**FAIR DICE GAME REPORT**

**CHATZOPOULOS IASON (p3312322) - STENTOUMIS NIKOLAOS**

This document provides an overview of the implemented solution for the "Fair Digital Dice" web application. The system is designed to allow a user to play a fair and secure game of dice against a server, with a focus on security, user authentication, and cryptographic fairness.

**Prerequisites to run:**

* WSL/Linux environment (we used ubuntu 22.04)
* Docker Desktop with docker compose (if on WSL, with WSL integration, 4.40.0 (187762))

**Build and Run:**

To build and run the application, navigate to the fair-dice-game directory of the deliverable and execute the compose script. This command will build the docker images and docker compose will create 3 containers, 1 for the database,1 one for nginx hosting the frontend of the application and finally 1 container for the backend of the application. If the script is invoked and any issues occur, adjust the permissions of the executables (chmod +x) and change the return carriages (e.g. dos2unix).

**Architecture and Technologies Used**

The solution is implemented as a full-stack web application, using the following technologies:

* The backend is developed using Java 21 with the Spring Boot framework, version 3.4.5 following a clean and modular architecture that separates domain logic, application services, persistence, and web controllers.
* The frontend is built using React version 19.1.0, providing a responsive and interactive user interface for registration, login, and gameplay.
* Communication between the client and server is secured using HTTPS with a custom TLS certificate.
* User authentication is implemented using JSON Web Tokens (JWT), allowing secure and stateless sessions.
* A relational database is used to store user credentials, with appropriate hashing applied to passwords.

**HTTPS and TLS**

The backend application enforces secure communication by using HTTPS. A TLS certificate, generated via OpenSSL, is integrated into the backend configuration. All HTTP requests are redirected to HTTPS to ensure encrypted communication between client and server.

**Authentication with JWT**

The application uses JWTs for user authentication. When a user logs in, the server generates a signed token containing user identity information. This token is stored on the client side and included in the Authorization header of all subsequent requests to secure API endpoints.

**Frontend Interface**

The React frontend provides pages for user registration, login, and the dice game interface. It stores the JWT token securely and uses it to access protected backend services. The game interface allows users to enter their guess and view the result of the game, with visual feedback and error handling.

**Backend API and Controllers**

The Spring Boot backend exposes a RESTful API that includes endpoints for user registration, login, starting a game session, and submitting a guess. The controller layer handles HTTP requests and delegates the processing to application services, which execute the game logic, authentication, and user management.

**Database and User Management**

User data is stored in a structured relational database. The system supports user registration and login functionalities. Passwords are securely stored using cryptographic hashing to protect user credentials.

Overall, the system integrates multiple modern technologies to offer a secure, fair, and responsive web-based dice game experience with strong cryptographic guarantees.

**Protocol Concept and Objectives**

The protocol ensures that the server cannot alter its dice roll after the client submits a guess. This is achieved using a commitment scheme, which involves the server committing to a value (its roll) without revealing it, and only revealing the roll after the client makes their guess.

**Implementation Overview**

The protocol is implemented across both the backend (Spring Boot) and the frontend (React). It includes:

• Generation and exchange of random strings.

• Secure hash computation on the server.

• Communication through authenticated HTTPS requests.

• Storage and verification of hashes on the client.

**Backend (Spring Boot) Implementation**

The main logic is handled by the `GameController.java` class located at:

backend-module/src/main/java/org/aueb/fair/dice/infrastructure/adapter/primary/web/controller/game/GameController.java

The two core endpoints are:

• “POST /api/game/{clientRandom}” – Accepts a random string from the client and initiates the server-side roll and commitment.

• “POST /api/game/guess/{guess}” – Accepts the client’s guess, reveals the server’s values, and returns the result.

The “startGame” method uses the interface “GamePort.java” to:

1. Roll a random number between 1 and 6 for the server.

2. Generate a random UUID string.

3. Concatenate: serverRoll + clientRandom + serverRandom.

4. Hash the result using SHA-256.

5. Return the hash to the frontend.

The server stores the full values internally (serverRoll, clientRandom, serverRandom) associated with the authenticated user session.

