



How to Configure Multiple Data Sources in a Spring Boot Application



By [SFG Contributor](#) | May 31, 2019 | [Spring Boot](#), [Spring Data](#)

 26 Comments

Introduction

Often, you will need to connect to more than one data source. Sometimes, this is for security reasons.

An example of this is the storage of credit card information. You may wish to store the data elements in multiple data sources. If one of the data sources is compromised the data retrieved is useless without the data from other data sources.

In this article, we will configure multiple data sources in Spring Boot and JPA.



Project Setup

Databases

We will use MySQL for our database server.

The credit card scenario described above, will use the following three databases:

1. **Member database(memberdb)**: Stores personal details of cardholders which include their full name and member id.
2. **Cardholder database(cardholderdb)**: Stores cardholder details which include the member id and credit card number.
3. **Card database(carddb)**: Stores the credit card information which includes the owner's full name and the credit card expiration date.

Since we are spreading the credit card data across three databases, all three would need to be compromised for a security risk.

NOTE: This scenario is for an example of using multiple data sources with Spring Boot. This article is not a security recommendation.

Dependencies

To support MySQL, our classpath must include the MySQL database connector dependency.

Here is the list of Maven dependencies.

```
1.    <dependencies>
2.        <dependency>
3.            <groupId>org.springframework.boot</groupId>
4.            <artifactId>spring-boot-starter-data-jpa</artifactId>
5.        </dependency>
6.        <dependency>
7.            <groupId>mysql</groupId>
8.            <artifactId>mysql-connector-java</artifactId>
9.            <scope>runtime</scope>
10.    </dependency>
11.    <dependency>
12.        <groupId>org.projectlombok</groupId>
13.        <artifactId>lombok</artifactId>
14.        <optional>true</optional>
15.    </dependency>
16.    <dependency>
17.        <groupId>org.springframework.boot</groupId>
18.        <artifactId>spring-boot-starter-test</artifactId>
19.        <scope>test</scope>
20.    </dependency>
21.    <dependency>
22.        <groupId>commons-dbcp</groupId>
23.        <artifactId>commons-dbcp</artifactId>
24.        <version>${commons.dbcp.version}</version>
25.    </dependency>
26. </dependencies>
```

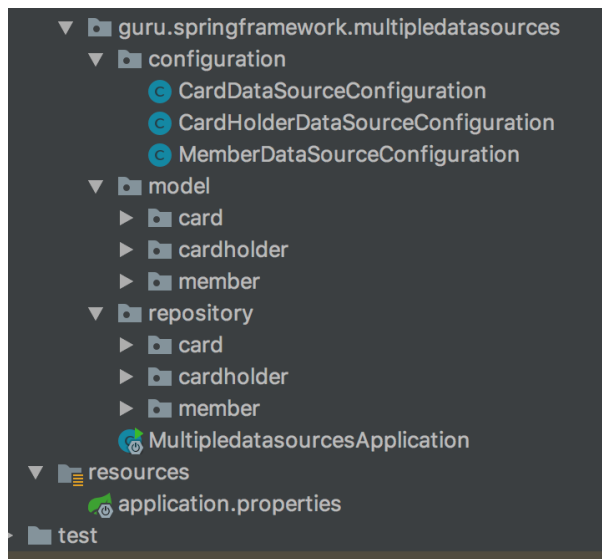
Packaging



The project packaging structure is very important when dealing with multiple data sources.

The data models or entities belonging to a certain datastore must be placed in their unique packages.

This packaging strategy also applies to the JPA repositories.



As you can see above, we have defined a unique package for each of the models and repositories.

We have also created Java configuration files for each of our data sources:

- `guru.springframework.multipledatasources.configuration.CardDataSourceConfiguration`
- `guru.springframework.multipledatasources.configuration.CardHolderDataSourceConfiguration`
- `guru.springframework.multipledatasources.configuration.MemberDataSourceConfiguration`

Each data source configuration file will contain its data source bean definition including the entity manager and transaction manager bean definitions.

