WP4-A1. Erstellung der Datenbank für das E-Learning-Tool.

„Finanziert durch die Europäische Union. Die geäußerten Ansichten und Meinungen sind jedoch ausschließlich die der Autoren und spiegeln nicht unbedingt die der Europäischen Union oder der Exekutivagentur Bildung, Audiovisuelles und Kultur (EACEA) wider. Weder die Europäische Union noch die EACEA können dafür verantwortlich gemacht werden.“

Inhalt

1. EINLEITUNG 4
2. ÜBERBLICK ÜBER DIE DATENARCHITEKTUR 5
2.1. Allgemeine Konzepte 5
2.2. Hauptkomponenten 6
2.2.1. React Native Mobile Client 6
2.2.2. Firebase-Backend: Firestore und Functions 7
2.2.3. WebSocket-Server auf Google Cloud Run (Echtzeit-Ebene) 9
3. DATENMODELL UND DATENBANKDESIGN 11
3.1. Firestore-Sammlungen und -Dokumente 11
3.1.1. Spiele – globaler Spielstatus 11
3.1.2. Benutzer – Profil- und spielbezogene Daten 12
3.1.3. blockchain/{gameId}/blocks – Mining-Probleme und gewonnene Blöcke 13
4.1.4. Produkte – globaler Produktkatalog 14
4.1.5. Markt – Marktstruktur nach Runden 14
3.2. Beziehungen zwischen Entitäten 15
3.2.1. Spiel – Spieler 15
3.2.2. Spiel – Mining-Blöcke 16
3.2.3. Spiel – Markt 16
3.2.4. Produkte – Markt / Lagerbestände 16
3.2.5. Wie werden der Status eines Spiels und eines Spielers rekonstruiert? 16
4. WICHTIGE DATENFLÜSSE IN DER PRODUKTION 17
4.1. Erstellung und Beitritt zu Spielen 17
4.1.1. Erstellung von Spielen 17
4.1.2. An einem bestehenden Spiel teilnehmen 17
4.1.3. Verbindung zur Echtzeit-Ebene 18
4.2. Marktaktualisierungen und Runden 18
4.2.1. Markterzeugung auf dem WebSocket-Server 19
4.2.2. Emission in Echtzeit und Rundenverlauf 19
4.3. Mining, Validierung und Belohnungen 20
4.3.1. Erstellung von Blöcken und Problemen 20
4.3.2. Echtzeit-Verteilung und Einreichung von Antworten 20
4.3.3. Validierung und Persistenz der Mining-Ergebnisse 21

4.3.4. Konsolidierung der Belohnungen am Ende der Runde 22
5. SICHERHEIT, DATENSCHUTZ UND PRODUKTIONSREIFE 24
5.1. Firestore-Sicherheitsregeln und Authentifizierung 24
5.1.1. Authentifizierung als Zugangstor 24
5.1.2. Eingeschränkter Zugriff auf Profil- und Spieldaten 24
5.1.3. Durch Firebase-Funktionen gesteuerte Schreibvorgänge 24
5.1.4. WebSocket-Server als vertrauenswürdiger Dienst 25
5.2. DSGVO und Datenschutz 25
5.2.1. Datenminimierung und Zweckbindung 26
5.2.2. Pseudonymisierung von Spielern 26
5.2.3. Rechtsgrundlage und Informationen für die Teilnehmer 26
5.2.4. Speicherort und Sicherheitsmaßnahmen 26
5.2.5. Aufbewahrung, Anonymisierung und Löschung 27
5.3. Backups, Bereinigung und grundlegende Wartung 27
5.3.1. Backups und Wiederherstellungsoptionen 27
5.3.2. Bereinigung alter Spiele und Pilotdaten 28
5.3.3. Überwachung und grundlegende Betriebsprüfungen 28
6. NÄCHSTE SCHRITTE UND SCHLUSSFOLGERUNGEN 29

1. EINLEITUNG

Dieser Bericht fasst die endgültigen Ergebnisse von WP4.A1 zusammen: Erstellung der Datenbank für das E-Learning-Tool. Im Gesamtrahmen des RockChain-Projekts konzentriert sich diese Aktivität auf den Entwurf und die Realisierung der Backend-Datenebene, die das Serious Game unterstützt, sodass Multiplayer-Sitzungen, Marktdynamiken und Mining-Ereignisse in Echtzeit auf einer stabilen und produktionsreifen Infrastruktur ausgeführt werden können.

Die entscheidende Aufgabe von WP4.A1 besteht darin, eine Datenbank und eine Backend-Architektur zu definieren und zu entwickeln, die eine konsistente Verarbeitung von Benutzerprofilen, Spielsitzungen, Runden, Belohnungen und Entscheidungen im Spiel gemäß den im Projekt definierten Lernszenarien unterstützen kann. Die aktuelle Implementierung einer solchen Architektur basiert auf *Firebase Firestore* als zentraler Datenbank, einer Reihe von Firebase-Funktionen, die sensible Schreibvorgänge und Spielelogik kapseln, sowie einem dedizierten *WebSocket-Server*, der auf *Google Cloud Run* bereitgestellt wird und die Echtzeitkommunikation mit geringer Latenz zwischen den Spielern verwaltet.

Dieser Bericht hat einen bewusst engen und technischen Fokus: Er dokumentiert, wie die RockChain-Datenebene strukturiert und in der Produktion implementiert ist. Er beschreibt die Hauptkomponenten der Architektur und deren Hosting, die Struktur der wichtigsten Firestore-Sammlungen und -Dokumente sowie die relevantesten Datenflüsse, die diese Informationen während eines Spiels erstellen und aktualisieren. Außerdem fasst er wichtige Produktionsaspekte in Bezug auf Sicherheitsstandards, Leistung und grundlegende Datenbankwartung zusammen.

Mit dieser prägnanten Beschreibung der Datenschicht bietet WP4.A1 eine praktische Referenz, die Entwickler und technische Partner für die Implementierung, den Betrieb oder die Erweiterung der RockChain-Infrastruktur nutzen können. Die daraus resultierende Architektur bildet die Grundlage, auf der sowohl das Front-End-E-Learning-Tool als auch die Pilotaktivitäten der nachfolgenden Arbeitspakete zuverlässig aufgebaut werden können.

2. ÜBERBLICK ÜBER DIE DATENARCHITEKTUR

2.1. Allgemeine Konzepte

Das RockChain-E-Learning-Tool läuft in der Produktion auf einer dreischichtigen Architektur, die sich um ein einziges Firebase-Projekt und einen dedizierten Echtzeitdienst dreht:

- Der **mobile Client React Native** (Android/iOS) bietet die Benutzeroberfläche und verbindet die Lernenden mit dem Spiel.
- **Das Firebase-Backend** mit Firestore, Functions und Authentication speichert alle persistenten Daten und kapselt sensible Vorgänge.
- Der auf Google Cloud Run bereitgestellte **WebSocket-Server** wickelt die Echtzeitkommunikation zwischen den Spielern während eines Spiels mit geringer Latenz ab.

Auf hoher Ebene folgt die Architektur einem hybriden Modell:

- Firestore dient als autoritative Datenbank, in der Spiele, Benutzer, Blöcke, Produkte, Märkte und Belohnungen gespeichert werden.
- Firebase Functions fungiert als kontrolliertes Gateway für kritische Schreibvorgänge wie das Erstellen von Spielen, das Beitreten zu Spielen und das Aktualisieren von Runden- und Belohnungsinformationen und wendet Validierungs- und Geschäftsregeln auf der Serverseite an.
- Der WebSocket-Server speichert den Status jedes aktiven Spiels im Arbeitsspeicher und überträgt Echtzeitereignisse wie Rundenstart, Marktdaten, Mining-Probleme und Ergebnisse an alle verbundenen Clients, wobei relevante Ergebnisse nur an genau definierten Punkten in Firestore gespeichert werden.

Da alle Komponenten dieselbe Firebase-Identität und denselben Sicherheitskontext teilen, sind Authentifizierung, Autorisierung und Datenzugriff über den gesamten Prozess hinweg konsistent. Diese Architektur kann für die Entwicklung und das Testen mit einem anderen Firebase-Projekt und einer anderen Implementierung von WebSockets repliziert werden, wobei die genaue Struktur beibehalten wird, aber nicht produktionsbezogene Anmeldedaten und URLs verwendet werden.

