

**Design document**

**Author** : Andrii Matviienko

**Version**: 1.0

**Date**: 28.03.2025

[**Context :** 2](#_Toc198831504)

[**C1:** 3](#_Toc198831505)

[**Software Principles Applied:** 4](#_Toc198831506)

[**С2:** 5](#_Toc198831507)

[**Software Principles Applied:** 6](#_Toc198831508)

[**С3:** 6](#_Toc198831509)

[**Software Principles Applied:** 7](#_Toc198831510)

[**Frontend:** 8](#_Toc198831511)

[**Main page:** 8](#_Toc198831512)

[**Login page:** 9](#_Toc198831513)

[**Register page:** 10](#_Toc198831514)

# **Context :**

This design document defines the architecture of WiredSpace, explains technology choices, and provides C4 Model diagrams. It ensures clarity, consistency, and a shared understanding among developers and stakeholders.

# **C1:**

Изображение выглядит как текст, снимок экрана, диаграмма, Шрифт

Контент, сгенерированный ИИ, может содержать ошибки.

This **System Context (C1) diagram** shows how different entities interact with **WiredSpace**:

1. **Users** create, consume, and interact with social media content through the system.
2. **Admins** manage, moderate, and configure the platform.
3. **WiredSpace** handles API requests and processes interactions.
4. **Email System** sends confirmation emails and notifications to users.

## **Software Principles Applied:**

* **KISS (Keep It Simple, Stupid)** – The architecture ensures simplicity by clearly defining roles for each component, reducing unnecessary complexity.
* **YAGNI (You Ain’t Gonna Need It)** – Features that are not immediately required are not included to keep the system lightweight.

# **С2:**

Изображение выглядит как текст, снимок экрана, диаграмма, Шрифт

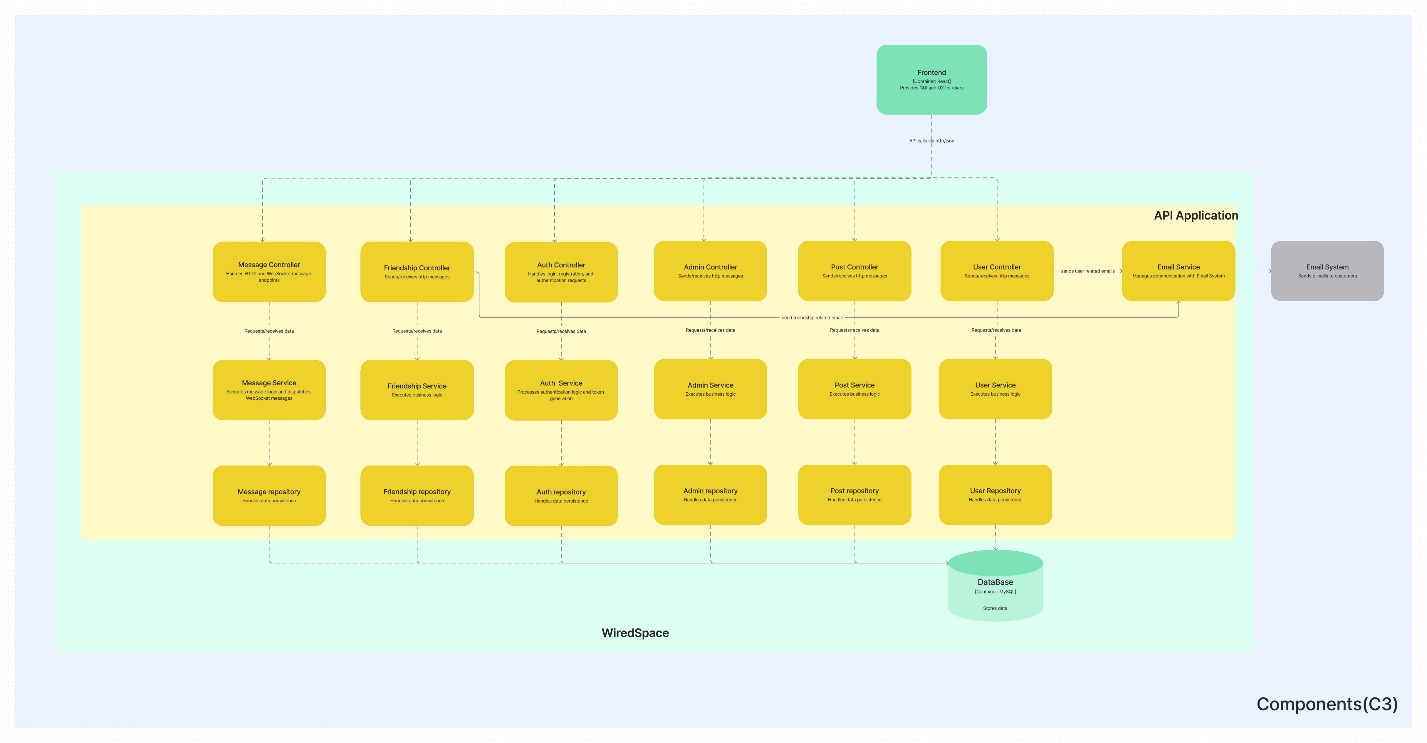
Контент, сгенерированный ИИ, может содержать ошибки.