Database Connection Settings

Since we are configuring three data sources we need three sets of configurations in the `application.properties` file.

Here is the code of the `application.properties` file.

```
1. #Store card holder personal details
2. app.datasource.member.url=jdbc:mysql://localhost:3306/memberdb?
   createDatabaseIfNotExist=true
3. app.datasource.member.username=root
4. app.datasource.member.password=P@ssw0rd#
5. app.datasource.member.driverClassName=com.mysql.cj.jdbc.Driver
6.
7. #card number (cardholder id, cardnumber)
8. app.datasource.cardholder.url=jdbc:mysql://localhost:3306/cardholderdb?
   createDatabaseIfNotExist=true
9. app.datasource.cardholder.username=root
10. app.datasource.cardholder.password=P@ssw0rd#
11. app.datasource.cardholder.driverClassName=com.mysql.cj.jdbc.Driver
12.
13. #expiration date (card id, expiration month, expiration year)
14. app.datasource.card.url=jdbc:mysql://localhost:3306/carddb?createDatabaseIfNotExist=true
```



```

15. app.datasource.card.username=root
16. app.datasource.card.password=P@ssw0rd#
17. app.datasource.card.driverClassName=com.mysql.cj.jdbc.Driver
18.
19. spring.jpa.hibernate.ddl-auto=update
20. spring.jpa.generate-ddl=true
21. spring.jpa.show-sql=true
22. spring.jpa.database=mysql

```

Data Source Configuration

It is important to note that during the configuration of multiple data sources, one data source instance must be marked as the primary data source.

Else the application will fail to start-up because Spring will detect more than one data source of the same type.

Steps

In this example, we will mark the member data source as our primary data source.

Here are the data source configuration steps.

1. Data source bean definition
2. Entities
3. Entity Manager Factory bean definition
4. Transaction Management
5. Spring Data JPA Repository custom settings

Data Source Bean Definition

To create a data source bean we need to instantiate the

`org.springframework.boot.autoconfigure.jdbc.DataSourceProperties` class using the data source key specified in the `application.properties` file. We are going to use this `DataSourceProperties` object to get a data source builder object.

The data source builder object uses the database properties found in the

`application.properties` file to create a data source object.

The following code shows the bean definitions of our data sources.

Primary Data Source

```

1. @Bean
2. @Primary
3. @ConfigurationProperties("app.datasource.member")
4. public DataSourceProperties memberDataSourceProperties() {
5.     return new DataSourceProperties();
6. }
7.
8. @Bean
9. @Primary
10. @ConfigurationProperties("app.datasource.member.configuration")
11. public DataSource memberDataSource() {
12.     return memberDataSourceProperties().initializeDataSourceBuilder()

```



```

13.         .type(HikariDataSource.class).build();
14.     }

```

Secondary Data Sources

```

1.  /*cardholder data source */
2.  @Bean
3.  @ConfigurationProperties("app.datasource.cardholder")
4.  public DataSourceProperties cardHolderDataSourceProperties() {
5.      return new DataSourceProperties();
6.  }
7.
8.  @Bean
9.  @ConfigurationProperties("app.datasource.cardholder.configuration")
10. public DataSource cardholderDataSource() {
11.     return cardHolderDataSourceProperties().initializeDataSourceBuilder()
12.         .type(BasicDataSource.class).build();
13. }
14.
15. /*card data source*/
16. @Bean
17. @ConfigurationProperties("app.datasource.card")
18. public DataSourceProperties cardDataSourceProperties() {
19.     return new DataSourceProperties();
20. }
21.
22. @Bean
23. @ConfigurationProperties("app.datasource.card.configuration")
24. public DataSource cardDataSource() {
25.     return cardDataSourceProperties().initializeDataSourceBuilder()
26.         .type(BasicDataSource.class).build();
27. }

```

Entities

Since we are going to store `Member`, `Card`, and `Cardholder` objects we must declare them as JPA entities using `@Entity` annotation. These entities will be mapped to relational database tables by JPA.