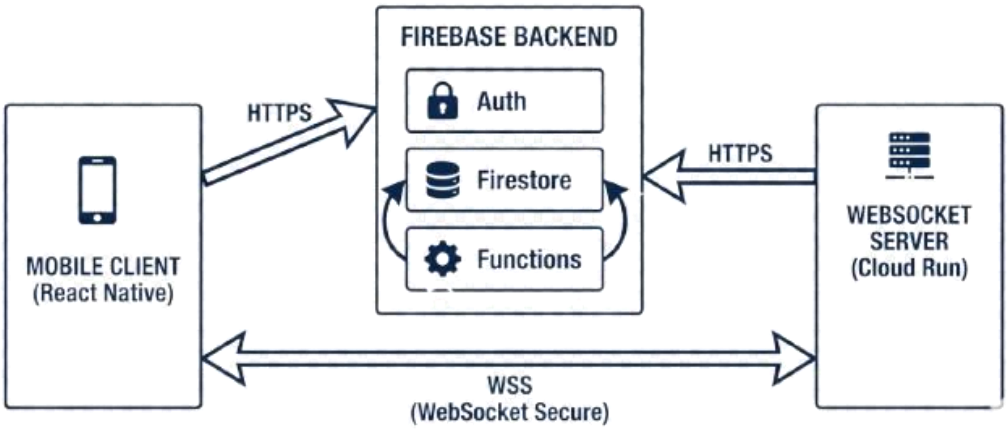

Abbildung1 : Architekturdiagramm

Durch diese Aufgabentrennung kann RockChain eine dauerhafte, gut strukturierte Speicherung der Ereignisse jeder Sitzung mit schnellen, ereignisbasierten Aktualisierungen der aktuellen Ereignisse während des Spiels kombinieren.

2.2. Hauptkomponenten

2.2.1. React Native Mobile Client

Die mobile Anwendung React Native ist **der einzige Einstiegspunkt für Benutzer**. Der Zweck besteht darin, die Spieloberfläche darzustellen und auf den Backend-Status zu reagieren, während die App selbst keine Persistenz verarbeitet.

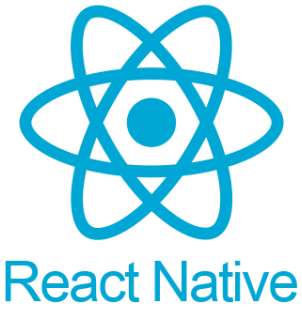

Abbildung2 : React Native

In der Produktion führt der mobile Client Folgendes aus:

- Authentifizierung mit Firebase Authentication und Abruf einer Benutzeridentität (*userId*), die im gesamten Backend konsistent verwendet wird.
- Tritt über dedizierte Firebase-Funktionsaufrufe, wie z. B. „Spiel erstellen“ oder „Spiel beitreten“, die zur Validierung der Anfrage und anschließenden Aktualisierung von Firestore dienen, an Spielen teil.

- Er öffnet eine WebSocket-Verbindung zum Cloud Run-Endpunkt, sobald ein Benutzer in einem Spiel ist, und sendet die *Game-ID*, *die* authentifizierte *User-ID* und grundlegende Spielerinformationen, damit der Socket in den entsprechenden Spielraum gestellt werden kann.
- Abonniert Backend-Ereignisse (entweder von Firestore-Listenern oder WebSocket-Nachrichten) und ordnet sie dem lokalen App-Status zu, der dann auf dem Bildschirm dargestellt wird.

Der Client schreibt bei kritischen Vorgängen nicht direkt in Firestore. Stattdessen:

- Firebase-Funktionen sollten immer dann aufgerufen werden, wenn eine dauerhafte Änderung des Datenmodells erforderlich ist, beispielsweise beim Aktualisieren des Spielstatus und beim Schreiben von Belohnungen.
- Sie senden WebSocket-Ereignisse wie *„Spielbereit“*, *„Produkt kaufen“* oder *„Mining-Antwort senden“*, die auf dem Server verarbeitet und gegebenenfalls gespeichert werden.

Durch dieses Design bleibt die mobile Anwendung eine schlanke, reaktive Schicht, die sich auf die Benutzerinteraktion konzentriert, während alle maßgeblichen Entscheidungen über den Spielstatus im Backend getroffen werden.

2.2.2. Firebase-Backend: Firestore und Functions

Das Firebase-Backend besteht aus Cloud Firestore mit Firebase Funktion, die zusammen eine sichere und skalierbare Datenschicht bilden.

Abbildung3 : Firebase

Cloud Firestore (Datenbank und maßgeblicher Speicher)

Firestore speichert alle Informationen, die eine WebSocket-Verbindung überstehen und für Analysen oder Wiederherstellungen verfügbar sein müssen. Zu den in RockChain gespeicherten Informationen gehören unter anderem

- **Spielstatus** in der Spielesammlung: Jedes Dokument steht für ein Spiel. Es enthält den Spielcode, die Liste der Spieler, die aktuelle Runde und den Status, die Gewinnerbranche pro Runde, Timer und zugewiesene Belohnungen.
- **Benutzerstatus** in der Benutzersammlung: Jedes Dokument speichert das Benutzerprofil – Anzeigename, Avatar – und spielbezogene Daten: rockCoins, angesammelte Abfälle, ausgewählte Branche, Inventar

- **Mining- und „Blockchain-inspirierte" Daten** in blockchain/{*gameId*}/*blocks*: Für jedes Spiel speichert diese Unterkollektion Mining-Probleme, erhaltene Antworten und den endgültigen Gewinner jedes Blocks.
- **Produktkatalog** in der Produktsammlung: Vorlagen für die verfügbaren Marmorprodukte (Typ, Marmortyp und Beschreibung), die zum Aufbau von Marktangeboten pro Runde verwendet werden.
- *Markt*: **Marktkonfiguration** pro Spiel. Die Ausgabe ist eine übersichtliche Darstellung der Marktparameter nach Runde und Produkttyp. Dies ist nützlich, um den Markt in anderen Kontexten als den Echtzeit-Flachlisten neu aufzubauen oder anzuzeigen.

In diesem Fall ist Firestore für kurze, häufige Lesevorgänge durch einen mobilen Client – aktueller Spielstatus, aktualisierter Bestand usw. – und kontrollierte Schreibvorgänge aus dem Backend-Code optimiert, die die Integrität und Konsistenz des Spielmodells gewährleisten.

Firebase-Funktionen (Geschäftslogik und kontrollierte Schreibvorgänge)

Firebase-Funktionen zentralisieren die wichtigsten Geschäftsregeln und sensiblen Änderungen an Firestore. Der Client schreibt nicht direkt in kritische Dokumente wie Spiele oder Benutzer-Spieldaten, sondern ruft Funktionen auf, die Folgendes tun:

- **Spiel erstellen**, *createGame*: Erstellt ein Dokument in Spielen mit dem Spielcode, dem Host, dem Status, der maximalen Spieleranzahl und anderen Standardfeldern.
- Einem Benutzer die **Teilnahme an einem Spiel** ermöglichen, zum Beispiel: *joinGame* überprüft, ob ein Spiel existiert und ob man daran teilnehmen kann; fügt den Benutzer zur Liste der Spieler hinzu; aktualisiert *playerCount* und erstellt/aktualisiert deren Startwerte: *rockCoins*, *waste*, *status*, timestamps….
- **Aktualisieren** der sensiblen Felder im Spiel: zum Beispiel Rundeninformationen, aggregierte Ergebnisse oder Belohnungen, die durch Client-Ereignisse oder als Ergebnis von Backend-Prozessen am Ende einer Runde ausgelöst werden.

```
exports.createGame = onCall({
  memory: '256MiB',
  timeoutSeconds: 60,
  region: 'us-central1',
  minInstances: 0,
  maxInstances: 10,
  concurrency: 80,
  retry: false,
  ingressSettings: 'ALLOW_ALL'
}, async (request) => {
  const userId = request.auth?.uid;
  const gameCode = generateGameCode();

  if (!userId) {…
  }

  const gameData = {…
  };

  try {…
  } catch (error) {
    console.error('❌ Error al crear el juego:', error);
    throw new Error(`Failed to create game: ${error.message}`);
  }
});

function generateGameCode() {
  const characters = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789';
  let code = '';
  for (let i = 0; i < 6; i++) {
    code += characters.charAt(Math.floor(Math.random() * characters.length));
  }
  return code;
}
```

Abbildung4 : createGame-Funktion

Durch die Weiterleitung dieser Vorgänge über Funktionen erreicht das Projekt Folgendes:

- sorgt für konsistente Validierung und Regeln innerhalb jeder Sitzung.
- vereinfacht die Firestore-Sicherheitsregeln, da komplexe Entscheidungen im Backend-Code verbleiben und nicht in umfangreichen Regelsätzen.
- Reduziert das Risiko böswilliger oder inkonsistenter Schreibvorgänge von nicht vertrauenswürdigen Clients.

In der Praxis ist der maßgebliche Pfad immer: *Client → Firebase-Funktion/Backend-Logik → Firestore***.**

2.2.3. WebSocket-Server auf Google Cloud Run (Echtzeit-Ebene)

Der WebSocket-Server ist ein mit Node.js erstellter Dienst, der als Container auf Google Cloud Run ausgeführt wird. Seine Hauptaufgabe besteht darin, **alle Spieler** während des Spiels **miteinander zu verbinden** und eine schnelle, unterbrechungsfreie bidirektionale Kommunikation zu gewährleisten.

