section provides additional information needed to complete the architecturalpicture. It consists of subsections for (respectively) elements, relations, interfaces, behavior, and constraints.
- Section 3.i.4: Context diagram. This section provides a context diagram showing the context of the part of the system represented by this view packet. It also designates the view packet's scope with a distinguished symbol, and shows interactions with external entities in the vocabulary of the view.
- Section 3.i.5: Variability mechanisms. This section describes any variabilities that are available in the portion of the system shown in the view packet, along with how and when those mechanisms may be exercised.
- Section 3.i.6: Architecture background. This section provides rationale for any significant design decisions whose scope is limited to this view packet.

1.7 Relationship to Other SADs

Not applicable

1.8 Process for Updating this SAD

As this document describes a system developed within the scope of an academic project the maintenance and updates are expected to cease after the project submission date.

2 Architecture Background

2.1 Problem Background

As part of the ISEL's 2024/2025 Fall Semester bachelor's degree project for the course of DAW(Desenvolvimento de Aplicações Web), students were tasked with building a fully functional multi-user Instant Messaging (IM) system. The problem focused on creating a platform capable of supporting online communication between authenticated users, organizing conversations into channels, and enforcing access rules.

Beyond merely exchanging messages, the challenge centered on designing a complete system-from persistent data storage to backend logic and frontend interaction, capable of supporting real-world messaging requirements such as authentication, authorization, message ordering, channel privacy, and extensible user interfaces.

2.1.1 System Overview

The IM system consists of two major components:

The Backend Component

- A PostgreSQL relational database responsible for durable storage of users, channels, invitations, and message history.
- A JVM-based server application that:
    - Exposes an HTTP API.
    - Interacts with the database for all data operations.
    - Ensures domain rules such as access control, message integrity, and channel membership.
- Supporting infrastructure, such as load balancers.

Frontend Application

- A browser-based UI, enabling users to interact with the system's functionality.
- Responsible for sending authenticated requests to the backend API.
- Provides features such as:
    - Registration via invitation

- Login and session management
- Channel browsing, creation, joining, and leaving
- Viewing and posting messages

The architecture also considers future compatibility with alternative frontends (e.g., Android or iOS), requiring the backend API to remain platform-agnostic.

2.1.2 Goals and Context

The primary goal of the project was to design and implement a robust, maintainable, and extensible end-to-end messaging platform. This includes:

- **Enforcing secure and reliable user authentication.**
- **Supporting invitation-based account creation.**
- Managing public and private channels with distinct access levels (read-only vs. read-write).
- Ensuring users can view, search, join, and leave channels seamlessly.
- Enabling users to create and retrieve messages in an ordered, consistent manner.
- Providing a clean and functional frontend interface that supports all required actions.

Development Context

The project was completed in two distinct phases:

1. Backend Delivery (Due: 2024-10-19)
   Focused on implementing the API, data models, authentication, authorization, and core domain logic.
2. Frontend Delivery (Due: 2024-11-30)
   Focused on implementing the UI, making the frontend fully operational, and refining or extending backend behaviors as needed.

2.1.3 Significant Driving Requirements

Behavioral Goals

- Correctness: All domain rules such as channel membership, message ownership, and access control, must be correctly enforced.
- Consistency: Messages must be stored and retrieved in proper chronological order.
- Secure Authentication: Proper credential handling and token-based session management.
- Role-based Access: Different channel permission levels must be respected (public access, private read-only, private read-write).

Quality Attribute Goals

- Scalability: Architecture should allow the addition of load balancers or multiple frontend clients without major redesign.
- Maintainability: Clear separation between frontend and backend through a stable HTTP API.
- Reliability: Database-backed message persistence ensures no loss of data and stable behavior across sessions.
- Usability: Frontend must present an intuitive and simple interface for messaging workflows.
- Security: Data integrity and access control are central, especially due to private channels and invitation-based registration.

2.2 Solution Background

Overall Rationale: The system uses a two-tiered architecture (a JVM backend + a React SPA frontend) to separate concerns: backend enforces domain rules, persistence, and authentication; frontend provides the interactive UI and consumes a stable HTTP API. This separation supports maintainability, extensibility (mobile or alternate frontends), and scalability (backend can be load-balanced independently).

Persistence & Durability: PostgreSQL (configured via Jdbi in MessagingAppApplication.kt) stores users, channels, memberships, invitations, and message history. SQL queries in

JdbiChannelsRepository.kt and JdbiUsersRepository.kt implement durable operations and ordering guarantees (e.g., message ordering by created_at ASC).

Authentication & Session Management: Token-based sessions use UUID tokens (see UsersService.kt and UUIDTokenEncoder). Tokens are issued and revoked by UsersService and validated by AuthenticationInterceptor + RequestTokenProcessor. Tokens are returned as secure, httpOnly cookies in UserController.kt (login).

Domain Enforcement: Core domain behavior (username/password rules, invitation lifecycle, channel naming, membership roles) is encapsulated in domain classes and enforced by service layer logic (UsersService.kt, ChannelService.kt) and repository checks (e.g., ensuring unique invitation codes in UsersService.createRegistrationInvitation).

Real-time Messaging: Server-Sent Events (SSE) are used for near-real-time message delivery. ChannelController.kt exposes a listen endpoint and uses per-channel ChannelEmitter instances to broadcast messages to connected clients. The frontend subscribes using EventSource in ChannelsService.listenToMessages.

API Stability & Documentation: Controllers use DTO input/output models under http/model/input and http/model/output, and OpenAPI generation is configured (MessagingAppApplication.kt) so alternative clients can rely on a stable API.

Error Model: The API returns structured problem responses (RFC-style JSON problem objects) from Errors.kt and http/model/output/problems/*, enabling consistent error handling in the frontend.

2.2.1 **Architectural Approaches**

Layered Architecture (Presentation → Service → Domain → Repository):

Controllers in http/* handle HTTP concerns and error mapping (UserController.kt, ChannelController.kt).

Services (UsersService.kt, ChannelService.kt) implement use cases and coordinate transactions.

Domain modules (domain/*) encapsulate business rules (e.g., UserDomain, ChannelDomain).

Repositories (repository/jdbi/*) handle persistence via JDBI and SQL.

Rationale: This clean separation facilitates reasoning about responsibilities, unit testing each layer, and replacing layers independently (e.g., swapping JDBI for another persistence library).

Repository Pattern with JDBI:

Concrete data access is implemented in Jdbi*Repository classes. SQL is explicit (e.g., JdbiChannelsRepository.createMessage), enabling precise control over queries (indexes, ordering).

Rationale: JDBI offers simple mapping and clear SQL control; chosen over heavy ORMs for explicit SQL and fine-grained performance control for messaging workloads.

Token-based Authentication & Argument Resolution:

Authentication uses a RequestTokenProcessor and AuthenticationInterceptor to authenticate requests; AuthenticatedUserArgumentResolver injects the authenticated user into controller method parameters.

