

Specyfikacja Projektu Zaliczenowego (Programowanie w Języku Java)

Radosław Ciepał

Patryk Kowalczyk

10 czerwca 2025

1. Tytuł Projektu

Chatterly

2. Opis i Cele Projektu

Projekt ma na celu stworzenie aplikacji desktopowej, która umożliwia użytkownikom szyfrowaną komunikację. Główne cele projektu:

- Bezpieczna rejestracja i logowanie użytkowników.
- Tworzenie, wysyłanie i odbieranie wiadomości tekstowych w czasie rzeczywistym
- Wyświetlanie listy kontaktów / znajomych
- Powiadomienia o przychodzących wiadomościach
- Wyświetlanie statusu online/offline
- Obsługa wylogowania

3. Wymagania Funkcjonalne

- Rejestracja, logowanie, zarządzanie sesją.
- Szyfrowanie wiadomości „end-to-end”
- Przechowywanie historii czatu w relacyjnej bazie danych na serwerze (zaszyfrowanej)
- Przechowywanie tablicy użytkowników w relacyjnej bazie danych na serwerze
- Cache kilku ostatnich wiadomości oraz kontaktów w lokalnej relacyjnej bazie danych (zaszyfrowanej)

- Zarządzanie kluczami kryptograficznymi (generowanie par kluczy asymetrycznych oraz klucza symetrycznego, wymiana publicznych kluczy)
- Bezpieczne zamykanie sesji po upływie timeout

4. Wymagania Niefunkcjonalne

- **Wydajność:** Czas przesyłu danych poniżej sekundy.
- **Skalowalność:**
 - Obsługiwanie wielu wątków na serwerze - obsługiwane domyślnie przez Eclipse Tyrus
- **Bezpieczeństwo:**
 - W serwerowej bazie danych znajdują się hashowane hasła (bcrypt)
 - Szyfrowanie komunikacji TCP serwer-klient (rejestracja, logowanie, wymiana endpointów API) (TLS/SSL)
 - * Uzgodnienie wersji protokołu (TLS 1.2/1.3),
 - * Wymiana kluczy asymetrycznych (certyfikat + klucz prywatny serwera),
 - * Uzgodnienie klucza sesji (symetrycznego),
 - * Szyfrowanie ruchu symetrycznym algorytmem (np. AES-GCM).
 - Uwierzytlić serwer za pomocą TLS/SSL certyfikat X.509 dla domeny serwera (Let's Encrypt)(KeyStore)
 - Szyfrowanie „end-to-end” wiadomości (hybrydowy schemat kryptograficzny) (KeyStore)
 - Utrzymywanie Ping/Pong(Heartbeat) między klientem a serverem (WebSocket idle timeout)
 - Weryfikacja klienta i kontrola czy ma uprawnienia do przesyłu (JWT (JSON Web Token))
- **Użyteczność:** Interfejs zgodny z odpowiednikiem z web WCAG (AA).

5. Architektura Systemu

Poniżej opisano ogólną architekturę systemu czatu, jego główne komponenty oraz zależności między nimi. Typ architektury: Klient-Serwer (MVC / warstwowa).

System składa się z dwóch głównych części: klienta desktopowego (JavaFX) oraz serwera (Java + REST + WS). Komponenty te komunikują się ze sobą za pośrednictwem TLS/SSL, a autoryzacja klienta odbywa się przy pomocy JWT. Ponadto do wymiany wiadomości w czasie rzeczywistym wykorzystywany jest WebSocket. Eclipse Tyrus implementuje Java API for WebSocket. Działa na serwerze i zarządza handshakami, sesjami WebSocket, cyklem życia połączenia, heartbeat, konfiguracją zabezpieczeń oraz endpointami. Jersey obsługuje REST API endpoints implementując JAX-RS.

REST (HTTPS)

- Służą do operacji CRUD i pojedynczych wywołań „na żądanie”:
 - Rejestracja (POST /api/register)
 - Logowanie (POST /api/login) → zwraca JWT
 - Odświeżanie tokena (POST /api/refreshToken)
 - Pobranie listy znajomych (GET /api/users/friends)
 - Pobranie historii czatu (GET /api/messages?withUser=id limit=N)
 - Pobranie/aktualizacja klucza publicznego (GET/PUT /api/users/id/publicKey)
- Każde żądanie REST zawiera w nagłówku Authorization: Bearer <JWT>, który został uzyskany po zalogowaniu.
- Połączenia REST są bezstanowe (każde żądanie musi mieć ważny JWT) i szyfrowane TLS-em (HTTPS).

