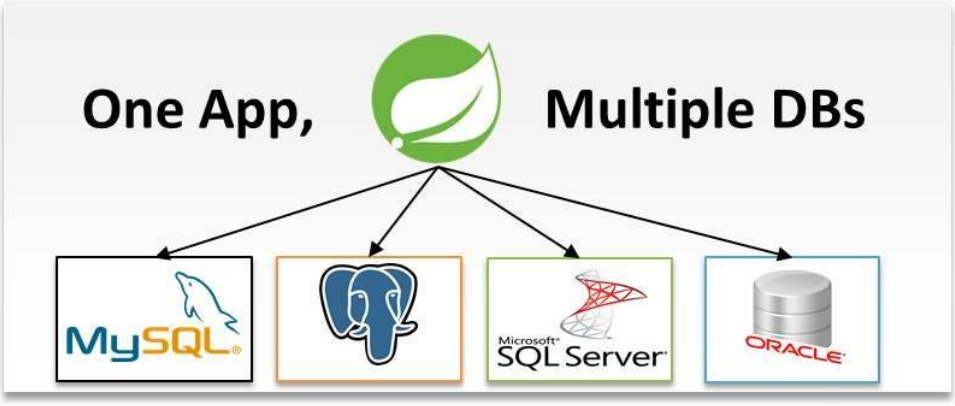
How to Set Up and Manage Multiple Databases in Spring Boot Microservices

In a microservices environment, it's common for services to interact with different databases. This can be due to integrating legacy systems, meeting performance needs, or utilizing specialized data stores. **Spring Boot** provides excellent support for managing multiple data sources, making it an ideal choice for flexible, modern architectures.

Table of Contents



---

1. Introduction

Modern microservices often need to communicate with various databases, each optimized for specific use cases. Spring Boot's flexible configuration and robust data management make it a strong solution for handling multiple databases efficiently.

---

2. Why Use Multiple Databases?

Common reasons for using multiple databases in a microservice include:

- **Legacy System Integration**: Seamlessly connecting new services with existing databases.
- **Optimized Performance**: Using specialized databases, like SQL for structured data and NoSQL for unstructured data.
- **Data Segregation**: Keeping sensitive or high-priority data in isolated storage.
- **Scalability**: Spreading data across databases to avoid bottlenecks.

---

3. Setting Up a Spring Boot Project

Start by creating a new Spring Boot project using Spring Initializr or your preferred IDE. Add dependencies for Spring Data JPA and any database drivers (e.g., **H2**, **PostgreSQL**, **MySQL**).

Maven Dependencies

In your `pom.xml`, include necessary dependencies:

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
  </dependency>
  <dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <scope>runtime</scope>
  </dependency>
</dependencies>
```

---

4. Configuring Multiple Data Sources

Define configurations in `application.yml` or `application.properties` to establish connections to each database.

Example `application.yml` Configuration:

```
spring:
  datasource:
    primary:
      url: jdbc:h2:mem:primarydb
      driver-class-name: org.h2.Driver
      username: sa
      password: password
    secondary:
      url: jdbc:postgresql://localhost:5432/secondarydb
      driver-class-name: org.postgresql.Driver
      username: postgres
      password: password

jpa:
  primary:
    database-platform: org.hibernate.dialect.H2Dialect
    hibernate:
      ddl-auto: update
  secondary:
    database-platform: org.hibernate.dialect.PostgreSQLDialect
    hibernate:
      ddl-auto: update
```

---

5. Creating Data Source Configuration Classes

Next, define separate configuration classes for each data source. These classes will handle the setup of data sources, `EntityManagerFactory`, and `TransactionManager` for each database.

Primary Data Source Configuration:

```
@Configuration
@EnableJpaRepositories(
   basePackages = "com.example.primary.repository",
   entityManagerFactoryRef = "primaryEntityManagerFactory",
   transactionManagerRef = "primaryTransactionManager"
)
public class PrimaryDataSourceConfig {

   @Bean(name = "primaryDataSource")
   @ConfigurationProperties(prefix = "spring.datasource.primary")
   public DataSource primaryDataSource() {
      return DataSourceBuilder.create().build();
   }