**Frontend (React) Implementation**

The frontend logic is found in:

frontend-module/src/components/GuessDice.js

**When the user enters a guess:**

1. A random string is generated and sent to the backend using “/api/game/{clientRandom}”.

2. The backend returns a hash, which is stored in localStorage as `gameHash`.

3. The user submits their guess via `/api/game/guess/{guess}`.

4. The backend reveals the original data.

5. The frontend recomputes the hash and compares it to `gameHash`.

If the hashes match, the server did not cheat. If not, the result is rejected. The frontend code explicitly checks this with state flags like `isHashMatch` and displays the outcome accordingly.

**Correctness and Full Compliance**

• The server does not know the user’s guess before committing.

• The client verifies that the server did not alter its roll.

• All communication occurs over HTTPS with JWT-protected routes.

• The result is only accepted if the computed and returned hash match.

• Code is cleanly structured across backend and frontend, following best practices.

This protocol offers true fairness through cryptographic integrity and bidirectional commitment without trust assumptions.

**Swimlane Flowchart Analysis**

**A green and black text

AI-generated content may be incorrect.**

**Decision-Based FlowChart**

**A diagram of a server

AI-generated content may be incorrect.**

The above flowcharts illustrates a secure and fair interaction protocol between a client and a server within a web-based dice game, designed to prevent either party from cheating. The process begins with the client a React frontend application generating a random string, which serves as a commitment input. This string is submitted to the backend via a POST /api/game/{clientRandom} API call. This is the first step of the commit-reveal scheme and marks the client’s contribution to the fairness of the session. This value is purely random and does not give away any sensitive game state, but it plays a vital role in the integrity check that follows later.

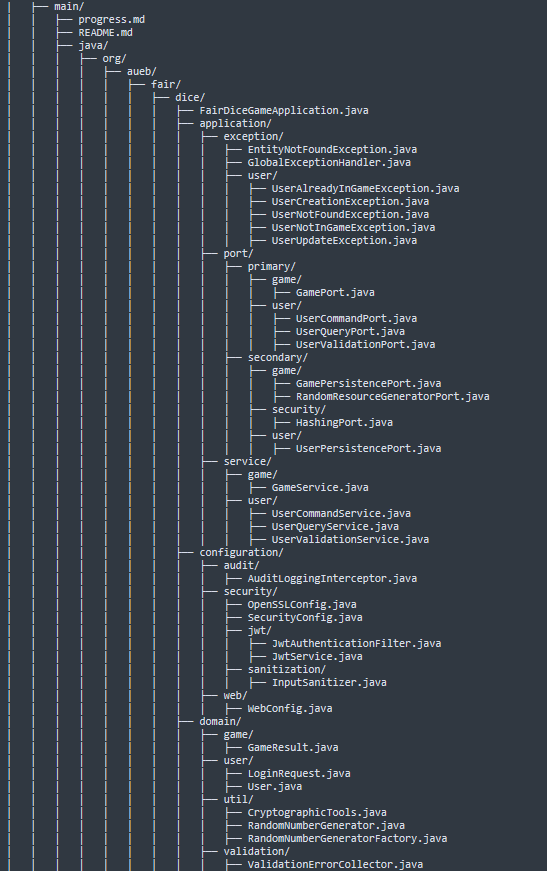
Upon receiving the client’s random input, the Spring Boot backend generates its own secret values: a dice roll (an integer from 1 to 6, simulating the server’s move) and an additional server-side random string. The backend then concatenates these values the server roll, the client’s random string, and its own server-generated string and hashes the resulting string using a cryptographic hash function (likely SHA-256, though the algorithm is abstracted in the service logic). This hash acts as a commitment: it locks the server’s response in a way that cannot be altered without being detected. The backend sends this hash back to the client, which stores it securely (in localStorage, as seen in GuessDice.js), but the actual dice roll and server random remain hidden.

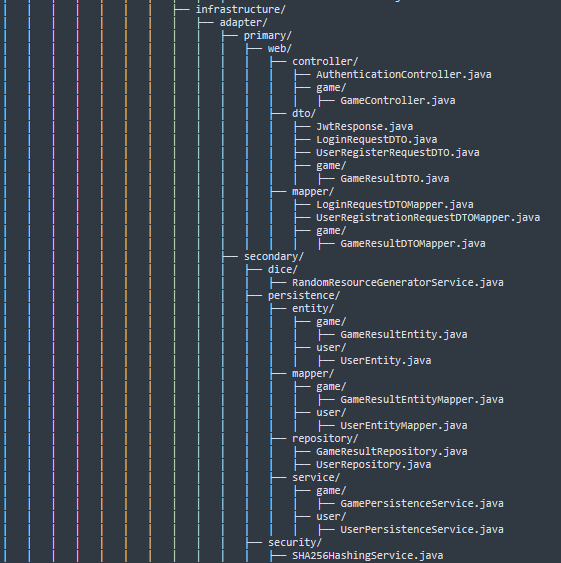
At this point, the game pauses for user input. The client interface prompts the user to guess a number between 1 and 6. When the user makes their selection, the guess is submitted via a second request to the backend: POST /api/game/guess/{guess}. This marks the reveal phase. In response, the backend returns the full original data — the server’s dice roll, the server’s random string, and the client’s own submitted value — along with an indicator of whether the user guessed correctly. This transmission allows the client to independently verify the server's honesty.