WebSocket (WSS)

- Służą wyłącznie do wymiany wiadomości w czasie rzeczywistym (push/pull).
- To „trwałe” połączenie TCP nad TLS:
 - Przy handshake (otwieraniu WSS) klient dołącza ten sam JWT w nagłówku Authorization.
 - Serwer sprawdza podpis i datę wygaśnięcia tokena. Gdy JWT jest poprawny, sesja WS jest akceptowana.
- Po otwarciu WebSocket klient wysyła/odbiera kolejne zaszyfrowane (E2E) pakiety JSON bez potrzeby ponownego dołączania JWT w każdej wiadomości.

Logika przepływu:

1. Klient wysyła REST-owe POST /api/login → dostaje JWT.
2. Klient (WSS) otwiera wss://server/chat z nagłówkiem Authorization: Bearer <JWT>.
3. Serwer w handshake weryfikuje JWT i przyznaje sesję WS.
4. Od teraz klient wysyła szyfrowane E2E-wiadomości przez WebSocket, a serwer tylko je przekazuje (plus buforuje offline).
5. Gdy JWT wygaśnie, przy próbie ponownego connectToServer() WebSocket handshake się nie uda → klient musi przez REST odświeżyć token, a potem ponowić WSS.

Schemat:

- Frontend: JavaFX + CSS
- Backend: Java + Tyrus + Jersey + Grizzly
- Server: Render Hobby Plan
- Baza danych: PostgreSQL, SQLite, JDBC (Java Database Connectivity)
- WebSocket: WebSocket
- Autoryzacja: JWT, certyfikat X.509
- Hashowanie: bcrypt, SQLCipher
- Przechowywanie kluczy i certyfikatów: KeyStore
- Generowanie TLS/SSL certyfikat X.509: Let's Encrypt

6. Zastosowane Algorytmy

- **Algorytmy uwierzytelniania:**

- *JWT:*

- * Algorytm sygnatury: HMAC-SHA256 (HS256) lub RSA-SHA256 (RS256)
 - w zależności od tego, czy używasz klucza symetrycznego (sekret) czy klucza prywatnego/publicznego.
 - * Claims: `sub: userId, iat: issuedAt, exp: expiresAt, roles: []`
 - .
 - * Weryfikacja podpisu oraz pola `exp` przed przyznaniem dostępu.

- *Hashowanie haseł:*

- * bcrypt z solą (np. 12 rund generowania soli, parametr `cost=12`).
 - * Przechowywanie w bazie tylko 60-znakowego ciągu zawierającego sol i hash.
 - * Weryfikacja `BCrypt.checkpw(plainPassword, storedHash)`.

- **Zarządzanie sesją:**

- *Access Token (JWT):*

- * Czas życia (TTL) typowo 15–60 minut (`exp = now + TTL_minutes`).
 - * Po wygaśnięciu odrzucany w każdej weryfikacji (`if now > exp → 401`).

- *Refresh Token:*

- * Dłuższy TTL (np. 7 dni), zapisany w bazie z flagą `revoked`.
 - * Endpoint `POST /api/refreshToken`: weryfikacja podpisu i obecności w bazie → wydanie nowego Access Tokena (i opcjonalnie nowego Refresh Tokena, unieważnienie starego).

- *Middleware / Interceptor:*

- * Każde wywołanie REST i handshake WebSocket weryfikuje JWT; jeśli exp przekroczone → próba odświeżenia (jeśli nagłówek Refresh-Token jest prawidłowy), w przeciwnym razie zwraca 401 i wymaga ponownego logowania.
- * W WebSocket: przy otrzymaniu błędu „token expired” klient automatycznie próbuje odświeżyć token i wznowić połączenie WSS.

