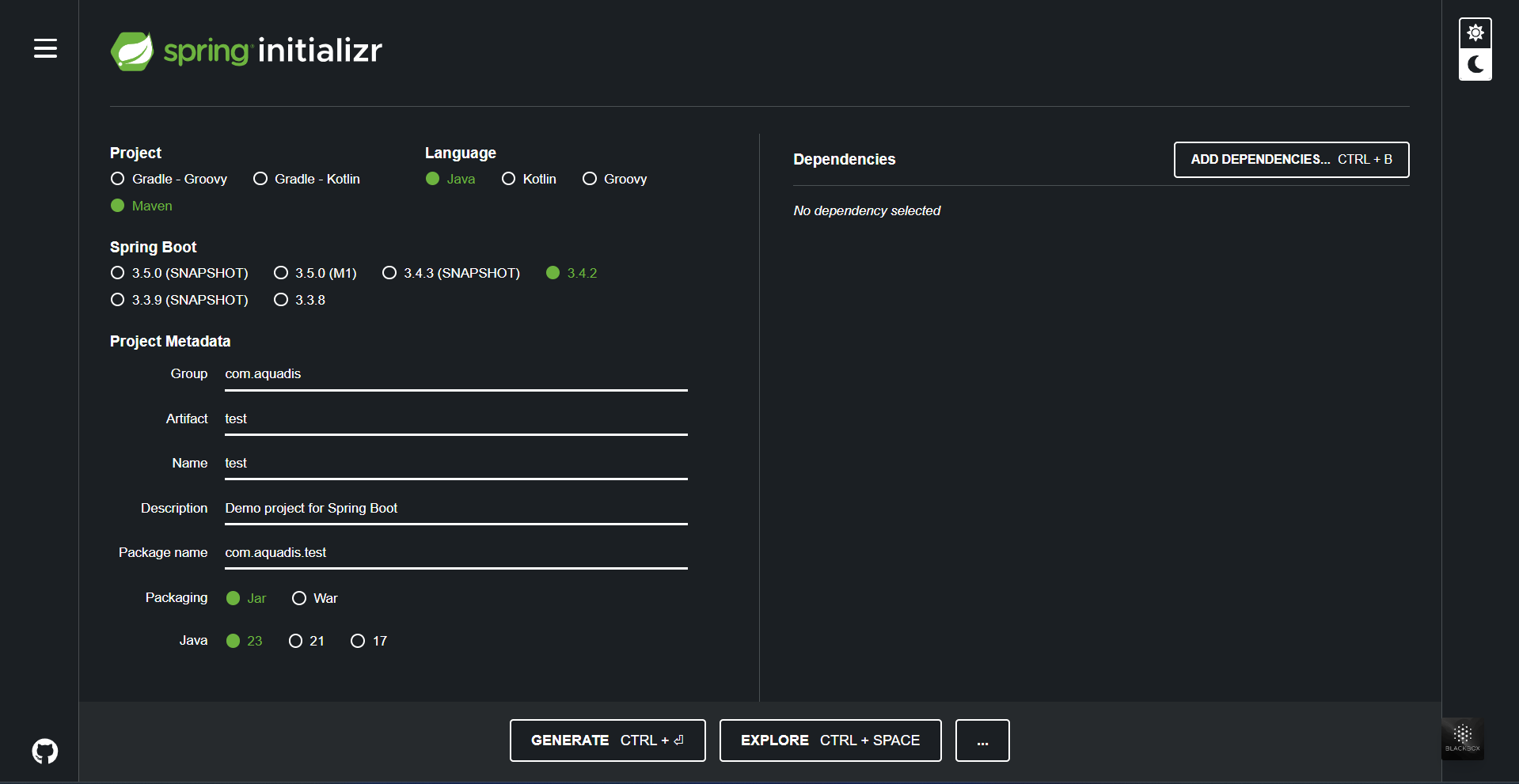
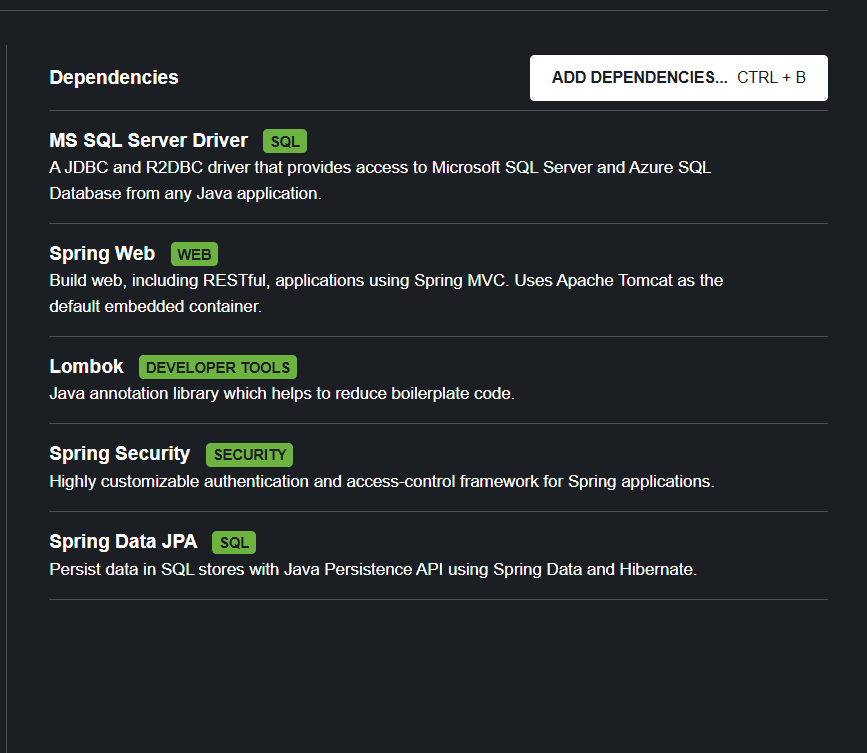
Aquadis Spring Boot + Angular Test Steps