Rationale: Centralized auth logic decouples auth processing from controllers and enables flexible token sources (Authorization header or cookie).

SSE for Real-time Updates:

SseEmitter and ChannelEmitter implement publish/subscribe per channel. Rationale: SSE is simple for server->browser streaming and integrates with standard HTTP without requiring WebSockets for this project scope.

Design Patterns Observed:

Factory/Builder: UserDomainConfig configuration in MessagingAppApplication.kt.

Strategy/Policy: Domain validations (password, username) centralized in UserDomain.

DTOs/Adapters: http.model.* act as translation layer between domain types and wire format.

Alternatives Considered (and implicit rejection reasons):

WebSockets for bi-directional real-time messaging — rejected in favor of SSE for simplicity and because only server->client message broadcasting was required.

JWT tokens — not used; the code uses opaque UUID tokens with server-side lookup which simplifies revocation and rolling TTL logic (server controls token lifecycle).

Full ORM (e.g., Hibernate) — not used to keep SQL explicit and tests deterministic; JDBI + hand-crafted SQL gives tighter control for message ordering and pagination.

COTS :

Spring Boot (web layer), Spring Security primitives (password encoder), JDBI (persistence), SpringDoc/OpenAPI (API docs). Tradeoffs: Spring provides fast bootstrapping and testing support; JDBI reduces ORM complexity.

2.2.2 **Analysis Results**

Domain rule enforcement

- Username/password validation location
- Invitation uniqueness, TTL, and “used” semantics
- Role-based access rules in `ChannelService`

Security model

- UUID-based tokens
- httpOnly cookies + Authorization header support
- Token TTL, rolling TTL, and max-token limits

Consistency guarantees

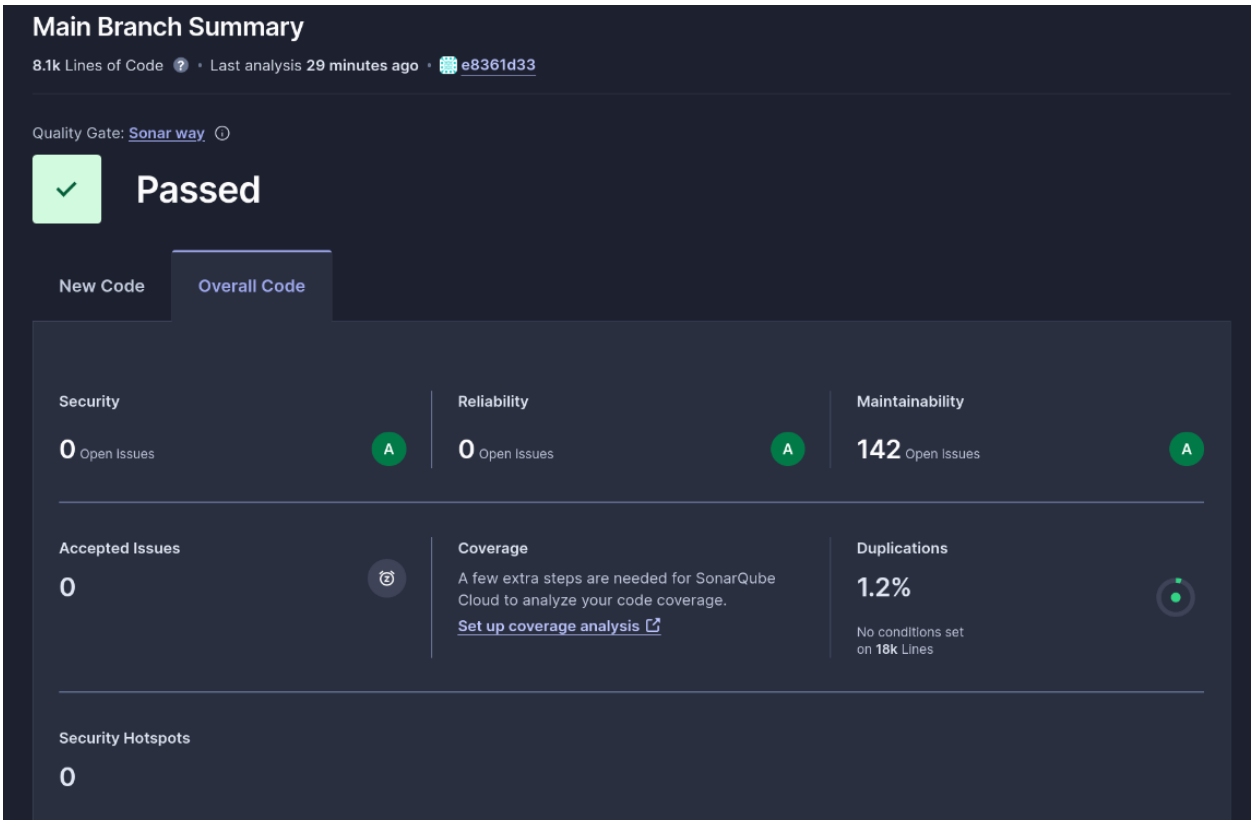
- Explicit ordering via `ORDER BY created_at`
- Server-side timestamping

Scalability awareness

- Stateless controllers
- DB-backed sessions
- Honest limitation note about SSE + in-memory emitter

Evidence-based claims

- Unit tests named and scoped
- Pagination to prevent unbounded queries



Evidence:
Our qualitative assessment is reinforced by automated static analysis using **SonarQube** on the main branch of the repository. The project **passes the Sonar Quality Gate**, with **no security or reliability issues**, low duplication (~1.2%), and an overall **A rating** across key quality dimensions. This independently confirms our suspicion that the codebase is well-structured, maintainable, and adheres to established best practices, providing additional confidence in the correctness and robustness of the implementation.