- **Szyfrowanie „end-to-end” wiadomości:**

- *Asymetryczne (klucze długoterminowe):*

- * RSA 2048-bit z OAEP (RSA/ECB/OAEPWithSHA-256AndMGF1Padding)
 - **RSA** Algorytm szyfrowania asymetrycznego, w którym każdy użytkownik ma parę kluczy.
 - **ECB (Electronic Codebook)** etykieta trybu szyfrowania dla bloków danych
 - **OAEP (Optimal Asymmetric Encryption Padding)** wypełniania klucz do stałej długości
 - **SHA (Secure Hash Algorithm)** konkretny algorytm skrótu używany wewnątrz OAEP do generowania maski i mieszanki bajtów.
- * **lub** ECC na krzywej secp256r1 (ECDH + AES-GCM) – generacja kluczy za pomocą `KeyPairGenerator.getInstance("EC")`, krzywa **SECP256R1**.
 - **ECC (Elliptic Curve Cryptography)** Szyfrowanie asymetryczne oparte na własnościach krzywych eliptycznych
 - **Krzywa secp256r1** Jeden z powszechnie stosowanych parametrów ECC
 - **ECDH (Elliptic Curve Diffie–Hellman)** Protokół wymiany klucza, który korzysta z ECC
 - **AES-GCM (Advanced Encryption Standard – Galois/Counter Mode)** Algorytm symetryczny łączący szyfrowanie blokowe AES z trybem GCM
- * Klucz publiczny A i B przechowywany w bazie; klucz prywatny użytkownika zaszyfrowany hasłem w lokalnym keystore.

- *Symetryczne (klucze sesyjne):*

- * AES-GCM 256-bit (`Cipher.getInstance("AES/GCM/NoPadding")`), generacja IV 12 bajtów losowych.
 - **AES (Advanced Encryption Standard)** symetryczny algorytm szyfrowania blokowego
 - **Szyfrowanie (Counter Mode):** blok danych jest liczony sekwencyjnie (nonce + licznik), zaszyfrowany AES, a wynik XORowany z tekstem jawnym.
 - **Autentykacja (Galois Field):** generuje tag (128-bitowy) HMAC-podobny, obliczony nad całym szyfrogramem i opcjonalnymi danymi dodatkowo uwierzytelnianymi (AAD(Additional Authenticated Data))

- * Przy każdej wiadomości:

1. `SecretKey sessionKey = KeyGenerator.getInstance("AES").init(256).gener`

2. Cipher aes = Cipher.getInstance("AES/GCM/NoPadding");
 3. GCMParameterSpec spec = new GCMParameterSpec(128, iv);
 4. ciphertext = aes.doFinal(plaintext);
- *Hybrydowy schemat pakowania:*
- * encryptedKey = RSA_OAEP(publicKeyRecipient, sessionKey) lub ECDH→derive sharedKey→AES-kdf→encryptedKey.
 - * JSON-payload:


```
{
    "to": "userB",
    "encryptedKey": "Base64(...)",
    "iv": "Base64(...)",
    "ciphertext": "Base64(...)",
    "timestamp": 163XXXYYYY
}
```
 - * Odbiorca:
 1. Odzyskuje sessionKey = RSA_OAEP_decrypt(privateKeyB, Base64(encryptedKey))
 2. Odszyfrowuje plaintext = AES_GCM_decrypt(sessionKey, iv, ciphertext).
- *Podpisywanie (opcjonalnie) dla integralności dodatkowej:*
- * **HMAC (Hash-based Message Authentication Code)** mechanizm, który pozwala jednocześnie zweryfikować integralność danych oraz autentyczność ich nadawcy
 - * HMAC-SHA256 nad całym JSON-em lub nad ciphertext + iv z kluczem pochodnym od shared secret (w ECDH).
 - * Weryfikacja HMAC przed odszyfrowaniem w celu wykrycia manipulacji.

7. Tablice baz danych:

users (przechowuje dane o kontach użytkowników)

- id SERIAL PRIMARY KEY
- username VARCHAR(50) UNIQUE NOT NULL
- passwordHash VARCHAR(60) NOT NULL – bcrypt-hash
- publicKey TEXT NOT NULL – klucz publiczny RSA/ECC w formacie Base64
- createdAt TIMESTAMP WITH TIME ZONE DEFAULT NOW()
- updatedAt TIMESTAMP WITH TIME ZONE DEFAULT NOW()

friends (lista relacji „znajomości” między użytkownikami)

- id SERIAL PRIMARY KEY
- userId INTEGER NOT NULL — REFERENCES users(id) ON DELETE CASCADE

- friendId INTEGER NOT NULL — REFERENCES users(id) ON DELETE CASCADE
- status VARCHAR(10) NOT NULL – np. pending, accepted, blocked
- createdAt TIMESTAMP WITH TIME ZONE DEFAULT NOW()
- UNIKALNY INDEX (userId, friendId)

messages (zapis wszystkich wysłanych paczek – także tych już dostarczonych)

