Etoile de bagarre - Technical documentation

TESSIER Cécile, CARREAU Thomas, PICHOU Charles

Table of contents

Third-party libraries used

The libraries that we use:

1. **Jetpack Compose**: application design.
2. **LibGdx**: game engine.
3. **Fleks**: A Fast, Lightweight, Entity component System library written in Kotlin.
4. **Ktor**: Websocket.
5. **Firebase plugin**: for communication with Firebase.
6. **GoogleId**, **gmsGoogleServices** and **CredentialsPlayServicesAuth**: to connect to the application using Google.

Library Choices

Game engine

For our project, we initially considered KorGE, a game engine designed for Kotlin. However, installation and compatibility issues between KorGE, Gradle, and Jetpack Compose led us to change direction.

We decided to go with LibGDX, which is also compatible with Kotlin. Although a bit older, it is open source, has solid documentation, and offers numerous online examples. Additionally, we found a tutorial suited to the type of game we want to develop.

We also used Fleks, which is an extension of libgdx.

Client-server synchronization

We use Ktor, it's a library frequently used in Kotlin to create asynchronous client-server applications.

Authentication

Firebase Authentication simplifies user management in Android projects by providing a secure and scalable solution for various sign-in methods, including email/password, and google sign-in account.

Game engine (LibGdx)

Entity-Component-System (ECS)

"**Entity–component–system**[1] **(ECS)** is a software architectural pattern mostly used in video game development for the representation of game world objects. An ECS comprises entities composed from components of data, with systems which operate on the components."

- **Entity:** An entity represents an "object" in the game (e.g., a player or an enemy). An entity itself is just an identifier, often an integer or a lightweight class.
- **Component:** Components are "building blocks" that describe the data or attributes of an entity (e.g., its position, speed, health). These are simple classes that contain only data.
- **System:** Systems contain the game logic. They act on entities that have specific components during each game loop.

Firebase configuration

First, you need to go to https://console.firebase.google.com, create a new project, enable Google Analytics, and select Default Account for Firebase.

Authentication (with email)

1. In the left-hand bar, click **All products**.
2. Select **Authentication** and click **Get Started**.
3. In the Sign-in method tab:
    a. Enable **Anonymous Native provider** and click Save.
    b. Click **Add new provider**, select **Email/Password**, enable it (do not enable the Email link option), and click Save.

Authentication (with Google Sign-In)

1. In the left-hand bar, click **All products**.
2. Select **Authentication** and click **Get Started**.
3. In the Sign-in method tab:
    a. Enable **Anonymous Native provider** and click Save.
    b. Click on **Add new provider**.
    c. Click on the **Google icon**, enable it.

[1] *https://en.wikipedia.org/wiki/Entity_component_system*

    d. In **Support email for project**, select your email.
    e. Click Save
4. In Android Studio:
    a. Click on the **Gradle icon**.
    b. Click on **Tasks > android > signingReport.**
    c. Search for **SHA1**, copy.
5. In Firebase, in the project setting:
    a. Go to **Your apps** section.
    b. Click on **Add fingerprint**.
    c. Paste the **SHA1**.

Firestore Database

1. In the left-hand bar, click **All products**.
2. Select **Cloud Firestore** and click **Create database**.
3. Choose a European location, select **Test mode**, and click **Create**.

Server

Main features

This **WebSocket** implementation is designed for a multiplayer game to handle real-time communication between players and the server. It manages player connections, assigns players to rooms, tracks their positions, health, and game states, and broadcasts updates to all players in a room. Additionally, it handles game events like hits, deaths, and game completion, ensuring seamless synchronization between players. The server supports dynamic room creation and provides an efficient game loop for real-time updates.

Refer to **NomDuJeu/backend/server.md** for detailed instructions on preparing and deploying the WebSocket server using Docker.

Technologies

- **Node.js:** Server management.
- **WebSocket:** Real-time communication between clients
- **Modules:**
    - **room-manager.js**: Manages rooms and positions.
    - **client-handler.js**: Handles player actions.
- **Docker:** Used to set up the WebSocket on a server (see *backend/server.md* for server preparation and WebSocket deployment).