Configuration

Configuration
Annotation-based Configuration
Spring Namespace
Custom Namespace Attributes
Bootstrap Mode
Recommendations

Edit this Page
GitHub Project
Stack Overflow

This section describes configuring Spring Data JPA through either:

- "Annotation-based Configuration" (Java configuration)
- "Spring Namespace" (XML configuration)

Annotation-based Configuration

The Spring Data JPA repositories support can be activated through both JavaConfig as well as a custom XML namespace, as shown in the following example:

Example 1. Spring Data JPA repositories using JavaConfig

```java
@Configuration
@EnableJpaRepositories
@EnableTransactionManagement
class ApplicationConfig {

  @Bean
  public DataSource dataSource() {

    EmbeddedDatabaseBuilder builder = new EmbeddedDatabaseBuilder();
    return builder.setType(EmbeddedDatabaseType.HSQL).build();
  }

  @Bean
  public LocalContainerEntityManagerFactoryBean entityManagerFactory() {

    HibernateJpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter();
    vendorAdapter.setGenerateDdl(true);

    LocalContainerEntityManagerFactoryBean factory = new LocalContainerEntityManager
    factory.setJpaVendorAdapter(vendorAdapter);
    factory.setPackagesToScan("com.acme.domain");
    factory.setDataSource(dataSource());
    return factory;
  }

  @Bean
  public PlatformTransactionManager transactionManager(EntityManagerFactory entityM

    JpaTransactionManager txManager = new JpaTransactionManager();
    txManager.setEntityManagerFactory(entityManagerFactory);
    return txManager;
  }
}
```

NOTE

You must create `LocalContainerEntityManagerFactoryBean` and not `EntityManagerFactory` directly, since the former also participates in exception translation mechanisms in addition to creating `EntityManagerFactory`.

The preceding configuration class sets up an embedded HSQL database by using the `EmbeddedDatabaseBuilder` API of `spring-jdbc`. Spring Data then sets up an `EntityManagerFactory` and uses Hibernate as the sample persistence provider. The last infrastructure component declared here is the `JpaTransactionManager`. Finally, the example activates Spring Data JPA repositories by using the `@EnableJpaRepositories` annotation, which essentially carries the same attributes as the XML namespace. If no base package is configured, it uses the one in which the configuration class resides.

Spring Namespace

The JPA module of Spring Data contains a custom namespace that allows defining repository beans. It also contains certain features and element attributes that are special to JPA. Generally, the JPA repositories can be set up by using the `repositories` element, as shown in the following example:

Example 2. Setting up JPA repositories by using the namespace

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:jpa="http://www.springframework.org/schema/data/jpa"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    https://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/data/jpa
    https://www.springframework.org/schema/data/jpa/spring-jpa.xsd">

  <jpa:repositories base-package="com.acme.repositories" />

</beans>
```

TIP

Which is better, JavaConfig or XML? XML is how Spring was configured long ago. In today's era of fast-growing Java, record types, annotations, and more, new projects typically use as much pure Java as possible. While there is no immediate plan to remove XML support, some of the newest features MAY not be available through XML.

Using the `repositories` element it activates persistence exception translation for all beans annotated with `@Repository`, to let exceptions being thrown by the JPA persistence providers be converted into Spring's `DataAccessException` hierarchy.

Custom Namespace Attributes

Beyond the default attributes of the `repositories` element, the JPA namespace offers additional attributes to let you gain more detailed control over the setup of the repositories:

Table 1. Custom JPA-specific attributes of the `repositories` element

`entity-manager-factory-ref`	Explicitly wire the `EntityManagerFactory` to be used with the repositories being detected by the `repositories` element. Usually used if multiple `EntityManagerFactory` beans are used within the application. If not configured, Spring Data automatically looks up the `EntityManagerFactory` bean with the name `entityManagerFactory` in the `ApplicationContext`.
`transaction-manager-ref`	Explicitly wire the `PlatformTransactionManager` to be used with the repositories being detected by the `repositories` element. Usually only necessary if multiple transaction managers or `EntityManagerFactory` beans have been configured. Default to a single defined `PlatformTransactionManager` inside the current `ApplicationContext`.

NOTE

Spring Data JPA requires a `PlatformTransactionManager` bean named `transactionManager` to be present if no explicit `transaction-manager-ref` is defined.

Bootstrap Mode

By default, Spring Data JPA repositories are default Spring beans. They are singleton scoped and eagerly initialized. During startup, they already interact with the JPA `EntityManager` for verification and metadata analysis purposes. Spring Framework supports the initialization of the JPA `EntityManagerFactory` in a background thread because that process usually takes up a significant amount of startup time in a Spring application. To make use of that background initialization effectively, we need to make sure that JPA repositories are initialized as late as possible.

As of Spring Data JPA 2.1 you can now configure a `BootstrapMode` (either via the `@EnableJpaRepositories` annotation or the XML namespace) that takes the following values:

- `DEFAULT` (default) — Repositories are instantiated eagerly unless explicitly annotated with `@Lazy`. The lazification only has effect if no client bean needs an instance of the repository as that will require the initialization of the repository bean.
- `LAZY` — Implicitly declares all repository beans lazy and also causes lazy initialization proxies to be created to be injected into client beans. That means, that repositories will not get instantiated if the client bean is simply storing the instance in a field and not making use of the repository during initialization. Repository instances will be initialized and verified upon first interaction with the repository.
- `DEFERRED` — Fundamentally the same mode of operation as `LAZY`, but triggering repository initialization in response to an `ContextRefreshedEvent` so that repositories are verified before the application has completely started.

Recommendations

If you're not using asynchronous JPA bootstrap stick with the default bootstrap mode.

In case you bootstrap JPA asynchronously, `DEFERRED` is a reasonable default as it will make sure the Spring Data JPA bootstrap only waits for the `EntityManagerFactory` setup if that itself takes longer than initializing all other application components. Still, it makes sure that repositories are properly initialized and validated before the application signals it's up.

`LAZY` is a decent choice for testing scenarios and local development. Once you are pretty sure that repositories can properly bootstrap, or in cases where you are testing other parts of the application, running verification for all repositories might unnecessarily increase the startup time. The same applies to local development in which you only access parts of the application that might need to have a single repository initialized.