We must tell Spring which tables belong to a certain data source. There are two ways of achieving this. You can use the `'schema'` field of the `@Table` annotation as indicated in the code snippet below at line 2.

```

1.  @Entity
2.  @Table(name = "member", schema = "memberdb")
3.  @Data
4.  public class Member {
5.
6.      @Id
7.      @GeneratedValue(strategy = GenerationType.AUTO)
8.      private Long id;
9.      private String name;
10.     private String memberId;
11. }

```

Or you may link the entities to their data source is via the

`org.springframework.boot.orm.jpa.EntityManagerFactoryBuilder` class method `packages()`. We can pass the packages or classes to be scanned for `@Entity` annotations in this method.

Spring will use this setting to map these entities to tables which will be created in the data source set through the `datasource()` method of this EMF builder class.



See code snippet in the next section.

Entity Manager Factory Bean Definition

Our application will be using Spring Data JPA for data access through its repository interfaces that abstract us from the EM(Entity Manager). We use the EMF bean to obtain instances of EMs which interact with the JPA entities.

Since, we have three data sources we need to create an EM for each data source.

This is done by providing the EMF builder class with reference to the data source and location of entities.

In our example, we will define this EMF using the

`org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean` class like this.

```
1.      /*Primary Entity manager*/
2.      @Primary
3.      @Bean(name = "memberEntityManagerFactory")
4.      public LocalContainerEntityManagerFactoryBean
memberEntityManagerFactory(EntityManagerFactoryBuilder builder) {
5.          return builder
6.              .dataSource(memberDataSource())
7.              .packages(Member.class)
8.              .build();
9.      }
10.
11.     /*Secondary Entity Managers*/
12.     @Bean(name = "cardHolderEntityManagerFactory")
13.     public LocalContainerEntityManagerFactoryBean cardHolderEntityManagerFactory(
14.         EntityManagerFactoryBuilder builder) {
15.         return builder
16.             .dataSource(cardholderDataSource())
17.             .packages(CardHolder.class)
18.             .build();
19.     }
20.
21.     @Bean(name = "cardEntityManagerFactory")
22.     public LocalContainerEntityManagerFactoryBean cardEntityManagerFactory(
23.         EntityManagerFactoryBuilder builder) {
24.         return builder
25.             .dataSource(cardDataSource())
26.             .packages(Card.class)
27.             .build();
28.     }
```

Transaction Management

The bean definition of a transaction manager requires a reference to the entity manager factory bean. We will use the `@Qualifier` annotation to auto-wire the entity manager specific to the data source's transaction manager.

A transaction manager is needed for each data source.

The following is a snippet of code showing the member data source transaction manager bean definition.

```
1.     @Primary
2.     @Bean
3.     public PlatformTransactionManager memberTransactionManager(
```



```

4.         final @Qualifier("memberEntityManagerFactory")
LocalContainerEntityManagerFactoryBean memberEntityManagerFactory) {
5.         return new JpaTransactionManager(memberEntityManagerFactory.getObject());
6.     }

```

JPA Repository Configuration

Since we are going to have multiple data sources we must provide the specific information for each data source repository using Spring's `@EnableJpaRepositories` annotation. In this annotation, we are going to set the reference to an entity manager, the repositories location and the reference to the transaction manager.

Below is the 'member' data source's JPA repository settings.

```

1.     @Configuration
2.     @EnableTransactionManagement
3.     @EnableJpaRepositories(basePackages =
"guru.springframework.multipledatasources.repository.member",
4.         entityManagerFactoryRef = "memberEntityManagerFactory",
5.         transactionManagerRef= "memberTransactionManager"
6.     )
7.     public class MemberDataSourceConfiguration { .... }

```

Line number 3:

`basePackages` : We use this field to set the base package of our repositories. For instance, for the member data source, it must point to the package `guru.springframework.multipledatasources.repository.member`

Line number 4:

`entityManagerFactoryRef` : We use this field to reference the entity manager factory bean defined in the data source configuration file. It is important to take note of the fact that the `entityManagerFactoryRef` value must match the bean name (if specified via the name field of the `@Bean` annotation else will default to method name) of the entity manager factory defined in the configuration file.