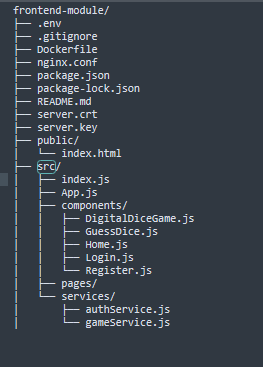
The critical step now unfolds on the client side: using the same hash function, the client recomputes the commitment hash using the revealed values. It compares this newly calculated hash with the one originally received and stored. If the hashes match, the integrity of the session is confirmed — the server did not alter its roll after the client’s guess. If they do not match, it is evidence of manipulation, and the game result is invalidated. In the correct implementation — which your project demonstrates — the game only displays the result if this integrity check passes.

The flowcharts reflect this dual-phase cryptographic interaction elegantly. The decision-style chart emphasizes the conditional logic: especially the pivotal "Is hash valid?" check, which acts as the cryptographic gatekeeper to the result. In contrast, the swimlane diagram highlights the responsibility boundaries: what the client does versus what the server controls. Together, they represent a secure and transparent system where neither party can retroactively affect the game’s outcome. This ensures compliance with the assignment's requirement for fairness, cryptographic integrity, and full lifecycle verification.

**Code Structure**

Code structure is presented below.





As already mentioned, the project is a full-stack web application designed to deliver a secure and fair online dice game between a user and a server. It integrates a Spring Boot backend and a React frontend, structured according to modern software engineering principles and built with a focus on cryptographic integrity, modularity, and end-to-end security.

At its core, the application implements a commit-reveal protocol to guarantee fairness in the game. The user interacts with a web interface built in React, which handles registration, login, and gameplay through well-structured components and service layers. The gameplay component, in particular, is responsible for generating a random string on the client side, initiating the game session by sending this to the backend, and later submitting the user’s guess. It then verifies the integrity of the server’s commitment by recomputing the hash locally using values returned by the backend and comparing it to the one previously received.

The backend, built using Java and Spring Boot, exposes a REST API responsible for user authentication, game logic, and cryptographic validation. Controllers handle HTTP requests and delegate business logic to application service ports like GamePort and UserCommandPort. These services orchestrate the commit-reveal sequence by generating a server-side dice roll, combining it with the client’s random string and a new server random value, hashing the combination, and returning the hash to the client as a cryptographic commitment. The game result is only finalized after the client submits a guess and the server reveals its values. This structure ensures the backend cannot alter its roll post-commitment.

The backend also manages user data securely, using DTO mappers, exception handlers, and Spring Data JPA repositories. Passwords are securely stored in a relational database with hashing applied before persistence. Authentication is stateless and token-based: upon successful login, the backend issues a JWT signed using the HS256 algorithm. This token is stored client-side and included in all subsequent API requests using the Authorization: Bearer <token> header.

For communication security, the application enforces HTTPS using a TLS certificate generated via OpenSSL. The backend is configured to serve HTTPS traffic and redirect HTTP requests automatically. This ensures encrypted data exchange between frontend and backend. The frontend is hosted on nginx, and also redirects http to https via the nginx configuration, This way a secure channel is enforced to interact with the application, while also validating the integrity of the game logic through hash verification.

The project is organized into clearly defined directories and packages. The backend separates concerns using a hexagonal architecture in conjunction with clean architecture: domain holds core entities, application defines services and interfaces, infrastructure contains web controllers and database access layers, and security manages token issuance and validation via the Spring Security Beans. The frontend mirrors this clarity, separating logic into components, pages, and services, with React Router handling navigation and a secure API layer managing all backend communication.