The **Container Diagram (C2)** illustrates the high-level architecture of WiredSpace. The **Frontend (React)** provides the user interface and interaction layer. The **Backend (Spring Boot)** processes business logic and manages API requests. The **Database (MySQL)** stores user data and content. The **Email System** sends notifications and confirmations via SMTP. **Users** and **Admins** interact with the system through the frontend.

## **Software Principles Applied:**

* **SOLID**
  + **Single Responsibility Principle (SRP):** Each service (frontend, backend, database, email system) has a well-defined responsibility.
  + **Dependency Inversion Principle (DIP):** The backend communicates with the database via repositories, avoiding direct dependencies.
* **DRY (Don't Repeat Yourself):** Common functionalities such as authentication and notifications are centralized to avoid redundant code.

# **С3:**

С3 diagram represents the high-level architecture of the **WiredSpace** API application.

1. The Frontend (React) allows users to interact with the system and send API requests.
2. The API Application (Spring Boot) contains controllers, services, and repositories for business logic and data access.
3. PostController and UserController handle HTTP requests for posts and users, delegating logic to services.
4. MessageController manages private messaging between users.
5. FriendshipController handles friend requests, confirmations, and status updates.
6. ReportController allows users to report inappropriate posts or messages.
7. PostService, UserService, MessageService, FriendshipService, and ReportService contain business logic and use corresponding repositories.
8. PostRepository, UserRepository, MessageRepository, FriendshipRepository, and ReportRepository access and manage data in MySQL.
9. EmailService sends confirmation and notification emails via the Email System.
10. The Database (MySQL) stores posts, users, messages, reports, friendships, and other persistent data.

## **Software Principles Applied:**

**SOLID**

* **Single Responsibility Principle (SRP)**: Each controller, service, and repository has a clearly defined purpose:
  + Controllers handle HTTP requests.
  + Services contain business logic.
  + Repositories manage data access.
* **Open/Closed Principle (OCP):** Services are designed to be extensible without modifying existing code.
* **Liskov Substitution Principle (LSP):** Repositories and services follow interfaces, ensuring flexibility in implementation.
* **Interface Segregation Principle (ISP):** Interfaces are small and specific (e.g., UserRepository, ReportRepository), so classes only implement what they actually use.
* **Dependency Inversion Principle (DIP):** High-level services depend on abstractions (interfaces), not on concrete classes — e.g., ReportService depends on ReportRepository interface, not its implementation.

**DRY:** Common logic (e.g., email sending, validation, error handling) is reused across multiple services.

**KISS:** Architecture is modular and easy to understand, with minimal dependencies between components.

**YAGNI:** Advanced optimizations (e.g., caching, load balancing) are deferred until necessary to avoid premature complexity.

