📌 1️⃣ Entity Layer (`com.cts.smartspend.entity`)

📍 **Purpose**: Represents **database tables** using JPA (Java Persistence API).
📍 **How It Works**:

- Each **class = a database table**.
- Uses **JPA annotations** (`@Entity` , `@Table` , `@OneToMany` , etc.).
- Defines **relationships** between tables.

🔹 `User.java` – Represents Users in the System

📍 **Purpose:**

- Stores **username, password (hashed), and role (ADMIN/USER)**.
- Used for **authentication & authorization**.

📍 **Code Breakdown:**

java Copy Edit

```
@Entity @Table(name = "users") public class User {
```

✅ **Marks this class as a JPA entity** (`@Entity`) and maps it to the `users` table (`@Table(name = "users")`).

java Copy Edit

```
@Id @GeneratedValue(strategy = GenerationType.IDENTITY) private Long id;
```

✅ Defines `id` **as the primary key** and sets it to auto-increment (`IDENTITY`).

java Copy Edit

```
@Column(unique = true, nullable = false) private String username;
```

✅ **Ensures usernames are unique** and **cannot be null**.

java Copy Edit

```
@Column(nullable = false) private String password;
```

✅ Stores **hashed passwords** (encrypted using `BCrypt`).

java Copy Edit

```
@Enumerated(EnumType.STRING) private Role role;
```

✅ Stores user roles (`ADMIN` or `USER`) in **string format**.

java Copy Edit

```
@OneToMany(mappedBy = "user", cascade = CascadeType.ALL) private List<Expense> expenses;
```

✅ **One user can have multiple expenses.**
✅ If a user is deleted, **all their expenses are deleted too** (`CascadeType.ALL`).

🔹 `Expense.java` – Represents Expenses

📌 **Purpose:**

- Stores **amount, description, date, category, and user**.
- Linked to **User** (`@ManyToOne`) and **Category** (`@ManyToOne`).

📌 **Code Breakdown:**

java Copy Edit

```
@Entity @Table(name = "expenses") public class Expense {
```

✅ Defines the `expenses` table.

java Copy Edit

```
@ManyToOne @JsonBackReference private Category category;
```

✅ **Each expense belongs to one category.**

java Copy Edit

```
@ManyToOne @JsonBackReference private User user;
```

✅ **Each expense belongs to one user.**

java Copy Edit

```
private Double amount; private String description; private LocalDate date;
```

✅ Stores **amount, description, and date.**

🔹 `Category.java` – Represents Expense Categories

📌 **Purpose:**

- Stores **category names** (e.g., "Food", "Transport").
- Linked to **Expenses** (`@OneToMany`).

📌 **Code Breakdown:**

java Copy Edit

```
@Entity @Table(name = "categories") public class Category {
```

✅ Defines the `categories` table.

java Copy Edit

```
@OneToMany(mappedBy = "category", cascade = CascadeType.ALL) @JsonManagedReference
private List<Expense> expenses;
```

✅ **A category can have multiple expenses.**

🔹 `Budget.java` – Represents Budget Limits

📌 **Purpose:**

- Stores **budget limits** for categories.
- Ensures **users don't overspend** in a category.

📌 Code Breakdown:

java Copy Edit

```java
@Entity @Table(name = "budgets") public class Budget {
```

✅ Defines the `budgets` table.

java Copy Edit

```java
@ManyToOne @JoinColumn(name = "category_id", nullable = false) private Category category;
```

✅ **Each budget is linked to a category.**

java Copy Edit

```java
private double amount; private LocalDate startDate; private LocalDate endDate;
```

✅ Stores **budget amount, start date, and end date.**

📌 Repository Layer (`com.cts.smartspend.repo`)

📍 **Purpose**:

- **Handles database operations** (fetching, saving, updating, deleting).
- Uses **Spring Data JPA** (`JpaRepository`) to simplify queries.
- **Interacts with the Service Layer** to execute business logic.

📍 **How It Works**:

1️⃣ The **Controller Layer** calls the **Service Layer**.
2️⃣ The **Service Layer** calls the **Repository Layer** to fetch/save data.
3️⃣ The **Repository Layer** interacts with the **database**.

🔹 1️⃣ `UserRepo.java` – Managing Users in the Database

📍 **Purpose**:

- Fetch users **by username** (for authentication).
- Fetch users **by role** (for admin operations).
- Standard **CRUD operations** (create, read, update, delete).

🔍 Code Breakdown

java Copy Edit

```
package com.cts.smartspend.repo;
```

✅ Defines the **package** for repository classes.

java Copy Edit

```
import com.cts.smartspend.entity.User; import
org.springframework.data.jpa.repository.JpaRepository;
```

✅ **Imports** `JpaRepository` to use built-in database operations.

```java
public interface UserRepo extends JpaRepository<User, Long> {
```

✅ Extends `JpaRepository<User, Long>`, which provides:

- `findAll()` → Fetch all users
- `findById(Long id)` → Fetch user by ID
- `save(User user)` → Save/update user
- `deleteById(Long id)` → Delete user

```java
User findByUsername(String username);
```

✅ **Custom method** to fetch users **by username**.
✅ **Used in** `CustomUserDetailsService.java` for authentication.

```java
List<User> findByRole(User.Role role);
```

✅ **Custom method** to fetch users **by role** (`ADMIN` or `USER`).
✅ **Used in** `UserService.java` **to fetch all ADMINs or USERS**.

🔹 2 `CategoryRepo.java` – Managing Expense Categories

📍 **Purpose**:

- Manages **CRUD operations** for expense categories.
- Fetches categories for **users to select when adding expenses**.

🔍 Code Breakdown

```java
```

```
package com.cts.smartspend.repo; import com.cts.smartspend.entity.Category; import
org.springframework.data.jpa.repository.JpaRepository;
```

✅ Imports `JpaRepository` to simplify database operations.

java Copy Edit

```
public interface CategoryRepo extends JpaRepository<Category, Long> { }
```

✅ **Extends** `JpaRepository<Category, Long>`, providing:

- `findAll()` → Fetch all categories.
- `findById(Long id)` → Fetch category by ID.
- `save(Category category)` → Save/update category.
- `deleteById(Long id)` → Delete category.

📍 **Where It's Used**:

- `CategoryService.java` to fetch & manage categories.
- `ExpenseService.java` when assigning a category to an expense.

🔹 3️⃣ `ExpenseRepo.java` – Managing Expenses

📍 **Purpose**:

- Fetches **expenses based on category**.
- Fetches **expenses within a specific date range**.
- Standard **CRUD operations** for expenses.

