CS763 Secure Software Development

Department of Computer Science
Boston University Metropolitan College

Battal Cevik

Project Assignment 2: Security requirements and threat modeling

In this assignment, you will perform a security analysis on your project. This include:

1. Identify if there are any compliance related to the application. (if any, a couple of sentences)

   In my app, it stores personally identifiable information (PII) such as name, email, phone, address, 7-digit SSN and also financial account info. It would generally bring the application under the Gramm-Leach-Bliley Act (GLBA) / Federal Financial Institutions Examination Council (FFIEC) expectations for safeguarding customer financial data in the USA. In the European Union, General Data Protection Regulation(GDPR) privacy obligations would also apply. Until now, it doesn't handle payment cards so PCI DSS doesn't apply yet. In the production deployments we will enforce HTTPS, least-privilege access, encryption in transit/ at rest especially SSN, strong authentication, auditing, and breach-notification procedures consistent with those regulations.

2. Consider possible adversaries, abuse scenarios and vulnerabilities to exploit. Create at least 5 possible abuse cases or abuser stories that may be used by the attackers to abuse the current use cases/user stories.

   I. Steal an account with the reset link.
   As an attacker, I want to take over someone's account by getting a password reset token, from stealing it from email or the browser so I can log in as them.

   II. Peek at other people's data
   As a logged in user, I want to change user IDs and account IDs in API calls so I can see another person's profile or account balances if the app does not check ownership.

III. Move money from different account
As a logged in user, I want to submit a transfer using someone's account so I can push funds to my accounts.

IV. Break in with password tries which is credential stuffing
As an attacker, I want to try many email/passwords combos on the login endpoint so I can use someones password from a breach.

V. Abuse the app through the browser(XSS)
As an attacker, I want to inject script into a form or a message that the app shows to others without escaping so that I can read their tokens and make actions as them.

3. Use those abuse cases as the root of attack trees to show detailed possible paths to achieve them. Try to have at least 3 levels of depth in your attack tree. Each attack tree lists subgoals or conditions in a hierarchy structure to achieve the ultimate goal of the attack described by the root node, with the leaf nodes being **specific, tangible** actions, steps, conditions, or outcomes that that an attack must achieve first. The leaf nodes can be specific well-known vulnerability exploits or attacks. The leaf nodes should be easily assessed in terms of possibility, complexity, and cost, so the risk analysis can be done on the root nodes.

I. Steal an account with the reset link

Attack tree: Take over account via password reset token.
Root goal: Take over victim's account via password reset mechanism
Subgoal: I can gain access to victims's email inbox.

Leaf1: Use previously breached credentials to log in victim's email such as no 2FA or password reuse.
It is highly possible since breached credentials are widely available. Complexity is low because automated tools like selenium or bots. Cost is also low because of free or cheap credentials lists.

Leaf2: Phishing attack to capture email login credentials
Possibility is medium because of depends on user awareness.
Complexity is also medium because it requires crafting convincing phishing pages.
Cost is also medium with phishing kits or hosting

Leaf3: Voicemail hijack if email recovery/OTP is phone based.
Possibility is medium because of bots and AI voices hijack.
Complexity is also medium because it requires voicemail. Cost is also medium with lots of voicemail hijack options.

II. Peek at other people's data

Attack tree: View data belonging to other users
Root goal: View or enumerate other users' profiles and account data
Subgoal: Discover vulnerable endpoints and parameters
Leaf1: Use browser dev tools and proxy to inspect APIs calls for userId, accountId, and customerId.
Possibility is high because publicly accessible in browser.
Complexity is low because basic web debugging skills. Cost is also low because of free tools like Chrome DevTools etc.

Leaf2: User documentation, error messages or URL patterns to find object identifiers
Possibility is medium, complexity is low with basic research skills and cost is also low free.

Leaf3: Intercept API calls using a proxy.
Possibility is high and it requires setup but straightforward. Complexity is medium with a proxy tool knowledge. Cost is low with free community versions of tools.

III. Move money from different account

Attack tree: Move funds from accounts the attacker does not own
Root goal: Transfer money from another user's account to attacker controlled destination
Subgoal: Gain access to the transfer functionality
Leaf1: Use stolen credentials from a data breach, credential stuffing
Possibility is high, complexity is low and cost is also low.

