Technical Design Document for Bartr(v2.1)

Contents:

1. Introduction

1.1 Purpose:

This Technical Design Document (TDD) elaborates on the technical architecture, design patterns, data model, and technology stack required to implement the insurance platform outlined in the Functional Requirements Document (FRD) version It serves as a blueprint for the development team to build and deploy the system.

1.2 Intended Audience:

This document is intended for the development team (software engineers, database administrators, DevOps engineers), architects, technical leads, and quality assurance engineers involved in the insurance platform project.

1.3 Scope:

This TDD covers the high-level and low-level technical design aspects of the core functionalities described in the FRD, including creating account, login to the account, view profile, creating a course, enrolling into a course, search for a course, purchasing of XPs. It also addresses the non-functional requirements related to performance, usability, reliability, and scalability.

1.4 Relationship to FRD:

This document directly references and expands upon the functional and non-functional requirements defined in the FRD. Each technical design decision aims to fulfil one or more of the stated requirements.

2. Technologies Used

1. Frontend – HTML5, CSS3, Bootstrap, Angular
2. Backend API – Spring Boot (Java)
3. Database – MySQL (via Spring Data JPA)
4. Authentication – Spring Security with JWT

3. Database Implementation

The database will be designed to store and retrieve data related to users, categories, payments, transactions, courses, enrollments. Below is the design of key entities and their attributes:

3.1 users:

- id int primary key
- username varchar
- email varchar
- password varchar
- phone varchar
- fullname varchar
- xp int
- avatar_url varchar
- created_at timestamp

3.2 categories:

- id int primary key
- name varchar
- description text
- xp_cost int

3.3 courses:

- id int primary key
- title varchar
- description text
- category_id int foreign key referencing categories.id
- creator_id int foreign key referencing users.id
- created_at timestamp
- level varchar

3.4 enrollments:

- id int primary key
- course_id int foreign key references courses.id

- learner_id int foreign key references users.id
- enrollment_date timestamp

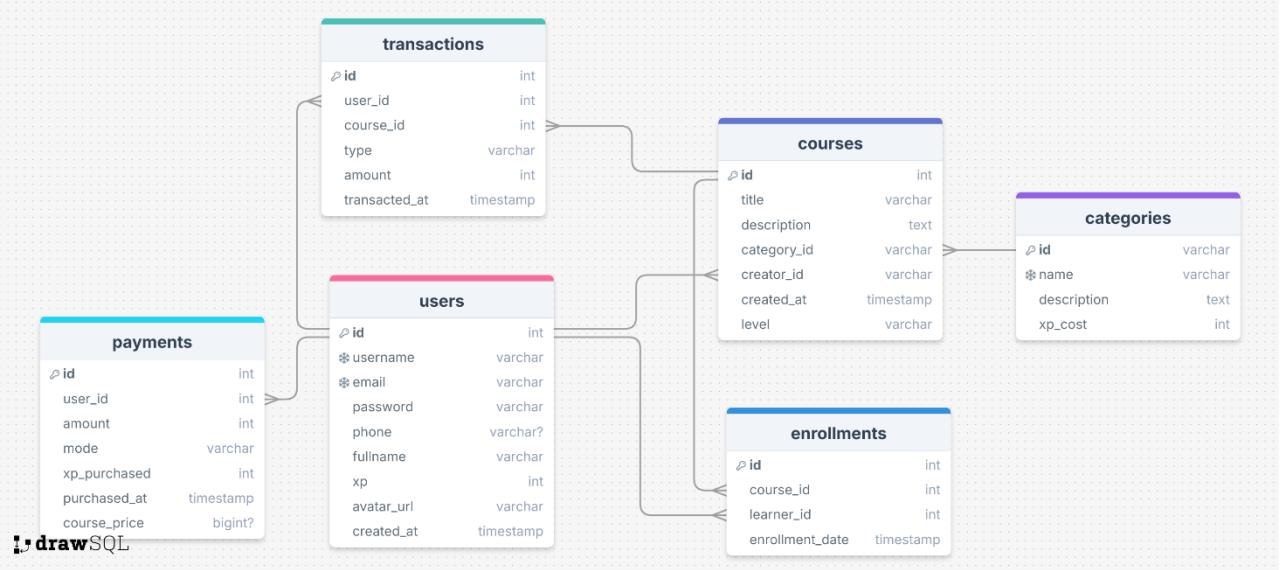
3.5 transactions:

- id int primary key
- user_id int foreign key references users.id
- course_id int foreign key references courses.id
- type varchar
- amount int
- transacted_at timestamp

3.6 payments:

- id int primary key
- user_id int foreign key references users.id
- amount int
- mode varchar
- xp_purchased int
- purchased_at timestamp

3.7 ER Diagram:



4.UI Implementation:

4.1 Overview:

The UI architecture is designed in Angular and follows a modular component-based approach. It uses services for API communication, guards for route protection, and state management to improve data flow and performance.

4.2 Component Structure:

- **App Module (Root):**
  - Shared Module (reusable components like button, headers)
- **Navbar Module:**
  - Bartr logo
  - Create a course button
  - Log in button (if user is not logged in)
  - Sign up button (if user is not logged in)
- **Auth Module:**
  - Login Component:
    - Username field
    - Password field
    - Forgot password button
    - Login button
    - Alternative login methods button
    - Register now button (if account is not present)
  - Register Component:
    - Full name field
    - Username field
    - Email field
    - Password field
    - Create account button
    - Alternative registration buttons (Google, Twitter)
    - Login button (if already a user)
  - Auth Service
- **Dashboard Module:**
  - User Profile Component:

        - Username avatar image
        - Username text
        - Email text
        - Name card with profile name text
        - Contact card with contact text
        - Xp card with xp amount text
        - Country card with country text
    - Sidebar Component:
        - Personal Information button
        - Courses Enrolled button
        - Courses Created button
        - Profile Settings button
- **Course Module:**
    - Course List Document
    - Course Detail Component
    - Create Course Component
    - Enroll Course Component
- **XP Module:**
    - Purchase XP Component
    - XP History Component
- **Footer Module:**
    - Bartr logo text
    - Social media handle buttons
    - Basic Information texts

4.3 Routing Configuration:

- **Public Routes:**
    - /login: Login Component
    - /register: Register Component
- **Protected Routes:**
    - /profile: User Profile Component
    - /courses: Course List Component
    - /course/id: Course Detail Component

    - /create-course: Create Course Component
    - /purchase-xp: Purchase XP Component

4.4 UI Libraries Used:

- **Angular Material:**
    - Buttons
    - Form Fields
    - Dialog
    - Table
- **Ngx-toaster:**
    - Alerts
    - Feedback
- **Ngx-spinner:**
    - Loading Indicators

4.5 Error Handling:

- Global Error handler for API Failures
- Route fallback (404 page)
- Loading and empty states for components

5. Backend Implementation:

5.1 Models (Entities / POJOs):

5.1.1 User:

Represents the user's core details and XP balance.

Fields: id, username, email, password, phone, fullname, avatarUrl, createdAt, xp

5.1.2 Course:

Represents a category/course available for barter.

Fields: id, title, description, skill, category (reference to Skill), creator (reference to User), createdAt, level

5.1.3 Category:

Defines course categories and the XP cost for each.

Fields: id, name, description, xpCost

5.1.4 Enrollment:

Represents which users have enrolled in which courses.

Fields: id, course (reference), learner (reference), enrollmentDate

5.1.5 Payment:

Logs real-money transactions to buy XP.

Fields: id, user (reference), amount, mode, xpPurchased, purchasedAt

5.1.6 Transaction:

Logs in-app XP movements for transparency.