🔍 Code Breakdown

java Copy Edit

```
package com.cts.smartspend.repo; import com.cts.smartspend.entity.Expense; import
org.springframework.data.jpa.repository.JpaRepository; import java.time.LocalDate; import
java.util.List;
```

✅ Imports necessary components, including `LocalDate` for **date-based queries**.

```java
public interface ExpenseRepo extends JpaRepository<Expense, Long> {
```

✅ **Extends** `JpaRepository<Expense, Long>`, which provides:

- `findAll()` → Fetch all expenses.
- `findById(Long id)` → Fetch expense by ID.
- `save(Expense expense)` → Save/update expense.
- `deleteById(Long id)` → Delete expense.

```java
List<Expense> findByCategoryId(Long categoryId);
```

✅ **Fetches all expenses belonging to a specific category.**
✅ **Used in** `ExpenseService.java` **to get expenses per category.**

```java
List<Expense> findByCategoryIdAndDateRange(Long categoryId, LocalDate startDate,
LocalDate endDate);
```

✅ **Fetches expenses for a category within a date range.**
✅ **Used in** `BudgetService.java` **to calculate remaining budget.**

🔹 4 `BudgetRepo.java` – Managing Budgets

📍 **Purpose:**

- Fetches budgets **by category** and **date**.
- Helps ensure **users do not overspend their budget**.

🔍 Code Breakdown

```java
```

```
package com.cts.smartspend.repo; import com.cts.smartspend.entity.Budget; import
org.springframework.data.jpa.repository.JpaRepository; import java.time.LocalDate; import
java.util.Optional;
```

✅ Imports `JpaRepository` for budget operations.
✅ Imports `LocalDate` for **date-based queries**.

java Copy Edit

```
public interface BudgetRepo extends JpaRepository<Budget, Long> {
```

✅ **Extends** `JpaRepository<Budget, Long>` , which provides:

- `findAll()` → Fetch all budgets.
- `findById(Long id)` → Fetch budget by ID.
- `save(Budget budget)` → Save/update budget.
- `deleteById(Long id)` → Delete budget.

java Copy Edit

```
Optional<Budget> findBudgetByCategoryAndDate(Long categoryId, LocalDate date);
```

✅ **Fetches a budget for a specific category on a given date.**
✅ **Used in** `BudgetService.java` **to check if a budget exists before adding expenses.**

📌 How the Repository Layer Interacts with Other Layers

📍 **Example Flow – When a User Adds an Expense**
1️⃣ **UserController (** `POST /expenses/add`) receives request.
2️⃣ Calls `ExpenseService.createExpense()` .
3️⃣ `ExpenseService` calls `CategoryRepo.findById()` to get the category.
4️⃣ `ExpenseService` calls `UserRepo.findById()` to get the user.
5️⃣ `ExpenseService` calls `ExpenseRepo.save()` to store the expense.
6️⃣ Expense is saved in the **database**.

📍 **Example Flow – When Checking Remaining Budget**
1️⃣ **UserController (** `GET /budget/get/{categoryId}`) receives request.
2️⃣ Calls `BudgetService.getBudgetByCategory()` .

3. `BudgetService` calls `BudgetRepo.findBudgetByCategoryAndDate()`.
4. `BudgetService` calls `ExpenseRepo.findByCategoryIdAndDateRange()` to check total spending.
5. Returns the **remaining budget** to the user.

📌 Summary of Repository Layer

Class	Purpose	Key Methods
UserRepo.java	Manages users	`findByUsername()`, `findByRole()`
CategoryRepo.java	Manages categories	Standard CRUD (`findAll()`, `save()`)
ExpenseRepo.java	Manages expenses	`findByCategoryId()`, `findByCategoryIdAndDateRange()`
BudgetRepo.java	Manages budgets	`findBudgetByCategoryAndDate()`

📌 DTO Layer (`com.cts.smartspend.dto`)

📍 **Purpose**:

- **Transfers data between the frontend and backend**.
- **Prevents exposing entity classes** directly to avoid security risks.
- **Helps structure API responses properly**.

📍 **How It Works**:

1️⃣ The **Controller Layer** receives a request from the frontend with a DTO.
2️⃣ The **Service Layer** converts the DTO into an **Entity** and saves it.
3️⃣ The **Repository Layer** fetches data from the database.
4️⃣ The **Service Layer** converts the **Entity back to a DTO** and returns it to the frontend.

🔹 1️⃣ `UserDTO.java` – Transferring User Data

📍 **Purpose**:

- Transfers **user data** (ID, username, role, and password).
- Used when **creating, updating, or fetching users**.
- Prevents exposing the **User entity directly** to the frontend.

🔍 Code Breakdown

```java
package com.cts.smartspend.dto;
```

✅ Defines the **package** for DTO classes.

```java
public record UserDTO(Long id, String username, String password, String role) { }
```

✅ **Defines a DTO as a** `record` (simpler than a class).
✅ **Fields:**

- `id` → Unique user ID.
- `username` → User's login name.
- `password` → User's encrypted password.
- `role` → `ADMIN` or `USER`.

📍 **Where It's Used:**

- `UserController.java` → When registering or fetching users.
- `UserService.java` → When processing user data before saving it.

🔹 2️⃣ `LoginDTO.java` & `LoginResponseDTO.java` – Handling Login Requests

📍 **Purpose:**

- `LoginDTO.java` → Transfers **login credentials** (`username` , `password`).
- `LoginResponseDTO.java` → Returns **JWT token** after successful login.

🔍 Code Breakdown

java Copy Edit

```java
public record LoginDTO(String username, String password) { }
```

✅ **Used when a user logs in** (`POST /login/authenticate`).
✅ **Contains username & password.**

java Copy Edit

```java
public record LoginResponseDTO(String token, String role) { }
```

✅ **Returns a JWT token** after successful login.
✅ **Role is included** to control frontend access (`ADMIN` vs `USER`).

📍 **Where It's Used:**

- `LoginController.java` → Processes login requests.
- `UserService.java` → Returns token upon successful login.

🔹 3️⃣ `ExpenseDTO.java` & `ExpenseResponseDTO.java` – Transferring Expense Data

📍 **Purpose:**

- `ExpenseDTO.java` → Transfers expense data **from frontend to backend**.
- `ExpenseResponseDTO.java` → Transfers expense data **from backend to frontend**.

🔍 Code Breakdown

```java
public record ExpenseDTO(Long id, Double amount, String description, Long categoryId, Long userId, LocalDate date) { }
```

✅ **Used when adding/updating expenses (** `POST /expenses/add` **).**
✅ **Fields:**

- `id` → Unique expense ID.
- `amount` → Expense amount.
- `description` → Short note on the expense.
- `categoryId` → Links expense to a category.
- `userId` → Links expense to a user.
- `date` → Date of expense.

```java
public record ExpenseResponseDTO(Long id, String description, Double amount, LocalDate date, String categoryName, Long userId, String remainingBudget) { }
```

✅ **Used when fetching expenses (** `GET /expenses/get/all` **).**
✅ **Includes extra data:**

- `categoryName` → Converts `categoryId` to a readable name.
- `remainingBudget` → Shows the **remaining budget** after expenses.

📍 **Where It's Used:**

- `ExpenseController.java` → Handles expense-related API calls.
- `ExpenseService.java` → Converts DTO to entity before saving.

🔹 4️⃣ `CategoryDTO.java` – Transferring Category Data

📍 **Purpose:**

- Transfers **category data** (`id` , `name`).
- Used when **adding, updating, or fetching categories**.

🔍 Code Breakdown

```java
public record CategoryDTO(Long id, String name) { }
```

✅ **Used when adding/updating categories** (`POST /categories/add`).
✅ **Prevents exposing the** `Category` **entity** directly.

