Projektdokumentation (PDF)

IPERKA

Informieren

Ausganslage:

Für ein Online-Game soll ein Backend mit Java und einer MySQL-Datenbank erstellt werden. Das System verwaltet User, Rollen (Admin/Player), Speicherschnappschüsse (Stats-History), ein Inventar mit limitierten Items sowie Achievements.

Ziele:

- Userverwaltung inkl. Rollen (Admin, Player)
- Stats als History: nicht änderbar/löschbar, nur anschauen
- Der aktuelle Spielstand wird im User gespeichert. SaveStats() erstellt daraus einen Snapshot in der History (stats). Der Client muss keine Werte mitsenden.
- Limitierte Items: globale Stückzahl (Stock)
- Achievements: pro User nur einmal freischaltbar

Rahmenbedingungen:

- Backend: Java Spring Boot
- Datenbank: MySQL
- Datenkonsistenz wichtig (kein „doppelt vergebenes letztes Item“)
- Saubere Trennung: Datenmodell (ERD) + Business-Logik (Services/Transaktionen)

Planen

Datenmodell (MySQL Tabellen – übersicht):

1) users

- user_id (PK, AUTO_INCREMENT)
- username (UNIQUE)
- email (UNIQUE)
- password_hash
- role (ADMIN/PLAYER)
- level INT DEFAULT 1
- xp INT DEFAULT 0
- coins INT DEFAULT 0
- created_at (DEFAULT CURRENT_TIMESTAMP)

2) stats (History)
- stats_id (PK, AUTO_INCREMENT)
- user_id (FK -> users.user_id)
- level
- xp
- coins
- saved_at (DEFAULT CURRENT_TIMESTAMP)

3) item (Item-Typ + globaler Bestand)
- item_id (PK, AUTO_INCREMENT)
- code (UNIQUE)
- name, desc, rarity
- stock (global vorhanden)

4) user_item (Aktueller Besitz)
- PRIMARY KEY (user_id, item_id)
- user_id (FK)
- item_id (FK)
- quantity (INT NOT NULL DEFAULT 0)

5) achievement
- achievement_id (PK, AUTO_INCREMENT)
- code (UNIQUE)
- name, description

6) user_achievement
- user_id (FK)
- achievement_id (FK)
- unlocked_at (DEFAULT CURRENT_TIMESTAMP)
- PRIMARY KEY (user_id, achievement_id)

Wichtige Business-Regeln
- Stats:
  - SaveStats() macht IMMER INSERT in stats
  - Kein UPDATE/DELETE für stats (nur SELECT)
- Items (limitiert):
  - Globaler Bestand: item.stock
  - Take/Drop immer in Transaktionen (damit bei gleichzeitigen Requests nur einer das letzte Item bekommt). Mit ACID (Atomicity, Consistency, Isolation, Durability)

---

0) Projektidee und Scope (Vorbereitung 1–3)

- Projektidee als *Elevator Pitch* (max. 1/2 A4) schreiben
- Prozess definieren (was kann der Client machen?)
- Grob festlegen: Entities + Kernfunktionen

1) Anforderungen und User Stories (Vorbereitung 4–5)

- 5–10 User Stories (Admin/Player) inkl. Akzeptanzkriterien
- Aufwand schätzen pro Story + Zeitplan als Tabelle
- Anforderungskatalog (Kernaufgaben) erstellen

2) Datenmodell (ERD) + Tabellenschema (Vorbereitung 6)

- ERD finalisieren (users, stats-history, item, user_item, achievement, user_achievement)
- Tabellen-Details festlegen: Datentypen, PK/FK, UNIQUE, DEFAULT CURRENT_TIMESTAMP, Indizes

3) REST-Schnittstellen planen (Vorbereitung 7)

- CRUD-Endpunkte planen (GET/POST/PUT/DELETE) passend zu den User Stories
- Request/Response-Modelle (DTOs) definieren
- Fehlerfälle definieren (404, 400 Validierung, 409 Konflikt etc.)