2.2.3 Requirements Coverage

Authentication & Secure Session Handling

- **Implemented in:**
  - UsersService.kt — token creation and revocation

    - UserController.kt — secure cookie issuance
    - AuthenticationInterceptor.kt, RequestTokenProcessor.kt — authentication enforcement
- **Configuration:**
    - MessagingAppApplication.kt (UserDomainConfig: token TTLs, rolling TTL, max tokens)

Invitation-Based Registration

- **Implemented in:**
    - UsersService.createRegistrationInvitation
    - UsersRepository methods (getRegistrationInvitation, registrationInvitationIsUsed, createRegistrationInvitation)

Channel Management (create / search / join / leave)

- **Implemented in:**
    - ChannelController.kt — REST endpoints
    - ChannelService.kt — application and use-case logic
    - JdbiChannelsRepository.kt — persistence layer

Role-Based Access Control

- **Implemented in:**
    - ChannelService.kt — role checks using ChannelDomain and MembershipRole
- **Enforced before:**
    - Inviting users
    - Creating messages
    - Removing memberships

Message Persistence & Ordering

- **Implemented in:**
    - JdbiChannelsRepository.createMessage
    - JdbiChannelsRepository.listMessages (ORDER BY created_at ASC)

Pagination & List Endpoints

- **Implemented in:**
  - ChannelsRepository and ChannelService
- **Configuration:**
  - Settings.kt (DEFAULT_PAGE, DEFAULT_PAGE_SIZE)
- **Mechanism:**
  - SQL LIMIT / OFFSET

Front-End API Integration & UI Flows

- **Implemented in:**
  - js/src/services/* (e.g., UsersService.ts, ChannelsService.ts)
  - js/src/pages/* (login, registration, channel views)

Real-Time Message Delivery (SSE)

- **Implemented in:**
  - ChannelController.kt — SSE listen endpoint
  - ChannelEmitter — server-side broadcast
  - ChannelsService.listenToMessages — frontend subscription

Error Handling & API Responses

- **Implemented in:**
  - Errors.kt
  - Problem models under http/model/output/problems/*
  - Controllers map service errors to RFC-style problem responses

2.2.4 Summary of Background Changes Reflected in Current Version

The current JVM-based backend is designed as a platform-agnostic HTTP API, making it highly reusable for mobile applications (Android/iOS) with minimal adaptation. The core domain logic, data models, and persistence layers are identical for both web and mobile products. Auth/Token Policy Tuned: Configuration in MessagingAppApplication.kt (token TTL, rolling TTL, max

tokens, invitation TTL) indicates iterative tuning of security/session behaviors — likely following feedback about session duration and token rotation.

SSE Added for Real-Time Delivery: The presence of ChannelEmitter and SSE endpoints shows the architecture evolved to include real-time delivery (instead of polling) — a decision to improve user experience without adding WebSockets complexity.

Explicit SQL & Pagination: Use of JDBI with explicit SQL and pagination suggests the architecture moved away from heavyweight ORMs to gain predictable performance and clear ordering guarantees for messages.

Error Modeling Standardized: The addition of many http/model/output/problems/* classes and a Problem responder implies a decision to standardize error payloads for consistent frontend behavior.

Testing Added: Service-level unit tests (e.g., UsersServiceTests.kt, ChannelsServiceTests.kt) added to lock down behavior and catch regressions; this is part of maturing the design.

2.3 Product Line Reuse Considerations

The current JVM-based backend is designed as a platform-agnostic HTTP API, making it highly reusable for mobile applications (Android/iOS) with minimal adaptation. The core domain logic, data models, and persistence layers are identical for both web and mobile products.

3 Views

This section contains the views of the software architecture. A view is a representation of a whole system from the perspective of a related set of concerns [IEEE 1471]. Concretely, a view shows a particular type of software architectural elements that occur in a system, their properties, and the relations among them. A view conforms to a defining viewpoint.

Architectural views can be divided into three groups, depending on the broad nature of the elements they show. These are:

- Module views. Here, the elements are modules, which are units of implementation. Modules represent a code-based way of considering the system. Modules are assigned areas of functional responsibility, and are assigned to teams for implementation. There is less emphasis on how the resulting software manifests itself at runtime. Module structures allow us to answer questions such as: What is the primary functional responsibility assigned to each module? What other software elements is a module allowed to use? What other software does it actually use? What modules are related to other modules by generalization or specialization (i.e., inheritance) relationships?

- Component-and-connector views. Here, the elements are runtime components (which are principal units of computation) and connectors (which are the communication vehicles among components). Component and connector structures help answer questions such as: What are the major executing components and how do they interact? What are the major shared data stores? Which parts of the system are replicated? How does data progress through the system? What parts of the system can run in parallel? How can the system's structure change as it executes?

- Allocation views. These views show the relationship between the software elements and elements in one or more external environments in which the software is created and executed. Allocation structures answer questions such as: What processor does each software element execute on? In what files is each element stored during development, testing, and system building? What is the assignment of the software element to development teams?

These three kinds of structures correspond to the three broad kinds of decisions that architectural design involves:

- How is the system to be structured as a set of code units (modules)

- How is the system to be structured as a set of elements that have run-time behavior (components) and interactions (connectors) ?

- How is the system to relate to non-software structures in its environment (such as CPUs, file systems, networks, development teams, etc.)?

Often, a view shows information from more than one of these categories. However, unless chosen carefully, the information in such a hybrid view can be confusing and not well understood.

The views presented in this SAD are the following:

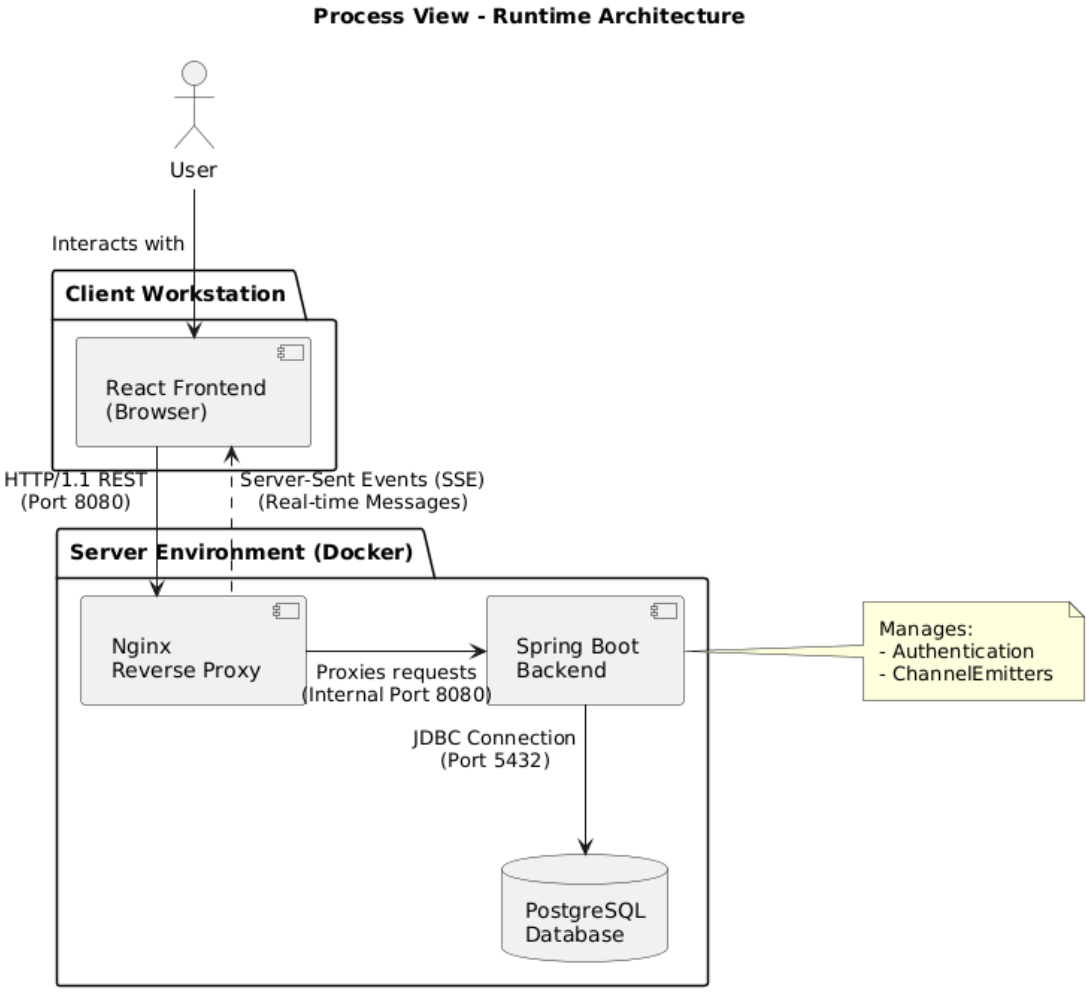
Name of view	**Viewtype that defines this view**	**Types of elements and relations shown**		**Is this a module view?**	**Is this a component-and-connector view?**	**Is this an allocation view?**
Component-and-Connector view	Component-and-Connector	Elements: Runtime components.	Relations: Connectors/Protocols.	No	Yes	No
Decomposition view	Module viewtype	Elements: Code units.	Relations: Dependencies.	Yes	No	No
Deployment View	Allocation viewtype	Elements: Software artifacts and Environment nodes.	Relations: "Allocated-to", "Hosted-on".	No	No	Yes
Data Model View	Module viewtype	Elements: Entities	Relation: Relations	Yes	No	No