📍 **Where It's Used:**

- `CategoryController.java` → Handles API requests for categories.
- `CategoryService.java` → Converts DTO into an entity before saving.

🔹 5 `BudgetDTO.java` – Transferring Budget Data

📍 **Purpose:**

- Transfers **budget details** (`amount` , `startDate` , `endDate` , `categoryId`).
- Used when **creating/updating budgets**.

🔍 Code Breakdown

```java
public record BudgetDTO(Long id, double amount, LocalDate startDate, LocalDate endDate, Long categoryId) { }
```

✅ **Used when adding/updating budgets** (`POST /budgets/add`).
✅ **Prevents exposing the** `Budget` **entity** directly.

📍 **Where It's Used:**

- `BudgetController.java` → Handles API requests for budgets.
- `BudgetService.java` → Converts DTO into an entity before saving.

📌 How the DTO Layer Interacts with Other Layers

📍 Example Flow – When a User Logs In

1. **Frontend (`React`)** sends `LoginDTO (username, password)` to `POST /login/authenticate`.
2. `LoginController` calls `UserService.loginUser(LoginDTO)`.
3. `UserService` verifies credentials & generates JWT token.
4. `LoginResponseDTO (token, role)` is returned to the frontend.

📍 Example Flow – When a User Adds an Expense

1. **Frontend (`React`)** sends `ExpenseDTO (amount, categoryId, userId)` to `POST /expenses/add`.
2. `ExpenseController` calls `ExpenseService.createExpense(ExpenseDTO)`.
3. `ExpenseService` converts `ExpenseDTO` into `Expense` entity & saves it.
4. `ExpenseResponseDTO (categoryName, remainingBudget)` is returned to the frontend.

📌 Summary of DTO Layer

Class	Purpose	Key Fields	Used In
UserDTO.java	Transfers user data	`id, username, role`	`UserController, UserService`
LoginDTO.java	Handles login requests	`username, password`	`LoginController, UserService`
LoginResponseDTO.java	Returns JWT token	`token, role`	`LoginController, UserService`
ExpenseDTO.java	Transfers expense data	`amount, categoryId, userId`	`ExpenseController, ExpenseService`
ExpenseResponseDTO.java	Returns expense details	`categoryName, remainingBudget`	`ExpenseController, ExpenseService`
CategoryDTO.java	Transfers category data	`id, name`	`CategoryController, CategoryService`
BudgetDTO.java	Transfers budget data	`amount, startDate, categoryId`	`BudgetController, BudgetService`

📌 Service Layer Overview

📍 **Purpose**:

- Contains **business logic** for handling users, expenses, budgets, and categories.
- Calls the **Repository Layer** to fetch/store data in the database.
- Converts **DTOs to Entities** before saving, and **Entities to DTOs** before returning data.
- Ensures **data validation, security, and exception handling**.

📍 **How It Works**:

1. **Controller Layer** receives a request from the frontend and calls the **Service Layer**.
2. **Service Layer** processes the request and interacts with the **Repository Layer**.
3. **Repository Layer** fetches/stores data in the **database**.
4. **Service Layer** converts the data into DTOs and sends it back to the **Controller**.
5. **Controller Layer** returns the response to the **frontend**.

🔹 1 `IUserService.java` (Interface)

📍 **Purpose**:

- Defines **method signatures** for managing users.
- Implemented by `UserService.java` .

java Copy Edit

```
public interface IUserService { UserDTO createUser(UserDTO userDTO); List<UserDTO> getAllUsers(); List<UserDTO> getUserByRole(String role); UserDTO updateUser(Long id, UserDTO userDTO); void deleteUser(Long id); Object loginUser(LoginDTO loginDTO); }
```

✅ **Declares CRUD methods for user management**.
✅ **Ensures `UserService.java` must implement these methods**.

🔹 2 `UserService.java` (Implements `IUserService`)

📍 **Purpose**:

- Implements **business logic** for user management.
- Hashes passwords using `BCrypt`.
- Uses `JwtUtils` to generate JWT tokens for authentication.

🔍 Code Breakdown

```java
@Service public class UserService implements IUserService {
```

✅ **Marks this as a Spring Service (`@Service`) so it can be injected.**

```java
@Autowired private UserRepo userRepo; @Autowired private JwtUtils jwtUtils; @Autowired private PasswordEncoder passwordEncoder;
```

✅ Injects the **User Repository**, **JWT Utility**, and **Password Encoder**.

🔹 Registering a User (`createUser()`)

```java
@Override @Transactional public UserDTO createUser(UserDTO userDTO) { if (userRepo.findByUsername(userDTO.username()) != null) { throw new RuntimeException("Username already exists!"); } User user = new User(); user.setUsername(userDTO.username()); user.setPassword(passwordEncoder.encode(userDTO.password())); // Hash password user.setRole(User.Role.valueOf(userDTO.role().toUpperCase())); return convertToUserDTO(userRepo.save(user)); }
```

✅ **Checks if the username is already taken.**
✅ **Encrypts the password using `BCryptPasswordEncoder`.**
✅ **Saves the user in the database.**
✅ **Converts the saved entity into a DTO before returning it.**

📍 **Where It's Used:**

- Called by `UserController.java` (`POST /users/add`) when a new user registers.

🔹 Logging in a User (`loginUser()`)

```java
```

```
@Override public Object loginUser(LoginDTO loginDTO) { User user =
userRepo.findByUsername(loginDTO.username()); if (user == null ||
!passwordEncoder.matches(loginDTO.password(), user.getPassword())) { throw new
RuntimeException("Invalid credentials"); } String token = jwtUtils.generateToken(user); return
new LoginResponseDTO(token, user.getRole().name()); }
```

- ✅ Finds user by username.
- ✅ Verifies password using `BCryptPasswordEncoder.matches()`.
- ✅ Generates JWT token using `JwtUtils`.
- ✅ Returns `LoginResponseDTO` containing the token and role.

📍 Where It's Used:

- Called by `LoginController.java` (`POST /login/authenticate`) when a user logs in.

🔹 Getting All Users (`getAllUsers()`)

java

```
@Override public List<UserDTO> getAllUsers() { return userRepo.findAll().stream()
.map(this::convertToUserDTO) .collect(Collectors.toList()); }
```

- ✅ Fetches all users from the database.
- ✅ Converts entities to DTOs before returning.

📍 Where It's Used:

- Called by `UserController.java` (`GET /users/get/all`).

🔹 Updating a User (`updateUser()`)

java

```
@Override @Transactional public UserDTO updateUser(Long id, UserDTO userDTO) { User user =
userRepo.findById(id).orElseThrow(() -> new RuntimeException("User not found"));
user.setUsername(userDTO.username());
user.setPassword(passwordEncoder.encode(userDTO.password())); // Re-encrypt password
user.setRole(User.Role.valueOf(userDTO.role().toUpperCase())); return
convertToUserDTO(userRepo.save(user)); }
```

- ✅ Finds the user by ID.
- ✅ Encrypts new password before saving.
- ✅ Converts entity to DTO before returning.

📍 **Where It's Used:**

- Called by `UserController.java` (`PUT /users/update/{id}`).