Line number 5:

`transactionManagerRef` : This field references the transaction manager defined in the data source configuration file. Again it is important to ensure that the `transactionManagerRef` value matches with the bean name of the transaction manager factory.

Complete Data Source Configuration File

Below is the complete data source configuration for our primary data source(member database). The complete card and cardholder configuration files are available on GitHub. They are similar to this one except that they are secondary data sources.

```

1.     @Configuration
2.     @EnableTransactionManagement
3.     @EnableJpaRepositories(basePackages =
"guru.springframework.multipledatasources.repository.member",
4.         entityManagerFactoryRef = "memberEntityManagerFactory",
5.         transactionManagerRef= "memberTransactionManager"
6.     )
7.     public class MemberDataSourceConfiguration {
8.
9.         @Bean
10.        @Primary

```



```

11.         @ConfigurationProperties("app.datasource.member")
12.         public DataSourceProperties memberDataSourceProperties() {
13.             return new DataSourceProperties();
14.         }
15.
16.         @Bean
17.         @Primary
18.         @ConfigurationProperties("app.datasource.member.configuration")
19.         public DataSource memberDataSource() {
20.             return memberDataSourceProperties().initializeDataSourceBuilder()
21.                 .type(HikariDataSource.class).build();
22.         }
23.
24.         @Primary
25.         @Bean(name = "memberEntityManagerFactory")
26.         public LocalContainerEntityManagerFactoryBean
27.             memberEntityManagerFactory(EntityManagerFactoryBuilder builder) {
28.             return builder
29.                 .dataSource(memberDataSource())
30.                 .packages(Member.class)
31.                 .build();
32.         }
33.         @Primary
34.         @Bean
35.         public PlatformTransactionManager memberTransactionManager(
36.             final @Qualifier("memberEntityManagerFactory")
37.             LocalContainerEntityManagerFactoryBean memberEntityManagerFactory) {
38.             return new JpaTransactionManager(memberEntityManagerFactory.getObject());
39.         }
40.     }

```

Important Points to note:

entity manager factory bean: Please make sure that you are referencing the correct data source when creating the entity manager factory bean otherwise you will get unexpected results.

transaction manager bean: To ensure that you have provided the correct entity manager factory reference for the transaction manager, you may use the `@Qualifier` annotation.

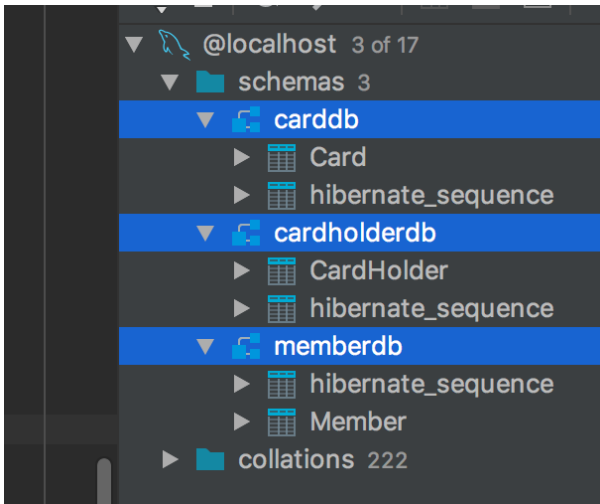
For example, the transaction manager of the ‘member’ data source will be using the entity manager factory bean with the name “memberEntityManagerFactory”.

Testing our application

After running the application, the schemas will be updated.

