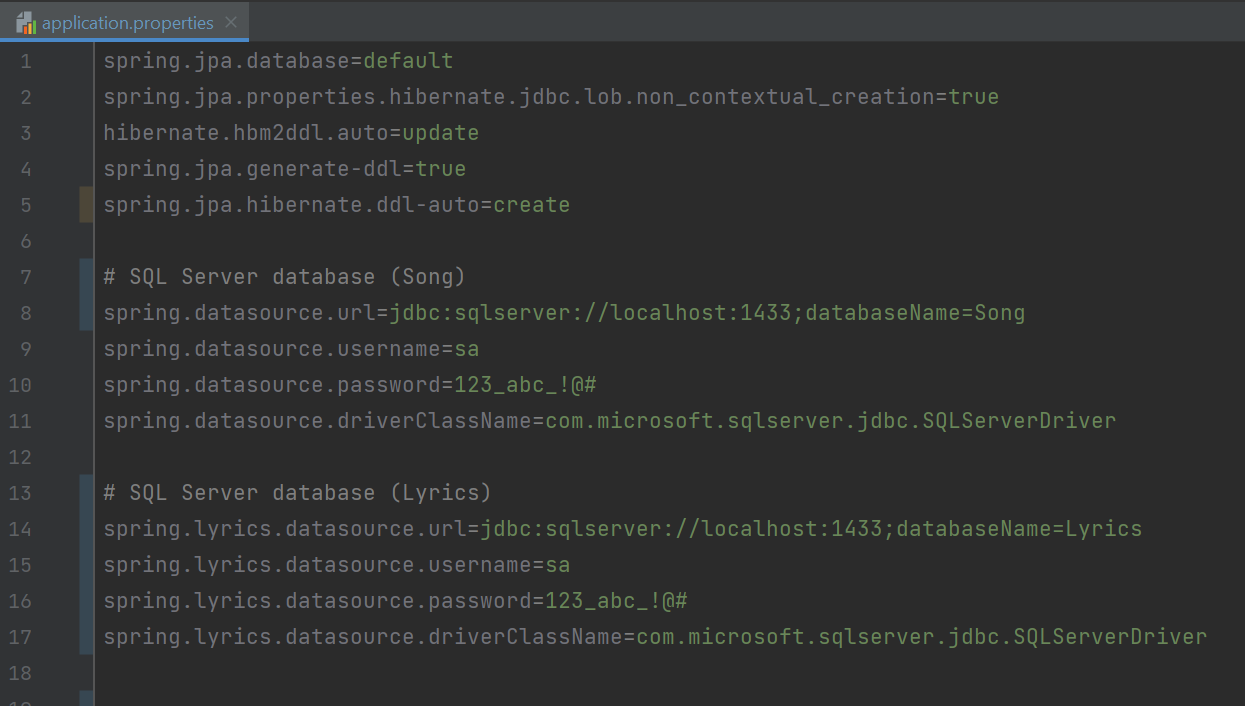
**Spring Boot Application With Multiple Data sources**

To make a spring boot application that can communicate with more than one database you need to define some configuration to let spring boot when to execute each source.

let’s agree first that your /**models** and /**repositories** and /**controllers** are already defined as you always do.

now let’s take a walk through the **application.properties** file



as you can see I have set the configuration for two databases. The first one is for **Song** and the second one for **Lyrics**. Note that the Song configuration prefixed with **(spring.datasource.\*)** and Lyrics configuration prefixed with **(spring.lyrics.datasource.\*)**, making the Song prefixed with (spring.song.datasource) would lead to error due to (spring.datasource.\*) not provided so at least one of the databases should act like if it was the default database.

You should organize your models and repositories directories like this; in order to let the configuration code knows which model and repo to apply the data source to.