🔹 Deleting a User (`deleteUser()`)

java Copy Edit

```
@Override @Transactional public void deleteUser(Long id) { if (!userRepo.existsById(id)) { throw new RuntimeException("User with ID " + id + " not found"); } userRepo.deleteById(id); }
```

✅ **Checks if the user exists** before deleting.
✅ **Deletes the user from the database.**

📍 **Where It's Used:**

- Called by `UserController.java` (`DELETE /users/delete/{id}`).

🔹 Converting Entities to DTOs (`convertToUserDTO()`)

java Copy Edit

```
private UserDTO convertToUserDTO(User user) { return new UserDTO(user.getId(), user.getUsername(), user.getPassword(), user.getRole().name()); }
```

✅ **Prevents exposing the raw User entity.**
✅ **Ensures the frontend receives only required user details.**

📌 How `UserService.java` Interacts with Other Layers

📍 **Example Flow – When a User Registers**

1. **Frontend** (`React`) sends `UserDTO (username, password, role)` to `POST /users/add`.
2. `UserController` calls `UserService.createUser()`.
3. `UserService`:

- Checks for duplicate username.
- Hashes the password.

- Saves the user via `UserRepo.save()`.
- Converts the user entity to `UserDTO`.

4️⃣ **Returns `UserDTO` to the frontend.**

📍 **Example Flow – When a User Logs In**

1️⃣ **Frontend (`React`) sends `LoginDTO (username, password)` to `POST /login/authenticate`.**
2️⃣ `LoginController` calls `UserService.loginUser()`.
3️⃣ `UserService`:

- Fetches user by username via `UserRepo.findByUsername()`.
- Validates password using `BCrypt`.
- Generates JWT token using `JwtUtils.generateToken()`.

4️⃣ **Returns `LoginResponseDTO (token, role)` to the frontend.**

📌 1 IExpenseService.java (Interface)

📍 **Purpose:**

- Defines **method signatures** for managing expenses.
- Implemented by `ExpenseService.java`.

```java
public interface IExpenseService { ExpenseDTO createExpense(ExpenseDTO expenseDTO); List<ExpenseResponseDTO> getAllExpenses(); List<ExpenseResponseDTO> getExpensesByCategory(Long categoryId); void deleteExpense(Long id); }
```

✅ **Declares CRUD methods for expense management.**
✅ **Ensures `ExpenseService.java` must implement these methods.**

📌 2 ExpenseService.java (Implements IExpenseService)

📍 **Purpose:**

- Implements **business logic** for expense management.
- Fetches **category & user before saving expenses.**
- Uses **DTOs for structured data transfer.**

🔍 Code Breakdown

```java
@Service public class ExpenseService implements IExpenseService {
```

✅ **Marks this as a Spring Service (`@Service`).**

```java
@Autowired private ExpenseRepo expenseRepo; @Autowired private CategoryRepo categoryRepo; @Autowired private UserRepo userRepo; @Autowired private BudgetRepo budgetRepo;
```

✅ **Injects repositories to interact with the database.**

🔹 Creating an Expense (`createExpense()`)

```java
@Override @Transactional public ExpenseDTO createExpense(ExpenseDTO expenseDTO) {
```

✅ **Handles adding a new expense.**
✅ **Transactional (** `@Transactional` **) ensures data consistency (rollback if failure).**

```java
User user = userRepo.findById(expenseDTO.userId()) .orElseThrow(() -> new RuntimeException("User not found"));
Category category = categoryRepo.findById(expenseDTO.categoryId()) .orElseThrow(() -> new
RuntimeException("Category not found"));
```

✅ **Finds user & category before saving expense** (prevents invalid expenses).

```java
Expense expense = new Expense(); expense.setAmount(expenseDTO.amount());
expense.setDescription(expenseDTO.description()); expense.setCategory(category); expense.setUser(user);
expense.setDate(expenseDTO.date());
```

✅ **Creates an Expense object from the DTO.**

```java
return convertToExpenseDTO(expenseRepo.save(expense));
```

✅ **Saves expense in the database** and **returns an ExpenseDTO.**

📍 **Where It's Used:**

- Called by `ExpenseController.java` (`POST /expenses/add`).

🔹 Fetching All Expenses (`getAllExpenses()`)

```java
@Override public List<ExpenseResponseDTO> getAllExpenses() { return expenseRepo.findAll().stream()
.map(this::convertToExpenseResponseDTO) .collect(Collectors.toList()); }
```

✅ **Fetches all expenses from the database.**
✅ **Converts each** `Expense` **entity to** `ExpenseResponseDTO`.

📍 **Where It's Used:**

- Called by `ExpenseController.java` (`GET /expenses/get/all`).

🔹 Fetching Expenses by Category (`getExpensesByCategory()`)

```java
@Override public List<ExpenseResponseDTO> getExpensesByCategory(Long categoryId) { return
expenseRepo.findByCategoryId(categoryId).stream() .map(this::convertToExpenseResponseDTO)
.collect(Collectors.toList()); }
```

✅ Fetches all expenses belonging to a specific category.
✅ Converts entities to DTOs before returning.

📍 Where It's Used:

- Called by `ExpenseController.java` (`GET /expenses/get/{categoryId}`).

🔹 Deleting an Expense (`deleteExpense()`)

```java
@Override @Transactional public void deleteExpense(Long id) { if (!expenseRepo.existsById(id)) { throw new RuntimeException("Expense with ID " + id + " not found"); } expenseRepo.deleteById(id); }
```

✅ Checks if the expense exists before deleting.
✅ Deletes the expense from the database.

📍 Where It's Used:

- Called by `ExpenseController.java` (`DELETE /expenses/delete/{id}`).

📌 3 CategoryService.java (Implements ICategoryService)

📍 Purpose:

- Implements **business logic** for managing categories.
- Ensures categories **cannot be deleted if they have expenses**.

🔍 Code Breakdown

```java
@Service public class CategoryService implements ICategoryService {
```

✅ Marks this as a Spring Service (`@Service`).

```java
@Autowired private CategoryRepo categoryRepo; @Autowired private ExpenseRepo expenseRepo;
```

✅ Injects **repositories** to interact with the database.

🔹 Deleting a Category (`deleteCategory()`)

```java
```

```
@Override @Transactional public void deleteCategory(Long id) { if (!categoryRepo.existsById(id)) { throw new RuntimeException("Category not found"); }
```

✅ **Checks if the category exists** before deleting.

java Copy Edit

```
List<Expense> expenses = expenseRepo.findByCategoryId(id); if (!expenses.isEmpty()) { throw new RuntimeException("Cannot delete category with existing expenses"); }
```

✅ **Prevents deletion if the category has linked expenses.**

java Copy Edit

```
categoryRepo.deleteById(id); }
```

✅ **Deletes the category from the database** if no expenses exist.

📍 **Where It's Used:**

- Called by `CategoryController.java` (`DELETE /categories/delete/{id}`).

📌 4️⃣ `BudgetService.java` (Implements `IBudgetService`)

📍 **Purpose:**

- Implements **business logic** for managing budgets.
- Ensures **users do not exceed their budget**.