3.1 Component-and-Connector View

3.1.1 View Description

This view describes the execution architecture of the Instant Messaging system. Its goal is to identify the major components in execution (Browser Frontend, Server Backend, Database) and the connectors who allow communication between them (HTTP, Server Sent Events). It's mainly used to identify the requirements for communication in real-time and data persistence.

3.1.2 Primary presentation



3.1.3 Element Catalog

Elements:

- React Frontend: SPA being executed in client browser. It's responsible for showing the UI e managing the session state.

- Nginx Reverse Proxy: Web server acts as a gateway for the system, receiving external requests and redirecting for the JVM app.

- Spring Boot Backend: Server app based on JVM. Processes the app's logic, authentication, validation of requests and exposes the HTTP API.

- PostgreSQL Database: Relational Database system responsible for data persistence (users, channels, messages, etc)

Relations:

- HTTP/1.1 REST: Request/Response protocol for all operations between client and server.

- Server-Sent Events (SEE): Unidirectional channel started by the client at the endpoint /api/channel/listen. It's used for the server to send asynchronously new messages for the frontend.

- JDBC: Protocol used by the backend to execute SQL queries in the database.

Interfaces:

- API REST: Interface accessible in /api.

- Database Schema: A relational schema (users, channels, invites, message) defined in the initiation scripts.

Behavior:

- Sending Message: The Frontend sends a POST request to the Backend, which will validate the token and permissions. If the request is valid, it will send a query to the database and notify Channel Emitter so that it can alert all the clients connected via SSE.

- Login: Frontend user sends the credentials via POST request, and the system validates them. If they are valid, it will reply with a cookie (authToken) and the flags Secure and HttpOnly.

Constraints:

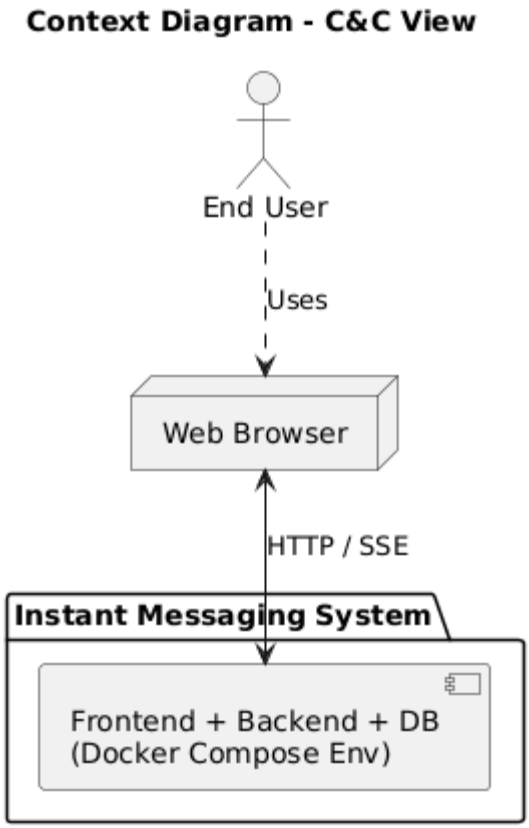
- The order of the messages in the chat only depends on the availability of the database.

- The SSE connections are kept open, and in case of network failure, the frontend is responsible for reconnecting.

3.1.4 Context Diagram

The context of the Component-and-Connector view is the Instant Messaging application system.

The external entities interacting with the system are the users who uses the web browser to access the application. The system acts as a containerized environment that exposes a standard HTTP interface to the outside world.



3.1.5 Variability Mechanisms

Environmental variables:

- The connection between the Backend and the Database is not hardcoded. It is configured via environment variables defined in the docker-compose.yml file.

Reverse Proxy Configuration:

- The routing logic is not in the application code. The nginx configuration file allows administrators to change listening ports, configure SSL/TLS termination, or adjust timeouts without recompiling the Java application.

3.1.6 Architecture Background

This section explains the rationale behind the key runtime architectural decisions.

Selection of Server-Sent Events (SSE):

- The primary requirement is server-to-client broadcasting (notifying users of new messages). Since the client (Frontend) sends messages via standard REST POST requests, a full bidirectional channel like WebSockets adds unnecessary complexity, and the purpose of the work was to use and understand SSE

Use of Nginx as a Reverse Proxy:

- It handles the "routing" of traffic and adds a layer of security by hiding the JVM application server from direct exposure.