Leaf2: Exploit session hijacking via stolen cookies via XSS. Possibility is Medium because it requires XSS and malware so complexity is also medium and cost is also medium.
Leaf3: Inspect network requests to find transfer endpoints. Possibility is high, it can be visible in the browser and complexity is low. It also has free tools so the cost might be very low too.

IV. Break in with password tries

Attack tree: Gain access via credential stuffing
Root goal: Log into legitimate user accounts using reused credentials

Subgoal: Obtain lists of email/password combinations
Leaf1: Download credential dumps from public leak sites or underground markets
Possibility is high because it is widely available on dark web marketplaces, complexity is low and cost is also low.
Leaf2: Use previous breach data from attacker's own earlier compromises
Possibility is high, complexity is low and cost is also low.
Leaf3: Create script bot to submit login requests with combos to /login
Possibility is high tools are accessible, complexity is medium we need a tool configuration and cost is low it is free open source tools.

V. Abuse the app through the browser XSS

Attack tree: Run attacker controlled JavaScript in victim's browser via the app
Subgoal: Find inputs that are stored and later rendered to other users
Leaf1: Submit test payloads such as <script>alert(1)</script> to forms.
Possibility is high if the input sanitization is weak. Complexity is low manual or scripted testing and cost is also low.

Leaf2: Use automated XSS scanners for example the XSStrike to find vulnerabilities
Possibility is high with an effective scanner. Complexity is medium with tool configuration and cost is low.

Leaf3: Share malicious link for reflected XSS via phishing or social engineering.
Possibility is medium because it depends on the user interaction. Complexity is medium and cost is also medium with phishing effort.

4. Identify any existing security feature that can help mitigate those abuse cases. Propose additional security requirements that can be used to help mitigate those abuse cases.

I. Reset link account takeover abuse case.
Right now I have BCrypt for passwords and a working reset flow but the reset token looks too demo style. It is returned in a JSON and I did not clearly state its strength and there is no rate limit. That makes it easier to guess, steal or reuse. I can fix it by generating a strong reset token at least 32 bytes of crypto random and store only a hash of the token in the Database. I can make each token single use and expire it after 15 minutes. I don't echo the token in API responses in a real set up, email a link instead and notify the user when a reset is requested or completed. I can add rate-limits to /auth/forgot-password for example 5/min per IP and per email and log/audit every reset attempt.

II. **Peek at other people's data**

I am using Spring Security and and ownership checks, but IDOR happens when **any** read or write path forgets to enforce this data belongs to the caller. So, we should never trust userId or accountId coming from the client. I must always derive the subject from the JWT/session and I must re-check the ownership in the service layer before hitting the repository. I have to add method security such as @PreAuthorize and a simple guard like *throw if entity.owner != authUser.* I am going to prefer **UUIDs** for public IDs, and add integration tests that try to access another user's info to prove it's blocked. So this way I can keep it secure users data from other people.

III. **Move money from another account**

In my application I have a transfer service and general authorization, but the risky gap is letting the client tell you the source account. I have to implement that the server must compute the source account from the authenticated user and then verify ownership again in the service. Also, I am planning to enforce sufficient funds, daily transfer limits, and require an Idempotency-Key which is a is a unique identifier sent with an API request for header so the same request can't be replayed. For larger amounts, I have to add step-up MFA before making an action and make sure every transfer is fully audited with who, when, amount, outcome.

IV. **Credential stuffing**

When I implement the BCrypt on my application it protects stored passwords, however the login endpoint will still accept unlimited tries unless I explicitly add brakes. I am going to put rate-limits on /auth/login by IP and by account for example 5/min, then cool-down. I can consider progressive back-off or a short temporary lock after many failures, and I can show generic errors to avoid username enumeration, and must require CAPTCHA which is a test designed to determine whether a user is a human or a bot after repeated failures. At signup/reset, I can check candidate passwords against breach lists (HIBP-style) and enforce a sane strength policy. I can send a new login email so users notice suspicious access.

V. XSS → token theft / actions

I build my application on React so it's default escaping helps, and Spring is also adding some headers. However, if tokens live in localStorage and there's no Content Security Policy(CSP), one XSS bug can steal them. I have to move tokens to httpOnly, Secure, SameSite cookies, and enable a strict CSP. I must sanitize any rich-text or user fields on the server and output-encode on render to avoid dangerously set inner HTML.