In this example, only one table for each datasource is created.





Spring Boot Test Class

The test class in the code snippet below contains test methods for each data source.

In each method, we are creating an object and persisting it to the database using the Spring Data JPA repository.

To verify, we check if that data is present in the database.

```

1.  @RunWith(SpringRunner.class)
2.  @SpringBootTest
3.  public class MultipledatasourcesApplicationTests {
4.
5.      /*
6.      * We will be using mysql databases we configured in our properties file for our
tests
7.      * Make sure your datasource connections are correct otherwise the test will fail
8.      * */
9.
10.     @Autowired
11.     private MemberRepository memberRepository;
12.
13.     @Autowired
14.     private CardHolderRepository cardHolderRepository;
15.
16.     @Autowired
17.     private CardRepository cardRepository;
18.
19.     private Member member;
20.     private Card card;
21.     private CardHolder cardHolder;
22.
23.     @Before
24.     public void initializeDataObjects(){
25.
26.         member = new Member();
27.         member.setMemberId("M001");
28.         member.setName("Maureen Mpofu");
29.
30.         cardHolder = new CardHolder();
31.         cardHolder.setCardNumber("4111111111111111");
32.         cardHolder.setMemberId(member.getMemberId());
33.
34.         card = new Card();
35.         card.setExpirationMonth(01);
36.         card.setExpirationYear(2020);
37.         card.setName(member.getName());
38.
39.     }
40.

```



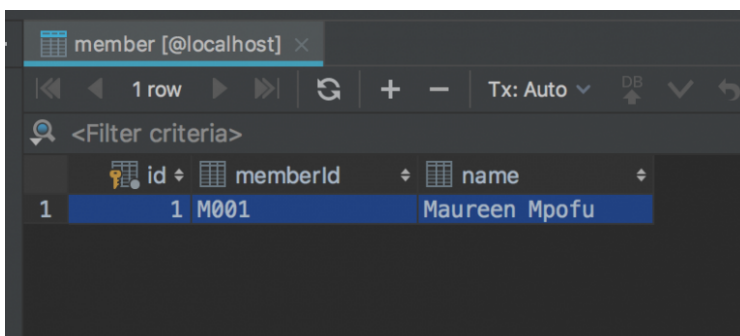
```

41.         @Test
42.         public void shouldSaveMemberToMemberDB() {
43.             Member savedMember = memberRepository.save(member);
44.             Optional<Member> memberFromDb= memberRepository.findById(savedMember.getId());
45.             assertTrue(memberFromDb.isPresent());
46.         }
47.
48.         @Test
49.         public void shouldSaveCardHolderToCardHolderDB() {
50.             CardHolder savedCardHolder = cardHolderRepository.save(cardHolder);
51.             Optional<CardHolder> cardHolderFromDb=
cardHolderRepository.findById(savedCardHolder.getId());
52.             assertTrue(cardHolderFromDb.isPresent());
53.         }
54.
55.         @Test
56.         public void shouldSaveCardToCardDB() {
57.             Card savedCard = cardRepository.save(card);
58.             Optional<Card> cardFromDb= cardRepository.findById(savedCard.getId());
59.             assertTrue(cardFromDb.isPresent());
60.         }
61.     }

```

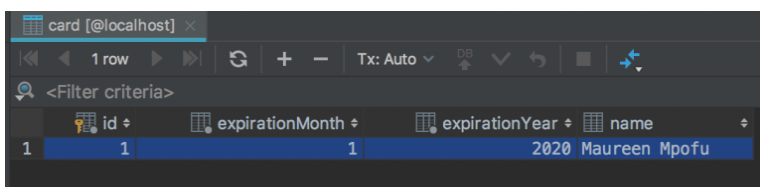
Our test cases passed and the database tables recorded the data persisted via the application(indicated by the screenshots below).