   @Bean(name = "primaryEntityManagerFactory")
   public LocalContainerEntityManagerFactoryBean primaryEntityManagerFactory(
         @Qualifier("primaryDataSource") DataSource dataSource) {
      LocalContainerEntityManagerFactoryBean em = new
LocalContainerEntityManagerFactoryBean();
      em.setDataSource(dataSource);
      em.setPackagesToScan("com.example.primary.entity");
      em.setJpaVendorAdapter(new HibernateJpaVendorAdapter());
      return em;
   }

   @Bean(name = "primaryTransactionManager")
   public PlatformTransactionManager primaryTransactionManager(
         @Qualifier("primaryEntityManagerFactory") EntityManagerFactory
entityManagerFactory) {
      return new JpaTransactionManager(entityManagerFactory);
```

```
    }
}
```

Secondary Data Source Configuration:

```
@Configuration
@EnableJpaRepositories(
    basePackages = "com.example.secondary.repository",
    entityManagerFactoryRef = "secondaryEntityManagerFactory",
    transactionManagerRef = "secondaryTransactionManager"
)
public class SecondaryDataSourceConfig {

    @Bean(name = "secondaryDataSource")
    @ConfigurationProperties(prefix = "spring.datasource.secondary")
    public DataSource secondaryDataSource() {
        return DataSourceBuilder.create().build();
    }

    @Bean(name = "secondaryEntityManagerFactory")
    public LocalContainerEntityManagerFactoryBean secondaryEntityManagerFactory(
            @Qualifier("secondaryDataSource") DataSource dataSource) {
        LocalContainerEntityManagerFactoryBean em = new
LocalContainerEntityManagerFactoryBean();
        em.setDataSource(dataSource);
        em.setPackagesToScan("com.example.secondary.entity");
        em.setJpaVendorAdapter(new HibernateJpaVendorAdapter());
        return em;
    }

    @Bean(name = "secondaryTransactionManager")
    public PlatformTransactionManager secondaryTransactionManager(
```

```
        @Qualifier("secondaryEntityManagerFactory") EntityManagerFactory
entityManagerFactory) {
      return new JpaTransactionManager(entityManagerFactory);
   }
}
```

---

6. Defining Entity Classes

Define your entity classes separately for each database.

Primary Database Entity:

```
@Entity
public class PrimaryEntity {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;
   private String name;
   // Getters and Setters
}
```

Secondary Database Entity:

```
@Entity
public class SecondaryEntity {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;
```

```
    private String description;
    // Getters and Setters
}
```

---

7. Creating Repository Interfaces

Create separate repositories for each database in the corresponding packages defined in the configuration classes.

Primary Repository

```
public interface PrimaryRepository extends JpaRepository<PrimaryEntity, Long> {}
```

Secondary Repository

```
public interface SecondaryRepository extends JpaRepository<SecondaryEntity, Long> {}
```

---

8. Testing the Configuration

To test the setup, create a REST controller that uses both repositories to interact with each database.

Sample Controller:

```
@RestController
```

```
public class TestController {

    @Autowired
    private PrimaryRepository primaryRepository;

    @Autowired
    private SecondaryRepository secondaryRepository;

    @GetMapping("/test")
    public String test() {
        PrimaryEntity primaryEntity = new PrimaryEntity();
        primaryEntity.setName("Primary Entity");
        primaryRepository.save(primaryEntity);

        SecondaryEntity secondaryEntity = new SecondaryEntity();
        secondaryEntity.setDescription("Secondary Entity");
        secondaryRepository.save(secondaryEntity);

        return "Entities saved!";
    }
}
```

Running the Application:

Start your Spring Boot application, then visit the `/test` endpoint to verify that both entities are saved in their respective databases.

---

9. Conclusion

Setting up and managing multiple databases in a Spring Boot microservice architecture is essential for efficient, scalable, and adaptable backend systems. By configuring separate data sources and repositories, Spring Boot enables clean separation of data access layers, allowing your microservices to evolve with minimal friction.