Fields: id, user (reference), course (reference), type (XP Spend, Refund), amount, transactedAt

5.2 Services (Business Logic Layer):

5.2.1 UserService:

- Fetch, create, update user details
- Update XP balance

- Handle user-related business logic

5.2.2 CourseService:

- Create courses
- Fetch courses (all, by category, by creator)
- Course-related validations

5.2.3 EnrollmentService:

- Enroll users in courses
- Deduct XP from learners, credit XP to course creators
- Save enrollment data

5.2.4 PaymentService:

- Handle XP top-up logic
- Show calculated amount for XP
- Display QR code for payment
- XP will be credited once user clicks payment completion button

5.2.5 AuthService:

- Handle user authentication using Spring Security
- Manage JWT Token generation and validation
- Encrypt and verify passwords using BCryptPasswordEncoder

5.3 Controllers (REST API Entry Points):

5.3.1 AuthController:

Register, login, return current user info.

Endpoints: /api/auth/*

5.3.2 UserController:

Fetch user profile, XP balance, user's created/enrolled courses.

Endpoints: /api/users/*

5.3.3 CourseController:

Course creation, course details, course listings by category.

Endpoints: /api/courses/*

5.3.4 EnrollmentController:

Enroll users in courses.

Endpoints: /api/courses/{id}/enroll

5.3.5 PaymentController:

XP top-up request, QR code display, and XP addition (on user-confirmed action)

Endpoints:

- POST: /api/payments/request – Submit XP top-up request
- GET: /api/payments/qr – Returns QR code for display
- POST: /api/payments/xp-credit – XP will be credited once user clicks button

5.4 Spring JPA & Security Integration:

Spring Data JPA is used for ORM mapping to the MySQL database. Each entity (User, Course, Payment, etc.) has a corresponding Repository interface.

Spring Security handles:

- User authentication with credentials stored in the database
- Role-based authentication
- JWT-based token validation for REST API endpoints

Password storage: BCrypt encoded

Token Format: Bearer Token (JWT)

6. Validations:

The Bartr Skill Barter System implements robust validations at both the frontend (Angular) and backend (Spring Boot) layers to ensure data integrity, enhance security, and deliver a seamless user experience.

6.1 Validation Types:

- Client-side (Angular): For immediate user feedback using reactive forms and validation messages.
- Server-side (Spring Boot): Final enforcement of rules before writing to the database or processing logic.
- Database-level (Appwrite): Schema validation where applicable.

6.2 Validation Rules by module:

- **User Registration and login:**

Field	**Validation Rule**	**Message**
Username	Required, min 3 characters	“username is required (min 3 characters)”
Email	Required, valid email format	“Enter a valid email address”
Password	Required, min 8 characters	“Password must be at least 8 characters long”
Phone	Numeric, 10-15 digits	“Enter a valid phone number”
Full Name	Required	“Full name is required”

- Course Creation:

Field	Validation Rule	Message
Title	Required, min 5, max 100 characters	“Course title is required (5-100 characters)”
Description	Required, min 10 characters	“Description must be at least 10 characters”
Skill/ Category	Must be selected from existing skill list	“Please select a valid skill category”
Level	One of [“Beginner”, “Intermediate”, “Advanced”]	“Select a valid difficulty level”
Video URL	Valid URL format (stored via Appwrite)	“Enter a valid course link”

- **XP Transaction (Course Enrollment):**

Field	Validation Rule	Message
User XP	Must be >= course XP cost	“Insufficient XP balance”
Duplicate Enroll	User must not be already enrolled in same course	“Already enrolled in same course”
Course ID	Must exist in DB	“Course not found”
Transaction Log	Log must be generated for each XP deduction/credit	“Transaction logging failed’

- **XP Purchase (Payment Integration):**

Field	Validation Rule	Message
XP Amount	Required numeric, >= minimum package (eg. 10xp)	“Enter a valid XP amount (minimum 10)”
User Confirmation	User must confirm after payment	“Please confirm once payment is done”
Admin Approval	Required for XP to be credited	“Admin will credit XP manually after review”

6.3 Backend API Validations:

- @Valid annotations for DTOs
- @NotNull, @Size, @Email, @Min/@Max used for field constraints
- Exception handlers for MethodArgumentNotValidException
- Custom validation: XPTransferValidator, CourseOwnershipValidator

6.4 Frontend Form Validations (Angular):

- Angular reactive forms with FormBuilder and Validators
- Error messages displayed on blur or submit
- Fields disabled until all validations are met
- Use of Angular Material mat-error messages