4) Testplan erstellen (Vorbereitung 8)

- Manueller Testplan für alle wichtigsten Endpunkte
- Positive + Negative Tests (Validierung, Not Found, Konflikte)
- Vorgehen festhalten (Insomnia)

5) Projekt-Setup (Entwicklung Start)

- GitHub Repo + saubere Commits
- Spring Boot Projekt mit Maven (pom.xml korrekt)
- MySQL: DB + User anlegen, Verbindung konfigurieren
- Grundstruktur Packages: controller / service / repository / model / dto / exception

6) Implementierung: CRUD + Validierung + Exceptions

- Entities + Repositories (JPA)
- CRUD Controller/Services für zentrale Ressourcen
- Bean Validation (z.B. @NotNull, @Size, @Email)
- Exception Handling (saubere Fehlermeldungen an Client)

7) Business-Logik

- SaveStats() => nur INSERT (History)
- Item Take/Drop mit Transaktionen (stock korrekt)
- Achievement unlock (einmalig pro User)
- Verschiedene Achievments
- Admin-Funktionen (z.B. Items anlegen/stock setzen, falls geplant)

8) Tests durchführen + Testprotokoll

- Manuelle Tests aus Testplan durchspielen
- Export aus Insomnia sichern
- Testprotokoll schreiben (was getestet, Resultat)

9) Unit-Tests (Automation)

- Unit-Tests für wichtigste Komponenten (mind. Positiv/Negativ pro Kernfunktion)
- Tests müssen laufen (mvn test)

10) Installationsanleitung + Final Doku

- Schritt-für-Schritt Setup (DB, User, Config, Start)
- Alle Hilfestellungen/Quellen dokumentieren
- Doku final als PDF (Titelblatt + Inhaltsverzeichnis + IPERKA-Struktur)

Deliverables: *Installationsanleitung*, *Hilfestellungen*, *Projektdokumentation.pdf (Alles auf Github)*

Entscheiden

Gewählte Lösung:

Ich setze das Backend als **Spring Boot REST-API** mit **MySQL** um. Die Persistenz erfolgt über **Spring Data JPA** (Repositories). Die API liefert und akzeptiert JSON.

Begründung:

- **Spring Boot + JPA** ist im Modul M295 behandelt und ermöglicht sauberes CRUD.
- **MySQL** erfüllt die Anforderung einer relationalen Datenbank.
- Durch die Trennung in **Controller → Service → Repository** bleibt der Code übersichtlich und wartbar.
- **Transaktionen** (@Transactional) stellen sicher, dass kritische Prozesse (z.B. Item nehmen/droppen, Stats speichern) konsistent bleiben (ACID).
- **Validierung** via Bean Validation verhindert fehlerhafte Daten in der Datenbank.
- **Exception Handling** liefert dem Client klare Fehlermeldungen (z.B. 400/404/409).

Architektur-Entscheidungen:

- **Controller**: nur HTTP + DTOs + Statuscodes
- **Service**: Business-Logik (SaveStats, Take/Drop Item, Unlock Achievement)
- **Repository**: DB-Zugriffe (JPA)
- **DTOs**: Request/Response getrennt von Entities
- **Fehlerbehandlung** erfolgt zentral über eine @RestControllerAdvice-Klasse, welche Validierungsfehler sowie fachliche Fehler (404/409) in ein einheitliches JSON-Format umwandelt.

Statuscodes (Fehlerfälle)

- **400**: Validierungsfehler (ungültige Eingaben)
- **404**: Ressource nicht gefunden (User/Item/Achievement existiert nicht)
- **409**: Konflikt (z.B. Item stock = 0, Achievement bereits freigeschaltet)

Realisieren

Projektstruktur

Maven-Projekt
Packages:

- controller
- service
- repository
- model (Entities)
- dto
- exception (eigene Exceptions + Handler)
- config (falls nötig)

Umsetzungsschritte (Technik)

Datenbank

- MySQL Benutzer + Datenbank erstellen
- Tabellen gemäss ERD anlegen (PK/FK/UNIQUE/DEFAULT CURRENT_TIMESTAMP)
- Indexe auf FK-Felder setzen (Performance)

Entities und Repositories

- Entities für: User, Stat, Item, UserItem, Achievement, UserAchievement
- Repositories für CRUD (JPA)

Services (Business-Logik)

- saveStats()
    - liest level, xp, coins aus users
    - macht INSERT in stats
- - keine Update/Delete Funktion für stats

- takeItem(userId, itemId, quantity)
    - prüft stock und aktualisiert item.stock
    - speichert Besitz in user_item (quantity erhöhen oder neuen Eintrag)
    - läuft transaktional
- dropItem(itemId, quantity)
    - reduziert user_item.quantity
    - erhöht item.stock
    - wenn quantity = 0, kann der Eintrag gelöscht werden
    - läuft transaktional
- unlockAchievement(achievementId)
    - Insert in user_achievement
    - doppelte Einträge werden durch PK verhindert → Konfliktbehandlung

