FLYWAY

Flyway is a database migration tool that helps to version control database schemas. It allows you to manage your database migrations easily, applying changes in a controlled manner, ensuring that your schema stays in sync with the application code.

Key Features of Flyway:

- **Version Control**: Tracks the history of changes to your database schema and applies migrations incrementally.
- **Easy Rollbacks**: Migrations are written as SQL scripts or Java classes, so rolling back or reapplying a migration is easy.
- **Integration**: Flyway integrates seamlessly with Spring Boot, Maven, Gradle, and other build tools.
- **Database Independence**: Flyway supports a variety of databases like MySQL, PostgreSQL, Oracle, SQL Server, etc.

Setting Up Flyway in a Spring Boot Project:

1. **Add Flyway Dependency**:
    - For **Maven**:

```
<dependency>
    <groupId>org.flywaydb</groupId>
    <artifactId>flyway-core</artifactId>
</dependency>
```

- For **Gradle**:

```
//DEPENDENCY FOR THE FLYWAY
// Flyway for database migrations
implementation 'org.flywaydb:flyway-core:9.0.0'
implementation 'org.postgresql:postgresql:42.5.0'
```

Configure Flyway in `application.properties` or `application.yml`:

- Example for `application.properties`:

```
spring.flyway.enabled=true
spring.flyway.url=jdbc:postgresql://localhost:5432/mydb
spring.flyway.user=your-username
spring.flyway.password=your-password
spring.flyway.locations=classpath:db/migration
```

Explanation:

- `spring.flyway.url`, `spring.flyway.user`, `spring.flyway.password` explicitly define the connection details for Flyway. This is **separate** from Spring's DataSource configuration.
- `spring.flyway.locations=classpath:db/migration` specifies where Flyway should look for migration scripts. This is helpful if you're using a custom directory for your migration scripts, but if you're using the default `src/main/resources/db/migration`, this isn't strictly necessary.

Alternative way to do that

```
# Database Configuration
spring.application.name=Credit_Card_Service
spring.datasource.url=jdbc:postgresql://localhost:5432/credit_
spring.datasource.username=postgres
spring.datasource.password=******
spring.jpa.hibernate.ddl-auto=none

spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect

# Server Configuration
server.port=8080

# location of the swagger json
#springfox.documentation.swagger.v2.path=/swagger.json

# Enable Flyway
spring.flyway.enabled=true

# Baseline the existing schema
spring.flyway.baseline-on-migrate=true

# Set initial baseline version
spring.flyway.baseline-version=1.0
```

- **Explanation:**
    - `spring.flyway.enabled=true` enables Flyway migrations in your Spring Boot project.
    - `spring.datasource.url`, `spring.datasource.username`, `spring.datasource.password` are used by **Spring DataSource** for database connection, and Flyway will use these settings as well (you don't need to define them separately for Flyway).
    - `spring.flyway.baseline-on-migrate=true` tells Flyway to create a baseline version if there's no `flyway_schema_history` table yet.
    - `spring.flyway.baseline-version=1.0` sets the baseline version number.

This is a **simplified configuration** where Spring Boot manages both the database connection and Flyway setup automatically through the main `datasource` properties.

Alternative Configuration:

- Example for `application.yml`:

```
spring:
  flyway:
    enabled: true
    url: jdbc:postgresql://localhost:5432/mydb
    user: your-username
    password: your-password
    locations: classpath:db/migration
```

Create Migration Scripts:

- Migration scripts should be placed in the `src/main/resources/db/migration` directory.
- The script files should be named in a specific format: `V<version>__<description>.sql`. For example:
  - `V1__initial_schema.sql`
  - `V2__add_new_table.sql`

1. **Flyway Migrations**:
   - When the application starts, Flyway checks if there are any pending migrations and applies them.
   - You can run the migrations manually by invoking:
     - **Maven**: `mvn flyway:migrate`
     - **Gradle**: `gradle flywayMigrate`

Flyway Functions (Commands):

1. **Migrate**:
   - `migrate`: Applies all pending migrations to the database.

   - Example: `mvn flyway:migrate` or `gradle flywayMigrate`.
2. **Clean**:
   - `clean`: Drops all objects in the configured database (useful when starting fresh).
   - Example: `mvn flyway:clean`.
3. **Info**:
   - `info`: Displays the current status of all migrations (applied and pending).
   - Example: `mvn flyway:info`.
4. **Validate**:
   - `validate`: Ensures that the migrations are applied correctly and the schema is in a consistent state.
   - Example: `mvn flyway:validate`.
5. **Baseline**:
   - `baseline`: Used when you want to start using Flyway in an existing database. It marks the current state as version 1.
   - Example: `mvn flyway:baseline`.
6. **Repair**:
   - `repair`: Repairs the Flyway metadata table if any problems occur (like a failed migration).
   - Example: `mvn flyway:repair`.

Flyway with Java Migration:

You can write custom migrations using Java if you prefer over SQL. To create a Java-based migration:

1. Create a class that implements `BaseJavaMigration` and overrides the `migrate()` method.
2. Place this class in the `src/main/java` directory under the `db/migration` package.

Example:

```
import org.flywaydb.core.api.migration.BaseJavaMigration;
import org.flywaydb.core.api.migration.Context;

public class V1__initial_migration extends BaseJavaMigratio
n {
    @Override
    public void migrate(Context context) throws Exception {
        // Custom migration logic here, such as creating a
table
        context.getConnection().createStatement().execute
("CREATE TABLE test_table (id INT PRIMARY KEY, name VARCHAR
(255))");
    }
}
```

Additional Configuration Options:

- **Flyway locations**: You can configure multiple locations for migrations.
- **Out-of-order migrations**: Allow Flyway to run migrations even if they are not applied in the exact order.
- **Callbacks**: Flyway allows you to define custom callbacks for actions before or after migrations, like `beforeMigrate`, `afterMigrate`, etc.

By integrating Flyway, you can ensure that your database schema evolves alongside your application code, maintaining consistency across different environments.