**Code Analysis - Backend**

The following classes are part of the backend of the application, depicting what their functionality and purpose.

**application.exception** — Global and Domain-Specific Error Handling

The application.exception package contains the centralized error handling mechanisms for the backend application. The class EntityNotFoundException.java defines a custom unchecked exception that is thrown when a requested entity (e.g., user or game) is not present in the database. It improves code readability and consistency by replacing ad hoc null checks and signaling HTTP 404 errors in a controlled manner. GlobalExceptionHandler.java uses Spring’s @ControllerAdvice to globally catch exceptions like EntityNotFoundException, translating them into meaningful HTTP responses. It ensures that API consumers always receive well-formatted error messages and status codes, avoiding returning to the user theexception stack traces or server errors and disclose information to the users that shouldn’t be exposed.

**application.exception.user** — User-Oriented Exception Definitions

The application.exception.user subpackage contains exception classes specific to user interactions. These exceptions are raised by the UserCommandService, UserQueryService, and game session logic to enforce business rules and constraints. UserAlreadyInGameException.java prevents users from launching a new game if one is already active. UserCreationException.java is used during registration failures, such as duplicate usernames or persistence issues. UserNotFoundException.java supports login and authentication by indicating invalid credentials. UserNotInGameException.java ensures game actions can only occur within valid sessions, and UserUpdateException.java is used when updates to user data cannot be persisted. These targeted exceptions allow the system to provide fine-grained feedback and improve robustness during runtime.

**port.primary.game** — Game Interaction Abstraction

The GamePort.java interface defines the primary entry point for initiating and resolving gameplay logic. It acts as a boundary between the controller layer and the application logic, exposing methods such as starting a game and submitting a user's guess. This abstraction enables the controller to remain decoupled from the specific implementation of the game protocol. In this architecture, GamePort serves as the contract that enforces the structure of the commit-reveal interaction, including generating the commitment hash, retrieving the result of the dice roll, and returning the necessary data for client-side verification.

**port.primary.user** — User Command and Query Ports

The interfaces UserCommandPort.java, UserQueryPort.java, and UserValidationPort.java define the contract for all primary operations related to user management. UserCommandPort includes methods for registering new users or modifying existing user data. UserQueryPort provides access to user retrieval operations, such as finding users by username, often used during login and authentication. UserValidationPort isolates logic that checks business rules, such as ensuring a user is in a valid state before game interaction. By separating command, query, and validation responsibilities, the system adheres to the principles of clean architecture and improves maintainability and testability.

**port.secondary.game** — Game Infrastructure Ports

GamePersistencePort.java defines how game results are stored and retrieved from persistence mechanisms, typically implemented using JPA. This allows the application layer to remain unaware of the underlying database technology. RandomResourceGeneratorPort.java abstracts the generation of random values used in the game protocol. By delegating randomness to a port, the application can be easily tested with fixed seeds or mocked values, ensuring consistent and reproducible behavior in test scenarios. These abstractions are essential for supporting a fair and secure dice game without binding the core logic to low-level implementations.

**port.secondary.security** — Cryptographic Hashing Ports

The HashingPort.java interface provides a contract for computing cryptographic hashes used in the commit-reveal phase of the protocol. It allows the game logic to delegate hash generation without being tied to a specific algorithm or library. This separation ensures that hash functions can be easily swapped or upgraded (e.g., from SHA-256 to SHA-3) without altering the main protocol logic. It contributes to the cryptographic modularity of the system.

**port.secondary.user** — User Persistence Ports

UserPersistencePort.java defines methods required to interact with persistent user data. It abstracts database operations such as saving new users, retrieving users by ID or username, and updating their state. This interface enables the application and domain layers to remain fully decoupled from JPA or any specific persistence technology. It supports clean separation of concerns and enables easy mocking during tests, reinforcing the architectural boundary between domain logic and infrastructure.

**application.service.game** — Gameplay Logic Implementation

GameService.java is the central class responsible for orchestrating the backend logic of the fair dice game. It implements the GamePort interface and coordinates the commit-reveal protocol by generating a dice roll, producing a server-side random string, and computing a commitment hash using the HashingPort. It also manages user state transitions during gameplay, enforcing one active game session per user. This class ensures that the hash returned to the client reflects a pre-committed roll and enables later validation by storing the random values and roll within the user entity. It integrates services such as RandomResourceGeneratorPort and UserPersistencePort, reflecting the clean separation of responsibilities defined by the architecture.