REST-Endpunkte (CRUD)

- Users: GET/POST/PUT/DELETE
- Items: GET/POST/PUT/DELETE (Admin)
- Achievements: GET/POST/PUT/DELETE (Admin)

- Stats: POST /users/{userId}/stats, GET /users/{userId}/stats (nur lesen)
- Inventory: POST /users/{userId}/items/{itemId}/take, POST /users/{userId}/items/{itemId}/drop

Validierung

- DTO-Validierung mit z.B. @NotNull, @Size, @Email, @Min(0)
- Bei Fehlern: Response mit **400** + klare Meldung

JavaDoc

- Alle Java-Klassen und public Methoden dokumentieren (ausser Getter/Setter)

GitHub

- Regelmässige, sinnvolle Commits
- README + Installationsanleitung

Kontrollieren

Ziel der Kontrollphase ist es, sicherzustellen, dass die Applikation korrekt, stabil und gemäss den Anforderungen funktioniert.

Alle wichtigen REST-Endpunkte werden manuell mit Insomnia getestet.
Für jeden Endpoint existieren Positive und Negative Tests.
Die Resultate werden im Testprotokoll festgehalten (Request, Response, Statuscode, Ergebnis).

Beispiele für wichtige Tests

SaveStats:

- Positiv: Aufruf von SaveStats() → neuer Datensatz in stats wird erstellt
- Negativ: User nicht authentifiziert → 401 Unauthorized
- Negativ: User existiert nicht → 404 Not Found

TakeItem:

- Positiv: stock > 0 → stock wird reduziert, Besitz wird gespeichert
- Negativ: stock = 0 → 409 Conflict
- Negativ: item_id existiert nicht → 404 Not Found

UnlockAchievement:

- Positiv: erstes Freischalten → Insert erfolgreich
- Negativ: zweites Freischalten → 409 Conflict (durch Primary Key verhindert)

Zusätzlich werden folgende Punkte kontrolliert:

- Fehlerhafte Requests werden mit passenden HTTP-Statuscodes beantwortet (400/401/404/409)
- Exceptions werden zentral behandelt und als JSON-Fehler zurückgegeben
- Transaktionen verhindern Inkonsistenzen (z.B. zwei User nehmen gleichzeitig das letzte Item)
- Die Datenbank bleibt konsistent (FKs, UNIQUE, keine negativen Stocks)

Automatisierte Tests

- Unit-Tests für die wichtigsten Services und Controller
- Pro Kernfunktion mindestens:
    - 3 positive Tests
    - 3 negative Tests
- Tests sind mit mvn test ausführbar

Abnahmekriterien

- SaveStats() erstellt immer einen neuen History-Eintrag
- Items können nicht doppelt oder über den Stock hinaus vergeben werden
- Achievements können pro User nur einmal existieren
- Die API ist gemäss Installationsanleitung startbar
- Alle Kernfunktionen sind über REST-Endpunkte erreichbar und funktionieren korrekt

---

Auswerten

Zielerreichung

Die gesetzten Ziele wurden erreicht:

- Es wurde eine lauffähige REST-API mit **Spring Boot** und **MySQL** umgesetzt.
- Die Applikation bietet CRUD-Operationen für die zentralen Ressourcen (User, Items, Achievements).
- Der aktuelle Spielstand wird im User gespeichert.
- SaveStats() erstellt daraus einen Snapshot in der History (stats).
- Items sind global limitiert und werden transaktional verwaltet.
- Achievements können pro User nur einmal freigeschaltet werden.
- Die Projektdokumentation ist nach **IPERKA** strukturiert und vollständig.

Reflexion

Was lief gut:

- Die klare Trennung zwischen Controller, Service und Repository erleichterte die Umsetzung.
- Das ERD half, das Datenmodell und die REST-Endpunkte sauber zu planen.
- Die Trennung zwischen aktuellem Stand (users) und History (stats) ist übersichtlich.
- Transaktionen verhinderten Logikfehler bei konkurrierenden Requests (z.B. „letzte Granate“).

Herausforderungen:

- Die korrekte Behandlung von Fehlerfällen mit passenden HTTP-Statuscodes.
- Die Umsetzung der Transaktionen beim Item-System.
- Das saubere Validieren und Abfangen von Ausnahmen.