# **Database:**

**Database:**

1. The MySQL database stores all persistent data used by the WiredSpace platform.
2. Tables are designed to support key platform features: user accounts, friendships, messaging, posts, comments, likes, and reports.
3. UUIDs (binary(16)) are used for user-related entities to ensure uniqueness and scalability across distributed systems.
4. accounts
   * Stores user profiles and credentials.
   * Fields: id, username, account\_type, email, name, password, registered\_at, note, comments\_count, friends\_count.
5. friendships
   * Stores friend connections and requests.
   * Fields: id, accepted, user\_id, friend\_id.
6. messages
   * Stores private messages between users.
   * Fields: id, sender, recipient, text, timestamp.
7. posts
   * Stores user-created posts.
   * Fields: id, title, content, created\_at, user\_id.
8. post\_comments
   * Stores comments on posts, including support for nested replies via parent\_id.
   * Fields: id, comment, created\_at, post\_id, user\_id, parent\_id.
9. post\_likes
   * Stores likes on posts.
   * Fields: id, liked\_at, post\_id, user\_id.
10. post\_reports
    * Stores reports submitted by users for inappropriate content.
    * Fields: id, reason, reported\_at, post\_id, reporter\_id.
11. Relationships:
    * user\_id, friend\_id, reporter\_id → reference accounts.id.
    * post\_id → references posts.id in post\_comments, post\_likes, and post\_reports.
    * user\_id → used in posts, post\_comments, post\_likes for ownership tracking.
    * parent\_id in post\_comments → self-referencing for threaded comments.
12. Indexes and foreign keys ensure data integrity and query performance across all relationships.

# **Frontend:**

This diagram illustrates the architecture of the WiredSpace frontend, implemented using React. The frontend is structured as a modular, role-based application, serving as the primary user interface layer for both end-users and administrators. It interacts with the backend API layer (Spring Boot) via HTTP requests and WebSocket connections.

**Core Elements:**

**React App (Container):**  
Acts as the root container that bootstraps all routes, layout, and context providers. It is responsible for client-side routing using React Router and loading lazy components based on the user’s role.

**Pages:**

* Main – Landing page with navigation to login and registration.
* Login – User login screen.
* Register – User registration screen.

**Auth Pages:**

* Dashboard – Main control panel after login.
* Manage Admins – Administrator management interface.
* Manage Users – User management panel.
* Manage Reports – Content report review and moderation.
* User List – Overview of all registered users.

**Admin Pages:**

* Admin Profile – Administrator profile page.
* Admin Feed – Administrative activity feed.
* Admin Messages – Internal messaging interface.
* Admin Friends – Admin friend list and requests.
* Admin Settings – Platform-related admin settings.

**User Pages:**

* User Profile – Standard user profile page.
* User Feed – User content feed (posts and updates).
* User Messages – Chat and private messaging.
* User Friends – Friend list and invitations.
* User Settings – Personal user preferences and settings.

**Components (Reusable):**

* **Sidebar:** is used in both admin and user interface.
* **PostCard, CommentCard:** Modular visual units for social interaction.
* **AccountCard:** Visual unit meant to show user information.

**Custom Hooks (Stateful Logic):**

* **useAttemptAutoLogin:** Automatically logs in users if a valid token exists.
* **useAuthRedirect:** Redirects users after authentication based on role.
* **useMessages:** Manages chat logic including sending and receiving messages.
* **useRoleGuard:** Ensures route/component access based on user roles.
* **useWebSocketMessages:** Enables real-time messaging via WebSocket connection.

**Service Layer:**

* **AuthService:** Manages authentication state, token storage, and auto-login logic.
* **UserService:** Retrieves and updates user profile and preferences.
* **PostService:** Handles creation, editing, and deletion of posts.
* **FeedService:** Loads the user's post feed and trending content.
* **MessageService:** Provides access to conversation history and message delivery.
* **FriendshipService:** Sends and accepts friend requests, manages friendships.
* **ReportService:** Submits reports for moderation review.
* **AdminService:** Manages moderation actions, such as banning users or reviewing reports.