Graphical user interface, text, application

Description automatically generated

now we’re done organizing the directories and submitting the database config in the application.properties we’re left with the spring boot configuration code for the data sources.

first, create a directory /**config** and create two java files for each datasource you have. mine will be **SongDBConfig.java** and **LyricsDBConfig.java**

let’s start with SongDBConfig.java

@Primary  
@Bean(name="songProps")  
@ConfigurationProperties("spring.datasource")  
public DataSourceProperties dataSourceProperties() {  
 return new DataSourceProperties();  
}  
  
@Primary  
@Bean(name="datasource")  
@ConfigurationProperties(prefix = "spring.datasource")  
public DataSource datasource(@Qualifier("songProps") DataSourceProperties properties){  
 return properties.initializeDataSourceBuilder().build();  
}  
  
@Primary  
@Bean(name="entityManagerFactory")  
public LocalContainerEntityManagerFactoryBean entityManagerFactoryBean  
 (EntityManagerFactoryBuilder builder,  
 @Qualifier("datasource") DataSource dataSource){  
 return builder.dataSource(dataSource)  
 .packages("com.example.multipledb.models.song")  
 .persistenceUnit("Song").build();  
}  
  
@Primary  
@Bean(name = "transactionManager")  
@ConfigurationProperties("spring.jpa")  
public PlatformTransactionManager transactionManager(  
 @Qualifier("entityManagerFactory") EntityManagerFactory entityManagerFactory) {  
 return new JpaTransactionManager(entityManagerFactory);  
}

since Song db is our **default** data source it is a must to include “**Primary**” annotation which defines that this bean going to be registered more than one time but I want this version to be the primary one. So, in the other class LyricsDBConfig.java the bean **won’t** have the “Primary” annotation.

For the “ConfigurationProperties” annotation we used it to let spring boot know which configuration from application.properties should it use.

and let’s not forget to annotate our class with these annotations

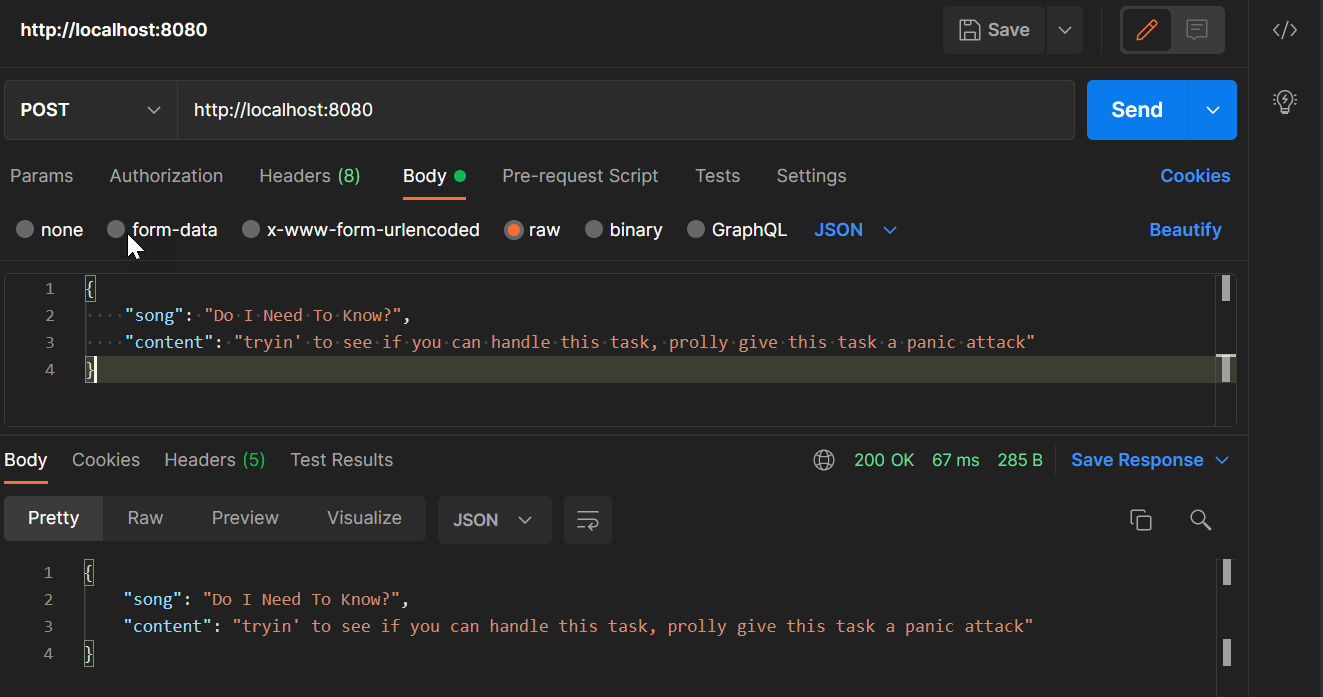
@Configuration  
@EnableTransactionManagement  
@EnableJpaRepositories(  
 entityManagerFactoryRef = "entityManagerFactory",  
 transactionManagerRef = "transactionManager",  
 basePackages = { "com.example.multipledb.repositories.song" })  
public class SongDBConfig {  
  