# Spring and DB Initialization:

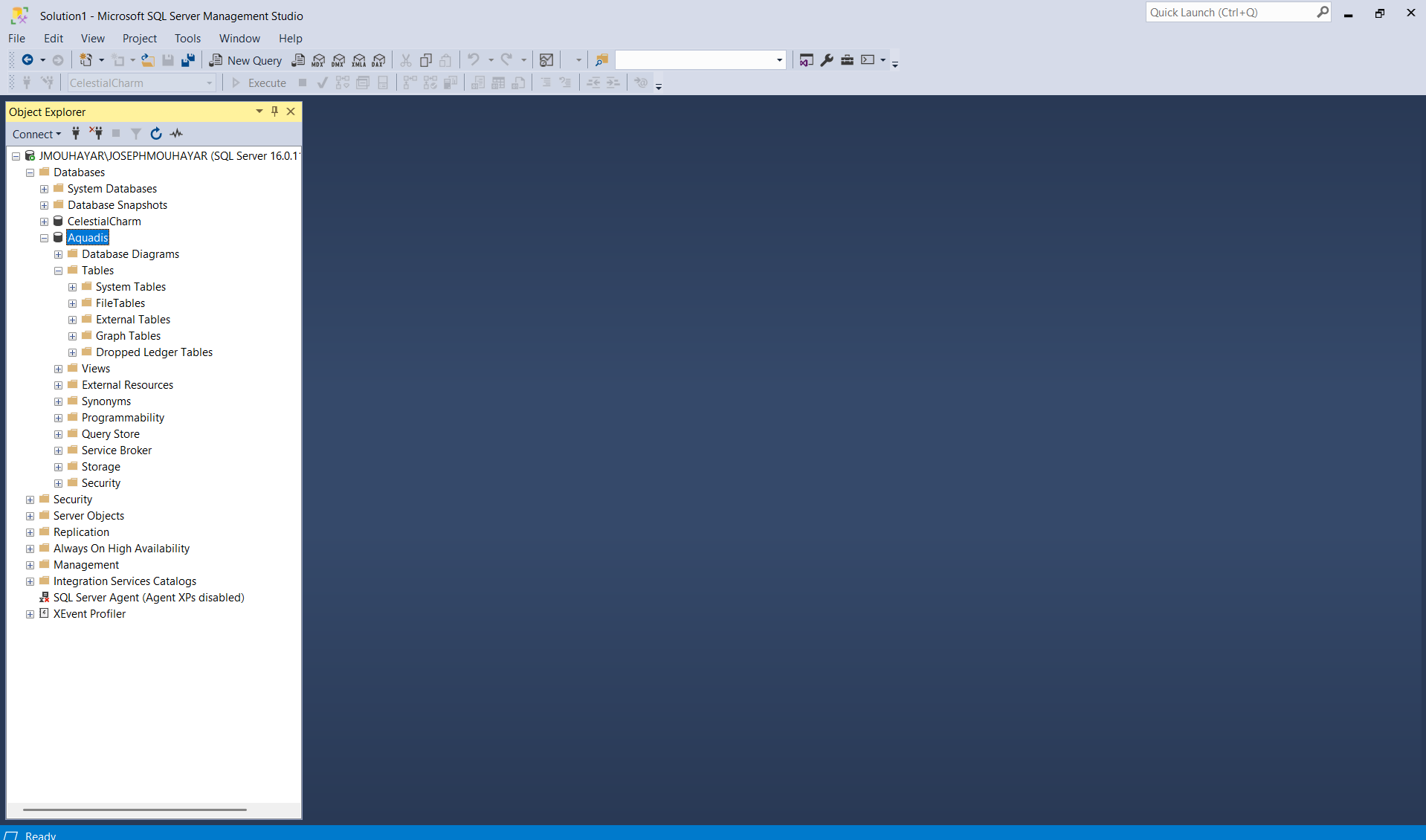
* + Create a spring project in spring initializr



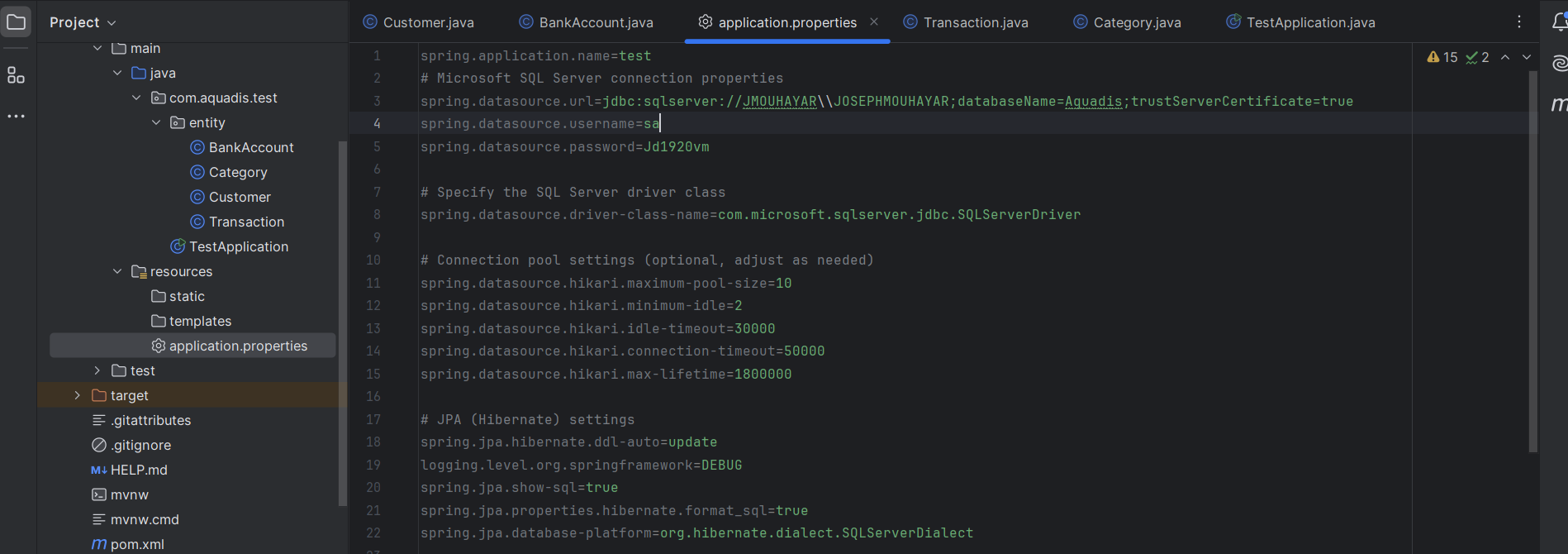
* + Add Dependencies:
    - MS SQL Server Driver (to connect to Microsoft SQL Server)
    - Spring Web (to build restful APIs)
    - Lombok (to create getters and setters)
    - Spring Security (for authentication and security against threats)
    - Spring Data JPA (to allow easy interaction with relational databases using repository interfaces) (similar to entity framework in .NET)



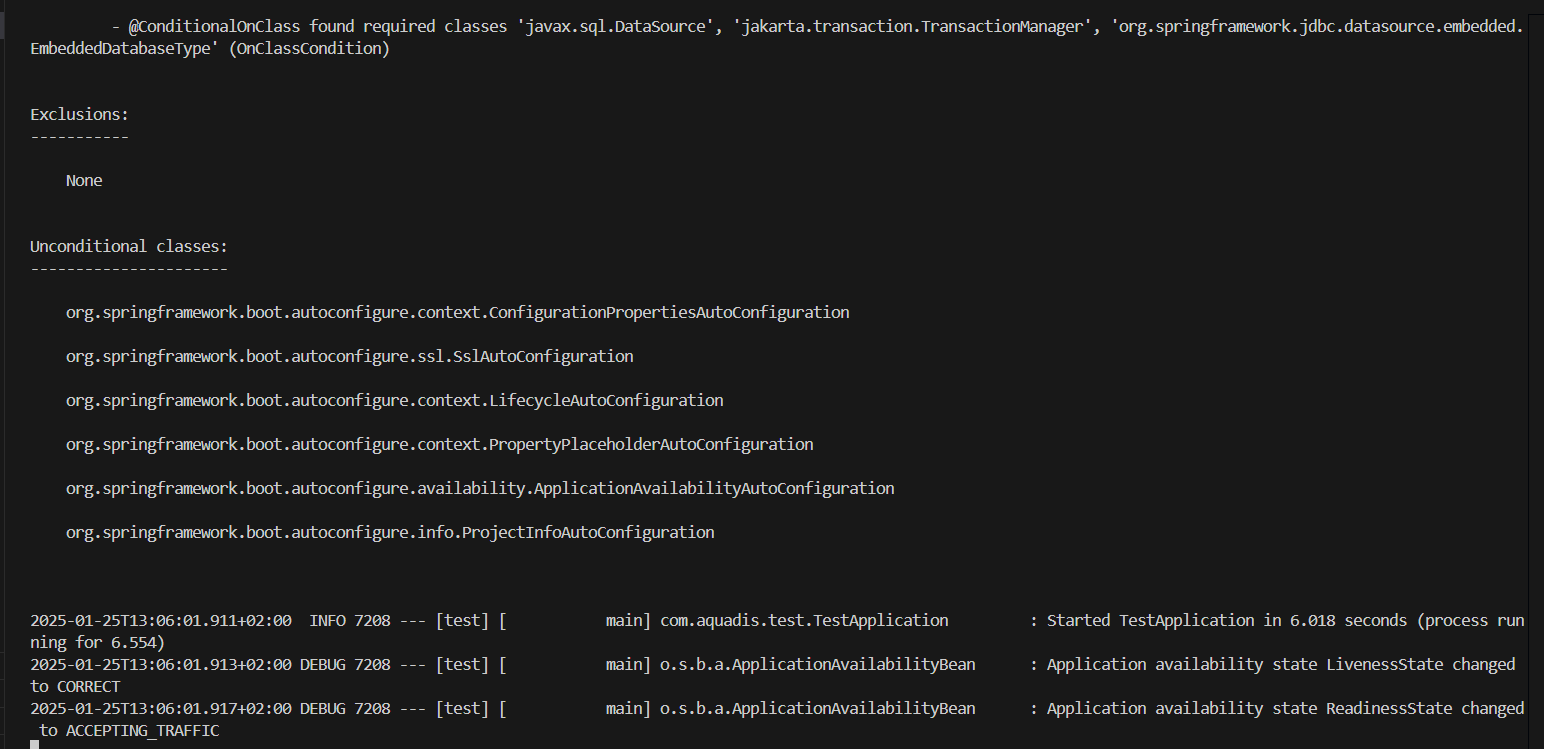
* + Create a new Database
    - Create a new Database in SSMS (Aquadis)