Member Database



	id	memberId	name
1	1	M001	Maureen Mpofu

Card Database



	id	expirationMonth	expirationYear	name
1	1	1	2020	Maureen Mpofu

CardHolder Database

Conclusion



When dealing with just one datasource and Spring Boot, data source configuration is simple. Spring Boot can provide a lot of auto configuration.

However, if you need to connect to multiple datasources with Spring Boot, additional configuration is needed.

You need to provide configuration data to Spring Boot, customized for each data source.

The source code of our sample application is available on GitHub. Please update the datasource to your own needs.

multiple JPA repositories

spring boot multiple data sources

About SFG Contributor

Staff writer account for Spring Framework Guru

26 comments on “How to Configure Multiple Data Sources in a Spring Boot Application”

Amir Choubani

September 24, 2019 at 11:33 am

Hi. I liked the blog but I didn't get how this import `@ConfigurationProperties("app.datasource.member.configuration")` works although it is not defined in `application.properties` file. Any idea please?

Borja Paz Rodríguez

October 19, 2019 at 11:12 am

I have the same question but I think the `*.configuration.*` part is for more advanced settings, such as the connection pool. Please, correct me if I'm wrong.

Naveen

February 12, 2021 at 10:30 pm



@ConfigurationProperties map the properties from application.properties file. In this example, it map all props start with "app.datasource.member.configuration" like user pad driverClass and spring automatically use these properties while data source creation

Naveen

February 12, 2021 at 10:29 pm

@ConfigurationProperties map the properties from application.properties file. In this example, it map all props start with "app.datasource.member.configuration" like user pad driverClass and spring automatically use these properties while data source creation

Neo

October 15, 2019 at 12:33 am

Most probably it is a mistake, you can try removing it and re running to check if it works.

Bharath Palaksha

December 13, 2019 at 4:29 am

EntityManagerFactoryBuilder bean is not defined

Bharath Palaksha

December 13, 2019 at 4:48 am

Entities are not getting scanned
java.lang.IllegalArgumentException: Not a managed type: is the error

Rod Madden

April 13, 2020 at 1:57 pm

Excellent tutorial and explanations ...well done !

jack sparrow

June 30, 2020 at 3:15 pm



org.springframework.beans.factory.BeanCreationException: Error creating bean with name 'cardEntityManagerFactory' defined in class path resource [guru/springframework/multipledatasources/configuration/CardDataSourceConfiguration.class]: Invocation of init method failed; nested exception is javax.persistence.PersistenceException: [PersistenceUnit: default] Unable to build Hibernate SessionFactory; nested exception is org.hibernate.exception.GenericJDBCException: Unable to open JDBC Connection for DDL execution

I am getting the above error. I wonder if I got the same error as me?

willem

August 17, 2020 at 4:40 pm

Thanks, a great example!
But here is my point.

After going over several others I found that that none of them implements more than one table and repo per DB.

So what happens in that case?

Does one need a configuration/ @EnableJpaRepositories for each table (declaring the same datasource).....

Kind of confussing this issue.

I Got it working inside a SpringBatch, but when I added some tables and repos I got this:

Caused by: org.springframework.beans.factory.BeanCreationException: Error creating bean with name 'socioActiveWriter' defined in file [C:\Users\artsgard\Documents\workspacejava\sociodbbatch\target\classes\com\artsgard\sociodbbatch\writers\SocioActiveWriter.class]: Invocation of init method failed; nested exception is java.lang.IllegalArgumentException: An EntityManagerFactory is required

Of course each entity/ repo needs its entitymanager, right? Who will take care of this?

You'll find the code here:

<https://github.com/thollem/sociodbbatch>

willem



It is hard to fall in love with Spring

willem

August 18, 2020 at 6:24 am

Concerning my previous post, I have to apologize!

The problem is not the dual/multi DB connection but the Spring-Batch part (transactions and entitymanager)

But anyway, any comment is welcome

willem

Sandeep shelar

September 26, 2020 at 4:24 pm

How to get session from session factory for each datasource as we have 3 different entity manager and through out application we have only one session factory.