Was genau macht er? Hier ist die Antwort:

- Er sorgt dafür, dass **die Kommunikation** in jedem Spiel **in Echtzeit** fließt. Dazu gehören Benachrichtigungen darüber, wer beitritt, wer bereit ist, wann eine Runde beginnt, wie sich der Markt entwickelt, neue Mining-Probleme, Ergebnisse und wann jede Runde oder das gesamte Spiel endet.

- Es verfolgt jedes aktive Spiel im Speicher. Auf diese Weise kann es die Vorgänge koordinieren, ohne Firestore zu überlasten. Zum Beispiel:
    - Wer nimmt teil und ob die Teilnehmer bereit sind.
    - In welcher Runde sie sich befinden und wie lange es noch dauert, bis sie endet.
    - Wie sich die Lagerbestände und der Markt zu diesem Zeitpunkt entwickeln.
    - Welche Mining-Probleme aktiv sind und welche Antworten gesendet wurden.
- Es speichert die Ergebnisse auch in Firestore, jedoch nur zu bestimmten Schlüsselmomenten: wenn jemand einen Block gewinnt, am Ende einer Runde oder wenn Belohnungen vergeben werden. Manchmal geschieht dies direkt, manchmal werden die Daten an Firebase-Funktionen weitergeleitet.

Da es auf Cloud Run läuft, skaliert sich der Dienst außerdem je nach Anzahl der aktiven Spiele. Und alles ist über einen sicheren WebSocket-Endpunkt (WSS) innerhalb desselben Firebase-Projekts verbunden, in dem sich Firestore und Functions befinden.

Zusammenfassend:

- Der WebSocket-Server **informiert Sie in Echtzeit über das Geschehen** im Spiel.
- **Firestore und Functions speichern den offiziellen Verlauf**: was passiert ist, wie das Spiel verläuft und in welchem Zustand sich die Spieler befinden.

Dank dieser Aufteilung kann RockChain Multiplayer-Spiele anbieten, die sich agil und flüssig anfühlen, ohne dabei die Kontrolle und Aufzeichnung der Ereignisse zu beeinträchtigen.

3. DATENMODELL UND DATENBANKDESIGN

In der Produktion dreht sich die Datenschicht von RockChain um einige wenige wichtige Sammlungen in Firestore. Jede davon hat eine klar definierte Funktion: Sie sind so konzipiert, dass sie ständig von der mobilen App gelesen werden, während Schreibvorgänge kontrolliert über Firebase-Funktionen und Backend-Prozesse ausgeführt werden.

Zusammen speichern diese Sammlungen:

- Den Gesamtstatus jedes Spiels.
- Den individuellen Status jedes Spielers innerhalb eines Spiels.
- Die Historie der Mining-Ereignisse.
- Die in jeder Runde verwendete Marktkonfiguration.

3.1. Firestore-Sammlungen und -Dokumente

3.1.1. Spiele – globaler Spielstatus

Die Spielesammlung speichert ein Dokument pro Spiel. Jedes games/{*gameId*} enthält Daten, die beschreiben, was in diesem Spiel im Allgemeinen geschieht.

Dies sind einige der wichtigsten Felder:

- *gameCode*: Der öffentliche Code, den die Spieler zum Beitreten verwenden.
- *players*: Die Liste der *Benutzer-IDs*, die am Spiel teilnehmen, zusammen mit Daten wie der Anzahl der Spieler (*playerCount*) und der maximal zulässigen Anzahl (*maxPlayers*).
- status: In welcher Phase sich das Spiel befindet (kann *„waiting”*, *„starting”*, *„in_progress”*, *„roundEnd”*, *„waitingForNextRound”* oder „finished” sein).
- round: Nummer der aktuellen Runde.
- *winningIndustry*: Branche, die die letzte vollständige Runde gewonnen hat.
- *roundTime*: Zeitstempel, der angibt, wann die aktuelle Runde endet; wird als Grundlage für die Anzeige von Zählern und Übergängen verwendet.
- *rewardsAssigned*: Ein boolescher Wert, der angibt, ob die Belohnungen für die letzte Runde bereits geschrieben wurden.
- *roundRewards*: Zusammenfassung der Belohnungen jedes Spielers in der vorherigen Runde (nach Branche, Abfallentsorgung, Bergbau usw.).

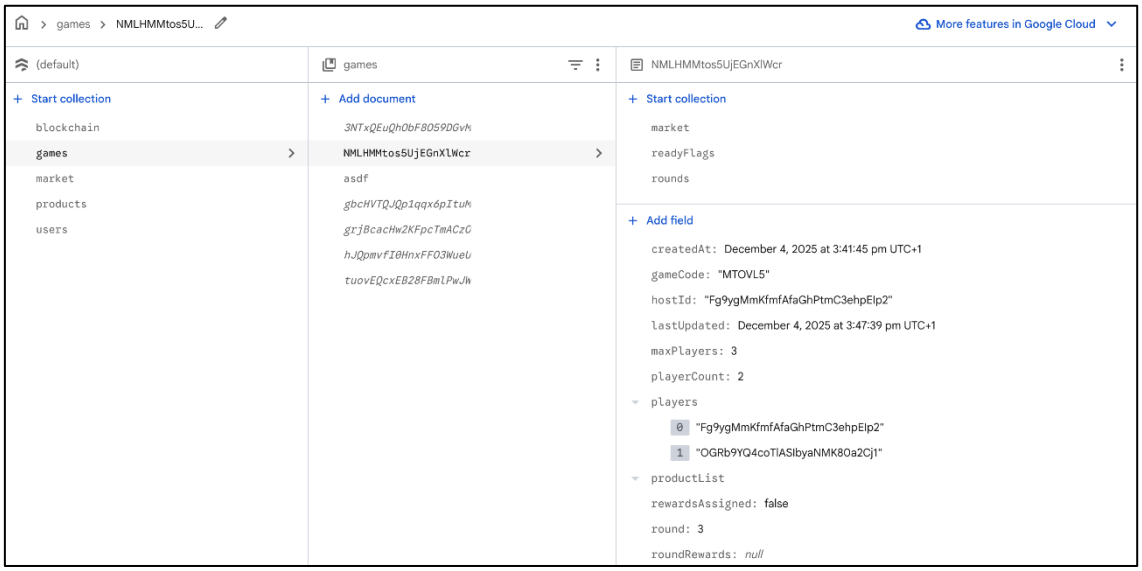

Abbildung5 : Erfassung von Spieldaten

Dieses Dokument ist das Hauptportal beim Laden oder Fortsetzen eines Spiels. Durch einfaches Lesen von games/{*gameId*} kann das System erkennen, wer spielt, in welcher Runde sich die Spieler befinden und ob die Belohnungen bereits zugewiesen wurden.

3.1.2. Benutzer – Profil- und spielbezogene Daten

Die Benutzersammlung speichert ein Dokument pro Benutzer, das anhand seiner *Firebase-Benutzer-ID* identifiziert wird. Jedes Dokument unter users/{*userId*} enthält zwei Ebenen von Informationen:

- **Benutzerprofil** mit Feldern wie *userName*, einem optionalen Avatar (*imageUrl*) und *lastUpdated*, das als Zeitstempel für grundlegende Audits dient.
- **Spieldaten**, die in einer Spielkarte organisiert sind, wobei jeder Eintrag mit einer *gameId* verknüpft ist. Für jedes Spiel, an dem der Benutzer teilnimmt, werden folgende Daten gespeichert:
    - Spielmünzen: *rockCoins* und angesammelte Abfälle.
    - Die Branche, die sie ausgewählt haben (*selectedIndustry*), um im Rahmen des Kreislaufwirtschaftsmodells Belohnungen zu verdienen.
    - Ihr Produktbestand (*products*), der zeigt, was sie auf dem Marktplatz gekauft haben.
    - Zusätzliche Daten wie G*ame-Code*, Status, Zeitpunkt des Beitritts (*joinedAt*) und die Mining-Warteschlange (*mining-Queue*).
    - Informationen über das letzte Mining-Problem, mit dem er interagiert hat (*lastMiningProblem*), einschließlich der *block-ID*, Problemdetails und eines Zeitstempels.

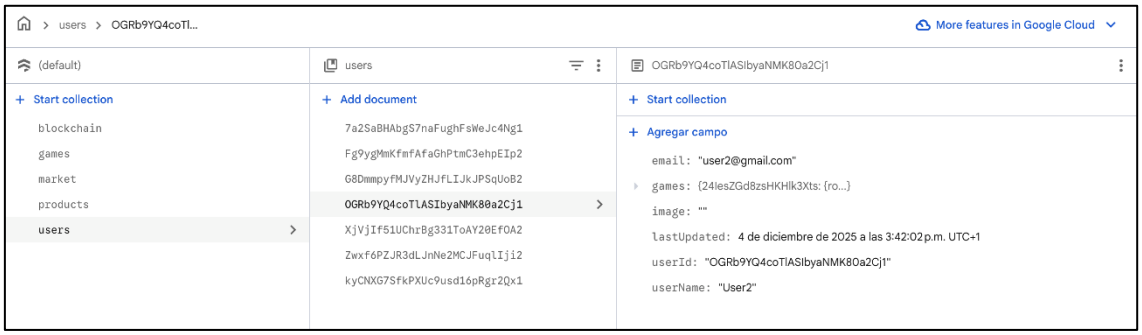

Abbildung6 : Datenerfassung von Benutzern

Die Konzentration all dieser Informationen in einem einzigen Dokument pro Benutzer vereinfacht die Dinge erheblich: So können Sie beispielsweise sofort die aktuelle Situation des Spielers in einem bestimmten Spiel erkennen, ohne mehrere Abfragen oder komplizierte Kombinationen durchführen zu müssen.