* + - Link the SQL Server database in my spring project (by updating the application.properties)

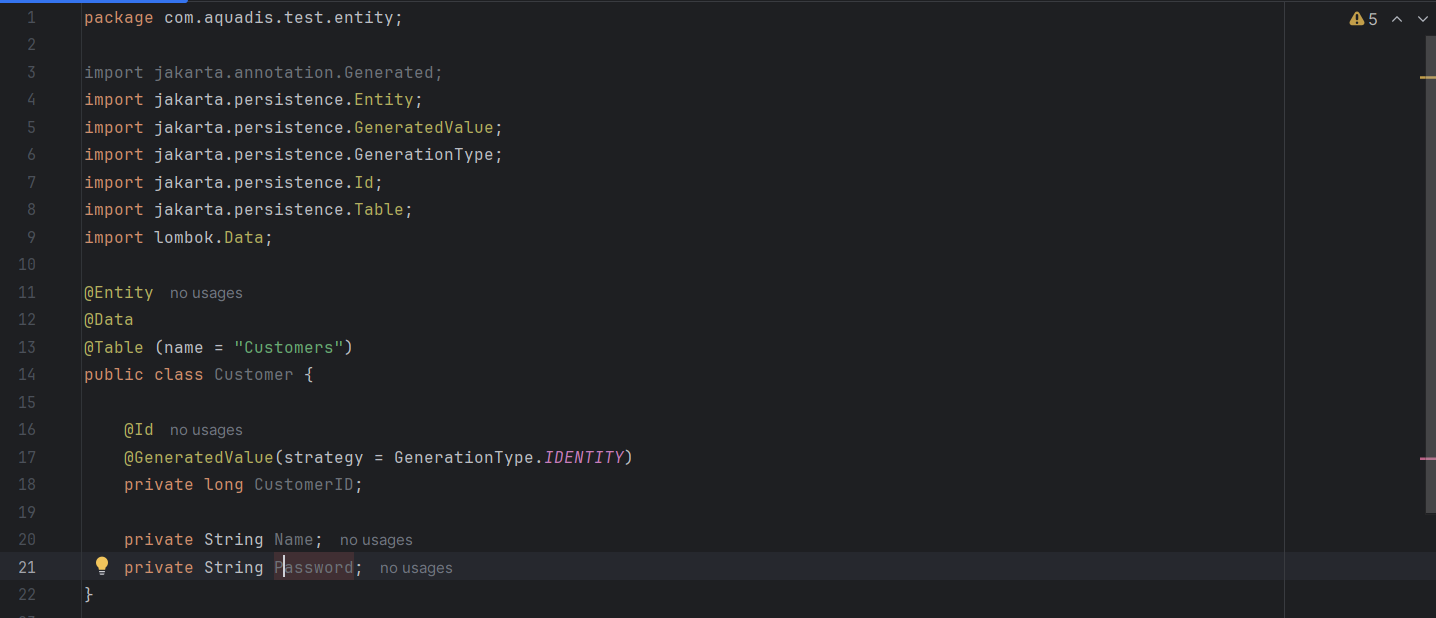


* + - Check connection by running [mvn clean spring-boot:run]

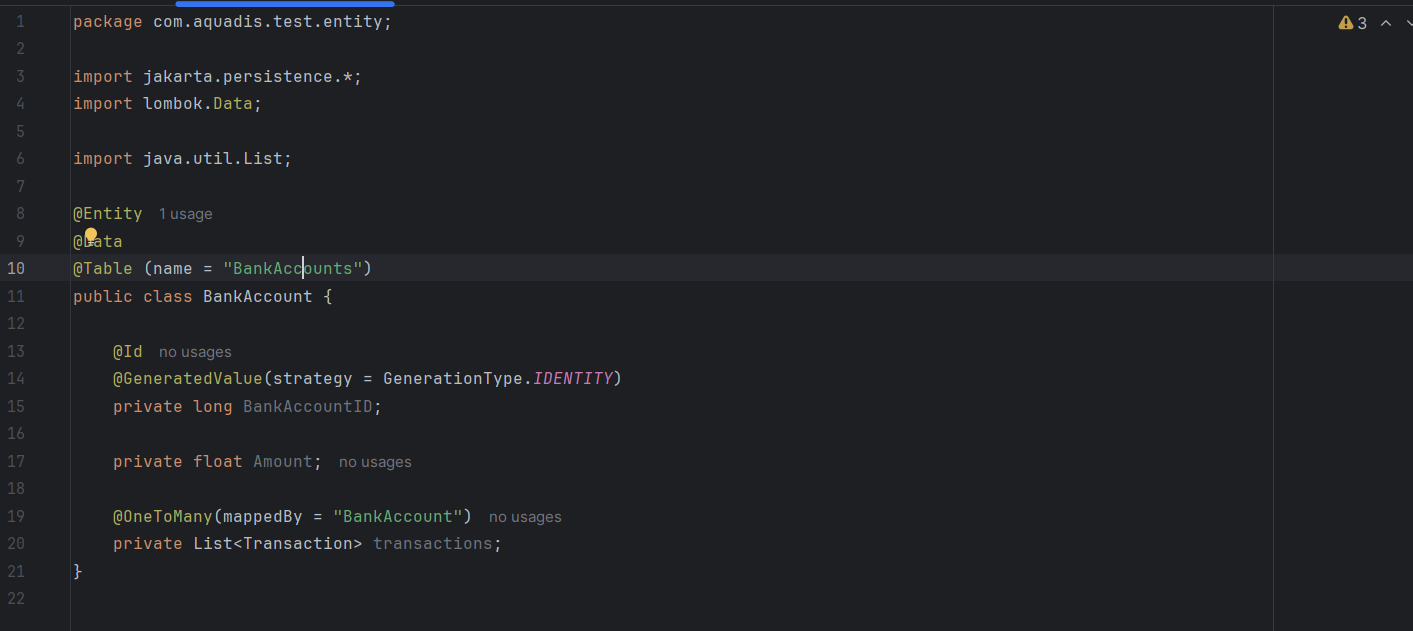


# Database and Entity Creation:

* + Create the database structure using code first JPA entities:
    - Customer: CustomerID (PK), Name, Password