 @Primary  
 @Bean(name="songProps")  
 @ConfigurationProperties("spring.datasource")  
 public DataSourceProperties dataSourceProperties() {  
 return new DataSourceProperties();  
 }  
  
 @Primary  
 @Bean(name="datasource")  
 @ConfigurationProperties(prefix = "spring.datasource")  
 public DataSource datasource(@Qualifier("songProps") DataSourceProperties properties){  
 return properties.initializeDataSourceBuilder().build();  
 }  
  
 @Primary  
 @Bean(name="entityManagerFactory")  
 public LocalContainerEntityManagerFactoryBean entityManagerFactoryBean  
 (EntityManagerFactoryBuilder builder,  
 @Qualifier("datasource") DataSource dataSource){  
 return builder.dataSource(dataSource)  
 .packages("com.example.multipledb.models.song")  
 .persistenceUnit("Song").build();  
 }  
  
 @Primary  
 @Bean(name = "transactionManager")  
 @ConfigurationProperties("spring.jpa")  
 public PlatformTransactionManager transactionManager(  
 @Qualifier("entityManagerFactory") EntityManagerFactory entityManagerFactory) {  
 return new JpaTransactionManager(entityManagerFactory);  
 }  
}

for the LyricsDBConfig.java it will be the same as SongDBConfig.java but we will prefix the entityManagerFactory and transactionManager with “**lyrics**” cause they’re already taken by the SongDBConfig.java

@Configuration  
@EnableTransactionManagement  
@EnableJpaRepositories(  
 entityManagerFactoryRef = "lyricsEntityManagerFactory",  
 transactionManagerRef = "lyricsTransactionManager",  
 basePackages = { "com.example.multipledb.repositories.lyrics" })  
public class LyricsDBConfig {  
  
 @Bean(name="lyricsProps")  
 @ConfigurationProperties("spring.lyrics.datasource")  
 public DataSourceProperties dataSourceProperties() {  
 return new DataSourceProperties();  
 }  
  
  
 @Bean(name="lyricsDatasource")  
 @ConfigurationProperties(prefix = "spring.lyrics.datasource")  
 public DataSource datasource(@Qualifier("lyricsProps") DataSourceProperties properties){  
 return properties.initializeDataSourceBuilder().build();  
 }  
  
  
 @Bean(name="lyricsEntityManagerFactory")  
 public LocalContainerEntityManagerFactoryBean entityManagerFactoryBean  
 (EntityManagerFactoryBuilder builder,  
 @Qualifier("lyricsDatasource") DataSource dataSource){  
 return builder.dataSource(dataSource)  
 .packages("com.example.multipledb.models.lyrics")  
 .persistenceUnit("Lyrics").build();  
 }  
  
  
 @Bean(name = "lyricsTransactionManager")  
 @ConfigurationProperties("spring.jpa")  
 public PlatformTransactionManager transactionManager(  
 @Qualifier("lyricsEntityManagerFactory") EntityManagerFactory entityManagerFactory) {  
 return new JpaTransactionManager(entityManagerFactory);  
 }  
}