6.5 Database Validations (MySQL + JPA):

- Required fields enforced using @NotNull, @Size, and @Column (nullable=false)
- Unique constraints defined using @Column(unique=true)
- Referential Integrity via @ManyToOne / @OneToMany with proper cascade and fetch types
- JPA schema validation via Hibernate

7. API Design:

7.1 Authentication APIs:

Base Path: /api/auth

Method Endpoint Request Body Response Description POST /register {username, email, password } 201 Created or 400 Error Registers a new user POST /login {email, password } { token, userInfo } Logs in user, returns JWT token GET /me Authorization: Bearer token { user details } Returns current logged-in user.

7.2 User APIs:

Base Path: /api/users

Method Endpoint Auth Description GET /{id} Yes Fetch user profile GET /{id}/xp Yes Fetch user's XP balance GET /{id}/courses Yes Fetch courses created by user GET /{id}/enrolled Yes Fetch courses user is enrolled in.

7.3 Course APIs:

Base Path: /api/courses

Method Endpoint Auth Request Body Response Description GET / No - List of all courses Get all available courses GET /{id} No - Course detail Get details of a specific course POST / Yes { title, description, categoryId } Course created Create a new course POST /{id}/enroll Yes { userId } Enrollment response Enroll in a course if XP is sufficient GET /category/{name} No - Courses list Get courses filtered by category name

7.4 Categories APIs:

Base Path: /api/categories

Method Endpoint Auth Description GET / No List all available skill categories and XP values

7.5 XP & Transactions APIs:

Base Path: /api/xp

Method Endpoint Auth Description GET /transactions Yes Get current user’s XP transaction history POST /transfer Yes Transfer XP between users (internal use).

7.6 Payment APIs:

Base Path: /api/payments

Method Endpoint Auth Request Body Description POST /request User requests to purchase XP by submitting amount GET /qr Returns a QR code for the

payment (Static UPI QR) POST /xp-credit XP will be credited automatically once user clicks payment complete button

7.7 Status Codes:

200 OK – Request successful

201 Created – Resource created

400 Bad Request – Invalid input

401 Unauthorized – Auth failure

403 Forbidden – Insufficient permissions

404 Not Found – Resource does not exist

500 Internal Server Error – Unexpected issue

8. Services:

This section outlines the Service Layer which is the heart of business logic for Bartr. It handles core functionalities such as user management, course creation, XP transactions, payments, and enrollments. Each service is designed with TDD in mind, ensuring test cases are defined before the implementation of any logic.

8.1 Service Layer Classes:

8.1.1 AuthService:

- Responsibilities:
    - Register new users with encoded password.
    - Authenticate users using email & password.
    - Generate and validate JWT tokens.
- Test Cases:
    - Valid login returns JWT and user info.
    - Wrong credentials return 401 Unauthorized.
    - JWT token correctly validates secured endpoints
    - Registration fails if email already exists.

8.1.2 UserService:

- Responsibilities:
  - Fetch, update user profile.
  - Manage XP balance.
  - Return enrolled/created courses.
- Test Cases:
  - XP updates correctly after course enrollment.
  - Invalid user ID returns 404 Not Found.
  - Fetch user returns full profile with avatar.

8.1.3 CourseService:

- Responsibilities:
  - Create, update, delete course data.
  - Fetch all courses or by filters (category, creator).
- Test Cases:
  - Course created with valid fields.
  - Invalid category ID returns 400 Bad Request.
  - Only creator can edit or delete course.

8.1.4 EnrollmentService:

- Responsibilities:
  - Enroll learners into courses.
  - Deduct XP from learners and credit XP to course creators.
  - Validate duplicate enrollments.
- Test Cases:
  - Enrollment successful if XP is sufficient.
  - Already enrolled returns 409 Conflict.
  - XP deducted and transaction recorded after enrollment.

8.1.5 TransactionService:

- Responsibilities:
  - Log all XP transactions.
  - Allow XP transfer (future use).
  - Maintain integrity of XP history.
- Test Cases:
  - Transaction logged on every XP change.
  - Invalid course/user ID throws error.
  - Transfers cannot exceed available XP.