**API Classes:**

* **AuthAPI:** Handles login, registration, and token refresh requests.
* **UserAPI:** Manages user data fetch/update operations.
* **PostAPI:** CRUD operations for posts.
* **FeedAPI:** Retrieves post feed.
* **MessageAPI:** Sends and receives messages.
* **FriendshipAPI:** Manages friend requests and status.
* **ReportAPI:** Sends content reports.
* **AdminAPI:** Interfaces with moderation and admin-level functionality.

**Routing Flow:**

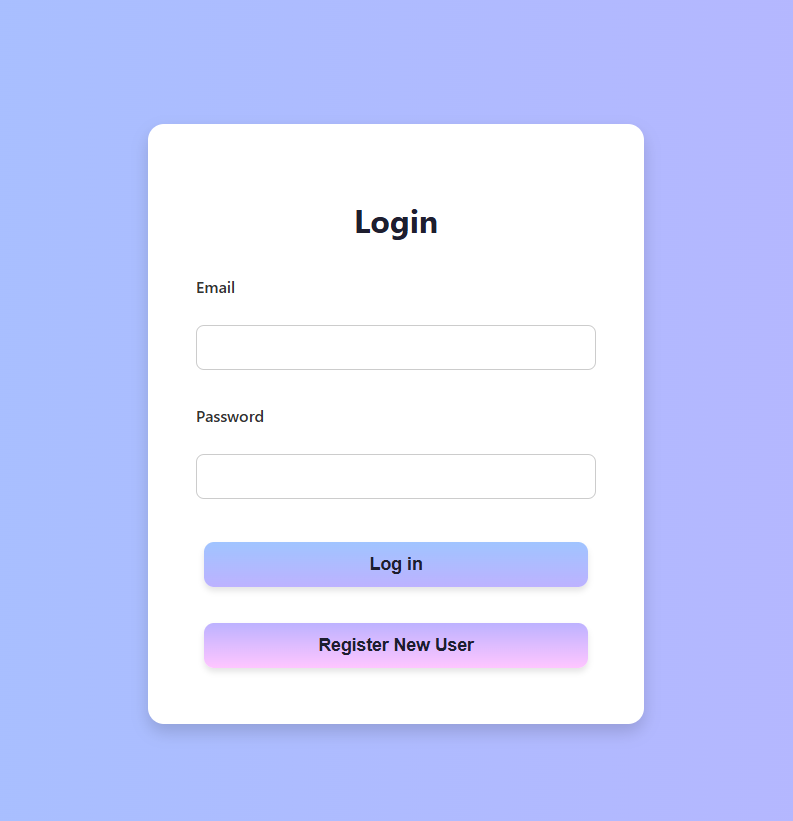
Upon login, the user is redirected to the appropriate dashboard depending on their assigned role. Protected routes ensure unauthorized users are redirected. Error boundaries wrap critical routes to ensure graceful degradation during runtime failures.

**Role-Based Access Control (RBAC):**

Admin and user interfaces are conditionally rendered based on the decoded JWT role. Navigation, layout, and accessible pages are dynamically scoped per role.

# **Application screenshots:**

Login



Register  
Изображение выглядит как текст, снимок экрана, Шрифт, число

Контент, сгенерированный ИИ, может содержать ошибки.

Admin dashboard

Изображение выглядит как текст, диаграмма, снимок экрана, линия

Контент, сгенерированный ИИ, может содержать ошибки.

Profile Page  
Изображение выглядит как снимок экрана, текст, дизайн

Контент, сгенерированный ИИ, может содержать ошибки.

Feed

Изображение выглядит как текст, программное обеспечение, Значок на компьютере, Операционная система

Контент, сгенерированный ИИ, может содержать ошибки.

Friends  
Изображение выглядит как текст, программное обеспечение, Значок на компьютере, Операционная система

Контент, сгенерированный ИИ, может содержать ошибки.

Settings

Изображение выглядит как текст, снимок экрана, дизайн

Контент, сгенерированный ИИ, может содержать ошибки.