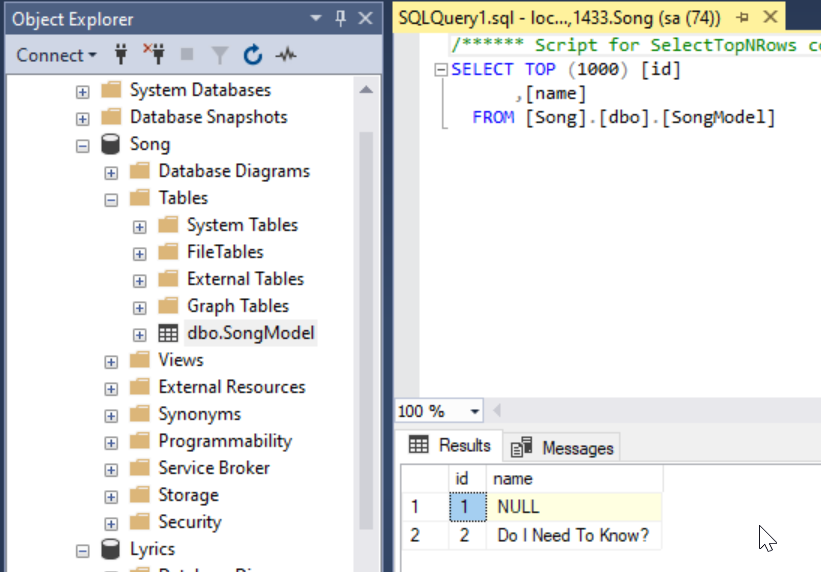
now the controller will call each repo and by their turn will communicate the selected DB.

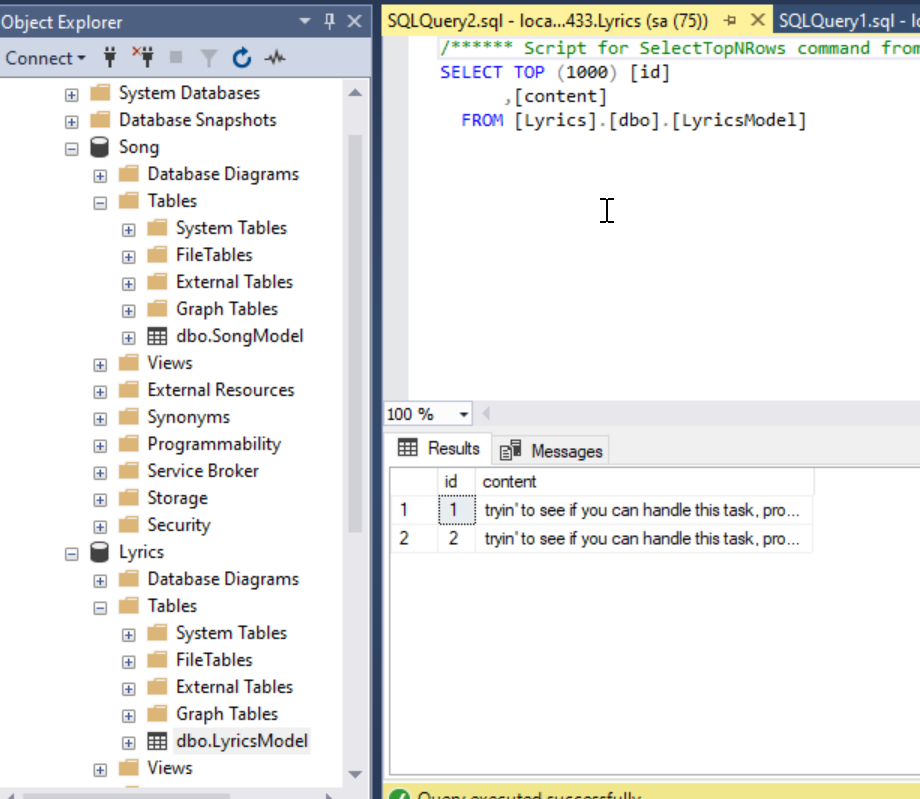
@RestController  
public class Controller {  
 @Autowired  
 LyricsRepo lyricsRepo;  
 @Autowired  
 SongRepo songRepo;  
  
 @PostMapping  
 public ResponseEntity add(@RequestBody request body){  
 SongModel songModel = new SongModel();  
 LyricsModel lyricsModel = new LyricsModel();  
 songModel.setName(body.getSong());  
 lyricsModel.setContent(body.getContent());  
  
// will insert in Song DB  
 songRepo.save(songModel);  
// will insert in Lyrics DB  
 lyricsRepo.save(lyricsModel);  
  
 return ResponseEntity.*ok*(body);  
 }  
}



Graphical user interface, application

Description automatically generated





source code: <https://gitlab.com/lamoboos223/spring-boot-multiple-datasource-song-project>