🔍 Code Breakdown

java Copy Edit

```
@Service public class BudgetService implements IBudgetService {
```

✅ **Marks this as a Spring Service** (`@Service`).

java Copy Edit

```
@Autowired private BudgetRepo budgetRepo; @Autowired private ExpenseRepo expenseRepo; @Autowired private CategoryRepo categoryRepo;
```

✅ Injects **repositories** to interact with the database.

🔹 Checking Remaining Budget (`getBudgetByCategory()`)

java Copy Edit

```
@Override public String getBudgetByCategory(Long categoryId, LocalDate date) { Optional<Budget> budgetOpt = budgetRepo.findBudgetByCategoryAndDate(categoryId, date); if (budgetOpt.isEmpty()) { return "Budget is not set for this category."; }
```

✅ **Finds the budget for a category on a given date.**
✅ **If no budget exists, returns** `"Budget is not set for this category."` .

java Copy Edit

```
Budget budget = budgetOpt.get(); double totalExpenses = expenseRepo.findByCategoryIdAndDateRange( categoryId, budget.getStartDate(), budget.getEndDate()) .stream() .mapToDouble(Expense::getAmount) .sum();
```

✅ **Calculates total expenses within the budget period.**

java Copy Edit

```
double remainingBudget = budget.getAmount() - totalExpenses; return remainingBudget >= 0 ? String.valueOf(remainingBudget) : "Budget exceeded!"; }
```

✅ **Returns the remaining budget or** `"Budget exceeded!"` if overspending has occurred.

📍 **Where It's Used:**

- Called by `BudgetController.java` (`GET /budgets/get/{categoryId}`).

1️⃣ `UserController.java` – Managing Users

📍 **Purpose:**

- Handles **user registration, fetching users, updating, and deleting users.**
- Calls `UserService.java` for business logic.
- Uses **DTOs (`UserDTO`)** to transfer data between frontend and backend.

🔍 Code Breakdown

```java
@RestController @RequestMapping("/users") public class UserController {
```

✅ **Marks this as a REST controller (`@RestController`).**
✅ **Base URL for all endpoints:** `/users`.

```java
@Autowired private IUserService userService;
```

✅ Injects `UserService.java` to call business logic.

🔹 Registering a User (`POST /users/add`)

```java
@PostMapping("/add") public ResponseEntity<UserDTO> createUser(@Valid @RequestBody UserDTO userDTO) { UserDTO user = userService.createUser(userDTO); return new ResponseEntity<>(user, HttpStatus.CREATED); }
```

✅ **Receives a `UserDTO` from the frontend** (React).
✅ **Calls `UserService.createUser(userDTO)` to register a new user.**
✅ **Returns `201 CREATED` with the saved user.**

📍 **Where It's Used:**

- Called when a new user registers **from the frontend (`React`).**

🔹 Fetching All Users (`GET /users/get/all`)

```java
@GetMapping("/get/all") public ResponseEntity<List<UserDTO>> getAllUsers() { List<UserDTO> users = userService.getAllUsers(); return new ResponseEntity<>(users, HttpStatus.OK); }
```

✅ **Calls `UserService.getAllUsers()` to fetch all users.**
✅ **Returns `200 OK` with the list of users.**

📍 **Where It's Used:**

- Used in **Admin Dashboard to view all users**.

🔹 Fetching Users by Role (`GET /users/get/{role}`)

java Copy Edit

```
@GetMapping("/get/{role}") public ResponseEntity<List<UserDTO>> getUserByRole(@PathVariable String role) { List<UserDTO> users = userService.getUserByRole(role); return new ResponseEntity<>(users, HttpStatus.OK); }
```

✅ **Fetches users based on their role** (`ADMIN` or `USER`).

📍 **Where It's Used:**

- Used when an admin wants to filter users based on roles.

🔹 Updating a User (`PUT /users/update/{id}`)

java Copy Edit

```
@PutMapping("/update/{id}") public ResponseEntity<UserDTO> updateUser(@PathVariable Long id, @Valid @RequestBody UserDTO userDTO) { UserDTO user = userService.updateUser(id, userDTO); return new ResponseEntity<>(user, HttpStatus.OK); }
```

✅ **Calls** `UserService.updateUser(id, userDTO)` **to update a user's details**.

📍 **Where It's Used:**

- Used when a user updates their **profile information**.

🔹 Deleting a User (`DELETE /users/delete/{id}`)

java Copy Edit

```
@DeleteMapping("/delete/{id}") public ResponseEntity<String> deleteUser(@PathVariable Long id) { userService.deleteUser(id); return new ResponseEntity<>("User deleted successfully", HttpStatus.OK); }
```

✅ **Calls** `UserService.deleteUser(id)` **to remove the user**.
✅ **Returns** `200 OK` **with a success message**.

📍 **Where It's Used:**

- Used by **Admin Dashboard** to remove users.

🔹 2️⃣ `ExpenseController.java` – Managing Expenses

📍 **Purpose:**

- Handles **adding, fetching, and deleting expenses**.

- Calls `ExpenseService.java` for business logic.

🔹 Adding an Expense (`POST /expenses/add`)

java Copy Edit

```
@PostMapping("/add") public ResponseEntity<ExpenseDTO> createExpense(@Valid @RequestBody
ExpenseDTO expenseDTO) { ExpenseDTO expense = expenseService.createExpense(expenseDTO);
return new ResponseEntity<>(expense, HttpStatus.CREATED); }
```

✅ Receives an `ExpenseDTO` from the frontend.
✅ Calls `ExpenseService.createExpense(expenseDTO)` to save it.
✅ Returns `201 CREATED` with the saved expense.

📍 Where It's Used:

- Called when a user **adds a new expense** from the frontend.

🔹 Fetching All Expenses (`GET /expenses/get/all`)

java Copy Edit

```
@GetMapping("/get/all") public ResponseEntity<List<ExpenseResponseDTO>> getAllExpenses() {
List<ExpenseResponseDTO> expenses = expenseService.getAllExpenses(); return new
ResponseEntity<>(expenses, HttpStatus.OK); }
```

✅ Fetches all expenses from the database.

📍 Where It's Used:

- Used in the **dashboard to display all expenses**.

🔹 Fetching Expenses by Category (`GET /expenses/get/{categoryId}`)

java Copy Edit

```
@GetMapping("/get/{categoryId}") public ResponseEntity<List<ExpenseResponseDTO>>
getExpensesByCategory(@PathVariable Long categoryId) { List<ExpenseResponseDTO> expenses =
expenseService.getExpensesByCategory(categoryId); return new ResponseEntity<>(expenses,
HttpStatus.OK); }
```

✅ Filters expenses based on category.

📍 Where It's Used:

- Used in the **expense breakdown page**.