* + - BankAccount: BankAccountID (PK), Amount



* Category: CategoryID (PK), Name

A screenshot of a computer

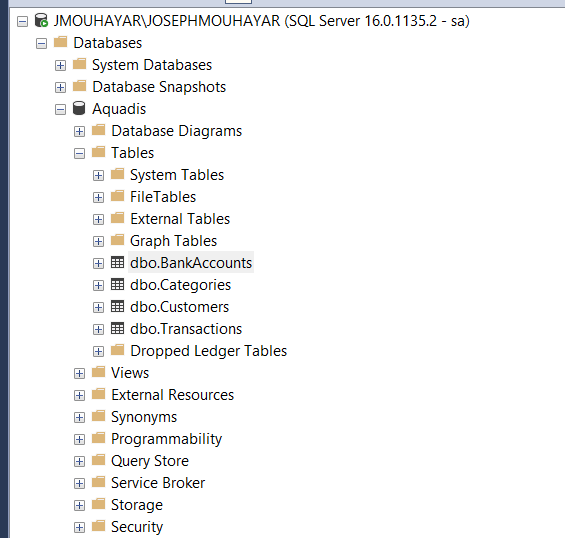
Description automatically generated

* + - Transaction: TransactionID (PK), Type (income or expenses), Amount, createdAt (date time that is directly generated on creation), BankAccountID (FK), CategoryID (FK)

A screenshot of a computer program

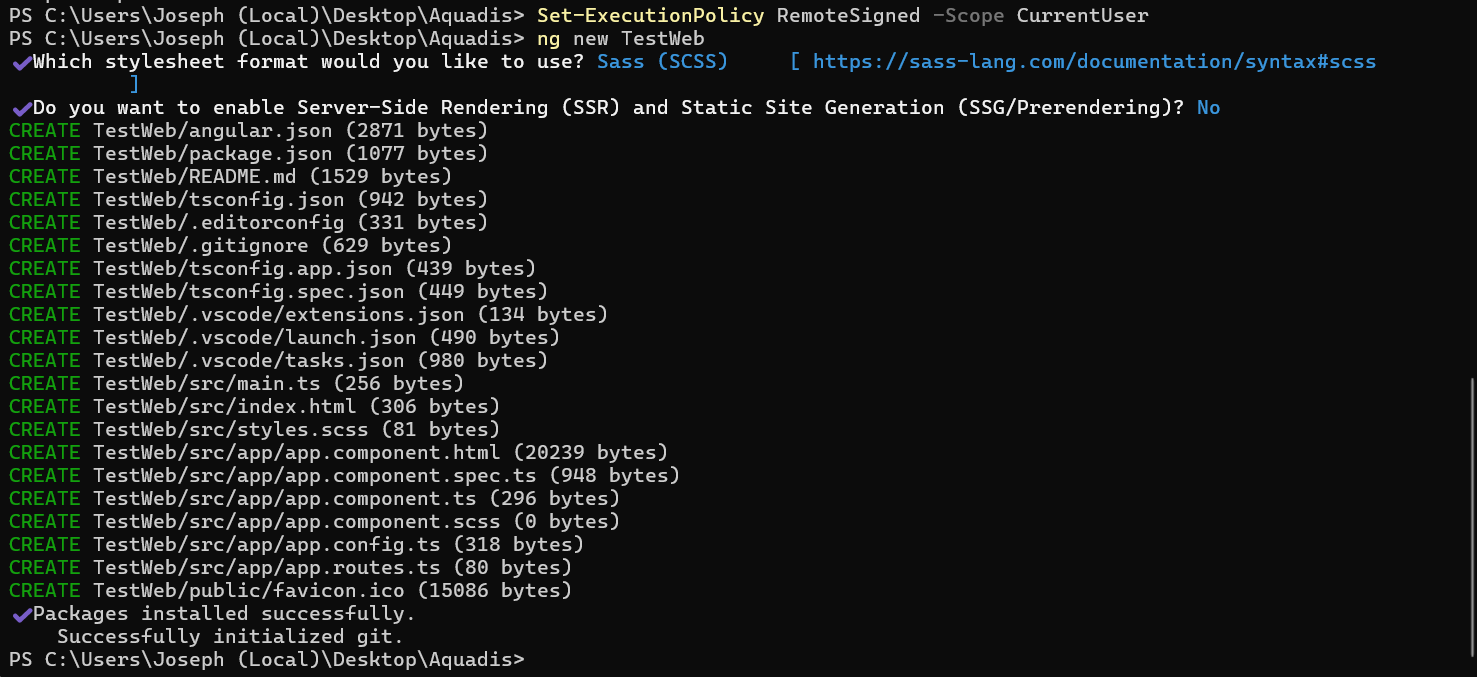
Description automatically generated

* + - Database Structure:

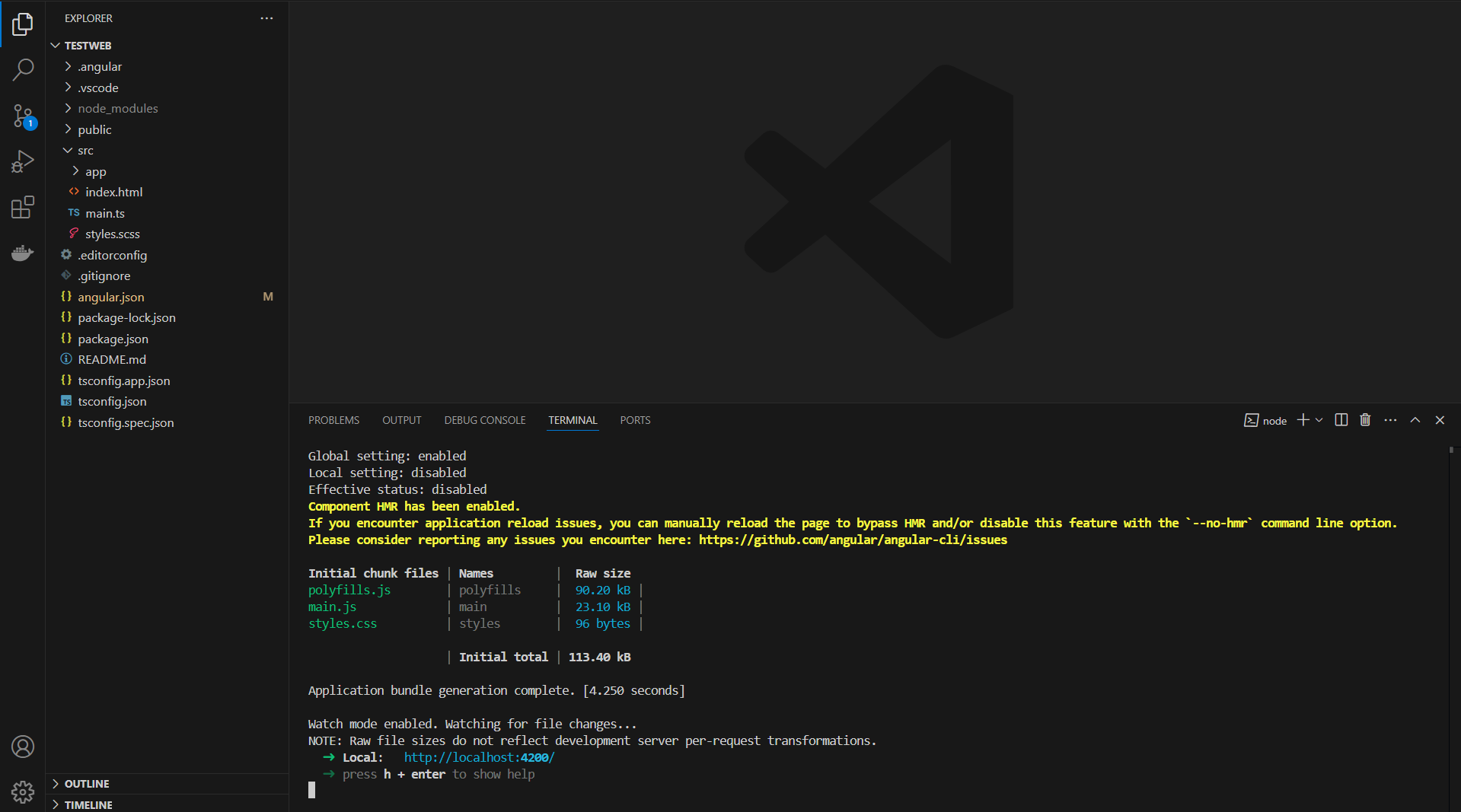


# Angular Initialization:

* PowerShell script



* VsCode View + ng serve:



Ng serve

# Login/SignUp pages:

* Spring-Boot:
  + Creation of JwtUserDetails:
    - Custom Implementation of UserDetails: The JwtUserDetails class implements the UserDetails interface from Spring Security, which is used to represent user authentication and authorization details, but in this case, it's based on the Customer entity.
    - Customer Information Mapping: The class maps customer data (such as password, username, and customerID) to the required UserDetails methods. It provides the customer's password and username, while also exposing the getCustomerID() method for customer-specific data.
    - Default Account Settings: It overrides the standard account status checks (isAccountNonExpired, isAccountNonLocked, etc.) to always return true, assuming the account is always active and valid in this example.