Was würde ich verbessern:

- Mehr Integrationstests für komplette Abläufe.
- Ein echtes Login-System mit Token für den aktiven User.

Zeitvergleich

- **Geplant:** gemäss Arbeitsplan (User Stories und Aufwandsschätzung).
- **Effektiv:** wird nach Projektabschluss eingetragen.

Hilfestellung

Chatgpt, Mitschüler, Google, Stack overflow und

Testdokumentation

1. Testkonzept

Das REST-API wird hauptsächlich durch **manuelle, funktionale Tests** überprüft.
Ziel der Tests ist es, sicherzustellen, dass die wichtigsten Endpunkte korrekt funktionieren und passende HTTP-Statuscodes zurückgeben.

Die Tests werden **nach der Implementierung** der jeweiligen Funktionalität durchgeführt.
Als Testwerkzeug wird **Insomnia** verwendet.
Zusätzlich existieren **Unit-Tests** für zentrale Controller- und Service-Klassen.

Der Fokus liegt auf:

- korrekter Funktionalität der Endpunkte
- Rollen- und Berechtigungsprüfung (Admin / Player)
- Validierung von Eingabedaten
- korrektem Fehlerverhalten (401 / 403 / 409)

2. Testplan (geplant – vor der Durchführung)

Nr.	Endpoint	Methode	Testfall	Voraussetzung	Erwartung
1	/users	POST	User erstellen	gültiger Request-Body	201 Created
2	/users	POST	User erstellen (ungültig)	ungültige Daten	400 Bad Request
3	/users/{id}/stats	POST	SaveStats als Player	User eingeloggt	201 Created
4	/users/{id}/stats	POST	SaveStats ohne Login	nicht authentifiziert	401 Unauthorized
5	/items	POST	Item erstellen (Admin)	Admin eingeloggt	201 Created
6	/users/{id}/items/{id}/take	POST	Item nehmen	Player eingeloggt, Stock > 0	201 Created
7	/users/{id}/items/{id}/take	POST	Item nehmen (Stock = 0)	Player eingeloggt	409 Conflict
8	/users/{id}/achievements	GET	Achievements anzeigen	Player eingeloggt	200 OK

3. Testfälle (durchgeführt mit Insomnia)

Die manuellen Tests wurden mit Insomnia anhand von User-Story-basierten Requests durchgeführt.
Dabei wurden **positive und negative Tests** (Auth, Rollen, Validierung, Konflikte) ausgeführt.

Beispiele:

- User erstellen (gültig / ungültig / Duplicate)
- SaveStats (Player erlaubt, Admin verboten, ohne Login verboten)
- Item nehmen (Stock > 0 / Stock = 0 / falsche Rolle)
- Item verwalten (Admin erlaubt, Player verboten)
- Achievements erstellen und anzeigen

4. Testresultate (Testprotokoll)

Nr.	Testfall	Erwartung	Ergebnis	Status
1	User erstellen	201 Created	201 Created	OK
2	User invalid	400 Bad Request	400 Bad Request	OK
3	SaveStats Player	201 Created	201 Created	OK
4	SaveStats ohne Login	401 Unauthorized	401 Unauthorized	OK
5	Item erstellen Admin	201 Created	201 Created	OK
6	TakeItem Stock > 0	201 Created	201 Created	OK
7	TakeItem Stock = 0	409 Conflict	409 Conflict	OK
8	Achievements anzeigen	200 OK	200 OK	OK

Die Testergebnisse sind durch **Insomnia-Requests und After-Response-Tests** reproduzierbar.

Wichtig: Beim zweiten Durchlauf der Tests treten erwartete Fehler auf, da bestimmte Daten bereits existieren (z.B. kann ein Benutzername nicht erneut verwendet werden).

5. Testwerkzeuge

- **Insomnia** – manuelle API-Tests
- **JUnit / Spring Boot Test** – Unit-Tests (Controller / Service)

6. Fazit der Tests

Die durchgeführten Tests zeigen, dass:

- alle zentralen REST-Endpunkte korrekt funktionieren
- Rollen- und Berechtigungen sauber durchgesetzt werden
- fehlerhafte Eingaben korrekt abgefangen werden
- Konflikte (z.B. Duplicate, Stock = 0) korrekt behandelt werden

Das System erfüllt damit die funktionalen Anforderungen des Projektes.