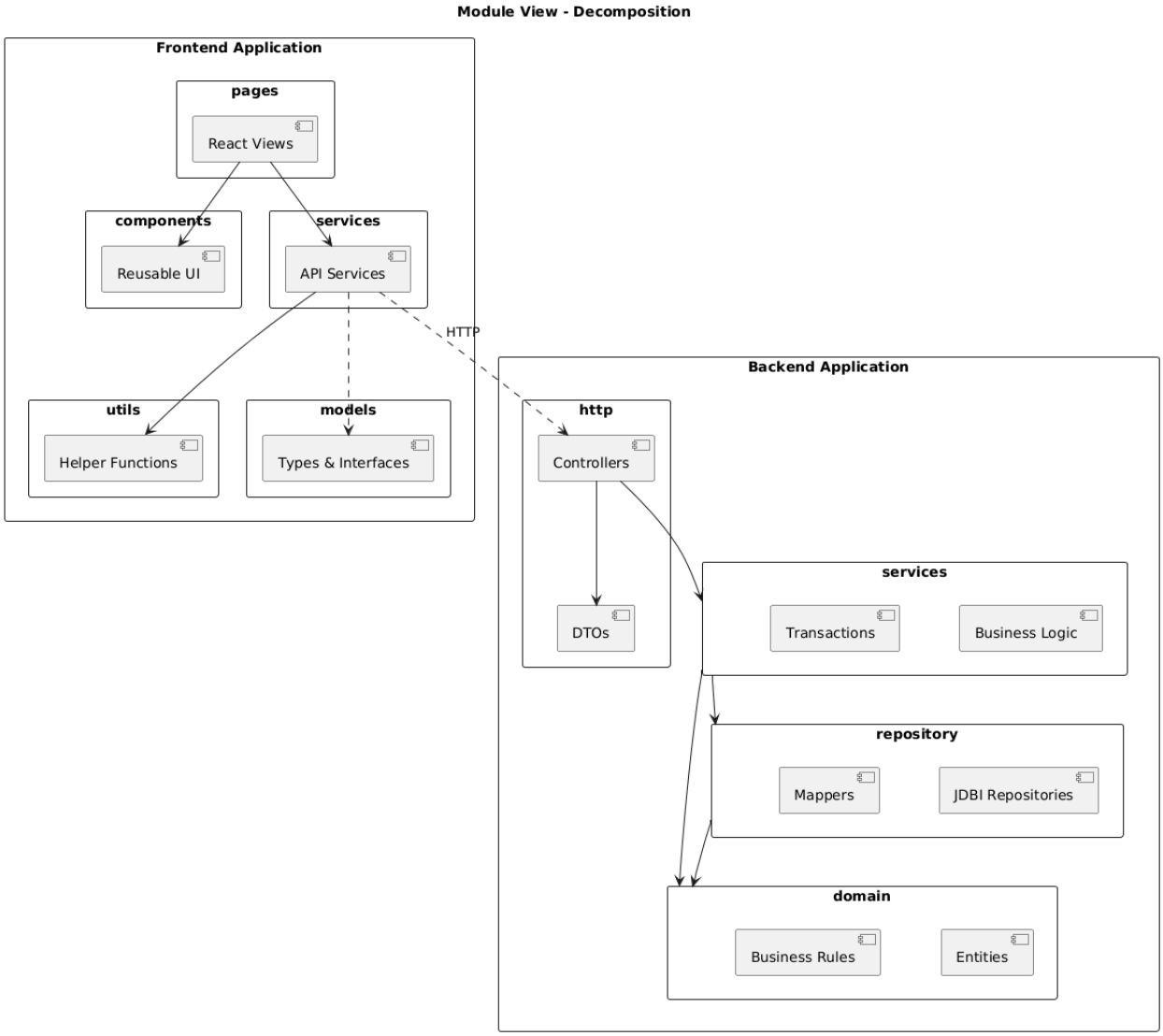
Database Persistence:

- A relational database ensures referential integrity (Foreign Keys) and transactional consistency, which are needed for a messaging system to prevent lost messages.

3.2 Decomposition View

3.2.1 View Description

3.2.2 Primary presentation



3.2.3 Element Catalog

Frontend Application

Pages:

Contains the top-level React components (views) corresponding to specific routes (e.g., HomePage, ChannelPage). These components orchestrate data fetching and layout.

Components:

Houses reusable UI elements (buttons, inputs, lists) used across multiple pages to ensure visual consistency.

Services:

Encapsulates external API communication. Modules here (e.g., ChannelsService) handle HTTP requests to the backend, decoupling the UI from network logic.

Models:

Defines TypeScript interfaces and types (e.g., ProblemModel, RoleModel) that enforce type safety for data structures used throughout the application.

Utils:

Contains shared helper functions and utilities, such as the fetch wrapper for handling HTTP headers and error parsing.

Backend Application

Http:

The entry point for the API. It contains Controllers that handle incoming HTTP requests and DTOs (Data Transfer Objects) that define the shape of data entering and leaving the API.

Services:

The core business logic layer. It implements use cases, manages transactions, and coordinates between the domain and repository layers.

Domain:

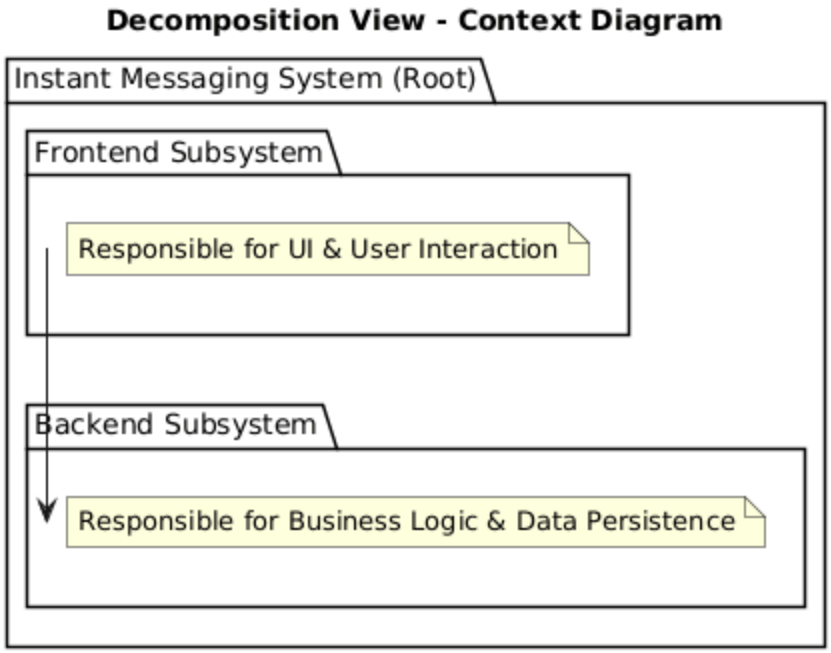
Represents the core business concepts (Entities) and rules. This layer is independent of frameworks and databases.

repository:

Responsible for data persistence. It uses JDBI to map domain objects to SQL queries and interact with the PostgreSQL database.

3.2.4 Context Diagram

The context diagram for the Decomposition View establishes the scope of the system. The Instant Messaging System is the top-level module which has all implementation units. The system is partitioned into two primary subsystems based on their execution environment and responsibilities: the Frontend Application and the Backend Application.