3.1.3. blockchain/{gameId}/blocks – Mining-Probleme und gewonnene Blöcke

Für jedes Spiel verfolgt die Unterkollektion *blockchain/{gameId}/blocks* Mining-Ereignisse unter Verwendung einer von der Blockchain inspirierten Struktur. Jedes Dokument *blocks/{block-ID}* repräsentiert einen Block mit den folgenden Informationen:

- Aktives Mining-Problem (*activeProblem*), das Folgendes umfasst:
  - *Problem-ID*, das mit dem *block-ID* übereinstimmt.
  - Eine einfache mathematische Aufgabe (*mathProblem*) zusammen mit ihrer richtigen Lösung.
  - Erstellungsdatum (*createdAt*) und Details der zugehörigen Transaktion (z. B. Produkt, Art der Operation und beteiligter Benutzer).
- Während der Aktivität des Problems erhaltene Antworten (*responses*).
- Die Rundennummer (*round*) und ein *createdAt,* das angibt, wann der Block erstellt wurde.
- Ein Gewinnerobjekt, das ausgefüllt wird, sobald der Gewinner des Mining-Wettbewerbs feststeht. Es enthält: *User-ID*, *Benutzername*, die gegebene Antwort, ob sie richtig war, die Zeit, die für die Antwort benötigt wurde (*elapsedTime*), und wann die Antwort gegeben wurde (*responseTime*), was für die Prüfung nützlich ist.
- Einen isActive-Indikator, der signalisiert, ob der Block noch offen ist oder ob es bereits einen Gewinner gibt.

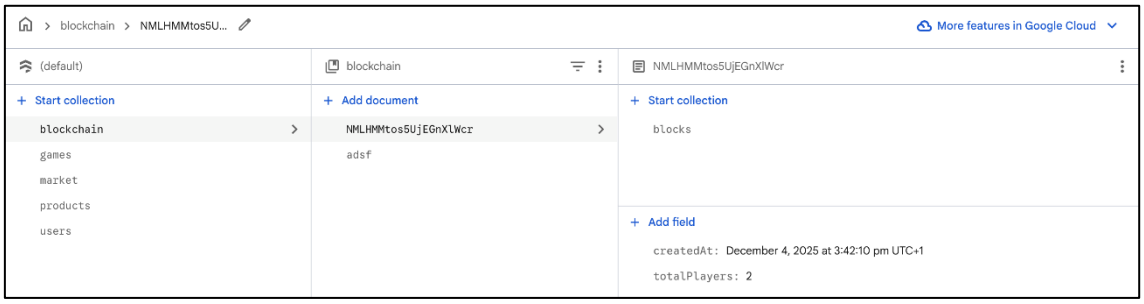

Abbildung7 : Blockchain-Datenerfassung

Diese Struktur ermöglicht es RockChain, eine klare und überprüfbare Aufzeichnung der generierten Probleme, der Antworten der Spieler und der Vergabe der Mining-bezogenen Belohnungen zu führen.

4.1.4. Produkte – globaler Produktkatalog

Die Produktsammlung definiert den globalen Katalog der im Spiel verfügbaren Marmorprodukte. Jedes Dokument fungiert als stabile Vorlage mit Feldern wie:

- *productId* (zum Beispiel: „product001")
- *Typ* („Block" oder „Platte")
- *marbleType* („Rot", „Weiß", „Grau", „Schwarz", „Creme")
- *quality* („A", „B", „C")
- Beschreibung und andere statische Daten

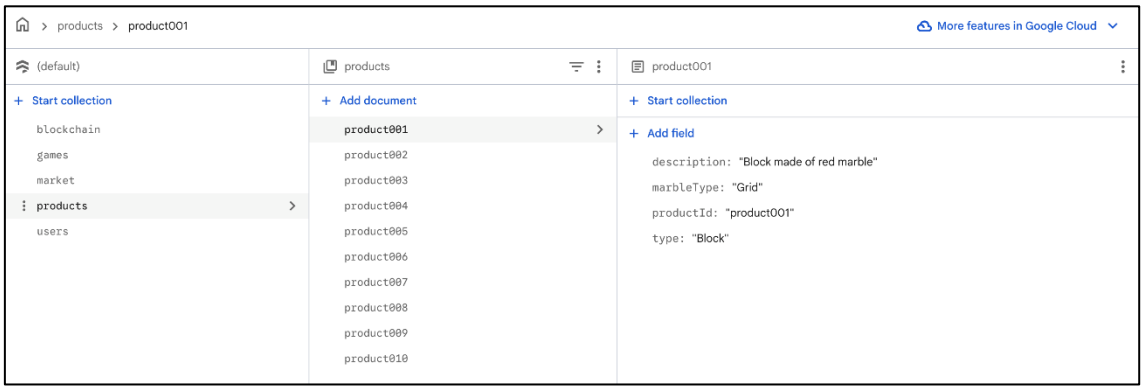

Abbildung8 : Produktdatenerfassung

Diese Vorlagen werden von der Backend-Logik verwendet, um die spezifische Liste der Produkte zusammenzustellen, die in jeder Runde auf dem Marktplatz angezeigt werden. Sie dienen als Grundlage für die Preisgestaltung und die Festlegung von Parametern im Zusammenhang mit Abfall.

4.1.5. Markt – Marktstruktur nach Runden

Für jedes Spiel speichert die Sammlung market/{*game-ID*} eine strukturierte Ansicht des Marktes. Diese Ansicht wird hauptsächlich verwendet, um die Konfiguration des Marktes in verschiedenen Runden zu rekonstruieren oder für nachfolgende Analysen.

Das Hauptfeld ist rounds: eine verschachtelte Karte mit der Struktur *rounds[round][marbleType][type][quality] = { price, waste }*, wobei:

- *„round"* die Rundennummer ist,
- *marbleType* einer der konfigurierten Murmeltypen ist,
- *type* „Block" oder „Slab" sein kann,
- *quality* „A", „B" oder „C" ist, jeweils mit den entsprechenden Werten für Preis und Abfallmenge.

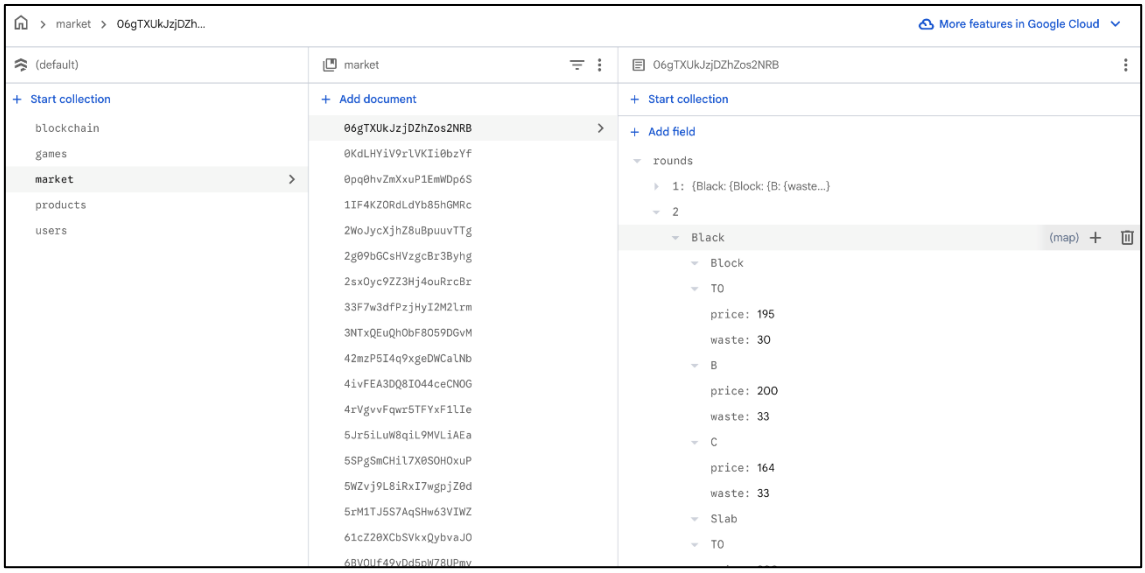

Abbildung9 : Marktdatenerfassung

Diese Struktur dient als eine Art Zusammenfassung oder historische Übersicht über den Markt. Sie ergänzt die Echtzeit-Produktlisten, die während des Spiels generiert und über WebSocket gesendet werden. Sie ist besonders nützlich, um zu überprüfen, wie sich der Markt in jeder Runde entwickelt hat, ohne jedes Produkt einzeln durchgehen zu müssen.