🔹 Deleting an Expense (`DELETE /expenses/delete/{id}`)

java Copy Edit

```
@DeleteMapping("/delete/{id}") public ResponseEntity<String> deleteExpense(@PathVariable
Long id) { expenseService.deleteExpense(id); return new ResponseEntity<>("Expense deleted
successfully", HttpStatus.OK); }
```

✅ **Calls** `ExpenseService.deleteExpense(id)` **to remove an expense.**

📍 **Where It's Used:**

- Allows users to **delete an expense entry.**

📌 Summary of Controller Layer

Controller	Purpose	Endpoints
UserController.java	Manages users	`/users/add`, `/users/get/all`, `/users/update/{id}`, `/users/delete/{id}`
ExpenseController.java	Manages expenses	`/expenses/add`, `/expenses/get/all`, `/expenses/get/{categoryId}`, `/expenses/delete/{id}`

📌 Next Steps

✅ Now you **fully understand the Controller Layer!** 🚀
Would you like me to:
1️⃣ **Continue with Security & Config Layer (JWT authentication, authorization)?**
2️⃣ **Clarify anything in the Controller Layer before moving on?** 🔍

no controller has not been fully explained only first 2 clases has been explained

Memory updated

You're absolutely right! We've only covered **UserController.java** and **ExpenseController.java** so far. Now, let's **deep dive into the remaining controllers**:

✅ `CategoryController.java` – Manages expense categories (CRUD operations).
✅ `BudgetController.java` – Handles budget assignments and tracking.
✅ `LoginController.java` – Handles user authentication and JWT token generation.

I'll go **line by line**, explaining:

- **Purpose of each method**
- **How it interacts with services & repositories**
- **Data flow from request to response**

🔹 3️⃣ `CategoryController.java` – Managing Expense Categories

📍 **Purpose:**

- Handles **adding, fetching, and deleting categories**.
- Calls `CategoryService.java` for business logic.

🔹 Code Breakdown

```java
@RestController @RequestMapping("/categories") public class CategoryController {
```

✅ **Marks this class as a REST controller (** `@RestController` **).**
✅ **Base URL for all category-related requests:** `/categories` .

```java
@Autowired private ICategoryService categoryService;
```

✅ Injects `CategoryService.java` to process category-related operations.

🔹 Adding a Category (`POST /categories/add`)

```java
@PostMapping("/add") public ResponseEntity<CategoryDTO> createCategory(@Valid @RequestBody CategoryDTO categoryDTO) { CategoryDTO category = categoryService.createCategory(categoryDTO); return new ResponseEntity<>(category, HttpStatus.CREATED); }
```

✅ **Receives a** `CategoryDTO` **from the frontend.**
✅ **Calls** `CategoryService.createCategory(categoryDTO)` **to save it.**
✅ **Returns** `201 CREATED` **with the saved category.**

📍 **Where It's Used:**

- Used in **Admin Dashboard** to add a new category.

🔹 Fetching All Categories (`GET /categories/get/all`)

```java
@GetMapping("/get/all") public ResponseEntity<List<CategoryDTO>> getAllCategories() { List<CategoryDTO> categories = categoryService.getAllCategories(); return new ResponseEntity<>(categories, HttpStatus.OK); }
```

✅ **Calls** `CategoryService.getAllCategories()` **to fetch all categories.**
✅ **Returns** `200 OK` **with a list of categories.**

📍 **Where It's Used:**

- Used when displaying **expense categories** in dropdown lists.

🔹 Deleting a Category (`DELETE /categories/delete/{id}`)

```java
@DeleteMapping("/delete/{id}") public ResponseEntity<String> deleteCategory(@PathVariable Long id) { categoryService.deleteCategory(id); return new ResponseEntity<>("Category deleted successfully", HttpStatus.OK); }
```

✅ **Calls** `CategoryService.deleteCategory(id)` **to remove a category.**
✅ **Returns** `200 OK` **with a success message.**

📍 **Where It's Used:**

- Used in **Admin Dashboard** to remove unused categories.

🔹 4️⃣ `BudgetController.java` – Managing Budgets

📍 **Purpose:**

- Handles **assigning budgets to categories.**
- Checks **remaining budget after expenses.**
- Calls `BudgetService.java` for budget logic.

🔹 Code Breakdown

```java
@RestController @RequestMapping("/budgets") public class BudgetController {
```

✅ **Marks this class as a REST controller (** `@RestController` **).**
✅ **Base URL for all budget-related requests:** `/budgets` .

```java
@Autowired private IBudgetService budgetService;
```

✅ Injects `BudgetService.java` for budget management.

🔹 Assigning a Budget (`POST /budgets/add`)

```java
@PostMapping("/add") public ResponseEntity<BudgetDTO> createBudget(@Valid @RequestBody BudgetDTO budgetDTO) { BudgetDTO budget = budgetService.createBudget(budgetDTO); return new ResponseEntity<>(budget, HttpStatus.CREATED); }
```

✅ **Receives a** `BudgetDTO` **from the frontend.**
✅ **Calls** `BudgetService.createBudget(budgetDTO)` **to save it.**
✅ **Returns** `201 CREATED` **with the saved budget.**

📍 **Where It's Used:**

- Used in **Admin Dashboard** to set category budgets.

🔹 Fetching a Budget by Category (`GET /budgets/get/{categoryId}`)

```java
@GetMapping("/get/{categoryId}") public ResponseEntity<String>
getBudgetByCategory(@PathVariable Long categoryId) { String remainingBudget =
budgetService.getBudgetByCategory(categoryId, LocalDate.now()); return new ResponseEntity<>
(remainingBudget, HttpStatus.OK); }
```

✅ **Calls** `BudgetService.getBudgetByCategory(categoryId)` **to check remaining budget.**
✅ **Returns** `200 OK` **with the remaining budget or "Budget exceeded!".**

📍 **Where It's Used:**

- Used to **track overspending** in categories.

🔹 5 `LoginController.java` – Handling User Authentication

📍 **Purpose:**

- Handles **user authentication (login).**
- Generates **JWT tokens** after successful login.
- Calls `UserService.java` for authentication logic.

🔹 Code Breakdown

```java
@RestController @RequestMapping("/login") public class LoginController {
```

✅ **Marks this as a REST controller (** `@RestController` **).**
✅ **Base URL for login requests:** `/login` **.**

```java
@Autowired private IUserService userService;
```

✅ **Injects** `UserService.java` **for authentication.**

🔹 Authenticating a User (`POST /login/authenticate`)

```java
```

```
@PostMapping("/authenticate") public ResponseEntity<Object> loginUser(@RequestBody LoginDTO loginDTO) { return new ResponseEntity<>(userService.loginUser(loginDTO), HttpStatus.OK); }
```

✅ Receives a `LoginDTO` (username & password) from the frontend.
✅ Calls `UserService.loginUser(loginDTO)` for authentication.
✅ Returns `200 OK` with `LoginResponseDTO (JWT token & role)`.

📍 Where It's Used:

- Used in **Login Page** to authenticate users.

📌 Summary of Controller Layer

Controller	Purpose	Endpoints
UserController.java	Manages users	`/users/add`, `/users/get/all`, `/users/update/{id}`, `/users/delete/{id}`
ExpenseController.java	Manages expenses	`/expenses/add`, `/expenses/get/all`, `/expenses/get/{categoryId}`, `/expenses/delete`
CategoryController.java	Manages categories	`/categories/add`, `/categories/get/all`, `/categories/delete/{id}`
BudgetController.java	Manages budgets	`/budgets/add`, `/budgets/get/{categoryId}`
LoginController.java	Handles login & authentication	`/login/authenticate`

📌 How the Controller Layer Interacts with Other Layers

📍 Example Flow – When a User Logs In
1️⃣ Frontend (`React`) sends `LoginDTO (username, password)` to `POST /login/authenticate`.
2️⃣ `LoginController` calls `UserService.loginUser(loginDTO)`.
3️⃣ `UserService`:

- Fetches user by username via `UserRepo.findByUsername()`.
- Verifies password using `BCrypt`.
- Generates JWT token using `JwtUtils.generateToken()`.