User Stories und Zeitaufwand

Player

US1 - Account erstellen
Als *Player* möchte ich mich registrieren können, damit ich einen eigenen Spielaccount habe.
Akzeptanz:

- Benutzername und E-Mail sind einzigartig
- Ungültige Daten → 400
- Erfolg → User wird erstellt

US2 – Stats speichern (SaveStats)
Als *Player* möchte ich meinen aktuellen Spielstand speichern, damit er in der History abgelegt wird.
Akzeptanz:

- Jeder Aufruf erzeugt **einen neuen Eintrag** in stats
- Der Client sendet keine Werte, sie werden aus users gelesen
- Nicht eingeloggt → 401

US3 – Item nehmen
Als *Player* möchte ich ein Item nehmen, damit es in meinem Inventar erscheint.
Akzeptanz:

- stock > 0 → Item wird vergeben, Stock reduziert
- stock = 0 → 409 Conflict
- Vorgang ist transaktional (ACID)
- Bei gleichzeitigen Requests bekommt nur einer das letzte Item (sonst 409).

US4 – Achievements sehen
Als *Player* möchte ich meine freigeschalteten Achievements anzeigen lassen, damit ich meinen Fortschritt sehe.
Akzeptanz:

- Nur eigene Achievements werden angezeigt
- Leere Liste, falls noch keine vorhanden

- Nicht eingeloggt → 401

Admin

US5 – Item verwalten

Als *Admin* möchte ich Items erstellen und deren Stock setzen, damit das Spiel verwaltet werden kann.

Akzeptanz:

- Nur Admin darf Items anlegen/ändern
- Code ist einzigartig
- Ungültige Daten → 400
- Kein Admin → 403

US6 – Achievements verwalten

Als *Admin* möchte ich Achievements erstellen, damit neue Ziele ins Spiel kommen.

Akzeptanz:

- Code ist einzigartig
- Nur Admin darf erstellen
- Erfolg → Achievement existiert im System
- Kein Admin → 403

User Story	Soll
US1 Account erstellen	30 min
US2 SaveStats	35 min
US3 Item nehmen (transaktional)	45 min
US4 Achievements anzeigen	20 min
US5 Items verwalten	40 min
US6 Achievements verwalten	30 min
Total	**200 min (3h 20m)**

Arbeitsplanung

IPERKA	Aufgabe	Soll (min)	Ist (min)
Informieren	Elevator Pitch + Scope	30	20
	Anforderungen zusammenfassen	30	20
	Zwischentotal Informieren	**60**	**40**
Planen	User Stories (5–10)	40	20
	Akzeptanzkriterien	20	10
	Aufwandsschätzung + Mini-Zeitplan	20	15
	ERD final prüfen	20	40
	SQL-Schema finalisieren	20	20
	REST-Endpunkte planen	20	20
	Testplan (manuell)	10	25
	Zwischentotal Planen	**150**	**150**
Entscheiden	Architektur-Entscheide formulieren	30	30
	Zwischentotal Entscheiden	**30**	**30**
Realisieren	Projekt-Setup (Spring, DB, Git)	45	30
	Entities (6 Klassen)	45	60
	Repositories	15	15
	Services (Business-Logik)	75	120
	Controller + DTOs	45	80
	Validierung + Exception Handling	15	30
	Zwischentotal Realisieren	**240**	**335**
Kontrollieren	Manuelle Tests (Insomnia)	45	90
	Testprotokoll	20	15
	Unit-Tests	25	90
	Zwischentotal Kontrollieren	**90**	**195**
Auswerten	Installationsanleitung / README	30	20
	Doku finalisieren (PDF)	45	40
	Zwischentotal Auswerten	**75**	**60**
Gesamt	**Totalzeit**	**645 (10 3/4h)**	**810 (13.5 h)**

Fazit: Ich habe insgesamt mehr Zeit benötigt als ursprünglich geplant. Der Hauptgrund dafür war, dass mir für das Projekt mehr Zeit zur Verfügung stand als angenommen. Zudem habe ich bewusst mehr umgesetzt, als ursprünglich vorgesehen war. Da mir die Arbeit an diesem Projekt sehr viel Freude bereitet hat, habe ich zusätzlich Zeit investiert, um die Funktionalität, Qualität und Tests weiter zu verbessern.

DatenKonzept + ERD