3.2. Beziehungen zwischen Entitäten

Obwohl Firestore eine dokumentenorientierte NoSQL-Datenbank ist und Beziehungen nicht wie eine relationale Datenbank verarbeitet, definiert das RockChain-Modell klare logische Beziehungen zwischen seinen Sammlungen.

3.2.1. Spiel – Spieler

Jedes games/{*game-ID*}-Dokument enthält seine Spieler in einem Spieler-Array, das die *user-IDs* enthält.

Gleichzeitig speichert jedes Dokument users/{*user-ID*} innerhalb der Zuordnung *games*[*game-ID*] den spezifischen Status dieses Benutzers in diesem Spiel. In der Praxis stellt dies eine viele-zu-viele-Beziehung zwischen Spielen und Benutzern dar, die mit Querverweisen implementiert wird.

3.2.2. Spiel – Mining-Blöcke

Jedes Spiel hat eine eigene Unterkollektion: blockchain/{*game-ID*}/*blocks*.

Alle Blöcke sind mit einer einzigen *game-ID* verknüpft und enthalten jeweils die Nummer der Runde, zu der sie gehören. Es handelt sich um eine Eins-zu-Viele-Beziehung: Ein Spiel kann mehrere Blöcke haben, aber jeder Block gehört nur zu einem Spiel.

3.2.3. Spiel – Markt

Jedes Spiel hat ein eindeutiges market/{*game-ID*}-Dokument, in dem der Markt nach Runden organisiert ist. Obwohl Firestore keine Beziehungen vorschreibt, handelt es sich logischerweise um eine Eins-zu-Eins-Beziehung zwischen games/{*game-ID*} und market/{*game-ID*}.

3.2.4. Produkte – Markt / Lagerbestände

Die globale Produktsammlung definiert die Basisproduktvorlagen.

Sowohl das Feld „Runden“ in „Markt/{*game-ID*}“ als auch die Produkt-Arrays in „Spiele/{*game-ID*}“ werden aus den in diesen *Produkt-IDs* definierten Werten und Parametern aufgebaut. Es handelt sich um eine Eins-zu-Viele-Beziehung: Ein einzelnes Produkt im Katalog kann in verschiedenen Märkten und in mehreren Spieler-Inventaren erscheinen.

3.2.5. Wie werden der Status eines Spiels und eines Spielers rekonstruiert?

Das Backend folgt in der Regel diesem Muster:

- Lesen Sie games/ *game-ID*}, um den Gesamtstatus des Spiels zu erhalten.
- Lesen Sie users/{*user-ID*} und überprüfen Sie *games*[*game-ID*], um zu sehen, wie sich dieser Spieler schlägt.
- Lesen Sie market/{*game-ID*}, wenn Sie einen vollständigen Überblick über die Preise und Verluste pro Runde benötigen.
- Lesen Sie blockchain/{*game-ID*}/*blocks* (und filtern Sie gegebenenfalls nach Runde), um die Mining-Historie zu analysieren.

Das Modell wendet eine leichte Denormalisierung an, um Aufrufe zu reduzieren und Abfragen vom Client zu vereinfachen, ohne die Rückverfolgbarkeit zu verlieren, die Projektpartner für die spätere Analyse der Daten benötigen.

4. WICHTIGE DATENFLÜSSE IN DER PRODUKTION

Über das statische Datenmodell hinaus funktioniert RockChain dank einer Reihe von wiederkehrenden Datenströmen, die während eines Spiels Dokumente in Firestore erstellen und aktualisieren. Diese Ströme verbinden den mobilen Client, Firebase-Funktionen, den WebSocket-Server und die zuvor beschriebenen Sammlungen. Zusammen stellen sie sicher, dass alle Spieler denselben Spielstatus in Echtzeit und mit Rückverfolgbarkeit sehen.

4.1. Erstellung und Beitritt zu Spielen

Die Erstellung von Spielen und die Aufnahme von Spielern hängen von Firebase-Funktionen ab, die die ersten Schreibvorgänge in die Spiele und Sammlungen der Benutzer vornehmen. Dadurch wird sichergestellt, dass jedes Spiel mit einem konsistenten Status beginnt und dass die Spieler sowohl im globalen Spieldokument als auch in ihrem Benutzerprofil korrekt registriert sind.

4.1.1. Erstellung von Spielen

Wenn ein Benutzer beschließt, ein neues Spiel zu hosten:

- Nach der Anmeldung ruft der Client eine Firebase-Funktion (*createGame*) auf.
- Die Funktion erstellt ein neues Dokument in games/{*game-ID*}.
- In users/{*user-ID*} erstellt oder aktualisiert die Funktion den Eintrag *games*[*game-ID*] mit Startwerten.

Dieser Prozess stellt sicher, dass sowohl der Gesamtstatus des Spiels als auch der individuelle Status des Host-Spielers von Anfang an klar definiert sind.

4.1.2. An einem bestehenden Spiel teilnehmen

Wenn ein Spieler einem bestehenden Spiel beitritt:

- Der Spieler gibt den *gameCode* im Client an.
- Der Client ruft eine Firebase-Funktion (*joinGame*) auf, die den *gameCode* in eine *game-ID* auflöst.
- Die Funktion überprüft, ob das Spiel beigetreten werden kann, und führt dann Folgendes aus:
    - Fügt den Spieler zu games/{*game-ID*}/*players* hinzu und erhöht *playerCount*.
    - Erstellt oder aktualisiert users/{*user-ID*}/*games*[*game-ID*] mit den Startdaten im Spiel (Währungen, Abfall, Status, Zeitstempel).

```
exports.joinGame = onCall({
  maxInstances: 10,
  memory: '256MiB',
}, async (request) => {
  console.log('🎮 joinGame iniciado');
  console.log('📥 Datos recibidos:', JSON.stringify(request.data, null, 2));

  // Extraer datos de la petición
  const { gameId } = request.data || {};
  const userId = request.auth?.uid;

  console.log(`🎯 Intentando unir usuario ${userId} al juego ${gameId}`);

  // Validaciones básicas
  if (!userId) {
  }

  if (!gameId) {
  }

  try {
  } catch (error) {
    console.error('❌ Error en joinGame:', error);

    // Mensajes de error específicos
    if (error.message.includes('Game not found')) {
      throw new Error('Game not found. Please check the code and try again.');
    } else if (error.message.includes('Game is full') || error.message.includes('No caben más jugadores')) {
      throw new Error('No caben más jugadores en esta sala. La sala está llena.');
    } else if (error.message.includes('not accepting new players')) {
      throw new Error('Game has already started. Please try another game.');
    } else if (error.message.includes('User not found')) {
      throw new Error('User not found. Please try logging in again.');
    } else {
      console.error('❌ Error no manejado:', error);
      throw new Error('Failed to join the game. Please try again.');
    }
  }
});
```

Abbildung10 : joinGame-Funktion

Dadurch werden direkte Schreibvorgänge des Clients in die Spielesammlung vermieden und sichergestellt, dass alle Spieler nach denselben Regeln registriert werden.

4.1.3. Verbindung zur Echtzeit-Ebene

Sobald die Firestore-Dokumente vorhanden sind:

- Jeder Client öffnet eine WebSocket-Verbindung zum Cloud Run-Endpunkt.
- Der Client sendet *die Spiel-ID*, *die Benutzer-ID* und grundlegende Profildaten.
- Der WebSocket-Server registriert den Socket im entsprechenden gameRooms-Eintrag und beginnt, diesen Benutzer in seinem In-Memory-Status zu verfolgen.

Zu diesem Zeitpunkt sind sowohl der persistente Status (Firestore) als auch der Echtzeitstatus (WebSocket-In-Memory-Maps) aufeinander abgestimmt, und das Spiel kann vom Wartezustand in die erste Runde übergehen.

4.2. Marktaktualisierungen und Runden

Für jede Runde wird der Markt hauptsächlich auf dem WebSocket-Server generiert und verwaltet, der die Produktliste im Speicher erstellt und sie über das Ereignis *economy:state* an die Clients sendet. Optional wird eine aggregierte Ansicht dieser Konfiguration in market/{game-*ID*} gespeichert, damit die Preise und Werte der

Rückstände später rekonstruiert werden können, ohne vom Status im Speicher abhängig zu sein.