5. List all entry points and API endpoints. Identify possible attack vectors among those interfaces.

Entry points where traffic can enter
Browser → Frontend
Dev: http://localhost:5173/ (Vite)
Prod (Docker): http://localhost:5173/ served by Nginx (static React build)
Risks: XSS via user-rendered data; mixed content if not HTTPS in prod.

Frontend → Backend API
http://localhost:8080/api/**
Risks: CORS misconfig, missing auth/ownership checks, brute force on auth routes.

Health/ops GET /actuator/health Risks: information leakage if other actuator endpoints are exposed.

Backend → Database
JDBC to Postgres on 5432 (Compose network)
Risks: leaked DB creds, weak DB role permissions, lack of TLS if remote.

CI/CD (GitHub Actions)
Pulls repo, builds images, runs scanners.
Risks: secrets in logs, dependency/CI supply chain, artifact poisoning.

API EndPoints:
- POST */api/auth/signup*
  The purpose of this endpoint is to create user email, password, name, address, phone, SSN-7. I think the common attack vectors are Input validation

bypass, duplicate email/SSN probing, weak password acceptance, error message user-enum, rate-limit needed

- POST */api/auth/login*
  The purpose of this end point is to issue JWT + session for UX. Possible common attack vectors are credential stuffing / brute force, username enumeration via messages/timing, token leak in logs, rate-limit needed

- POST */api/auth/forgot-password*
  The purpose of this end point is to generate a reset token. Possible common attack vectors are account enumeration, reset-token brute force, mail bombing, rate-limit needed.

- POST */api/auth/reset-password*
  The purpose of this end point is to use reset token to test a new password. Possible common attack vectors are token replay, low-entropy tokens, long TTL, lack of single-use

- GET */api/users/me*
  The purpose of this endpoint is to return the current user's profile (PII). Possible common attack vectors are IDOR if the server ever reads userId from the request instead of the token, and verbose error messages that leak PII.

- GET */api/accounts/me*
  The purpose of this endpoint is to return the user's own accounts (checking/savings). Possible common attack vectors are IDOR/data leakage if the query isn't scoped by the authenticated user and pagination scraping to harvest extra data.

- POST */api/transfers*
  The purpose of this endpoint is to create a funds transfer. Possible common attack vectors are unauthorized source account use if the API trusts sourceAccountId from the client, replay attacks without an IDempotency check, CSRF if cookie-based auth is used, and missing sufficient-funds/daily-limit checks.

6. Clearly describe the application architecture and the code structure and how they are mapped to each other.

   Frontend (React/Vite)
   I created this layer is to render the SPA (login, signup, dashboard, accounts, profile, transfer) and call the backend via /api/*. Code mapping: frontend/src/pages/* (screens), frontend/src/components/* (UI parts), frontend/src/api/* (axios client), App.jsx (routes), vite.config.js (dev), nginx.conf (prod proxy /api → backend).

Nginx, React / Vite
I created this component because the purpose of this component is to serve the built React files and reverse-proxy /api/* to the backend. Code mapping: frontend/nginx.conf; Dockerfile builds static assets and copies them into the Nginx image.

Backend API & Spring Boot with Java
I used sprint boot with java for backend and API. The purpose of this service is to expose REST endpoints for auth, users/accounts, and transfers, applying validation and security. Code mapping: backend/src/main/java/com/example/bankapp/**; BankAppApplication.java boots the app; application.yml/properties holds config.

Security layer - Spring Security + JWT
I created this layer to authenticate requests (JWT/session), authorize access, and add security headers. Code mapping: security/SecurityConfig.java (filter chain, public vs protected paths), security/JwtAuthFilter.java (verifies token, sets Authentication), security/JwtService.java (issue/verify JWT), optional @PreAuthorize on controller/service methods.

Auth module
The purpose of this module is to handle signup, login, password-reset, refresh tokens. Code mapping: auth/AuthController.java (HTTP endpoints), auth/AuthService.java (BCrypt checks, token issuance), auth/ResetToken* (entity/repo/service for reset tokens)

Users & Accounts module
This module is created to to return/update the current user and list their accounts (checking/savings). Code mapping: user/UserController.java and user/AccountController.java (endpoints like /api/users/me, /api/accounts/me), user/UserService.java (ownership checks, profile logic), user/AccountService.java, entities user/User.java, user/Account.java

Transfers module
This module is created to create and list transfers with business rules (ownership, balance, limits). Code mapping: transfer/TransferController.java (e.g., POST /api/transfers), transfer/TransferService.java (compute source from auth user, sufficient-funds, daily limits, idempotency), entity transfer/Transfer.java, repo transfer/TransferRepository.java.