- id BIGSERIAL PRIMARY KEY
- fromUser INTEGER NOT NULL — REFERENCES users(id) ON DELETE CASCADE
- toUser INTEGER NOT NULL — REFERENCES users(id) ON DELETE CASCADE
- iv BYTEA NOT NULL – wektor inicjalizacyjny AES-GCM (12 B)
- encryptedKey BYTEA NOT NULL – klucz AES zaszyfrowany RSA/OAEP (lub ECDH→KDF)
- ciphertext BYTEA NOT NULL – zaszyfrowana treść wiadomości (AES-GCM)
- timestamp TIMESTAMP WITH TIME ZONE DEFAULT NOW()
- delivered BOOLEAN NOT NULL DEFAULT FALSE – czy zostało „wysłane” do odbiorcy

pendingMessages (dla wiadomości, które odbiorca był offline)

- id BIGSERIAL PRIMARY KEY
- fromUser INTEGER NOT NULL — REFERENCES users(id) ON DELETE CASCADE
- toUser INTEGER NOT NULL — REFERENCES users(id) ON DELETE CASCADE
- iv BYTEA NOT NULL
- encryptedKey BYTEA NOT NULL
- ciphertext BYTEA NOT NULL
- timestamp TIMESTAMP WITH TIME ZONE DEFAULT NOW()

refreshTokens (przechowuje refresh tokeny do odświeżania JWT)

- id SERIAL PRIMARY KEY
- token VARCHAR(255) UNIQUE NOT NULL – przechowujemy cały token (np. Base64 JWT)

- `userId` INTEGER NOT NULL — REFERENCES `users(id)` ON DELETE CASCADE
- `expiresAt` TIMESTAMP WITH TIME ZONE NOT NULL
- `revoked` BOOLEAN NOT NULL DEFAULT FALSE

userStatus (szybki podgląd online/offline)

- `userId` INTEGER PRIMARY KEY — REFERENCES `users(id)` ON DELETE CASCADE
- `isOnline` BOOLEAN NOT NULL DEFAULT FALSE
- `lastSeen` TIMESTAMP WITH TIME ZONE DEFAULT NOW()

8. Struktura Klas

Model

- **User** – username, password, publicKey.
- **Message** – fromUser, toUser, iv, encryptedKey, ciphertext.

Serwisy

- **UserService** – rejestracja, logowanie, validateAccessToken, refreshTokens.
- **MessageService** – logika zarządzania wiadomościami.
- **NotificationService** – generowanie powiadomień nowych wiadomości.
- **AuthService** – JWT, walidacja, bezpieczeństwo.

Kontrolery

- **UserController** – REST API użytkownika.
- **MessageController** – REST API pobierania zaległych wiadomości.

Dodatkowe komponenty

- **WebSocketHandler** – wymiana wiadomości w czasie rzeczywistym.
- **DTO** – obiekty wymiany danych (Data Transfer Object).
- **DAO** – zapewnia abstrakcyjny interface do bazy danych (Data Access Object).

9. API

Punkty końcowe API.

Endpoints:

POST /api/users/register
POST /api/users/login
POST /api/users/refreshToken
POST /api/users/logout
GET /api/messages

10. Interfejs Użytkownika

Opis interfejsu graficznego aplikacji, jego struktury, logiki działania oraz dostępnych ekranów.

Główne widoki:

- **Ekran logowania i rejestracji:** Formularze z walidacją danych.
- **Pulpit użytkownika:**
 - Lista kontaktów z powiadomieniem o nowej wiadomości
 - Wyszukiwanie nowych użytkowników
 - Widok ostatnich wiadomości
 - Pole wpisywania i wysyłania wiadomości

11. Etapy Realizacji / Harmonogram

Plan wdrożenia projektu z podziałem na fazy i szacowanym czasem realizacji.

Faza	Opis	Czas
Planowanie	Określenie wymagań, wybór technologii	1 tydzień
Backend	Budowa API, modele danych	1 tydzień
Frontend	Interfejs, integracja z API	1 tydzień

12. Testowanie

Sposoby testowania aplikacji, typy testów oraz używane narzędzia.

- Testy jednostkowe (logika backendu i frontend).

- Testy integracyjne (API) JerseyTest, JUnit.
- Testy obciążeniowe – np. JMeter, Gatling.