A screenshot of a computer program

Description automatically generated

* + Creation of JwtUtil:
* WT Token Generation: The class manually constructs a JWT token by creating a header, payload (with claims such as customerID, name, and expiration time), and signature using HMAC SHA-256 and a secret key (SECRET\_KEY). It then combines these parts to form a valid JWT.
* Token Parsing and Claims Extraction: The class provides methods to decode a JWT, parse its payload, and extract claims like username (subject), customerID, and expiration. It uses Base64 URL decoding and simple string manipulation to extract and return these values.
* Token Validation: It validates a JWT by checking if the username in the token matches the provided UserDetails and if the token has not expired, using the expiration date in the payload and comparing it with the current date.

A screenshot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

* + Dtos:
    - CustomerDto:

A black rectangular object with a black strip

Description automatically generated

* SignUpRequest:

A black rectangular object with a black line

Description automatically generated

* AuthenticationRequest:

A black rectangular object with a blue stripe

Description automatically generated

* + UserRepository: The UserRepository interface extends JpaRepository to provide CRUD operations for the Customer entity and includes a custom method findByName to retrieve a Customer based on their name, returning an Optional<Customer>.

A screenshot of a computer

Description automatically generated

* + Services
    - AuthService: The AuthService interface defines methods for creating a customer from a sign-up request and checking if a customer with a specific name exists.

A screenshot of a computer program

Description automatically generated

* AuthServiceImpl:
  + **Hashing Password**: The hashPassword method hashes a given password using SHA-256 to store it securely.
  + **Create Customer**: The createCustomer method accepts a SignUpRequest, creates a new Customer entity, hashes the password, saves it to the repository, and returns a CustomerDto.
  + **Check Customer Existence**: The hasCustomerWithName method checks if a customer with the given name exists in the repository.

A screen shot of a computer program

Description automatically generated

* UserDetailsServiceImpl: The UserDetailsServiceImpl class implements UserDetailsService to load user details from the repository based on the username, returning a JwtUserDetails object for the authenticated user.

A screen shot of a computer program

Description automatically generated

* + JwtRequestFilter:
* Extracts Token and Customer Details: Checks the Authorization header for a JWT token, extracting customerName and customerID using JwtUtil.
* Validates the Token: Verifies the token's validity using JwtUtil and loads user details via UserDetailsServiceImpl if authentication is not yet set.
* Sets Authentication: Creates an authenticated UsernamePasswordAuthenticationToken and stores it in SecurityContextHolder, including the customerID in the authentication details.
* Continues Request Processing: Passes the request along the filter chain by calling filterChain.doFilter(), ensuring the customer’s ID is available for subsequent processing.

A screenshot of a computer program

Description automatically generated

* + Config:
    - SimpleCorsFilter:
* Handles CORS: It sets the necessary headers to allow cross-origin requests, including Access-Control-Allow-Origin, Access-Control-Allow-Methods, and Access-Control-Allow-Headers.
* Dynamic Origin Handling: The origin of the incoming request is dynamically set in the Access-Control-Allow-Origin header based on the Origin header from the request.
* Supports Preflight Requests: If the HTTP method is OPTIONS, which is used for preflight CORS requests, it responds with status 200 OK to allow the subsequent actual request.
* Filter Execution Order: The filter is configured to run at the highest precedence (Ordered.HIGHEST\_PRECEDENCE), ensuring it is applied first in the filter chain.

A screen shot of a computer

Description automatically generated

* + - WebSecurityConfiguration:
* API Security: It applies security settings only to endpoints under /api/\*\*, disabling CSRF protection and allowing specific endpoints like /authenticate, /signup, and /api/customer/\*\* to be accessed without authentication.
* Stateless Sessions: Configures stateless session management (no session is created or stored) by using SessionCreationPolicy.STATELESS, which is ideal for token-based authentication (e.g., JWT).
* JWT Filter Integration: It adds a JwtRequestFilter to the filter chain before the UsernamePasswordAuthenticationFilter to handle JWT validation in incoming requests.
* Password Encoding and Authentication Manager: It defines a BCryptPasswordEncoder bean for password encoding and provides an AuthenticationManager bean for managing authentication.

A screen shot of a computer

Description automatically generated

* Angular:
  + Creation of Components (Login and SignUp) and Customer Service

A screenshot of a computer

Description automatically generated

* Customer Service:
  + User Data Management: Uses BehaviorSubject to manage and update the current user data (name and password), with currentUserData observable for real-time updates.
  + Sign Up: signUpCustomer method sends customer data to the backend for creating a new customer.
  + Login: loginCustomer method sends login credentials (name and password) to authenticate the user and returns the full response, including headers (for token handling).
  + JWT Decoding: decodeToken method decodes the JWT token using the jwt-decode library to extract user claims.

A screen shot of a computer program

Description automatically generated

* Login Component:
  + HTML:

A screen shot of a computer program

Description automatically generated

* + Component:

A screen shot of a computer program

Description automatically generated

* SignUp Component:
  + HTML:

A screenshot of a computer program

Description automatically generated

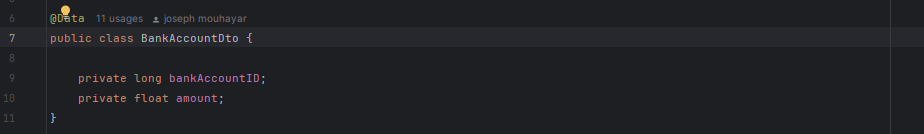
* + Component:

A screen shot of a computer program

Description automatically generated

# Dashboard:

* Spring (Backend):
  + Create Dtos:
    - BankAccountDto



* + - CategoryDto

A black line on a black surface

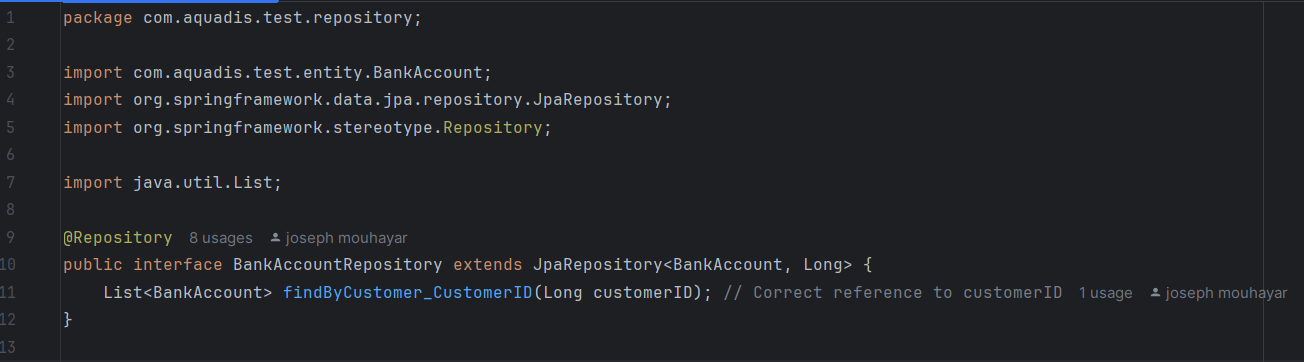
Description automatically generated

* + - TransactionDto

A black rectangle with white text

Description automatically generated

* + Create Repositories:
    - BankAccountService:
      * Extends JpaRepository<BankAccount, Long> → Provides CRUD operations for BankAccount entities with Long as the primary key type.
      * Custom Query Method findByCustomer\_CustomerID(Long customerID) → Fetches all bank accounts associated with a specific customer by their customerID.
      * Uses Spring Data JPA → No need for manual SQL queries; Spring automatically implements the method.