Validation & Error handling
I created validation and error handling to enforce input rules and return consistent errors. Code mapping: Bean Validation annotations on DTOs/entities (@Email, @Size, SSN 7-digit), and a global handler like web/

GlobalExceptionHandler.java (maps common exceptions to JSON, e.g., duplicate email/SSN).

Database - PostgreSQL
The purpose of the DB is to persist users, accounts, transfers, and reset tokens. Code mapping: JPA entities & repositories in user/, transfer/, auth/ packages; schema managed by Hibernate DDL or migrations; sensitive fields such as SSN, can use a JPA AttributeConverter for encryption.

Configuration (Web/Beans/Seeder)
The purpose of these classes is to wire cross-cutting concerns (CORS, object mapping) and seed initial data (e.g., admin user). Code mapping: config/WebConfig.java (CORS/json), config/DataSeeder.java (runs at startup to insert defaults).

Docker Compose - Runtime wiring
The purpose of Docker Compose is to run frontend + backend + postgres on one network. Code mapping: docker-compose.yml (service definitions, ports, env vars), Dockerfiles in frontend/ and backend/. Frontend proxies /api to backend:8080 in prod via Nginx config.

CI/CD (GitHub Actions)
The purpose of CI is to build, test, and scan the app on each push/PR. Code mapping: .github/workflows/java.yml (Maven build/tests), node.yml (Vite build), security.yml (Dependency-Check, Trivy), optional codeql.yml and zap-dast.yml.

7. Identify all components and draw a DFD diagram for the application, which includes data flow, data store, process, interactor/endpoints, and trust boundary. Then perform a STRIDE threat modeling by checking if any component in the diagram is subject to any of these 6 threat categories. List possible threats and the corresponding mitigations?

   I am going to explain each layer in below for STRIDE and possible corresponding mitigations.

   Interactor – User Browser
   The purpose of this actor is to load the SPA and call /api/*.
   Data in/out: credentials, JWT/session cookies, profile/forms.

   Trust boundary: crosses the public Internet into our stack.

STRIDE risks for Interactor:
Spoofing: phishing steals credentials
Info disclosure: tokens exposed by XSS/mixed content
Tampering: malicious extensions/DOM injection
DoS: automation/flooding

Mitigations: I can force to use HTTPS in prod (HSTS), strict CSP as I mentioned in the answer of above questions. I can store tokens in httpOnly+SameSite cookies or access token only in memory + refresh cookie, avoid localStorage, rate-limit auth endpoints.

Frontend (Nginx serving React SPA, reverse proxy for /api)
The purpose is to serve the built UI and forward API calls to the backend. Data in/out: HTML/JS/CSS, proxied JSON.

Trust boundary: Internet → edge container.

STRIDE risks for Frontend:
Spoofing: host header.
Tampering: bad proxy rules, cache poisoning
Info disclosure: misrouted requests
DoS: Slowloris,

Mitigations: I can use strict Nginx config (proxy only /api to backend), rate-limit at edge, allowlisted Host, security headers HSTS, X-Content-Type-Options, minimal server tokens.

Backend API (Spring Boot: Auth, Users/Accounts, Transfers, Security)
The purpose is to enforce authentication/authorization and business rules, then read/write Postgres.
Trust boundary: frontend container → backend container.

STRIDE risks for Backend API:
Spoofing: forged/ stolen JWT
Tampering: parameter tamper/replay
Repudiation: no audit
Info disclosure: PII in errors/logs
DoS: brute force/login floods
Elevation: missing ownership/method security

Mitigations: Spring Security for all non-auth routes, verify JWT (issuer/aud/alg/TTL), uniform error messages, structured audit logs, rate-limit /auth/*, method security (@PreAuthorize) and service-level ownership checks, idempotency on transfers.

Data Store – PostgreSQL
The purpose is to persist users, accounts, transfers, and reset tokens.

Trust boundary: backend container → DB container.