4.2.1. Markterzeugung auf dem WebSocket-Server

Für die aktive *game-ID* und *Runde führt* der Server folgende Schritte aus:

- Verwendet die globalen Produktvorlagen (Typen, Marmortypen, Qualitäten, Preisklassen), um alle möglichen Kombinationen zu generieren.
- wendet die Preis- und Abfalllogik (Qualität A/B/C, Block vs. Platte, Marmorunterschiede und andere Spielregeln) an.
- Wählt zufällig eine Teilmenge von Produkten für diese Runde aus und speichert sie im Speicher als aktuelle Marktliste für dieses Spiel.

```
const assignProductsToGame = async ({ gameId, round }) => {
  const productsRef = admin.firestore().collection('products');
  const gameRef = admin.firestore().collection('games').doc(gameId);
  const marketRef = admin.firestore().collection('market').doc(gameId);

  const marbleTypes = ['Red', 'White', 'Gray', 'Black', 'Cream'];
  const productTypes = ['Block', 'Slab'];
  const qualities = ['A', 'B', 'C'];

  const qualityMultipliers = {…
  };

  const calculateProductPrice = (basePrice, productType) => {…
  };

  const calculateProductWaste = (baseWaste, productType) => {…
  };

  try {…
  } catch (error) {
    console.error('Error assigning products to game:', error);
    throw new Error('Failed to assign products to game');
  }
};

module.exports = { assignProductsToGame };
```

Abbildung11 : Funktion „assignProductsToGame"

Dadurch entsteht ein rundenspezifischer Markt, der sowohl den statischen Katalog als auch die dynamischen Parameter des aktuellen Spiels widerspiegelt.

4.2.2. Emission in Echtzeit und Rundenverlauf

Für jede *Runde*:

- Der WebSocket-Server sendet ein *„economy*:state"-Ereignis an alle Spieler im Spiel mit einer Nutzlast, die die Liste der in dieser Runde verfügbaren Produkte und alle relevanten Metadaten enthält.
- Auf dem Client speichert der *GameContext* (*GameContextHybridRobust*) diese Informationen im lokalen Status und macht sie für die verschiedenen Bildschirme (*Markt*, *Statistiken*, *Kopfzeile*) sichtbar.

- Die Spieler interagieren mit diesem Markt (Kauf von Produkten, Änderung von Beständen) über WebSocket-Ereignisse, während der Server die Arbeitskopie des Marktes und der Bestände im Speicher behält.

Diese klare Aufteilung bedeutet, dass Firestore eine dauerhafte, aggregierte Ansicht des Marktes pro Runde beibehält, während die WebSocket-Schicht die konkrete Produktliste liefert und schnelle Interaktionen während jeder Runde verarbeitet.

4.3. Mining, Validierung und Belohnungen

Das Mining in RockChain kombiniert Echtzeit-Interaktion mit persistenter Speicherung. Der WebSocket-Server generiert und verteilt Mining-Probleme, während die Unterkollektion blockchain/{*game-ID*}*/blocks* eine permanente Aufzeichnung jedes Problems, der erhaltenen Antworten und des Gewinners speichert.

Am Ende jeder Runde werden alle diese Informationen zu einer Reihe von Belohnungen pro Spieler zusammengefasst, die dann zurück in Firestore geschrieben werden.

4.3.1. Erstellung von Blöcken und Problemen

Wenn ein Mining-Ereignis für ein bestimmtes *Spiel* und *eine bestimmte Runde* ausgelöst wird:

- Der WebSocket-Server erstellt ein neues Mining-Problem und ordnet es einer *block-ID* zu.
- Er erstellt oder aktualisiert ein Dokument in blockchain/{*game-ID*}/blocks/{*block-ID*}.

Dieser erste Schreibvorgang stellt sicher, dass jedes Mining-Ereignis vom Moment seiner Erstellung an einen dauerhaften Anker in Firestore hat.

4.3.2. Echtzeit-Verteilung und Einreichung von Antworten

Sobald der Block erstellt ist:

- Der Server sendet ein *Mining*:*Problem*-Ereignis an alle Spieler im Spiel, das das Problem und den relevanten Kontext enthält.
- Die Clients zeigen das Problem an und ermöglichen es den Benutzern, innerhalb eines begrenzten Zeitfensters Antworten einzureichen.
- Jeder Spieler sendet seine Antwort über ein *„mining*:submit"-WebSocket-Ereignis, das der Server im Speicher aufzeichnet und möglicherweise an das Antwort-Array im entsprechenden Blockdokument anhängt.

Diese Interaktion modelliert einen **„Mining-Wettlauf",** bei dem die Spieler darum konkurrieren, so schnell wie möglich die richtige Lösung zu finden.

4.3.3. Validierung und Persistenz der Mining-Ergebnisse

Wenn Antworten eintreffen:

- Der Server bewertet jeden Versuch anhand der richtigen Lösung.
- Wenn gemäß den Spielregeln ein gültiger Gewinner ermittelt wird (die **erste richtige Antwort**), führt der Server folgende Schritte aus:
  - Ermittelt den Gewinner.
  - Markiert den Block als geschlossen (*isActive* = false).
  - Aktualisiert blockchain/{*game-ID*}/blocks/{*block-ID*} mit einem Gewinnerobjekt, das Folgendes enthält: *user-ID*, *userName*, deren Antwort und ob sie richtig war, *elapsedTime* und *responseTime* für die Prüfung.
- Der Server sendet dann ein *mining*:*result*-Ereignis an alle Clients, einschließlich Informationen über den Gewinner und sofortiges Feedback.

```
// Función para manejar respuestas de minado con tiempo del cliente
const handleMiningSubmit = (blockId, userId, response, userName, clientResponseTime) => {
  console.log(`[MINING][SUBMIT] blockId=${blockId} user=${userId} response=${response} clientResponseTime=${clientResponseTime}`);

  // Definir timestamp actual al inicio de la función
  const now = Date.now();

  // Usar el tiempo del cliente si está disponible, sino calcular del servidor
  let elapsed;
  if (clientResponseTime !== undefined && clientResponseTime > 0) {···
  } else {···
  }

  // Obtener el problema para verificar la respuesta correcta
  const problemData = miningProblems.get(blockId);
  if (!problemData) {···
  }

  const correctAnswer = parseFloat(problemData.activeProblem.solution);
  const userAnswer = parseFloat(response);
  const isCorrect = userAnswer === correctAnswer;

  console.log(`[MINING][SUBMIT] blockId=${blockId} user=${userId} isCorrect=${isCorrect} elapsed=${elapsed}ms`);

  // Inicializar o actualizar respuestas del bloque
  if (!miningResponses.has(blockId)) {···
  }

  const blockData = miningResponses.get(blockId);

  // Verificar si el usuario ya respondió
  const existingResponseIndex = blockData.responses.findIndex(r => r.userId === userId);
  if (existingResponseIndex !== -1) {···
  } else {
    // Añadir nueva respuesta
  const newResponse = {···
  };
  blockData.responses.push(newResponse);
  }

  // Determinar ganador y estado de completado
  const correctResponses = blockData.responses.filter(r => r.isCorrect);
  let winner = null;
  let isCompleted = false;

  if (correctResponses.length > 0) {···
  }
```

Abbildung „12 “: Funktion „handleMiningSubmit“

Dieser Ablauf garantiert, dass jedes Mining-Ereignis über einen nachvollziehbaren, überprüfbaren Datensatz in Firestore verfügt und gleichzeitig vom WebSocket-Server in Echtzeit verarbeitet wird.

4.3.4. Konsolidierung der Belohnungen am Ende der Runde

Am Ende jeder Runde sammelt RockChain alle Ereignisse der Markt- und Mining-Phase, um eine einzigartige Reihe von Belohnungen pro Spieler zu berechnen. Diese Belohnungen werden dann in Firestore geschrieben, wodurch sowohl das globale Spieldokument als auch der individuelle Status jedes Spielers aktualisiert werden.

Auslösen des Rundenendes

Wenn die offizielle Rundenzeit abgelaufen ist (gemäß dem Feld *„roundTime“* und den Zeitstempeln in *„gameAuthorities“*) oder wenn alle erforderlichen Bedingungen erfüllt sind, ändert der WebSocket-Server den Spielstatus in die Rundenabschlussphase.

Zu diesem Zeitpunkt sammelt er die Daten, die er im Speicher hat, wie zum Beispiel:

- Geminierte Blöcke und deren Gewinner.
- Spieler-Inventare.
- Ausgewählte Branchen.
- Abfallmengen und andere Indikatoren.

Berechnung der Belohnungen pro Spieler

Das Backend berechnet für jede *Benutzer-ID*:

- **Mining-Belohnungen**: zum Beispiel eine feste Anzahl von RockCoins für jeden korrekt gewonnenen Block.
- **Abfallreduzierung**: basierend auf der Effizienz der gewählten Branche und den Spielregeln (z. B. wie viel Abfall durch die Anwendung von Kreislaufstrategien vermieden wird).

Schreiben der Ergebnisse in Firestore

In games/{*game-ID*} werden folgende Angaben aktualisiert:

- *winningIndustry für die Runde.*
- *roundRewards*[*user-ID*] = { *industryReward*, *wasteRemoved*, *miningReward*, ... } für alle Spieler.
- *rewardsAssigned* = true.
- *Status*, der je nach Folgenden auf roundEnd oder waitingForNextRound gesetzt wird.

