Spring Boot

CSIT 2024

github/TanPingZhi

Lombok

references: https://projectlombok.org/features/constructor

`@NoArgsConstructor`

- Generates a no-args constructor. eg. `Person() {}`

`@RequiredArgsConstructor`

- Generates a constructor for all final and not null fields, with parameter order same as field order.

`@AllArgsConstructor`

- Generates a constructor for all fields.

Spring Repository

Purpose

- Encapsulates data access logic, make it easier to do CRUD

Interface

- `CrudRepository<T, ID>`
- `PagingAndSortingRepository<T, ID>`
- `JpaRepository<T, ID>`

Methods

- `save(S entity)`
- `findById(ID primaryKey)`
- `findAll()`
- `deleteById(ID primaryKey)`

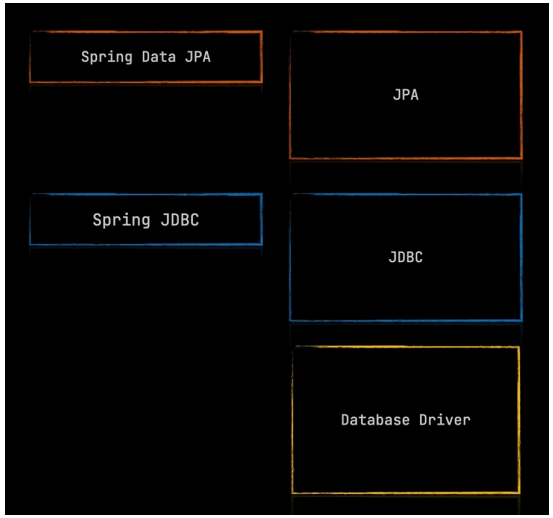
Example Usage

```
import org.springframework.data.repository.CrudRepository;
@Repository
public interface PersonRepository extends CrudRepository<Person, Long> {
  // custom queries
}
```

Usage in Service

```
@Service
public class PersonService {
  @Autowired
  private PersonRepository personRepository;
  public Person savePerson(Person person) {
    return personRepository.save(person);
  }
  // other methods
}
```

Database layers



Database drivers

- Allows you to interact with the database from the Java code

JDBC: Java Database Connectivity

- Allows developers to use custom SQL queries
- Have to handle mapping to Java objects manually

Spring JDBC

- Builds on top of JDBC
- Provides the JDBC template
- Makes interacting with the database with SQL easier

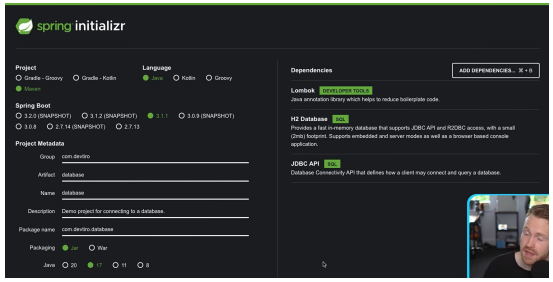
JPA: Java Persistence API

- Allows interaction with the database using Java objects
- Handles all the generation of the sql and the mapping to and from Java objects
- Builds on top of JDBC

Spring Data JPA

- Repository
- Hibernate is a ORM (Object Relational Mapping) framework

Connect to a H2 Database




```
package com.pingzhi.database;
import lombok.extern.java.Log;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
```

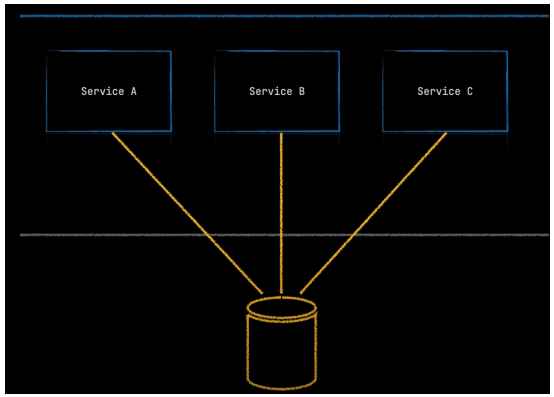
```
import org.springframework.jdbc.core.JdbcTemplate;
import javax.sql.DataSource;

@Log
@SpringBootApplication
public class DatabaseApplication implements CommandLineRunner {
  private final DataSource dataSource;
  public DatabaseApplication(final DataSource datasource) {
    this.dataSource = datasource;
  }
  public static void main(String[] args) {
    SpringApplication.run(DatabaseApplication.class, args);
  }
 @Override
 public void run(final String... args) {
  log.info("Datasource: " + dataSource.toString());
  final JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
  jdbcTemplate.execute("select 1");
 }
}
```