3.2.5 Variability Mechanisms

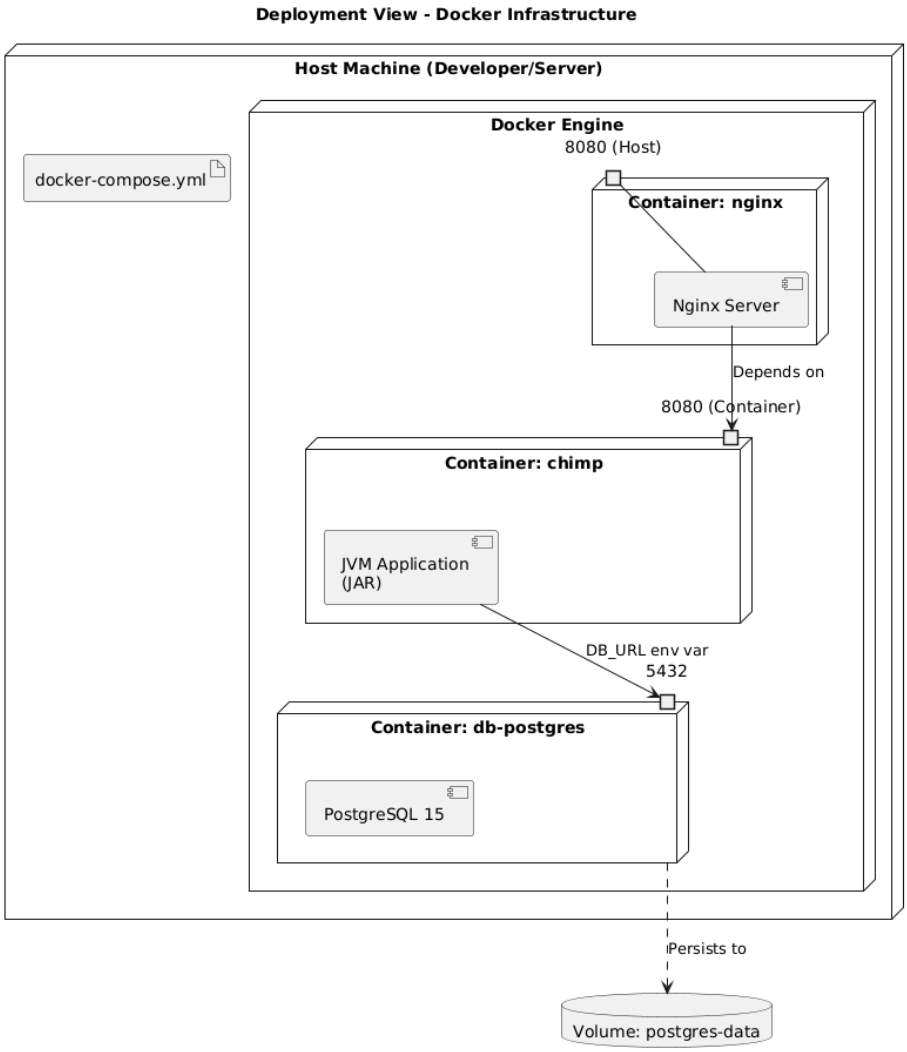
3.2.6 Architecture Background

3.3 Deployment View

3.3.1 View Description

This view describes the deployment architecture of the system. Its goal is to identify hosts, Docker containers, networks, and volumes, and to show how software components are physically deployed and interconnected. It is also used to explain the running, communication, resource isolation, and dependencies of the service.

3.3.2 Primary presentation



3.3.3 Element Catalog

Nodes:

- Host machine
- Docker engine

Containers:

- Nginx: Reverse proxy container that routes incoming HTTP requests through port 8080
- Chimp: Application container running a JVM-based application that handles the API logic
- Db-postgres: Database container that runs PostreSQL 15

Other elements:

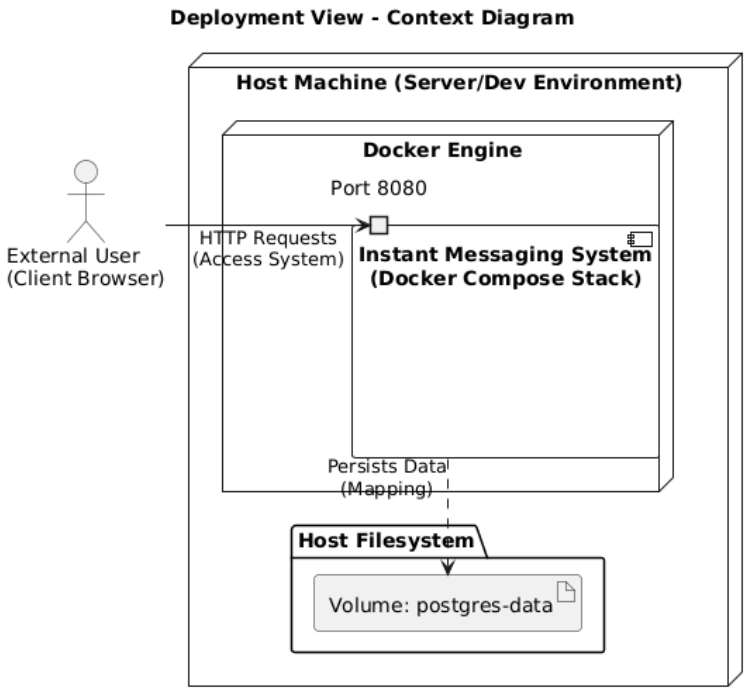
- Docker-compose.yml: Configuration artifact from Docker that defines dependencies, port mapping and environment variables
- Postgres-data: Persistent storage for PostgreSQL data.

3.3.4 Context Diagram

The system is deployed on a single host machine running Docker Compose. External users access the system through HTTP requests handled by the nginx container, which serves as the system's entry point.

The nginx container forwards requests to the chimp application serving as a reverse proxy container over the Docker network. The chimp container that is connected to the db-postgres container, applies all the logic needed.

Database data persists outside the container lifecycle using a Docker-managed volume.



3.3.5 Variability Mechanisms

Container Replication

- The nginx and chimp containers can be scaled horizontally by increasing the number of container instances.

Storage Strategy

- Docker volumes can be replaced with external or cloud-based storage solutions if needed.