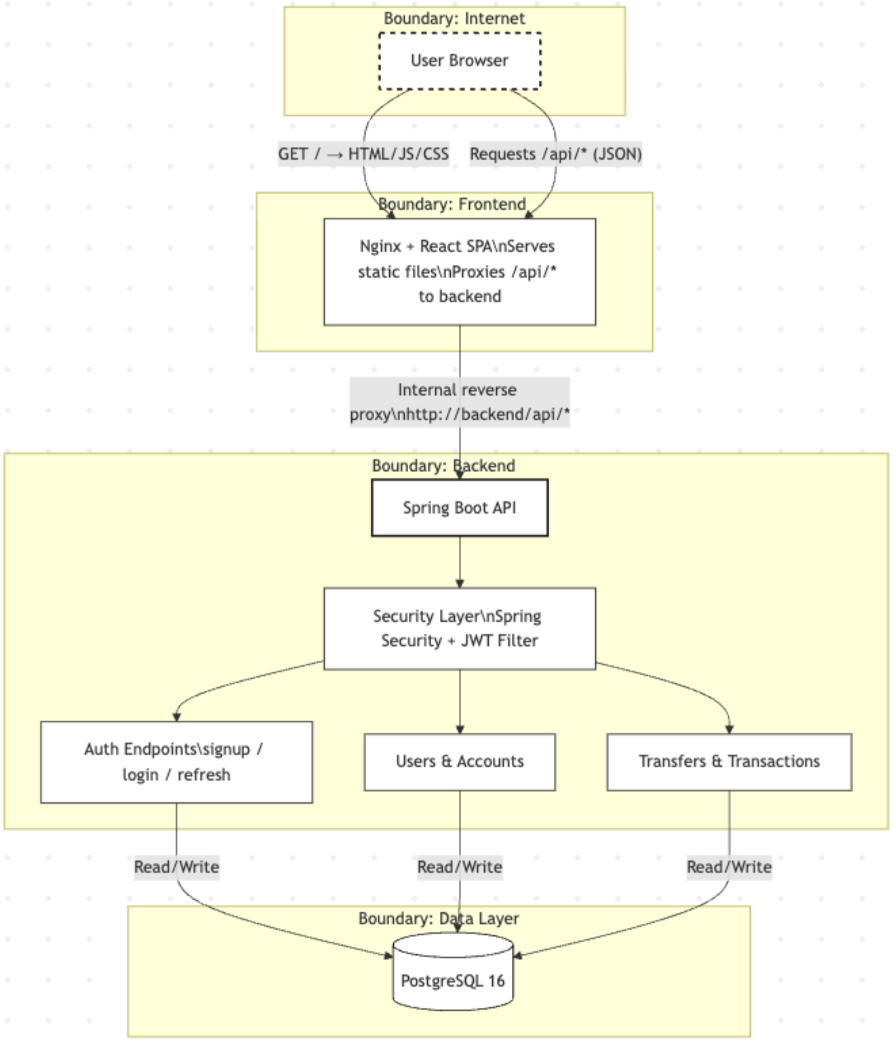
STRIDE risks:
Info disclosure: PII dump,
Tampering (balance edits),
Elevation (over-privileged DB user),
Repudiation (no immutable logs),
DoS (heavy queries/locks).

Mitigations: least-privileged DB role, network-only exposure, encrypt SSN at rest (AES-GCM via JPA AttributeConverter), at-rest encryption, TLS to DB if remote, backups with access control, indexes and sane query limits.

Aux – CI/CD (GitHub Actions)
The purpose is to build/test/scan images and code.
Trust boundary: GitHub cloud ↔ repo/runners.

STRIDE risks:
Tampering: build scripts/actions,
Info disclosure: secrets in logs/artifacts,
Elevation: over-broad permissions.

Mitigations: protected branches + reviews, pin actions, permissions: minimal, secret masking, Dependency-Check/Trivy/CodeQL (+ optional ZAP), signed releases.



8. Add a separate section to describe how you use the AI tool to help you perform analysis.

   I have used AI Assistant ChatGPT and Claude AI to tighten my STRIDE analysis and it helped my notes into clear diagrams and text. I provided the app context and my initial abuse stories and the data flow. AI helped me refine wording, structuring and completeness. I mostly get helped DFD Mermaid code after I described the components, data stores, endpoints and trust boundaries.