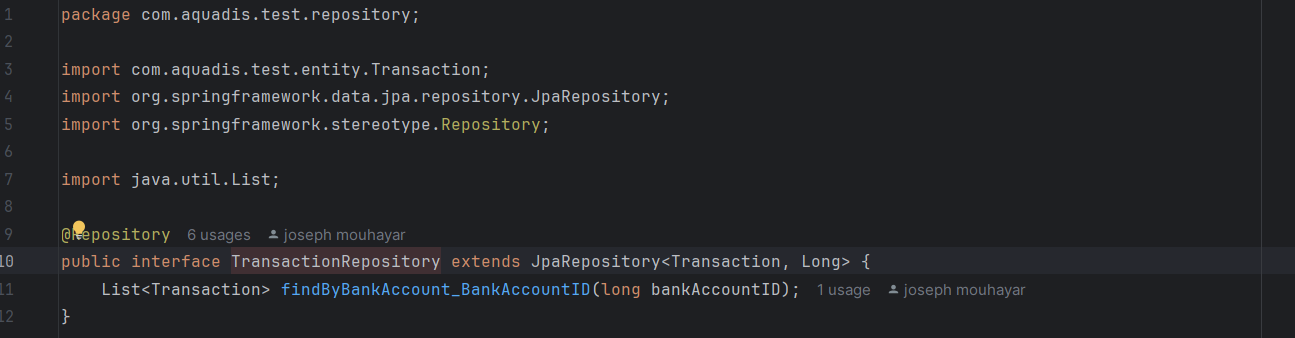


* + - CategoryService:
      * Extends JpaRepository<Category, Long> → Provides built-in CRUD operations for Category entities with Long as the primary key type.
      * No Custom Queries Defined → Uses default JPA methods like save(), findById(), and findAll().
      * Spring Manages Implementation → No need to write query logic; Spring Data JPA generates it automatically.

A screen shot of a computer

Description automatically generated

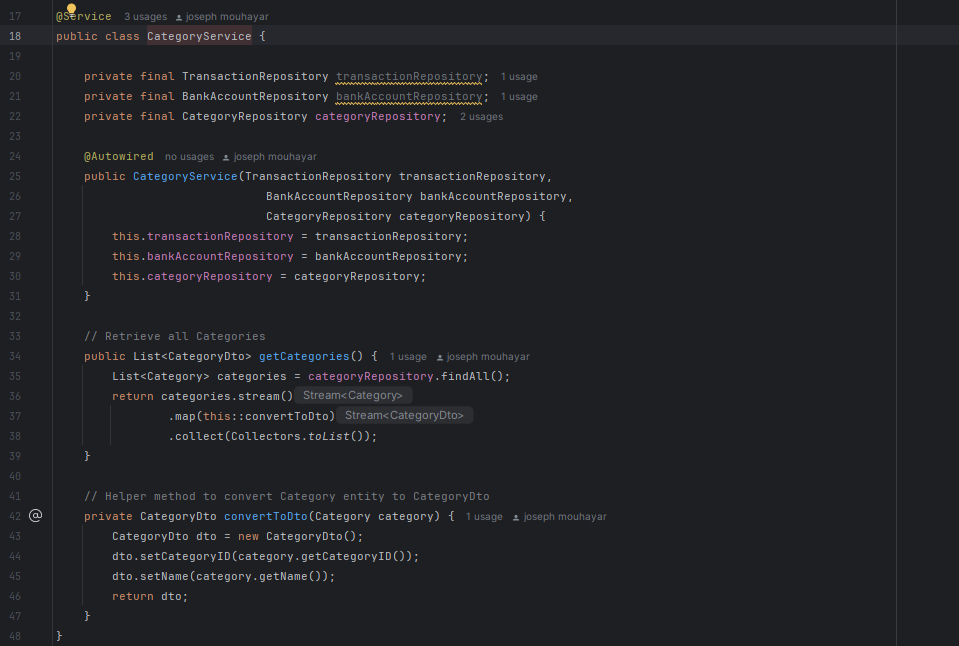
* + - TransactionService:
      * Extends JpaRepository<Transaction, Long> → Provides default CRUD operations for Transaction entities.
      * Custom Query Method → findByBankAccount\_BankAccountID(long bankAccountID) retrieves all transactions linked to a specific bank account.
      * Spring Data JPA Handles Query Generation → No need to write SQL; Spring generates the necessary query automatically.



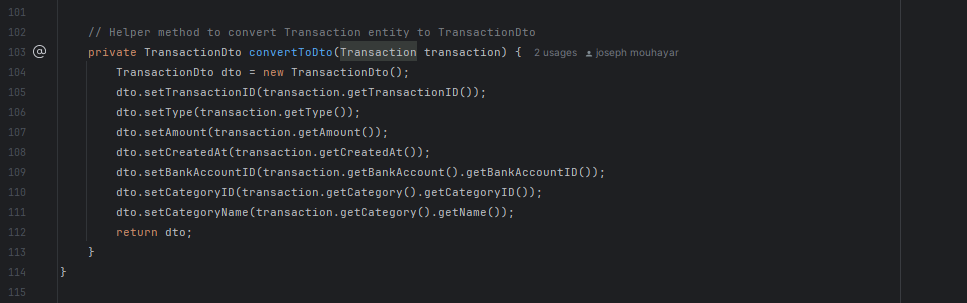
* + Create Services:
    - BankService:
      * Manages Bank Accounts → This service handles retrieving, creating, and deleting bank accounts for customers.
      * Uses BankAccountRepository and UserRepository → Fetches and manipulates bank account data while ensuring linkage to a valid customer.
      * DTO Conversion → Converts BankAccount entities to BankAccountDto for cleaner data transfer.
      * Includes Key Operations → Retrieves all bank accounts for a customer, initializes a new account with zero balance, and deletes bank accounts by ID.



* + - CategoryService:
      * Manages Categories → Handles retrieving all categories from the database.
      * Uses DTO Conversion → Converts Category entities to CategoryDto for structured data transfer.
      * Dependencies Injected via Constructor → Ensures proper initialization of TransactionRepository, BankAccountRepository, and CategoryRepository.