3.3.6 Architecture Background

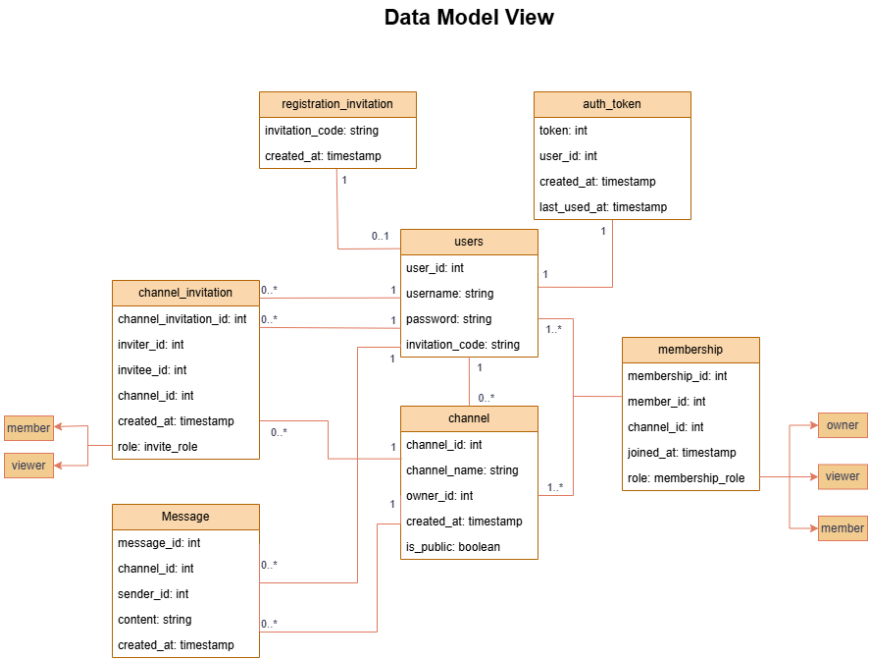
This structure was designed so that it could scale in the future for each container to run in independent machines.

3.4 Data Model View

3.4.1 View description

This view describes the data architecture of the message system. Its goal is to identify the main data entries and their relationships that ensure the project's data consistency and integrity. It’s used to understand data dependencies for each table and query patterns.

3.4.2 Primary Presentation



3.4.3 Element Catalog

Elements:

- users: Represents system users. Contains its credentials and references to the invitation code used for registration.
- auth_token: Authentication tokens for user sessions. Implements a token-based authentication system.
- registration_Invitation: Invitation codes required to register a new user. Each code can be reused 3 times.

- channel: Communication channels that can be public or private. Each channel has its owner who cannot leave the channel.

- membership: Many-to-many relationships that link channels and its users. Each membership has a specific role.

- message: Message sent within channels. Contains message content and references to the sender and channel.

- channel_invitation: Invitations for users to join specific channels. Includes the role that will be assigned.

Relationships:

- users <-> registration_invitation: Each user must use exactly one invitation. Each invitation can be used, but it is not mandatory. Each user can only generate 3 invites that do not have a user associated with it.

- users -> auth_token: Each user has a single authentication token associated. If the user performs an action and the token has not expired yet, its duration is extended.

- users -> channel: Each user can own zero or multiple channels. Each channel has exactly one owner.

- user <-> channel (membership): Users can be members of multiple channels, and channels have at least one member (owner). This relationship includes role information.

- Users -> message: A user can send zero or multiple messages. Each message has a sender.

- Channel -> message: A channel contains zero or multiple messages. Each message has a channel associated.

- Users -> channel_invitation (inviter): A user can send multiple channel invitations.

- Users -> channel_invitation (invitee): A user can receive multiple invitations.

- Channel -> channel_invitation: A channel can have multiple pending invitations.

Behavior:

- User registration:

    - A valid registration_invitation is required.
    - System creates a new user record
    - The invite is marked as used
- Channel creation:
    - User creates a channel with a unique name and its visibility
    - System automatically creates a membership record with role = ”owner”
- Sending messages:
    - The message is validated according to user membership permissions (viewers cannot send messages)
- Channel invitation
    - Channel member (everyone except viewers) invites another user
    - System creates a channel_invitation record
    - Invitees can either accept, generating a new membership and deleting the invitations, or decline, and the invitation is just deleted
- Token management:
    - If the user authenticates, a new auth_record is generated with created_at and last_used_at timestamps. For each action performed by the user, last_used_at is updated
    - If the user logs out, the token is deleted
    - Token expires based on absolute TTL (30 days) or rolling TTL (30 minutes)

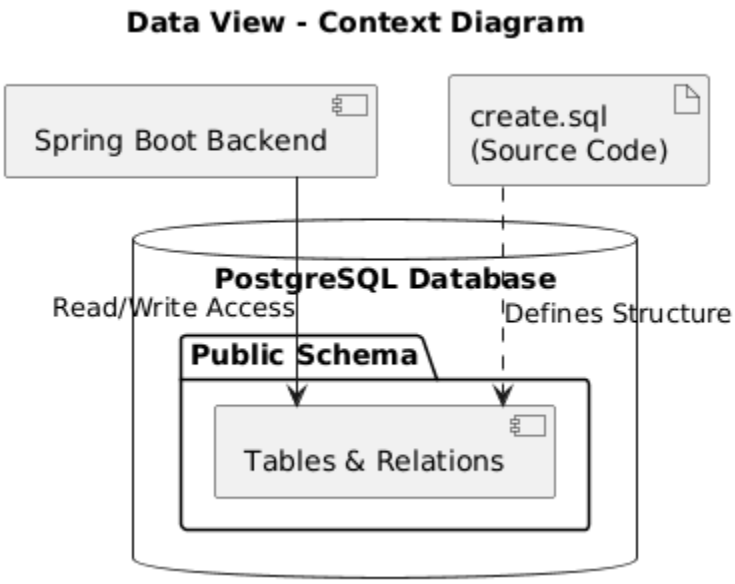
Constraints:

- Usernames and channel names must be unique
- Password requires a minimum of 8 characters, containing at least a number and a letter

- Each user can generate 3 active invitation_tokens (not used by any user)
- Each channel has a unique owner who cannot be changed or removed
- Viewers can only read messages
- Only owner and members can invite others to a channel
- Any user can join a public channel
- User needs invitation to join private channels

3.4.4 Context Diagram

The data model view operates within the PostgresSQL database system, and external operations occur through JDBC connections and SQL queries.



3.4.5 Variability Mechanisms

3.4.6 Architecture Background

Token-based authentication

- Automatic cleanup prevents token accumulation

- Reusable codes reduce administrative overhead

Enum types for roles:

- Ensured data consistency
- Prevents invalida role assignments

4 Relations Among Views

Each of the views specified in Section 3 provides a different perspective and design handle on a system, and each is valid and useful in its own right. Although the views give different system perspectives, they are not independent. Elements of one view will be related to elements of other views, and we need to reason about these relations. For example, a module in a decomposition view may be manifested as one, part of one, or several components in one of the component-and-connector views, reflecting its runtime alter-ego. In general, mappings between views are many to many. Section 4 describes the relations that exist among the views given in Section 3. As required by ANSI/IEEE 1471-2000, it also describes any known inconsistencies among the views.

4.1 General Relations Among Views

Structural consistency:

Data entities defined in the Data Model View are implemented and persisted in deployment elements identified in the Deployment View (e.g., database containers and volumes).