In users/{*user-ID*}, innerhalb von *games*[*game-ID*], wird Folgendes aktualisiert:

- *rockCoins*, wobei die in der Runde verdienten Belohnungen hinzugefügt werden.
- *waste*, wobei das Entfernte abgezogen wird (natürlich ohne unter Null zu fallen).

- Dadurch wird sichergestellt, dass der persistente Status des Spielers das Ergebnis der Runde genau widerspiegelt.

Benachrichtigung der Clients

- Sobald die Daten geschrieben sind, sendet der WebSocket-Server ein *round_end*-Ereignis mit einer Zusammenfassung der Belohnungen und der Gewinnerbranche.
- Die Clients aktualisieren ihren lokalen Status (Belohnungen, Inventar, Gewinnerbranche) und zeigen die Ergebnisse an, bevor sie zur nächsten Runde übergehen.

Dank dieses kombinierten Ablaufs fungiert Firestore als definitive Aufzeichnung dessen, was jeder Spieler gewonnen hat und wie sich seine Ressourcen verändert haben, während WebSocket dafür sorgt, dass sich alles sofort und synchronisiert auf allen Geräten anfühlt.

5. SICHERHEIT, DATENSCHUTZ UND PRODUKTIONSREIFE

Das RockChain-Backend wurde so konzipiert, dass nur vertrauenswürdige Komponenten offizielle Systemdaten ändern können und dass Informationen über Benutzer und Treiber begrenzt, geschützt und im Laufe der Zeit leicht zu verwalten bleiben. In diesem Abschnitt wird zusammengefasst, wie Sicherheit, Datenschutz und grundlegende Vorgänge auf der Datenebene in der Produktion gehandhabt werden.

5.1. Firestore-Sicherheitsregeln und Authentifizierung

Die RockChain-Produktionsumgebung kombiniert Firebase-Authentifizierung, Firestore-Sicherheitsregeln und Firebase-Funktionen, um sicherzustellen, dass nur autorisierte Benutzer und Dienste auf Daten zugreifen oder diese ändern können.

5.1.1. Authentifizierung als Zugangstor

- Jeder Zugriff auf das Backend erfordert eine vorherige Authentifizierung mit Firebase Authentication (entweder per E-Mail/Passwort oder über einen anderen kompatiblen Anbieter).
- Jedem Benutzer wird eine stabile *Benutzer-ID* zugewiesen, die in Firestore und auf dem WebSocket-Server verwendet wird.

5.1.2. Eingeschränkter Zugriff auf Profil- und Spieldaten

- Die Firestore-Sicherheitsregeln stellen sicher, dass jeder authentifizierte Benutzer nur Folgendes tun kann:
    - sein eigenes users/{*user-ID*}-Dokument lesen und aktualisieren.
    - Daten aus Spielen lesen, in denen er registriert ist.
    - Sie können nicht direkt auf wichtige Dokumente wie games/{*game-ID*} oder blockchain/{*game-ID*}/blocks/{*block-ID*} vom Client aus schreiben.
- Der Zugriff auf die Daten anderer Spieler ist auf das für das Funktionieren des Spiels erforderliche Minimum beschränkt (z. B. öffentliche Namen oder Ranglisten).

5.1.3. Durch Firebase-Funktionen gesteuerte Schreibvorgänge

- Die meisten Vorgänge, die sich auf das Gameplay auswirken (Erstellen oder Beitreten von Spielen, Aktualisieren des Spielstatus, Speichern von Belohnungen usw.), können nur über Firebase-Funktionen mit erhöhten Berechtigungen ausgeführt werden.
- Die Firestore-Regeln sind einfach gehalten und überprüfen im Wesentlichen Folgendes:
    - Authentifizierte Clients können bestimmte Dokumente lesen, wenn sie über die entsprechende Berechtigung verfügen.

- Nur Backend-Funktionen oder Dienstkonten können in geschützte Pfade schreiben (*Spiele*, *Blockchain*, bestimmte Felder in *Benutzern* usw.).

5.1.4. WebSocket-Server als vertrauenswürdiger Dienst

- Der WebSocket-Server läuft auf Cloud Run und verwendet ein Dienstkonto mit eingeschränkten Berechtigungen, um:
  - Dokumente zu lesen und zu schreiben, die für die Verwaltung von Runden, Mining und Belohnungen erforderlich sind.
  - Aufruf von Backend-Funktionen, wenn eine zusätzliche Validierung erforderlich ist.
- Die gesamte Kommunikation zwischen dem Client und dem WebSocket-Server erfolgt über sichere Verbindungen (WSS), und jede empfangene Nachricht wird anhand des aktuellen Status im Speicher und in Firestore validiert, bevor dauerhafte Änderungen vorgenommen werden.

Mit diesen Maßnahmen stellt RockChain sicher, dass kein nicht authentifizierter Benutzer auf die Daten zugreifen kann und dass selbst authentifizierte Clients nur Absichten lesen und senden können. Tatsächliche Änderungen an der Datenbank werden immer vom Backend kontrolliert, wodurch die Integrität des Spiels gewährleistet ist.

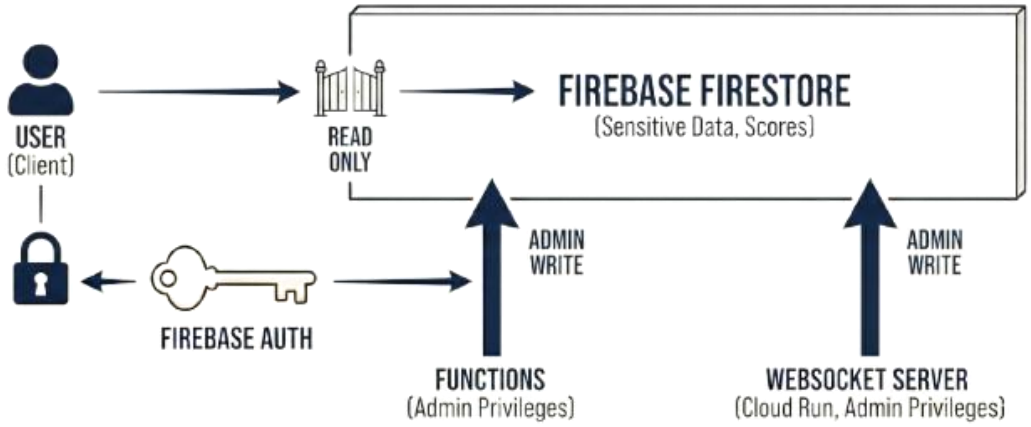

Abbildung13 : So funktioniert die Sicherheit in Rockchain.

5.2. DSGVO und Datenschutz

Da RockChain in Pilotaktivitäten mit realen Personen eingesetzt wird, wurde seine Datenschicht in Übereinstimmung mit den Grundsätzen der **Datenschutz-Grundverordnung (DSGVO) der Europäischen Union** und den geltenden nationalen Gesetzen konzipiert.

5.2.1. Datenminimierung und Zweckbindung

- Das System speichert nur Informationen, die für folgende Zwecke unbedingt erforderlich sind:
    - Durchführung von Spielsitzungen (Benutzer-IDs, Teilnahme, Ressourcen im Spiel).
    - Bewertung und Verbesserung von Schulungsaktivitäten (aggregierte Statistiken, anonymisierte Protokolle).
- Es werden keine besonderen Kategorien personenbezogener Daten (wie Gesundheit, Religion oder politische Meinungen) erfasst oder gespeichert.

5.2.2. Pseudonymisierung von Spielern

- Auf technischer Ebene werden Benutzer in erster Linie anhand ihrer *Firebase-Benutzer-ID* identifiziert. Optional können sie während der Lernerfahrung einen sichtbaren Namen oder Avatar verwenden.
- Die Projektpartner in jedem Pilotprojekt sind dafür verantwortlich, die Zuordnung zwischen einer *Benutzer-ID* und der tatsächlichen Identität des Teilnehmers separat und lokal zu verwalten. Das System arbeitet somit mit pseudonymen Identifikatoren.

5.2.3. Rechtsgrundlage und Informationen für die Teilnehmer

- RockChain wird in strukturierten Bildungskontexten (Erwachsenenbildung, Berufsausbildung, berufliche Weiterbildung) eingesetzt.
- Jeder Pilotstandort stellt den Teilnehmern Folgendes zur Verfügung:
    - Eine Datenschutzerklärung, in der erläutert wird, welche Daten in RockChain zu welchen Zwecken und für wie lange erfasst werden.
    - Klare Kontaktdaten für die Ausübung ihrer Rechte als betroffene Personen (Zugriff, Berichtigung, Löschung usw.).
- Je nach den rechtlichen Rahmenbedingungen des Landes kann die Rechtsgrundlage die Einwilligung oder das berechtigte Interesse an der Durchführung und Bewertung von Schulungen sein. Dies wird auf der Ebene jedes Pilotprojekts vom jeweiligen Partner dokumentiert.