4️⃣ Returns `LoginResponseDTO (token, role)` to the frontend.

📍 Example Flow – When a User Adds an Expense
1️⃣ Frontend (`React`) sends `ExpenseDTO (amount, categoryId, userId)` to `POST /expenses/add`.
2️⃣ `ExpenseController` calls `ExpenseService.createExpense(expenseDTO)`.
3️⃣ `ExpenseService`:

- Fetches category from `CategoryRepo`.
- Fetches user from `UserRepo`.

- Saves expense via `ExpenseRepo.save()`.
  4. **Returns `ExpenseDTO` to the frontend.**

🔹 1️⃣ `CustomUserDetails.java` – Wrapping User Data for Spring Security

📌 **Purpose:**

- **Implements** `UserDetails` (used by Spring Security).
- Provides **user authentication data** (username, password, roles).

🔍 Code Breakdown

```java
public class CustomUserDetails implements UserDetails {
```

✅ **Implements** `UserDetails`, which is required by Spring Security.

```java
private User user; public CustomUserDetails(User user) { this.user = user; }
```

✅ Stores the **User entity** inside this class.

```java
@Override public Collection<? extends GrantedAuthority> getAuthorities() { return
Collections.singletonList(new SimpleGrantedAuthority(user.getRole().name())); }
```

✅ **Returns user roles (** `ADMIN` **or** `USER` **)** so Spring Security can check permissions.

```java
@Override public String getPassword() { return user.getPassword(); } @Override public
String getUsername() { return user.getUsername(); }
```

✅ **Returns username & password** for authentication.

📌 **Where It's Used:**

- Used in `CustomUserDetailsService.java` to authenticate users.

🔹 2️⃣ CustomUserDetailsService.java – Fetching User from Database

📍 **Purpose:**

- **Loads user details from the database** for authentication.
- Used by **Spring Security's authentication manager**.

🔍 Code Breakdown

```java
@Service public class CustomUserDetailsService implements UserDetailsService {
```

✅ **Implements** `UserDetailsService` , which is required by Spring Security.

```java
@Autowired private UserRepo userRepo;
```

✅ Injects `UserRepo` **to fetch user details** from the database.

```java
@Override public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException { User user = userRepo.findByUsername(username); if (user == null) { throw new UsernameNotFoundException("User not found: " + username); } return new CustomUserDetails(user); }
```

✅ **Finds user by username** in the database.
✅ **Throws an exception if the user is not found.**
✅ **Wraps the user inside** `CustomUserDetails` **and returns it.**

📍 **Where It's Used:**

- Used by **Spring Security's authentication manager**.

🔹 3 JwtAuthFilter.java – Intercepting Requests to Validate JWT

📍 **Purpose:**

- **Intercepts incoming HTTP requests** to check for JWT tokens.
- **Extracts user details from JWT** and sets authentication context.

🔍 Code Breakdown

java Copy Edit

```
@Component public class JwtAuthFilter extends OncePerRequestFilter {
```

✅ **Extends** `OncePerRequestFilter` , meaning it runs once per request.

java Copy Edit

```
@Autowired private JwtUtils jwtUtils; @Autowired private CustomUserDetailsService
customUserDetailsService;
```

✅ Injects **JWT utility** & **CustomUserDetailsService** to validate users.

🔹 Extracting & Validating JWT

java Copy Edit

```
@Override protected void doFilterInternal(HttpServletRequest request, HttpServletResponse
response, FilterChain filterChain) throws ServletException, IOException { String token =
getTokenFromRequest(request); if (token != null){ String username =
jwtUtils.getUsernameFromToken(token); UserDetails userDetails =
customUserDetailsService.loadUserByUsername(username); if (StringUtils.hasText(username)
&& jwtUtils.isTokenValid(token, userDetails)){
```

✅ **Extracts JWT from** `Authorization` **header.**
✅ **Decodes JWT to get the username.**

✅ **Loads user details from the database.**
✅ **If valid, sets authentication context** so the user is authenticated.

```java
SecurityContextHolder.getContext().setAuthentication(authenticationToken);
```

✅ **Spring Security now recognizes the user as authenticated.**

📍 **Where It's Used:**

- Automatically applied to **all API requests** using `SecurityConfig.java`.

🔹 4️⃣ `JwtUtils.java` – Generating & Validating JWT Tokens

📍 **Purpose:**

- **Generates JWT tokens** for authenticated users.
- **Extracts & validates JWT tokens** when requests are made.

🔍 Code Breakdown

```java
@Service public class JwtUtils {
```

✅ **Marks this class as a Spring Service.**

```java
@Value("${secreteJwtString}") private String secreteJwtString;
```

✅ **Loads secret key from** `application.properties` to sign JWT tokens.

```java
```

```
public String generateToken(User user){ return Jwts.builder()
.subject(user.getUsername()) .issuedAt(new Date(System.currentTimeMillis()))
.expiration(new Date(System.currentTimeMillis() + EXPIRATION_TIME_IN_MILLISEC))
.signWith(key) .compact(); }
```

✅ Creates a JWT token with:

- Username (`subject`)
- Expiration time
- Signed using a secret key

java Copy Edit

```
public String getUsernameFromToken(String token){ return extractClaims(token,
Claims::getSubject); }
```

✅ Extracts the username from a given JWT token.

📍 Where It's Used:

- Used in `JwtAuthFilter.java` to authenticate API requests.

🔹 5 `SecurityConfig.java` – Configuring Spring Security

📍 Purpose:

- Defines which endpoints are secured/public.
- Integrates JWT authentication.

🔍 Code Breakdown

java Copy Edit

```
@Configuration @EnableWebSecurity @EnableMethodSecurity public class SecurityConfig {
```

✅ Enables Spring Security.

```java
@Bean public PasswordEncoder passwordEncoder() { return new BCryptPasswordEncoder(); }
```

✅ Ensures passwords are securely hashed with `BCrypt`.

🔹 Defining Security Rules

```java
@Bean SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
return http .csrf((csrf) -> csrf.disable()) .authorizeHttpRequests(authorize -> authorize
.requestMatchers("/login/**", "/users/add").permitAll() .anyRequest().authenticated())
.addFilterBefore(jwtAuthFilter, UsernamePasswordAuthenticationFilter.class) .build(); }
```

✅ Disables CSRF (not needed for JWT).
✅ Allows public access to `/login` & `/users/add` (all other endpoints require authentication).
✅ Adds `JwtAuthFilter` before processing user authentication.