**application.service.user** — User Management Services

The UserCommandService.java handles all operations that modify user state, including registration. It validates input, maps DTOs to domain objects, and persists users through UserPersistencePort. Errors such as duplicate usernames are caught and wrapped in custom exceptions to provide controlled failure handling. UserQueryService.java retrieves user information needed for authentication or gameplay using UserQueryPort. It is typically used during login or hash verification. UserValidationService.java enforces business rules such as ensuring that a user is currently in a game before allowing further actions. These services adhere to the command-query separation principle and encapsulate the core business logic for secure and consistent user handling.

**configuration.audit** — HTTP Logging Interceptor

The class AuditLoggingInterceptor.java is responsible for intercepting HTTP requests for logging and auditing purposes. It extends Spring’s interceptor mechanism to log incoming request details before they reach the controller layer. Its inclusion strengthens observability across the application and can support compliance and security auditing in production environments.

**configuration.security** — Spring Security and TLS Configuration

The root of the security package contains two core classes: OpenSSLConfig.java and SecurityConfig.java. The OpenSSLConfig class is responsible for loading the TLS keystore and ensuring that HTTPS is used for all communication. It integrates with the application’s TLS certificate setup to secure network traffic. SecurityConfig.java configures Spring Security, defining which endpoints require authentication, the use of JWT-based stateless sessions, and how filters are applied in the request chain. It specifies protected routes, CORS settings, and disables CSRF (appropriate for token-based authentication), thus forming the foundation of the application’s access control model.

**configuration.security.jwt** — JWT Filter and Service

This package contains the implementation of JWT-based security. JwtAuthenticationFilter.java extracts and verifies the JWT from the Authorization header of incoming HTTP requests. It uses Spring Security’s OncePerRequestFilter to ensure that authentication is applied consistently across the request lifecycle. JwtService.java handles token generation, signature creation, and decoding. It uses the HS256 algorithm to sign tokens with a shared secret. The service embeds user identity and roles into the token claims, allowing for lightweight and stateless authentication throughout the application.

**configuration.sanitization** — Input Normalization

InputSanitizer.java is a utility class responsible for sanitizing user input to prevent malformed or malicious data from propagating through the system. It standardizes inputs by trimming whitespace, validating character sets, or stripping unsafe sequences. This layer of sanitization complements security by mitigating injection risks and data inconsistency before persistence or logic execution.

**configuration.web** — Web Layer Setup

WebConfig.java configures application-wide web behavior such as static resource handling, CORS, and endpoint mappings. It may also register interceptors like AuditLoggingInterceptor, ensuring they apply to specific routes. This class centralizes the configuration for all Spring MVC–related components and plays a critical role in linking the security, auditing, and controller infrastructure.

**domain.game** — Game Outcome Representation

The GameResult.java class encapsulates the outcome of a dice game session. It contains all relevant fields such as the server’s dice roll, the client’s guess, and the random strings used in the commitment protocol. This class serves as a data structure to represent both the result itself (win or loss) and the necessary cryptographic materials to verify the game’s fairness. It is used internally in the application layer and returned to the client as part of the reveal phase, allowing the client to recompute and compare the commitment hash.

**domain.user** — Core User Model and Authentication Input

The User.java class represents the domain entity for an authenticated player. It contains fields such as username, password hash, role, and state information related to game participation. This class is essential for linking security credentials with gameplay data and is used across persistence, authentication, and game logic layers. LoginRequest.java is a lightweight DTO-like object that holds the username and password used during authentication. Although it's located in the domain layer, it acts more like a value object passed from the web layer into the application logic for credential validation.

**domain.util** — Cryptographic and Randomization Helpers

The util package includes supporting tools for cryptographic operations and randomness, central to the fairness of the dice game. CryptographicTools.java provides static methods for computing secure hashes (e.g., SHA-256), which are used to generate the commitment sent to the client. RandomNumberGenerator.java is likely an interface or class defining methods for secure random generation, while RandomNumberGeneratorFactory.java provides a way to instantiate RNGs, possibly with deterministic control for testing. These classes ensure the reproducibility and integrity of random values used in gameplay, contributing to both fairness and security.

**domain.validation** — Input Validation Utility

ValidationErrorCollector.java is a helper class for accumulating validation errors during user registration, updates, or game actions. It enables the backend to gather multiple validation failures and return a single coherent response to the client. This approach improves user feedback, as multiple input issues can be addressed in one response rather than triggering repeated client-server interactions. It enhances the robustness of input validation across the domain.