Resources:

- Microsoft STRIDE — baseline for the threat model structure we used.
  https://learn.microsoft.com/security/engineering/stride

- OWASP Top 10 — common web risks (IDOR, XSS, etc.) that shaped our abuse cases and fixes.
  https://owasp.org/Top10/

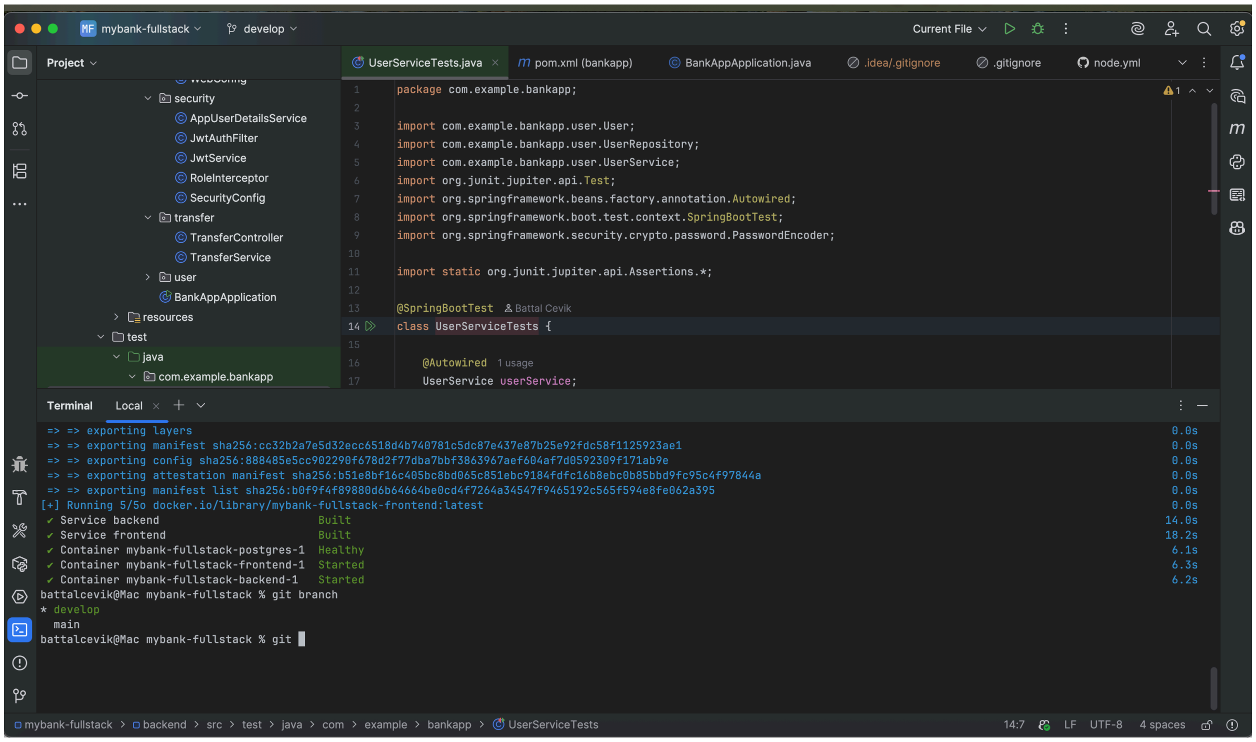
- OWASP API Security Top 10 — API-specific risks (BOLA/IDOR, auth, rate limits) for your endpoints.
  https://owasp.org/API-Security/

- OWASP Forgot Password Cheat Sheet — concrete guidance for secure reset tokens (entropy, TTL, single-use).
  https://cheatsheetseries.owasp.org/cheatsheets/Forgot_Password_Cheat_Sheet.html

- Spring Security Reference — how we wire authn/authz, headers, and method security in your stack.
  https://docs.spring.io/spring-security/reference/

Deliverables:

1. A github project : https://github.com/battalcevik/AgileBankProject
2. A report that describes the details of each of the above items.



