Repository Layer Documentation

Overview

The repository layer acts as an abstraction between the application's domain logic and its data sources (in this case, Supabase). It provides a consistent API for the rest of the system to interact with data without depending directly on database queries or client configurations.

By using repositories, the system achieves: - **Separation of concerns** between business logic and data persistence. - **Easier testing** via mocking repository interfaces. - **Simplified maintainability** when changing data sources (e.g., migrating from Supabase to another backend). - **Improved readability** and a centralized place for handling data-related logic.

Piece Repository → PieceRepository

Purpose

Handles all CRUD (Create, Read, Update, Delete) and filtering operations related to the `Piece` domain object. It also transforms database records into domain entities through the `PieceFactory`.

Responsibilities

- Fetch all clothing pieces or a specific piece by ID.
- Create, update, and delete clothing piece entries.
- Filter pieces based on multiple attributes such as name, category, color, size, brand, gender, price, and condition.
- Use the `PieceFactory` to convert between database DTOs and domain entities (ensuring consistent business logic).

Benefits

- **Abstraction:** The rest of the application doesn't need to know about the database structure or queries.
- **Reusability:** Common data operations are encapsulated, reducing code duplication.
- **Consistency:** Data is retrieved and transformed through a single, standardized interface.
- **Error Handling:** Centralized error catching and null safety mechanisms prevent propagation of invalid data.
- **Scalability:** Makes it easy to integrate caching or switch databases later without refactoring domain or UI layers.

Methods Summary

Method	Return Type	Description	Example
getPieces()	Promise<Array<Piece>>	Retrieves all pieces from the database	`repo.getPieces()`
getPieceById(id)	Promise<Piece | null>	Fetches a single piece by its unique ID	`repo.getPieceById("12")`
createPiece(piece)	Promise<Error | null>	Inserts a new piece record	`repo.createPiece(newPiece)`
updatePiece(piece)	Promise<boolean>	Updates an existing record	`repo.updatePiece(existingPiece)`
deletePiece(id)	Promise<boolean>	Deletes a record by ID	`repo.deletePiece("12")`
filterPieces(filters)	Promise<Array<Piece>>	Retrieves pieces based on provided criteria	`repo.filterPieces({ color: "red", size: "M" })`

Justification for Repository Pattern

The repository pattern is used to decouple the domain and data mapping layers, ensuring that the domain model remains free from data-access logic. This allows for: - **Cleaner architecture:** Reduces direct dependencies on Supabase queries or API responses. - **Easier unit testing:** Repositories can be mocked or stubbed. - **Enhanced flexibility:** Future migration to other databases (e.g., PostgreSQL, Firebase) can be done without altering core business logic. - **Better maintainability:** Centralizing data operations minimizes redundancy and potential inconsistencies across services.

Example Usage

```
const repo = new PieceRepository();

// Create a new piece
await repo.createPiece({
  name: "Summer T-shirt",
  category: "SHIRT",
  color: "red",
  brand: "H&M",
  gender: "UNISEX",
  size: "MEDIUM",
  price: 10,
  condition: "LIKE_NEW",
  reason: "Style change",
  images: ["tshirt1.png"],
  user_id: "847"
});

// Retrieve filtered pieces
const filtered = await repo.filterPieces({ category: "SHIRT", color: "red" });
console.log(filtered.length);
```