Behavioral alignment:

Interactions between components described in logical or application views are realized as container-to-container communication paths in the Deployment View.

Traceability:

Each runtime element in the Deployment View can be traced back to a logical responsibility and a data concern defined in the other views.

Together, these views provide a complete and consistent architectural description, enabling stakeholders to reason about the system from conceptual, logical, and operational perspectives without overlap or contradiction.

4.2 View-to-View Relations

1. Decomposition and Data Model views:

- The database described in the Data Model view is implemented in PostgereSQL 15, running inside the db-postgres container.

- Query patterns described in the Data Model View are executed by the chimp application container, which accesses the database through the configured DB_URL environment variable.
- The Docker volume (postgres-data) ensures that persistent data defined in the Data Model View survives container restarts and redeployments.
- The application services defined in the logical architecture are deployed as a JAR within the chimp container.
- The presentation and request-routing are implemented by the nginx container.
- The Data Model View defines what data exists and how it is structured.
- The Deployment View defines where that data lives and how it is accessed.

2. Component-and-Connector and Data Model Views:

- The PostgreSQL Database component stores all entities defined in the Data Model view
- JDBC operations: The Spring Boot Backend executes SQL queries through JDBC to perform the behaviors described in the Data Model view
- The auth_token table in the Data Model view supports the cookie-based authentication described in the Component-and-Connector login behavior.
- When the Backend writes new messages to the database (Data Model), it triggers the Channel Emitter to notify connected clients via SSE (Component-and-Connector).

3. Component-and-Connector and Deployment Views

- The React Frontend runs in client browsers, the Spring Boot Backend executes within the chimp container, and PostgreSQL runs in the db-postgres container.

- SSE connections described in Component-and-Connector are maintained between client browsers and the chimp container. Network failures require frontend reconnection logic.

- The JDBC connector uses the DB_URL environment variable defined in docker-compose.yml to connect the chimp container to db-postgres.

4. Decomposition and Deployment Views

- The application services defined in the logical architecture are packaged and deployed as a JAR within the chimp container.

- The presentation layer and request-routing logic from the Decomposition view are implemented by the nginx container.

- The database described in the Decomposition view is implemented in PostgreSQL 15, running inside the db-postgres container.

- The docker-compose.yml artifact defines how the decomposed modules interact, including dependencies, port mappings, and environment variables.

5. Component-and-Connector and Decomposition Views

- Components in the Component-and-Connector view (React Frontend, Spring Boot Backend, PostgreSQL Database) represent the runtime instances of modules defined in the Decomposition view.
- The HTTP/1.1 REST connector implements the service interfaces defined in the Decomposition view application layer.
- The React Frontend's session state management corresponds to the presentation layer's responsibilities in the Decomposition view.
- The Spring Boot Backend implements the business logic modules (authentication, validation, and message processing) described in the Decomposition view.

6. Data Model and Deployment Views

- All tables in the Data Model view (users, channels, messages, etc.) are stored in the db-postgres container using the postgres-data volume.
- Query patterns and behaviors described in the Data Model view are executed by the chimp application container through JDBC connections.
- The DB_URL environment variable in docker-compose.yml provides the connection string for accessing the database schema defined in the Data Model view.
- Container isolation in the Deployment view enforces the data access patterns and constraints specified in the Data Model view.

- Database initiation scripts defining the relational schema run when the db-postgres container starts.

5 Referenced Materials

Barbacci 2003	Barbacci, M.; Ellison, R.; Lattanze, A.; Stafford, J.; Weinstock, C.; & Wood, W. *Quality Attribute Workshops (QAWs)*, Third Edition (CMU/SEI-2003-TR-016). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2003. <http://www.sei.cmu.edu/publications/documents/03.reports/03tr016.html>.
Bass 2003	Bass, Clements, Kazman, *Software Architecture in Practice,* second edition, Addison Wesley Longman, 2003.
Clements 2001	Clements, Kazman, Klein, *Evaluating Software Architectures: Methods and Case Studies,* Addison Wesley Longman, 2001.
Clements 2002	Clements, Bachmann, Bass, Garlan, Ivers, Little, Nord, Stafford, *Documenting Software Architectures: Views and Beyond*, Addison Wesley Longman, 2002.
IEEE 1471	ANSI/IEEE-1471-2000, *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems*, 21 September 2000.
Moodle	University of Lisbon, “Mobile Robots”. Moodle. [Online]. Available: https://moodle.ciencias.ulisboa.pt/course/view.php?id=6204

6 Directory

6.1 Glossary

Term	Definition
software architecture	The structure or structures of that system, which comprise software elements, the externally visible properties of those elements, and the relationships among them [Bass 2003]. "Externally visible" properties refer to those assumptions other elements can make of an element, such as its provided services, performance characteristics, fault handling, shared resource usage, and so on.
view	A representation of a whole system from the perspective of a related set of concerns [IEEE 1471]. A representation of a particular type of software architectural elements that occur in a system, their properties, and the relations among them. A view conforms to a defining viewpoint.
view packet	The smallest package of architectural documentation that could usefully be given to a stakeholder. The documentation of a view is composed of one or more view packets.
viewpoint	A specification of the conventions for constructing and using a view; a pattern or template from which to develop individual views by establishing the purposes and audience for a view, and the techniques for its creation and analysis [IEEE 1471]. Identifies the set of concerns to be addressed, and identifies the modeling techniques, evaluation techniques,

	consistency checking techniques, etc., used by any conforming view.
Backend	The server-side part of the system running on the JVM, responsible for business logic, authentication, and database access.
Container	A standard unit of software that packages code and all its dependencies so the application runs quickly and reliably from one computing environment to another (Docker).
Controller	A component of the Web Layer (Spring Boot) responsible for handling incoming HTTP requests, validating input, and returning responses.
Emitter	n the context of SSE, refers to the object managed by the server to push events to a specific client connection.
Frontend	The client-side part of the system running in the user's browser (React), responsible for the visual interface and user interaction.
Middleware	Software that acts as a bridge between different applications or services. In this project, Nginx acts as a middleware reverse proxy.
Repository	A design pattern used to isolate data access logic. It defines an interface for interacting with the database.

6.2 Acronym List

API	Application Programming Interface; Application Program Interface; Application Programmer Interface
HTTP	Hypertext Transfer Protocol

API	Application Programing Interface
JDBC	Java Database Connectivity
UML	Unified Modeling Language
SPA	Single Page Application
UI	User Interface
UUID	Universally Unique Identifier
SSE	Server Sent Events
DTO	Data Transfer Object
TTL	Time To Live
IOS	IPhone Operating System
IM	Instant Messaging
JAR	Java Archive
SQL	Structured Query Language
ORM	Object Relational Mapping
COTS	Commercial Off-The-Shelf
IEE	International Education Evaluation
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
REST	Representational State Transfer