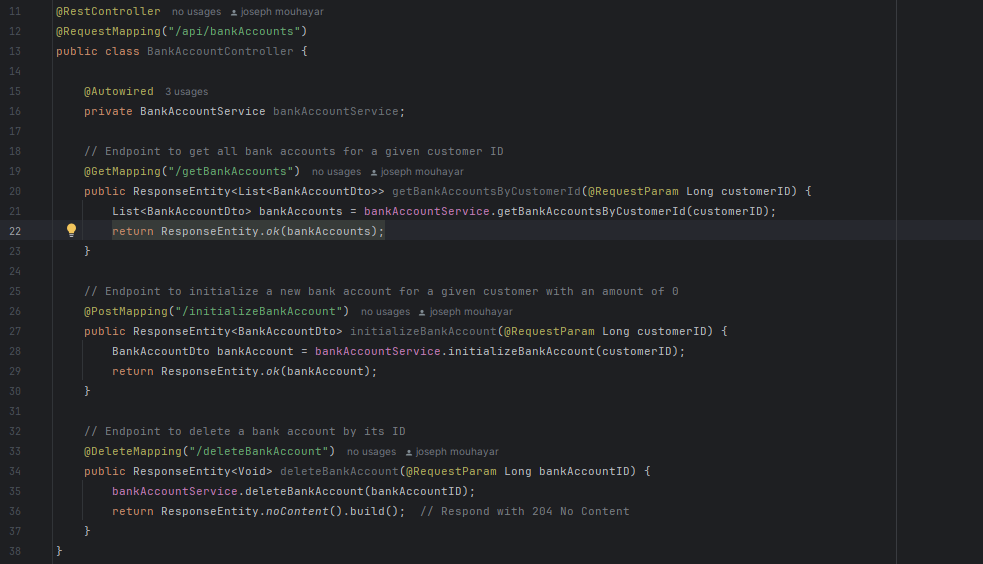


* + - TransactionService:
      * Manages Transactions → Handles retrieving, adding, and deleting transactions for bank accounts.
      * Updates Bank Account Balance → Adjusts the balance when a transaction is added or deleted, ensuring no negative balance occurs.
      * Validates Transactions → Prevents expenses if they exceed the available balance and ensures income deletion doesn’t cause an invalid balance.
      * Uses DTO Conversion → Converts Transaction entities to TransactionDto for structured data transfer.

A computer screen shot of a program code

Description automatically generated

* + Create Controllers:
    - BankAccountController:
      * Manages Bank Accounts → Handles retrieval, initialization, and deletion of bank accounts.
      * Customer-Based Operations → Retrieves bank accounts by customer ID and initializes new accounts with a zero balance.
      * RESTful API Endpoints → Provides structured endpoints for fetching, creating, and deleting bank accounts with proper HTTP responses.

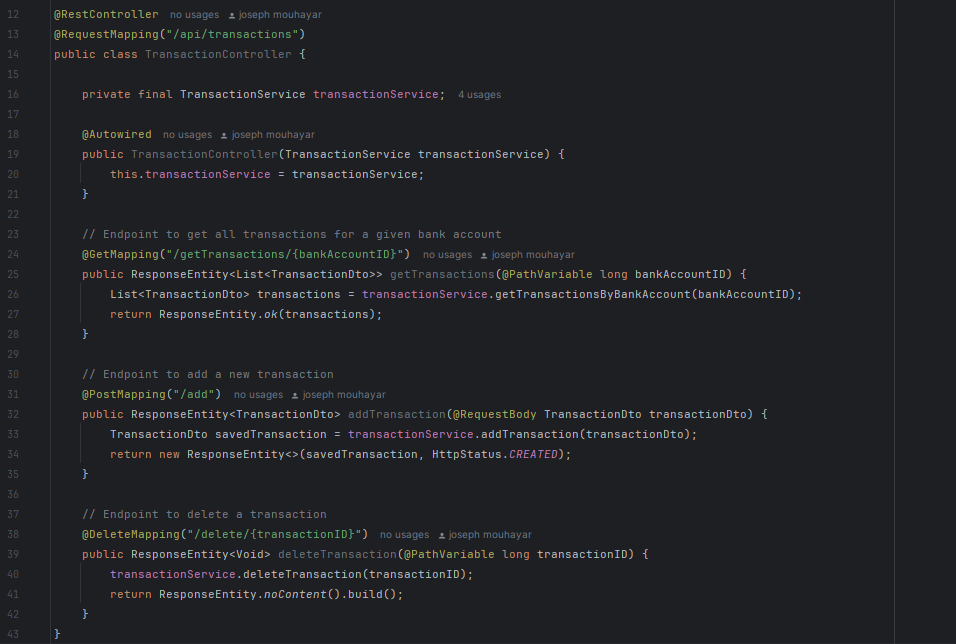


* + - CategoryController:
      * Class Overview: This is a Spring Boot CategoryController class, designed as a REST controller to handle API requests related to categories.
      * Dependency Injection: The CategoryService is injected into the controller using @Autowired to manage the logic related to categories.
      * API Endpoint: The @GetMapping("/getCategories") method defines an endpoint that retrieves all categories by calling the getCategories() method of CategoryService.

A computer screen shot of a program code

Description automatically generated

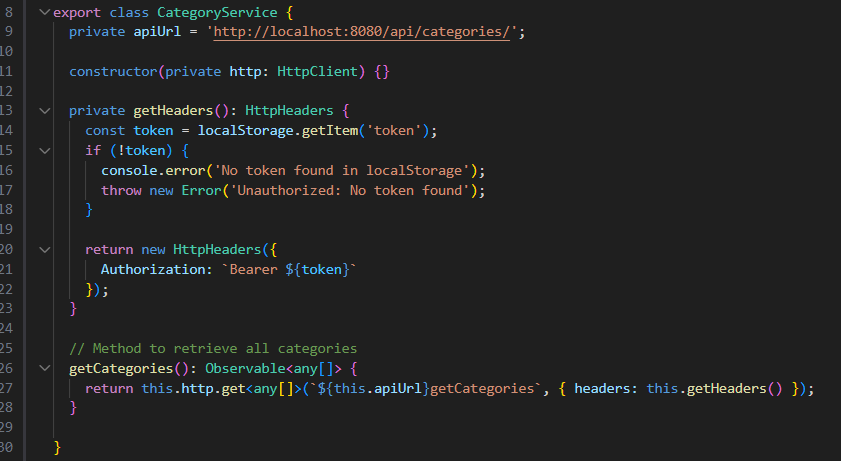
* + - TransactionController:
      * Class Overview: This is a Spring Boot TransactionController class, which provides RESTful endpoints for managing transactions.
      * Dependency Injection: The TransactionService is injected into the controller via @Autowired to handle business logic related to transactions.
      * RESTful API Endpoints → Provides structured endpoints for fetching, creating, and deleting Transactions with proper HTTP responses.