8.1.6 PaymentService:

- Responsibilities:
    - Accept XP top-up requests from users
    - Generate and return static UPI QR code
    - Credits XP to user account once user clicks the button
- Test Cases:
    - Payment success updates XP.

8.2 General Design Patterns & Best Practices:

- Follows Separation of Concerns using layered architecture.
- Uses Dependency Injection for testability.
- Business rules are unit-tested using mock repositories.
- Test files follow structure: ServiceNameTest.java.

8.3 Security in Services:

- Authentication and role-based authorization enforced before service-level access.
- Sensitive operations like payments and XP transfers validate ownership and integrity.

8.4 Service Interaction Flow:

8.4.1 Course Enrollment Flow:

Controller → EnrollmentService → UserService (XP check/update) → CourseService (course validation) → TransactionService (log XP)

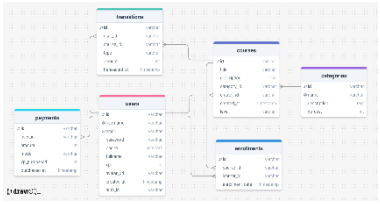
8.4.2 Payment Flow:

Controller → PaymentService → Stripe API / Webhook → UserService (update XP) → TransactionService (log XP purchase) Let me know if you want mock unit test examples (e.g., using JUnit or Mockito) or diagrams (like a flowchart) for visualizing service

Change Log:

Section	Initial Content	New Content	Changes
2	1. Frontend – HTML5, CSS3, Bootstrap, Angular 2. Backend API – Spring Boot (Java) 3. Database – Appwrite (Cloud/Hosted) 4. Authentication – Appwrite Auth 5. Payment Gateway – Stripe API	1. Frontend – HTML5, CSS3, Bootstrap, Angular 2. Backend API – Spring Boot (Java) 3. Database – MySQL (via Spring Data JPA) 4. Authentication – Spring Security with JWT	Changed Database and Authentication from Appwrite and Appwrite Auth to MySQL and Spring Security with JWT Also removed using Stripe API for Payment verification
3.1	• id varchar primary key • username varchar • email varchar • password varchar • phone varchar • fullname varchar • xp int • avatar_url varchar • created_at timestamp • auth_id varchar	• id int primary key • username varchar • email varchar • password varchar • phone varchar • fullname varchar • xp int • avatar_url varchar • created_at timestamp	Reverted to Previous Version
3.2	• id varchar primary key • name varchar • description text • xp_cost int	• id int primary key • name varchar • description text • xp_cost int	Reverted to Previous Version
3.3	• id varchar primary key • title varchar • description text • category_id varchar foreign key referencing categories.id	• id int primary key • title varchar • description text • category_id int foreign key referencing categories.id	Reverted to Previous Version

	• creator_id varchar foreign key referencing users.id • created_at timestamp • level varchar	• creator_id int foreign key referencing users.id • created_at timestamp • level varchar	
3.4	• id varchar primary key • course_id varchar foreign key references courses.id • learner_id varchar foreign key references users.id • enrollment_date timestamp	• id int primary key • course_id int foreign key references courses.id • learner_id int foreign key references users.id • enrollment_date timestamp	Reverted to Previous Version
3.5	• id varchar primary key • user_id varchar foreign key references users.id • course_id varchar foreign key references courses.id • type varchar • amount int • transacted_at timestamp	• id int primary key • user_id int foreign key references users.id • course_id int foreign key references courses.id • type varchar • amount int • transacted_at timestamp	Reverted to Previous Version
3.6	• id varchar primary key • user_id varchar foreign key references users.id • amount int • mode varchar • xp_purchased int purchased_at timestamp	• id int primary key • user_id int foreign key references users.id • amount int • mode varchar • xp_purchased int • purchased_at timestamp	Reverted to Previous Version