📌 Summary of Security & Config Layer

Class	Purpose	Key Functionality
CustomUserDetails.java	Wraps user data for Spring Security	Returns username, password, and roles
CustomUserDetailsService.java	Loads user from database	Used by authentication manager
JwtAuthFilter.java	Extracts & validates JWT token	Intercepts HTTP requests
JwtUtils.java	Generates & validates JWT tokens	Handles token creation & expiration
SecurityConfig.java	Configures Spring Security	Defines access rules

📌 Overview of the Full Project Flow

What Happens When a User Logs In and Uses the Application?

1️⃣ **User logs in** (`POST /login/authenticate`) → Gets a JWT token.
2️⃣ **Frontend stores JWT** and sends it with every API request.
3️⃣ **Backend validates JWT** before allowing access.
4️⃣ **Protected APIs fetch, save, update, or delete data.**
5️⃣ **Response is sent back to the frontend** for display.

🔹 Step 1: User Logs In (`POST /login/authenticate`)

📍 **What happens?**

- The user enters their **username and password**.
- `LoginController.java` receives the request.
- Calls `UserService.java` to validate credentials.

🔍 Request from Frontend

json Copy Edit

```
{ "username": "john_doe", "password": "securepassword" }
```

🔍 `LoginController.java`

java Copy Edit

```
@PostMapping("/authenticate") public ResponseEntity<Object> loginUser(@RequestBody LoginDTO loginDTO) {
return new ResponseEntity<>(userService.loginUser(loginDTO), HttpStatus.OK); }
```

✅ Calls `UserService.loginUser(loginDTO)` to handle login.

🔍 UserService.java

java Copy Edit

```
@Override public Object loginUser(LoginDTO loginDTO) { User user =
userRepo.findByUsername(loginDTO.username()); if (user == null ||
!passwordEncoder.matches(loginDTO.password(), user.getPassword())) { throw new RuntimeException("Invalid
credentials"); } String token = jwtUtils.generateToken(user); return new LoginResponseDTO(token,
user.getRole().name()); }
```

✅ **Finds user by username** in `UserRepo`.
✅ **Verifies password using** `BCryptPasswordEncoder.matches()`.
✅ **Generates JWT token** using `JwtUtils.generateToken(user)`.

🔍 JwtUtils.java – Generating JWT

java Copy Edit

```
public String generateToken(User user) { return Jwts.builder() .subject(user.getUsername())
.issuedAt(new Date(System.currentTimeMillis())) .expiration(new Date(System.currentTimeMillis() +
EXPIRATION_TIME_IN_MILLISEC)) .signWith(key) .compact(); }
```

✅ **Creates a JWT token with:**

- **Username** (`subject`)
- **Expiration time**
- **Signed using a secret key**

🔍 Response Sent to Frontend

json Copy Edit

```
{ "token": "eyJhbGciOiJIUzI1...", "role": "USER" }
```

✅ **Frontend stores this JWT token** and includes it in all future API requests.

🔹 Step 2: Frontend Makes an API Request with JWT Token

📍 Example: Fetching Expenses (`GET /expenses/get/all`)

🔍 Request from Frontend

http Copy Edit

```
GET /expenses/get/all Authorization: Bearer eyJhbGciOiJIUzI1...
```

✅ JWT token is included in the `Authorization` header.

🔍 `JwtAuthFilter.java` – Validating JWT

📍 Before Spring Security processes the request, `JwtAuthFilter` runs.

java Copy Edit

```
@Override protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response,
FilterChain filterChain) throws ServletException, IOException { String token =
getTokenFromRequest(request); if (token != null){ String username =
jwtUtils.getUsernameFromToken(token); UserDetails userDetails =
customUserDetailsService.loadUserByUsername(username); if (jwtUtils.isTokenValid(token, userDetails)){
SecurityContextHolder.getContext().setAuthentication( new
UsernamePasswordAuthenticationToken(userDetails, null, userDetails.getAuthorities()) ); } }
filterChain.doFilter(request, response); }
```

✅ Extracts JWT from the request header.
✅ Validates token using `JwtUtils`.
✅ If valid, Spring Security allows access to protected APIs.

🔍 `ExpenseController.java` – Fetching Expenses

java Copy Edit

```
@GetMapping("/get/all") public ResponseEntity<List<ExpenseResponseDTO>> getAllExpenses() {
List<ExpenseResponseDTO> expenses = expenseService.getAllExpenses(); return new ResponseEntity<>
(expenses, HttpStatus.OK); }
```

✅ Calls `ExpenseService.getAllExpenses()` to fetch data.

🔍 ExpenseService.java

```java
@Override public List<ExpenseResponseDTO> getAllExpenses() { return expenseRepo.findAll().stream()
.map(this::convertToExpenseResponseDTO) .collect(Collectors.toList()); }
```

✅ Fetches all expenses from the database using `ExpenseRepo.findAll()`.
✅ Converts `Expense` entity to `ExpenseResponseDTO` for the frontend.

🔍 ExpenseRepo.java

```java
public interface ExpenseRepo extends JpaRepository<Expense, Long> { List<Expense> findByCategoryId(Long
id); }
```

✅ Runs SQL query:

```sql
SELECT * FROM expenses;
```

🔍 Response Sent to Frontend

```json
[ { "id": 1, "description": "Lunch", "amount": 12.50, "date": "2024-02-08", "categoryName": "Food",
"userId": 2, "remainingBudget": "87.50" } ]
```

✅ Frontend receives and displays expense data.

🔹 Step 3: User Accesses a Protected Admin API (`DELETE /users/delete/{id}`)

📍 Admin tries to delete a user

🔍 Request from Frontend

http Copy Edit

```
DELETE /users/delete/5 Authorization: Bearer eyJhbGciOiJIUzI1...
```

🔍 `SecurityConfig.java` – Restricting Access

java Copy Edit

```
@Bean SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception { return http
.csrf(csrf -> csrf.disable()) .authorizeHttpRequests(authorize -> authorize
.requestMatchers("/login/**", "/users/add").permitAll() .requestMatchers(HttpMethod.DELETE,
"/users/delete/**").hasAuthority("ADMIN") .anyRequest().authenticated()) .addFilterBefore(jwtAuthFilter,
UsernamePasswordAuthenticationFilter.class) .build(); }
```

✅ Only `ADMIN` users can delete users.
✅ If a normal user tries, access is denied (`403 Forbidden`).

🔍 Response Sent to Frontend (If Unauthorized)

json Copy Edit

```
{ "error": "Access Denied", "message": "You do not have permission to access this resource" }
```

✅ User is blocked from accessing admin-only routes.

📌 Summary of the Full Project Flow

Step	Process	Components Involved
1️⃣ User Logs In	User enters credentials → Backend verifies → JWT token is returned	`LoginController` , `UserService` , `JwtUtils`

Step	Process	Components Involved
2 Frontend Stores JWT	Token is saved in browser local storage & used in API calls	Frontend React App
3 User Makes an API Request	JWT is sent with API request → `JwtAuthFilter` validates → Backend processes request	`JwtAuthFilter` , `SecurityConfig` , `ExpenseController` , `ExpenseService` , `ExpenseRepo`
4 Protected Admin Route	If user lacks `ADMIN` role, access is denied (`403 Forbidden`)	`SecurityConfig` , `JwtAuthFilter`