**infrastructure.adapter.primary.web.controller** — REST Controllers

The AuthenticationController.java class provides the REST endpoints for user login and registration. It delegates requests to application services such as UserCommandService and JwtService, validating incoming credentials and returning a signed JWT upon successful login. It forms the entry point for authentication flows and exposes clear, RESTful endpoints (/api/auth/login, /api/auth/register) that the frontend interacts with. GameController.java, located in the nested game/ package, manages the endpoints related to the fair dice game. It implements the commit-reveal interaction by accepting a random string from the client, generating the commitment hash, and then later receiving the user’s guess and returning the full reveal. It coordinates closely with GameService and integrates with user validation to enforce the game flow.

**infrastructure.adapter.primary.web.dto** — Web Layer Data Transfer Objects

This package contains DTOs used for communication between the client and the backend API. JwtResponse.java wraps the JWT token returned to the client upon successful authentication. LoginRequestDTO.java and UserRegisterRequestDTO.java model the payloads received from the client during login and registration, respectively. These DTOs are intentionally decoupled from internal domain models, supporting a clean separation between external inputs and internal logic. GameResultDTO.java, located in the game/ subpackage, contains the outcome of a game round, including the dice roll, random strings, and win/loss status. It is returned to the frontend after the guess phase so the client can verify fairness.

**infrastructure.adapter.primary.web.mapper** — DTO Mappers

This package contains mapping logic that converts between DTOs and internal domain models. LoginRequestDTOMapper.java and UserRegistrationRequestDTOMapper.java transform incoming user input into LoginRequest or User domain objects used by the application layer. The game/GameResultDTOMapper.java converts a GameResult domain object into a format suitable for client-side verification and display. These mappers support modularity by ensuring that controllers do not directly manipulate domain models and maintain the boundary between the web interface and the core logic.

**infrastructure.adapter.secondary.dice** — Randomness Provider

RandomResourceGeneratorService.java implements the RandomResourceGeneratorPort interface and is responsible for generating secure random strings and dice rolls used in the commit phase of the fair dice protocol. It ensures that the server’s roll and random string are unpredictable and unique per session, contributing to the integrity and fairness of the game. By encapsulating random generation in this service, the system supports testability and future replacement with more robust entropy sources if needed.

**infrastructure.adapter.secondary.persistence.entity** — JPA Entity Models

This package holds the persistence-layer representations of domain models. game/GameResultEntity.java is a JPA entity that stores the outcome of a game, including server guess, client guess, and associated random strings. It maps directly to a database table for historical or audit purposes. user/UserEntity.java maps the user domain model to a relational database structure, containing fields like username, hashed password, role, and current game state. These entities bridge the domain logic with the persistence layer and are used throughout repository and service classes.

**infrastructure.adapter.secondary.persistence.mapper** — Domain–Entity Translators

This subpackage includes mappers that convert between JPA entities and domain models. game/GameResultEntityMapper.java transforms GameResult domain objects into their corresponding persistent forms and vice versa. Similarly, user/UserEntityMapper.java handles the mapping between User and UserEntity, enabling seamless transitions between business logic and database operations. These mappers ensure that persistence concerns do not leak into the domain layer and maintain the integrity of the architecture.

**infrastructure.adapter.secondary.persistence.repository** — Data Access Interfaces

This package contains Spring Data JPA interfaces for direct database access. GameResultRepository.java and UserRepository.java define methods for querying, saving, and updating entities using standard JPA conventions. These repositories serve as the final link in the persistence chain and are automatically implemented by Spring at runtime, allowing the higher layers to remain agnostic to SQL and ORM details.

**infrastructure.adapter.secondary.persistence.service** — Persistence Service Implementations

GamePersistenceService.java and UserPersistenceService.java implement the secondary ports GamePersistencePort and UserPersistencePort, respectively. They mediate between the application services and the JPA repositories. These services handle entity conversion via mappers, enforce data access rules, and abstract away persistence logic from the core business logic. They provide transactional boundaries and are designed to be reusable and testable.

**infrastructure.adapter.secondary.security** — Hashing Service

SHA256HashingService.java implements the HashingPort and provides cryptographic hashing functionality using the SHA-256 algorithm. It is used by the game logic to compute commitment hashes during the commit-reveal protocol. This service encapsulates the hashing algorithm, allowing it to be reused or swapped without affecting higher-level logic. It is a core part of the application’s fairness and integrity guarantees.