3.7			Reverted to Previous Version
5.2.4	• Handle XP top-up logic • Create payment records • Update XP balance after payment verification	• Handle XP top-up logic • Show calculated amount for XP • Display QR code for payment • XP will be credited once user clicks payment completion button	Changed Payment Service without using Stripe API
5.2.5	• Wrap Appwrite SDK calls to create, update, delete, and read documents • Map Appwrite JSON responses to Java models	• Handle user authentication using Spring Security • Manage JWT Token generation and validation • Encrypt and verify passwords using BCryptPasswordEncoder	Changed Appwrite Auth Service responsibilities to Spring Security Auth Service
5.3.5	Initiate payments, verify webhooks, log XP purchase. Endpoints: /api/payments/*	XP top-up request, QR code display, and XP addition (on user-confirmed action) Endpoints: • POST: /api/payments/request – Submit XP top-up request • GET: /api/payments/qr – Returns QR code for display • POST: /api/payments/xp-credit – XP will be credited once user clicks button	Changed Payment Controller to have separate end points as Stripe API is not used.

5.4	Appwrite replaces direct DB (like JPA repositories). Interactions: • CreateDocument → insert new user, course, enrollment, etc. • ListDocuments → fetch records for listing APIs. • UpdateDocument → update XP or user profile fields. • DeleteDocument → (if required for admin features). • File APIs → store thumbnails, avatars, videos.	Spring Data JPA is used for ORM mapping to the MySQL database. Each entity (User, Course, Payment, etc.) has a corresponding Repository interface. Spring Security handles: • User authentication with credentials stored in the database • Role-based authentication • JWT-based token validation for REST API endpoints Password storage: BCrypt encoded Token Format: Bearer Token (JWT)	Changed the information regarding using Appwrite and Appwrite Auth in backend to using Spring Data JPA and Spring Security
6.5	• Required fields configured via Appwrite's document schema • Unique constraints: email, username • Referential integrity: creator_id → users, skill_id → skills	• Required fields enforced using @NotNull, @Size, and @Column (nullable=false) • Unique constraints defined using @Column(unique=true) • Referential Integrity via @ManyToOne / @OneToMany with proper cascade and fetch types • JPA schema validation via Hibernate	Changed Database validations from Appwrite to Spring Data JPA
7.6	Method Endpoint Auth Request Body Description POST /create Yes { userId, xpAmount } Initiates Razorpay payment and returns orderId POST /verify Verifies payment	Method Endpoint Auth Request Body Description POST /request User requests to purchase XP by submitting amount GET /qr Returns a QR code for the payment (Static UPI QR) POST /xp-credit XP	Changed Payment APIs without using Stripe API

	and updates XP GET /history Yes - Returns all payment history for user	will be credited automatically once user clicks payment complete button	
8.1.1	• Responsibilities: ○ Handle user login, registration. ○ Password encryption. ○ JWT token generation and validation (using Appwrite Auth). • Test Cases: ○ Successful login returns valid JWT. ○ Incorrect credentials return 401 Unauthorized. ○ Registration with all valid fields creates new user.	• Responsibilities: ○ Register new users with encoded password. ○ Authenticate users using email & password. ○ Generate and validate JWT tokens. • Test Cases: ○ Valid login returns JWT and user info. ○ Wrong credentials return 401 Unauthorized. ○ JWT token correctly validates secured endpoints ○ Registration fails if email already exists.	Changed Auth service using Appwrite Auth to Spring Security
8.1.6	• Responsibilities: ○ Initiate payments via Stripe. ○ Verify webhooks. ○ Credit XP post successful payment. • Test Cases:	• Responsibilities: ○ Accept XP top-up requests from users ○ Generate and return static UPI QR code ○ Credits XP to user account once	Changed Payment Service without using Stripe API

	○ Payment success updates XP. ○ Webhook signature mismatch returns 403 Forbidden. ○ History returns all valid payment records.	user clicks the button • Test Cases: ○ Payment success updates XP.	
8.1.7	• Responsibilities: ○ Abstract interaction with Appwrite SDK. ○ Manage CRUD operations for users, courses, enrollments, etc. • Test Cases: ○ CreateDocument inserts correct data schema. ○ Invalid document schema throws exception. ○ UpdateDocument reflects immediately on fetch.		Removed the Appwrite Service section since we are not using it anymore