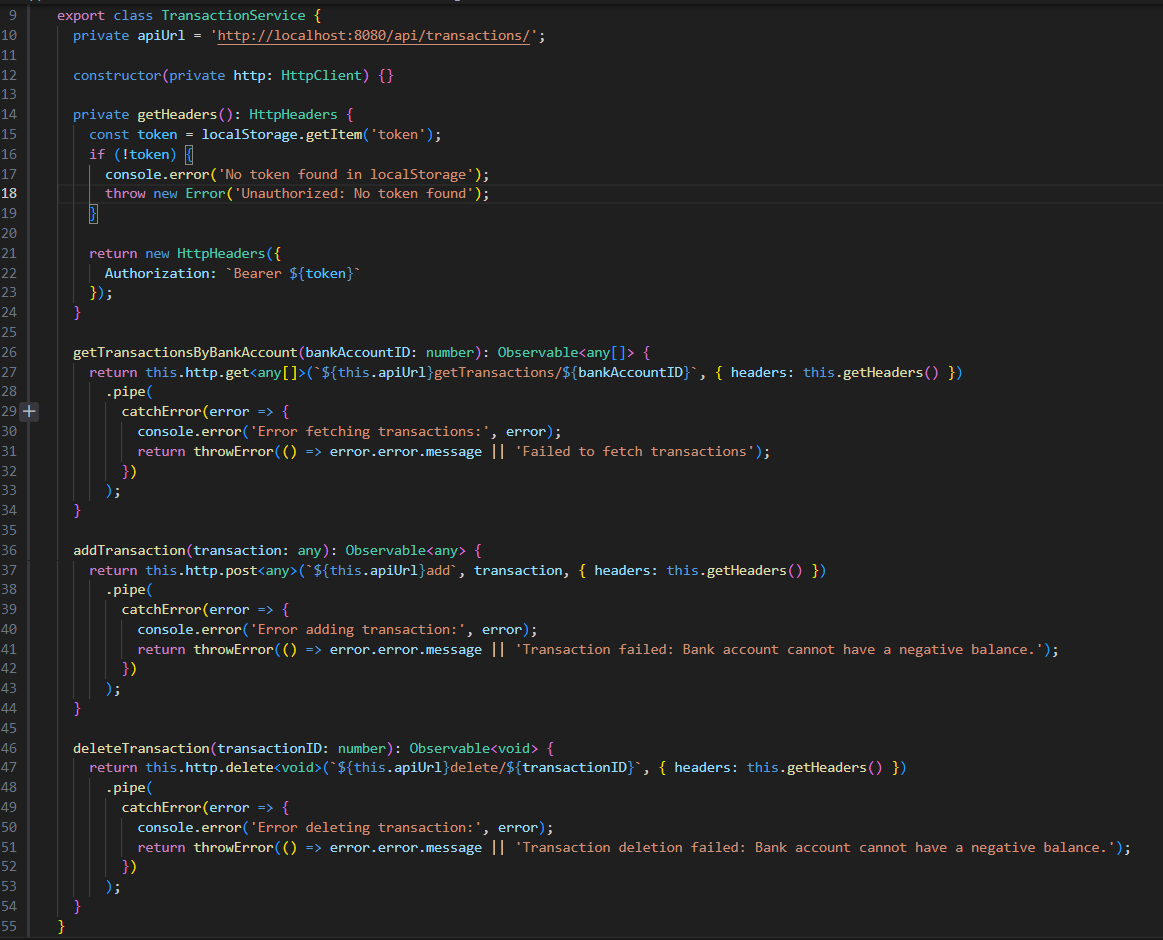
* Angular (Frontend):
  + Create Services:
    - BankAccountService:
      * Service Overview: The BankAccountsService is an Angular service that provides methods for interacting with an API related to bank accounts.
      * Authorization Header: The service uses a private method getHeaders() to retrieve an authentication token from localStorage and includes it in the HTTP request headers for authorization.
      * Get Bank Accounts: The getBankAccounts() method retrieves all bank accounts for a given customer by making a GET request to the API.
      * Bank Account Operations: The service includes methods to initialize a new bank account (initializeBankAccount()) and delete an existing bank account (deleteBankAccount()) by sending POST and DELETE requests, respectively.



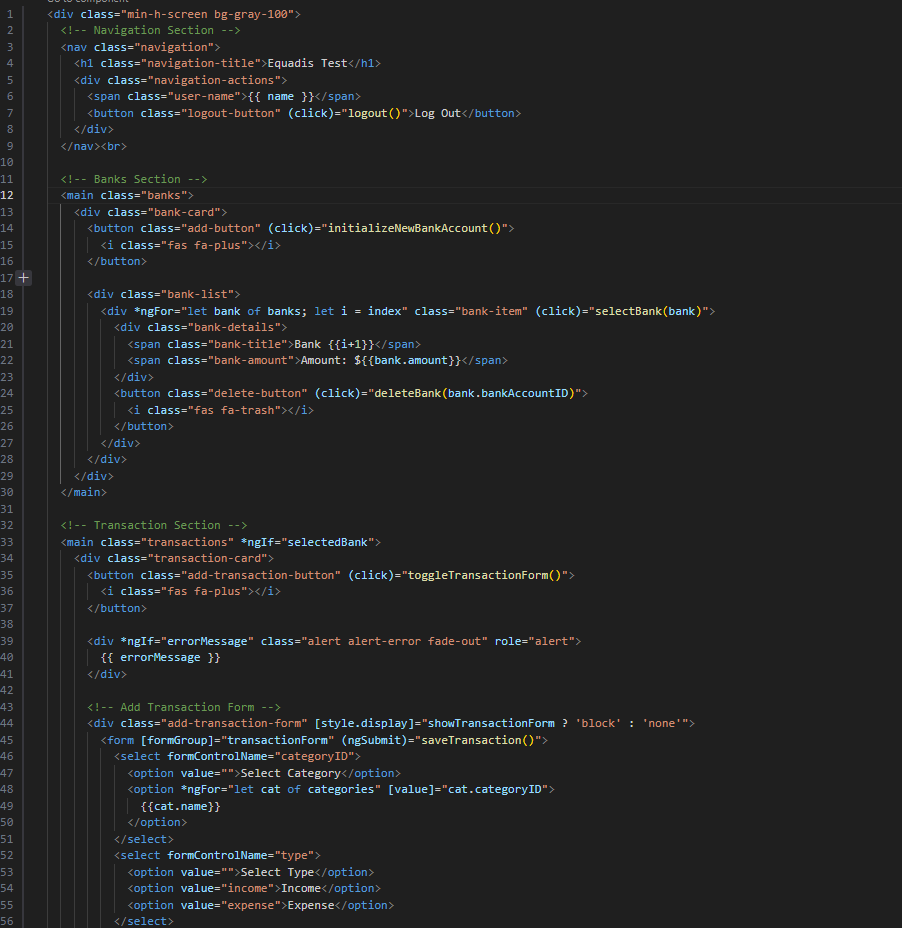
* + - CategoryService:
      * Service Overview: The CategoryService is an Angular service that handles HTTP requests related to categories in the application.
      * Authorization Header: The service includes a private method getHeaders() to fetch an authentication token from localStorage and add it to the request headers for authorization.
      * Get Categories: The getCategories() method retrieves all categories by making a GET request to the API endpoint, returning the data as an Observable of an array of categories.



* + - TransactionService:
      * Service Overview: The TransactionService is an Angular service that manages HTTP requests related to transactions, including fetching, adding, and deleting transactions.
      * Authorization Header: The service includes a private method getHeaders() to retrieve a token from localStorage and include it in the request headers for authorization.
      * Error Handling: The service uses catchError from rxjs to handle errors in each method, logging the error to the console and returning a custom error message via throwError.
      * Transaction Methods:
        + getTransactionsByBankAccount() retrieves transactions for a specific bank account.
        + addTransaction() adds a new transaction to a bank account.
        + deleteTransaction() deletes a transaction by its ID.



* + Dashboard Component:
    - HTML:
      * Navigation: Displays a title "Equadis Test" and the logged-in user's name with a "Log Out" button.
      * Banks Section: Lists all banks with their balance and options to add or delete a bank account.
      * Transactions Section: Shows transactions for a selected bank with details like category, date, type, and amount, along with an option to add or delete transactions.
      * Add Transaction Form: Allows users to add a new transaction by selecting a category, type, and amount, with validation on form submission.
      * Conditional Rendering: The transaction form and transaction list are conditionally shown based on the selected bank and the state of the form.

A computer screen shot of a program code

Description automatically generated

* + - Component:
      * Component Setup: This is an Angular component for a dashboard that handles bank accounts, transactions, and categories, with reactive forms for managing transactions.
      * Bank and Transaction Data: It manages bank accounts, retrieves transactions associated with each bank, and allows users to add or delete banks and transactions.
      * Customer Authentication: The customerID and name are retrieved from localStorage to ensure the user is logged in; if not, the user is redirected to the login page.
      * Reactive Form: It uses a reactive form (transactionForm) for adding transactions, which includes fields for category, type, and amount, with validation.
      * Bank Operations: Provides methods to initialize new bank accounts, delete existing ones, and select a bank to view and manage its transactions.
      * Transaction Management: Supports adding new transactions, toggling the transaction form, displaying transactions for the selected bank, and deleting transactions.
      * Category Management: Fetches categories from a service and uses them in the transaction form to categorize each transaction.
      * Error Handling: Displays error messages when actions fail (e.g., fetching data, adding/deleting items), which automatically disappear after 3 seconds.
      * Logout Functionality: Includes a logout method that clears localStorage and redirects to the login page.
      * Service Communication: Uses injected services (BankAccountsService, TransactionService, CategoryService) to interact with APIs for managing bank accounts, transactions, and categories.



A black background with colorful lines

Description automatically generated

A black screen with text on it

Description automatically generated

A screen shot of a computer

Description automatically generated