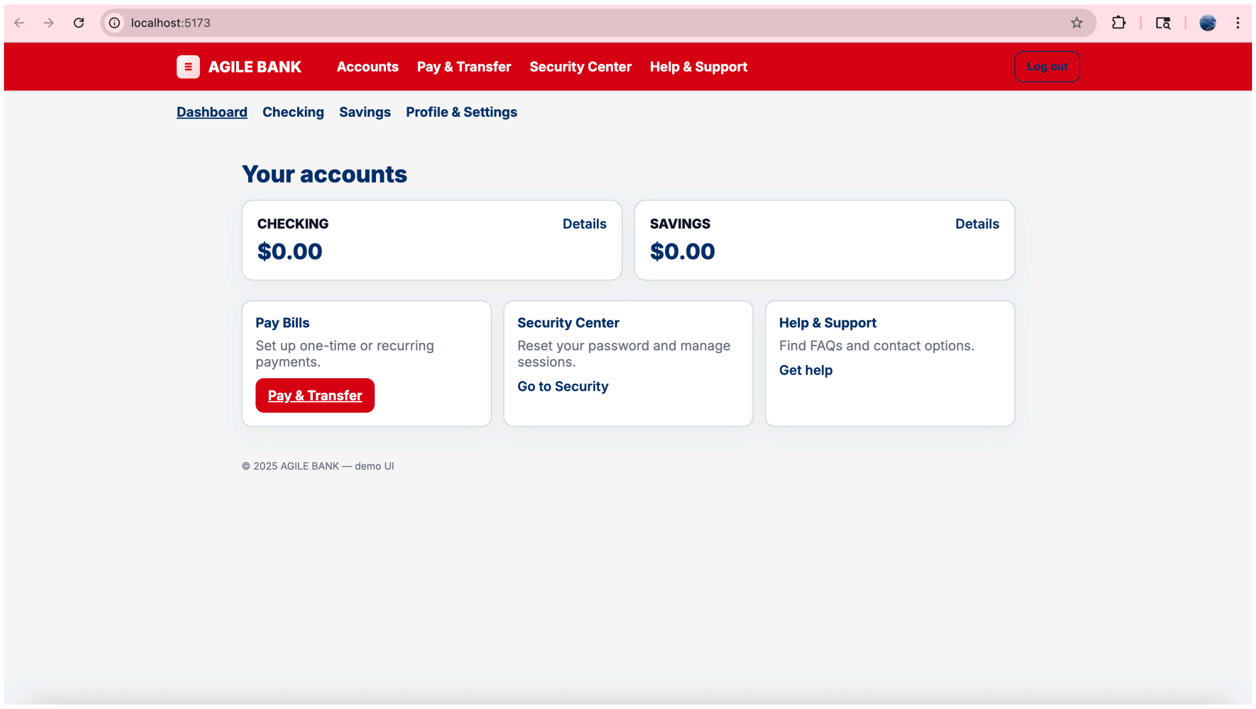
```java
package com.example.bankapp;

import com.example.bankapp.user.User;
import com.example.bankapp.user.UserRepository;
import com.example.bankapp.user.UserService;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.security.crypto.password.PasswordEncoder;

import static org.junit.jupiter.api.Assertions.*;

@SpringBootTest  Battal Cevik
class UserServiceTests {

    @Autowired  1 usage
    UserService userService;
```

```
 => => exporting layers                                                                                                    0.0s
 => => exporting manifest sha256:cc32b2a7e5d32ecc6518d4b740781c5dc87e437e87b25e92fdc58f1125923ae1                          0.0s
 => => exporting config sha256:888485e5cc902290f678d2f77dba7bbf3863967aef604af7d0592309f171ab9e                            0.0s
 => => exporting attestation manifest sha256:b51e8bf16c405bc8bd065c851ebc9184fdfc16b8ebc0b85bbd9fc95c4f97844a              0.0s
 => => exporting manifest list sha256:b0f9f4f89880d6b64664be0cd4f7264a34547f9465192c565f594e8fe062a395                     0.0s
[+] Running 5/5o docker.io/library/mybank-fullstack-frontend:latest                                                        0.0s
 ✔ Service backend                   Built                                                                                14.0s
 ✔ Service frontend                  Built                                                                                18.2s
 ✔ Container mybank-fullstack-postgres-1  Healthy                                                                          6.1s
 ✔ Container mybank-fullstack-frontend-1  Started                                                                          6.3s
 ✔ Container mybank-fullstack-backend-1   Started                                                                          6.2s
battalcevik@Mac mybank-fullstack % git branch
* develop
  main
battalcevik@Mac mybank-fullstack % git
```

The Process of building and running the application>

- The process of building the application is listed on the Github readme file.
- It is dockerized so it can be used below command from terminal to run on any local.
- docker compose up --build -d

- Manual Testing of the running Application



Your accounts

CHECKING Details
$0.00

SAVINGS Details
$0.00

Pay Bills
Set up one-time or recurring payments.
Pay & Transfer

Security Center
Reset your password and manage sessions.
Go to Security

Help & Support
Find FAQs and contact options.
Get help

© 2025 AGILE BANK — demo UI