5.2.4. Speicherort und Sicherheitsmaßnahmen

- Die gesamte Kommunikation mit dem Backend (Firestore, Funktionen, WebSocket) wird während der Übertragung mit HTTPS oder WSS verschlüsselt.
- Die Daten werden auch im Ruhezustand mit Standardmechanismen von Firebase und Google Cloud verschlüsselt.
- Der Zugriff auf die Firebase-Konsole und die Produktionsumgebung ist auf eine kleine Gruppe von Konsortiumsadministratoren beschränkt, die eine robuste Authentifizierung und rollenbasierte Berechtigungen verwenden.

5.2.5. Aufbewahrung, Anonymisierung und Löschung

- Spieldaten werden nur so lange aufbewahrt, wie es für folgende Zwecke erforderlich ist:
  - Durchführung der Pilotprojekte.
  - die vereinbarten Projektergebnisse (wie Berichte und allgemeine Nutzungs- und Lernstatistiken) zu erstellen.
- Nach Ablauf dieses Zeitraums können die Partner:
  - die Daten weiter anonymisieren und alle direkten Verknüpfungen zwischen der *Benutzer-ID* und lokalen Identitätslisten entfernen.
  - Spiele, Benutzer oder ganze Datensätze aus der Produktionsinstanz von Firestore löschen.
- Auf Wunsch eines Teilnehmers können dessen Daten gelöscht oder pseudonymisiert werden, indem das Dokument „users/{*user-ID*}" und die zugehörigen Daten geändert oder gelöscht werden, wie in der Datenschutzerklärung des Pilotprojekts erläutert.

Zusammenfassend lässt sich sagen, dass die RockChain-Datenschicht so konzipiert wurde, dass sie transparent, begrenzt und reversibel funktioniert und die Bildungsziele des Projekts unterstützt, ohne die Kontrolle über personenbezogene Daten zu verlieren, die stets in den Händen der lokalen Pilotpartner verbleiben.

5.3. Backups, Bereinigung und grundlegende Wartung

Um für die Produktion bereit zu sein, muss das Backend von RockChain nicht nur sicher, sondern auch einfach zu bedienen und zu warten sein. Die aktuelle Konfiguration umfasst einfache Verfahren für Backups, die Bereinigung veralteter Daten und die tägliche Wartung.

5.3.1. Backups und Wiederherstellungsoptionen

- Firestore bietet bereits integrierte Redundanz und Replikation auf Speicherebene.
- Darüber hinaus können Administratoren regelmäßig wichtige Sammlungen (wie *Spiele*, *Benutzer*, *Blockchain*, *Produkte*, *Markt*) entweder über geplante Skripte oder Cloud-Funktionen in Cloud Storage exportieren.
- Diese Backups dienen beiden Zwecken:
  - Wiederherstellung von Daten im Falle einer versehentlichen Löschung.
  - Erhaltung eingefrorener Snapshots bestimmter Pilotphasen zu Dokumentations- oder Analysezwecken, auch wenn diese später aus der aktiven Datenbank gelöscht werden.

5.3.2. Bereinigung alter Spiele und Pilotdaten

- Um zu verhindern, dass Daten unkontrolliert anwachsen, und um die Minimierung zu unterstützen:
    - Alte Spiele können als archiviert markiert und gelöscht werden, einschließlich ihrer Blockchain/{*game-ID*}-Untersammlungen und market/{*gameID*}-Dokumente.
    - Verwandte Einträge in users/{*user-ID*}.*games*[*game-ID*] können ebenfalls gelöscht oder anonymisiert werden.
- Diese Bereinigung kann erfolgen über:
    - Geplante Aufgaben (Cloud-Funktionen oder externe Skripte).
    - Manuelle Aktionen, die vom technischen Partner koordiniert werden.

5.3.3. Überwachung und grundlegende Betriebsprüfungen

- Die Dashboards von Firebase und Cloud Run bieten:
    - Nutzungsmetriken (Lesevorgänge, Schreibvorgänge, Bandbreite).
    - Fehlerprotokolle für Funktionen und den WebSocket-Server.
    - Leistungsindikatoren, mit denen Abfragen identifiziert werden können, für die möglicherweise neue Indizes erforderlich sind.
- Während der Pilotphase werden regelmäßige Überprüfungen durchgeführt, um:
    - sicherzustellen, dass Limits oder Kontingente nicht überschritten werden.
    - Fehlerhafte Konfigurationen (z. B. schlecht definierte Sicherheitsregeln oder fehlende Indizes) zu erkennen.
    - Anpassung der Ressourcen oder Indizierungsstrategien, wenn die Anzahl der gleichzeitigen Benutzer steigt.

Dank dieser Maßnahmen bleibt das Backend von RockChain schlank und überschaubar, sodass Partner das Tool sowohl während als auch nach dem Projekt ohne ein spezielles DevOps-Team betreiben können, während gleichzeitig die Datenintegrität und -verfügbarkeit gewährleistet ist.

6. NÄCHSTE SCHRITTE UND SCHLUSSFOLGERUNGEN

Die in WP4.A1 durchgeführten Arbeiten haben zu einer konkreten und funktionalen Datenschicht für das RockChain-Lerntool geführt. Diese basiert auf einer hybriden Architektur, die Firestore, Firebase-Funktionen und einen dedizierten WebSocket-Server kombiniert, der auf Google Cloud Run läuft und von einer in React Native erstellten mobilen App genutzt wird. Dieser Technologie-Stack bietet bereits eine solide Grundlage für die Verwaltung von Spielen, Benutzern, Märkten und Mining-Ereignissen in Echtzeit, mit klaren Verantwortlichkeiten für jede Komponente und einem kompakten Firestore-Modell, das die Schlüsselelemente des Spiels und des Belohnungssystems widerspiegelt.

Von hier aus geht es in den nächsten Schritten weniger um größere strukturelle Veränderungen, sondern vielmehr um die Konsolidierung und Stärkung des bereits Aufgebauten. Einerseits werden die Firestore-Sicherheitsregeln für die Hauptsammlungen (Spiele, Benutzer, Blockchain) unter realistischen Szenarien fein abgestimmt und getestet, um sicherzustellen, dass der Zugriff gut kontrolliert wird. Außerdem überprüfen wir, ob die Trennung zwischen Entwicklungs- und Produktionsumgebung in allen Teilen des Systems ordnungsgemäß umgesetzt ist: von mobilen Builds über Funktionen bis hin zum WebSocket-Server. Im Rahmen dieser Vorbereitungen werden kleine Pilotversuche mit internen Benutzern und Projektpartnern durchgeführt, um zu überprüfen, ob die Datenflüsse unter realen Spielbedingungen korrekt funktionieren.

Darüber hinaus werden für die sensibelsten Vorgänge, wie Rundenwechsel, Belohnungsberechnungen und Mining-Validierung, leichtgewichtige Überwachungs- und Protokollierungsmechanismen integriert, die dazu beitragen werden, potenzielle Fehler während der Pilotversuche zu beheben. Die Indizes der aktivsten Sammlungen werden ebenfalls auf der Grundlage tatsächlicher Abfragemuster überprüft und angepasst, um ein gutes Gleichgewicht zwischen Leistung und Schreibkosten zu erreichen. Außerdem werden einfache Wartungsroutinen definiert, um alte Sitzungen zu bereinigen und wichtige Sammlungen wie Spiele, Benutzer und Blockchain sowohl für Sicherungszwecke als auch für Analysen zu exportieren.

Mit Blick auf die Zukunft planen wir, gut dokumentierte Einstiegspunkte entweder über Funktionen oder die WebSocket-API offenzulegen, damit andere Aktivitäten innerhalb desselben WP4 eine Verbindung zu dieser Datenschicht herstellen können, ohne die Logik duplizieren zu müssen. Wir möchten auch die bereits gespeicherten Informationen (wie Mining-Verlauf, Marktkonfigurationen und Belohnungen) nutzen, um pädagogische Entscheidungen bei der Szenario-Gestaltung oder Lernanalyse in den folgenden Arbeitspaketen zu treffen.

Zusammenfassend lässt sich sagen, dass WP4.A1 RockChain von einem abstrakten Entwurf darüber, „was gespeichert werden soll", zu einer konkreten Dateninfrastruktur

gemacht hat, die produktionsreif ist und Multiplayer-Spiele auf mobilen Geräten unterstützen kann. Die aktuelle Architektur ist bewusst einfach und leicht verständlich, aber robust genug, um echte Sitzungen auszuführen, grundlegende Sicherheitsregeln durchzusetzen und alles, was in jedem Spiel geschieht, nachvollziehbar aufzuzeichnen. Indem sie sich bei den nächsten Schritten auf Stabilisierung, Überwachung und kleine inkrementelle Erweiterungen statt auf größere Neugestaltungen konzentrieren, können die Projektpartner diese Datenschicht als zuverlässiges Rückgrat des Tools betrachten: etwas, das sowohl während der Pilotphase als auch bei einer möglichen zukünftigen Weiterentwicklung über die Laufzeit des Projekts hinaus nach Bedarf eingesetzt, gewartet und angepasst werden kann.