**Code Analysis FrontEnd**

**frontend-module/src** — React Frontend Root

The src/ directory contains the core logic of the React frontend. It includes the root entry point (index.js) and the main routing component (App.js). The application uses React Router to manage navigation between authentication views and game states. The routing logic defined in App.js maps specific paths like /login, /register, /digital-dice-game, and /guess to their corresponding React components. The structure is clear and modular, separating concerns by purpose (pages, components, services).

**components/** — UI and Functional Game Pages

This directory holds the React components that implement the application's interactive features. DigitalDiceGame.js handles the initial phase of the protocol, where the client generates a random string and receives a cryptographic hash from the backend. GuessDice.js is responsible for submitting the user's guess and verifying the fairness of the server’s reveal using a recomputed hash. Login.js and Register.js implement JWT-based authentication, storing the token in localStorage and redirecting to the game on success. Home.js serves as the landing page with navigation options. These components use React’s useState and useNavigate hooks and implement input validation, token handling, and HTTP communication via the fetch API.

**services/** — Backend API Abstractions

The services directory contains reusable modules for backend interaction. authService.js handles POST requests for login and registration, encapsulating API endpoints and payload formatting. gameService.js manages gameplay-related communication, including initiating the commitment and submitting the guess. These modules isolate network logic from UI components, promoting maintainability and allowing components to focus on rendering and state transitions.

**public/ and TLS Assets** — Static and Security Configuration

The public/ folder includes index.html, the main HTML scaffold for the React application. TLS configuration files (server.crt, server.key, and nginx.conf) enable the frontend to serve over HTTPS locally using a reverse proxy, aligning with the backend’s secure configuration. This setup ensures encrypted communication across the application and satisfies the assignment's requirement for TLS-based data exchange.

**Presentation of the UI and a real game example**

**A screenshot of a login box

AI-generated content may be incorrect.A screenshot of a login register

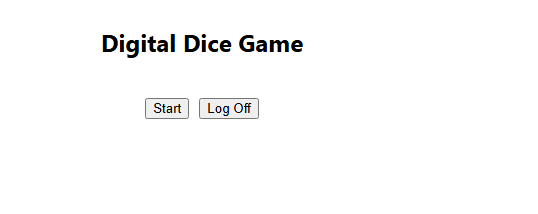
AI-generated content may be incorrect.**

On the left the home page of the app where the user can either login or register. On the right the login page where user gets prompted to enter username and password.

**A screenshot of a register

AI-generated content may be incorrect.**

The registration page is presented above, with all the mandatory fields the user has to complete in order to create a new account.

****

Once the account is created the user can either start the game or log off. If the game is started a random string is generated and send to server.

**A screenshot of a game

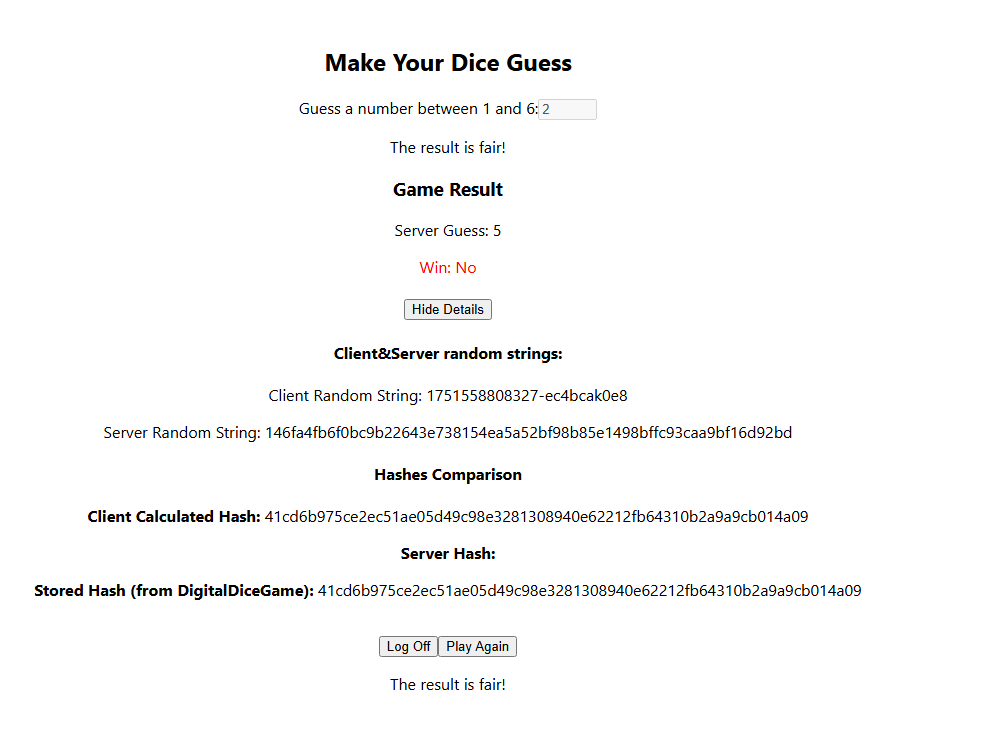
AI-generated content may be incorrect.**