- SpringApplication.run(...) is the driver to the Spring Boot application
- mistake in the tutorial: should use "jdbcTemplate" instead of "restTemplate"

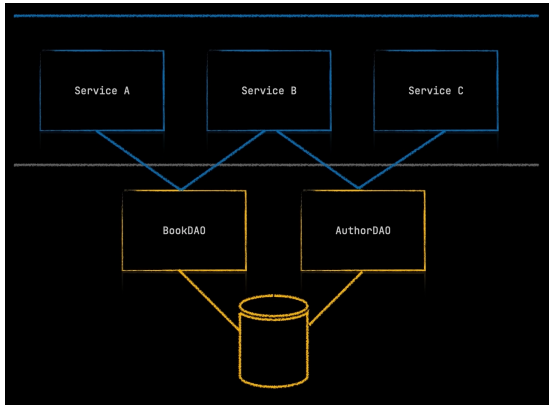
Introduction to DAO: Data Access Object pattern

Consider book and auther entities

- Book has 1 Author
- Author has ≥ 1 Book



- Lets say that we have 3 services that need to interact with the database.
- If we use jdbc, each services will need to convert between sql and java objects and will result in duplicate code.



- Abstracts the conversion between sql and java objects into a single class

Author Domain

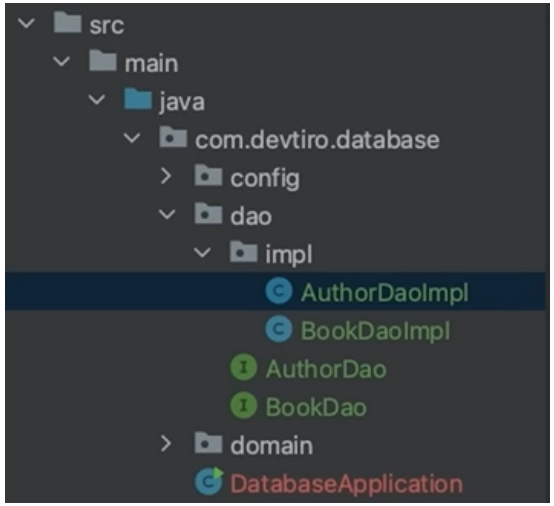
```
package com.devtiro.database.domain;
import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;
@Data
@AllArgsConstructor
@NoArgsConstructor
@Builder
public class Author {
  private Long id;
  private String name;
  private Integer age;
}
```

- @Data: a Lombok annotation that generates getters, setters, equals, hashcode, and toString
- @Builder: builder pattern
- Long id: so that it can be null instead of 0

Book Domain

```
// same as Author
public class Book {
  private String isbn;
  private String title;
  private Long authorId;
}
```

File structure



Author DAO

```
package com.devtiro.database.dao.impl;
import com.devtiro.database.dao.AuthorDao;
import org.springframework.jdbc.core.JdbcTemplate;
public class AuthorDaoImpl implements AuthorDao {
    private final JdbcTemplate jdbcTemplate;
    public AuthorDaoImpl(final JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }
}
```

• allows us to inject the JdbcTemplate into the AuthorDaoImpl

Book DAO

• Similar to Author Dao

Create DAO

```
public class AuthorDaoImpl implements AuthorDao {
  // ... continuing from previous snippet
  @Override
  public void create(Author author) {
      jdbcTemplate.update(
              "INSERT INTO authors (id, name, age) VALUES (?, ?, ?)",
              author.getId(), author.getName(), author.getAge()
      );
  }
}
```

Test Auther DAO

```
@ExtendWith(MockitoExtension.class)
public class AuthorDaoImplTests {
    @Mock
    private JdbcTemplate jdbcTemplate;
    @InjectMocks
    private AuthorDaoImpl underTest;
    @Test
    public void testThatCreateAuthorGeneratesCorrectSql() {
        Author author = Author.builder()
                .id(1L)
                .name("Abigail Rose")
                .age(80)
                .build();
        underTest.create(author);
        verify(jdbcTemplate).update(
                eq("INSERT INTO authors (id, name, age) VALUES (?, ?, ?)"),
                eq(1L), eq("Abigail Rose"), eq(80)
        );
    }
}
```

• Author.builder() allows us to create an Author object

10.1 REST API Design

We will be putting books and authors in a postgres db **RestController**

• Primarily designed for server-server communication and RESTful APIs
• Not meant for rendering content in a browser
• Good for distributed systems

10.2 Author Create Endpoint