Robert

November 3, 2020 at 11:26 am

Great explanation!! Saved me a lot of time!

Renan

December 7, 2020 at 6:14 pm

Hi. From where did you get "BasicDataSource.class"? I'm following your tutorial with two Oracle databases and I couldn't understand this part. Thank you

spanky

December 11, 2020 at 7:34 pm

Hi, Is it possible to join two entities from different Datasource?



Sudan Shrestha

December 14, 2020 at 12:22 am

How can we use a json file instead of application.properties to extract configurations?

Dominik

December 15, 2020 at 10:16 am

Thanks for the article.

Just a question: What happens when a transaction for one datasource fails – will a rollback happen for the transactions of the other datasources?

I. e. suppose we have the following class:

```
@Service
public class FooService {
    @Autowired
    private MemberRepository memberRepository;
    @Autowired
    private CardHolderRepository cardHolderRepository;
    @Autowired
    private CardRepository cardRepository;

    @Transactional
    public void createMemberWithCard(...) {
        memberRepository.save(...);
        cardHolderRepository.save(...);
        cardRepository.save(...);
    }
}
```

Then, FooService.createMemberWithCard() is called but cardRepository.save() fails (e.g. because of a connection issue) and the _cardTransactionManager_ will rollback the transaction for _cardDataSource_.

****Will the transactions for _memberDataSource_ and _cardHolderDataSource_ also be rolled back?****

shajahan

December 22, 2020 at 12:25 am



How to connect to multiple databases dynamically as connection string is dynamic which is available on runtime (not static). (With one permanent connection) ?

Bender

December 23, 2020 at 3:29 am

Thank you and very nice Tutorial, but do you have a repo for that (the repo you linked is STUFFED with so much else) ?

Would be nice if one could just clone any of your examples (kind of hard to do if they are all in 1 Repo).

mahdi

December 29, 2020 at 2:46 am

My Exception in project two db each db two table:

org.hibernate.AnnotationException: @OneToOne or @ManyToOne on

ir.fava.citybank.models.tbluser.TblUser.tblRoleType references an unknown entity:

ir.fava.citybank.models.tblroletype.TblRoleType

Alex82

January 25, 2021 at 10:17 am

i have the same problem. Did you solve?

Gnanasekaran

February 17, 2021 at 3:33 am

Thank you so much. This is very nice and useful tutorials.

Vicki

March 12, 2021 at 6:48 am

Thank you for the great post! It helped me a lot.

If you use in-memory databases for testing you would need to declare DataSourceInitializer for each db config to create and populate from the given sql files.

I have also added Qualifier to the second dataSource as I pass it to some other beans.

```
@Bean
@Profile("test")
public DataSourceInitializer
secondDBDataSourceInitializer(@Qualifier("secondDBDataSource") DataSource
datasource) {
    ResourceDatabasePopulator resourceDatabasePopulator = new
    ResourceDatabasePopulator();
    resourceDatabasePopulator.addScript(new ClassPathResource("schema2-
h2.sql"));
    resourceDatabasePopulator.addScript(new ClassPathResource("data2-h2.sql"));

    DataSourceInitializer dataSourceInitializer = new DataSourceInitializer();
    dataSourceInitializer.setDataSource(datasource);
    dataSourceInitializer.setDatabasePopulator(resourceDatabasePopulator);
    return dataSourceInitializer;
}
```

Gurjit

March 17, 2021 at 11:14 am

Not working. Give exception when using the datasource to initializing the proc bean

Skipa

March 16, 2022 at 7:08 am

Do you have one with multiple datasources but one datasource writes data to another database? for example, Cardholder is the datasource and we writing the contents of the cardholder DB to Member DB?

This site uses Akismet to reduce spam. [Learn how your comment data is processed.](#)

Copyright © 2021 Spring Framework Guru All Rights Reserved.