Then it is time to make a guess and submit it.

**A screenshot of a game

AI-generated content may be incorrect.**

The result is presented after the submission of the guess, along with the information on whether the game was fair or not. By pressing the show details button the user can see the hashes exchanged so that he know why the game was fair or not.

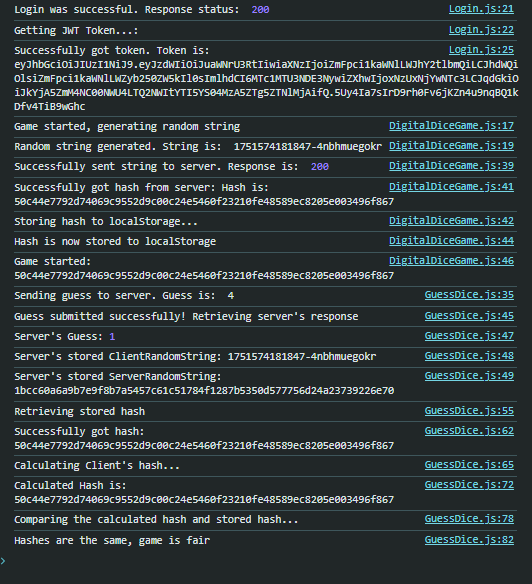
****

Finally he can choose to play again or log off, after the completion of the game.

**A screenshot of a game

AI-generated content may be incorrect.**

Everything that takes place in the web app, is logged into the browsers console, so that all the exchange between the client and the server is visible and transparent.



Prevention of SQL Injection:

The application leverages Springboot’s Spring JPA which automatically generates the prepared statements during runtime. The framework is an industry standard and a mature solution backing most if not all Spring/Springboot applications. The technology implements measures to avoid most well-known SQL injection attacks out of the box. Finally, the Data Transfer Objects (DTOs) use validation that is performed during the retrieval of the object. The configuration allows for a secure retrieval of serialized objects, which in turn will ignore all unknown fields that do not conform to the schema. For example, the user request DTO class uses the following annotations. If a single field doesn’t conform to the schema the request will not be processed and a response with a HTTP BAD RESPONSE will be returned to the invoker.

A screenshot of a computer program

AI-generated content may be incorrect.

Finally, we have implemented a Sanitization service on incoming Strings that removes special characters that could break the prepared statements, like the control characters, invisible characters and collapses all whitespaces as shown in following set of regular expressions:

A screen shot of a computer program

AI-generated content may be incorrect.

Logging:

The backend of the application uses extensive logging on almost the significant methods by utilizing the Slf4j library. Slf4j is an industry standard for allowing the secure logging of the application. The current appenders do not write on disk but on console (std out), this configuration can be adjusted but we didn’t deem as necessary for the project needs.

Logging adheres to the clean architecture standards and as such no user-related information is leaked when an operation is performed. We have also implemented audit logs by intercepting the HTTP requests that the spring framework detects. We are leveraging the HandlerInterceptor spring class by extending it, allowing us to log the inbound requests. The audit logs contain the HTTP endpoint (path and method) that was invoked as well as the user public IP address. This logging is transparent to the application and takes place before spring invokes our endpoint.

**Redirection of HTTP to HTTPS**

The implementation has foreseen the redirection of http to https. The nginx that hosts the frontend automatically has the following configuration:

nginx.conf A screenshot of a computer program

AI-generated content may be incorrect.