Spring Data Jpa vs Jpa vs Hibernate And Hibernate caching

https://medium.com/javarevisited/difference-between-hibernate-jpa-and-spring-data-jpa-7df55717692f

https://javarevisited.blogspot.com/2017/03/difference-between-first-and-second-level-cache-in-Hibernate.html

Configuring Second level Cache:

To enable the second level cache in a Hibernate-based application, you typically need to follow these steps:

1. **Configure Hibernate Properties**: First, you need to configure Hibernate properties to enable and configure the second level cache. This configuration is typically done in the `application.properties` or `application.yml` file.

   For example, in `application.properties`, you would add properties like these:

   ```properties
   # Enable the second level cache
   spring.jpa.properties.hibernate.cache.use_second_level_cache=true

   # Specify the cache provider (e.g., Ehcache, Hazelcast, etc.)
spring.jpa.properties.hibernate.cache.region.factory_class=org.hibernate.cache.ehcache.EhCacheRegionFactory

   # Optionally, configure caching settings
   spring.jpa.properties.hibernate.cache.use_query_cache=true
   spring.jpa.properties.hibernate.cache.use_minimal_puts=true
   ```

   Adjust the cache provider (`hibernate.cache.region.factory_class`) according to the caching provider you want to use. For example, `org.hibernate.cache.ehcache.EhCacheRegionFactory` for Ehcache.

2. **Annotate Entities**: To specify which entities should be cached, you can use Hibernate's `@Cacheable` annotation on entity classes.

   ```java
   import javax.persistence.Cacheable;
   import javax.persistence.Entity;

   @Entity
   ```

```
   @Cacheable
   public class YourEntity {
       // Entity attributes and methods
   }
```

   By default, Hibernate caches entities at the entity level. You can also cache collections by applying `@Cache` annotations to collection mappings in your entity classes.

3. **Configure Cache Providers**: Depending on the cache provider you choose, you may need to configure additional settings. For example, if you're using Ehcache, you'll need to provide an `ehcache.xml` configuration file to specify cache settings such as cache regions, eviction policies, and time-to-live settings.

4. **Verify Cache Usage**: After configuring the second level cache, you can verify its usage by monitoring the Hibernate logs or using caching monitoring tools provided by your chosen cache provider.

By following these steps, you can enable and configure the second level cache in your Hibernate-based application, which can help improve performance by reducing the number of database queries and improving overall application responsiveness.

Other way of doing:

Yes, there are alternative ways to enable the second level cache in Hibernate-based applications. One common approach is to configure the second level cache programmatically using Hibernate's `SessionFactory` or `EntityManagerFactory` configuration.

Here's how you can do it programmatically:

1. **Create Cache Configuration**: Create a cache configuration object to specify the cache provider and caching settings.

```java
   import org.hibernate.cache.ehcache.EhCacheRegionFactory;
   import org.hibernate.cfg.Configuration;

   public class HibernateConfig {

       public static Configuration configure() {
         Configuration configuration = new Configuration();
         configuration.setProperty("hibernate.cache.use_second_level_cache",
"true");
         configuration.setProperty("hibernate.cache.region.factory_class",
EhCacheRegionFactory.class.getName());
         // Optionally configure other caching settings
         return configuration;
      }
   }
```

2. **Build SessionFactory**: Use the cache configuration to build the `SessionFactory` instance.

   ```java
   import org.hibernate.SessionFactory;
   import org.hibernate.boot.MetadataSources;
   import org.hibernate.boot.registry.StandardServiceRegistry;
   import org.hibernate.boot.registry.StandardServiceRegistryBuilder;

   public class HibernateUtil {

      private static final SessionFactory sessionFactory = buildSessionFactory();

      private static SessionFactory buildSessionFactory() {
         StandardServiceRegistry registry = new
StandardServiceRegistryBuilder()
               .applySettings(HibernateConfig.configure().getProperties())
               .build();
         return new
MetadataSources(registry).buildMetadata().buildSessionFactory();
      }

      public static SessionFactory getSessionFactory() {
         return sessionFactory;
      }
   }
   ```

3. **Enable Caching for Entities**: In your entity classes, enable caching by specifying the `@Cacheable` annotation.

   ```java
   import javax.persistence.Cacheable;
   import javax.persistence.Entity;

   @Entity
   ```

```java
   @Cacheable
   public class YourEntity {
       // Entity attributes and methods
   }
```

4. **Use SessionFactory**: Use the `SessionFactory` instance to obtain `Session` objects and perform database operations.

```java
   import org.hibernate.Session;

   public class Main {

       public static void main(String[] args) {
           SessionFactory sessionFactory = HibernateUtil.getSessionFactory();
           try (Session session = sessionFactory.openSession()) {
               // Perform database operations using Hibernate session
           }
       }
   }
```

By configuring the second level cache programmatically, you have more flexibility to customize caching settings and integrate with different cache providers. However, this approach requires more manual configuration and management compared to using properties-based configuration. Choose the approach that best suits your application's requirements and architecture.

In Hibernate, the `StandardServiceRegistry` is an important component used for bootstrapping and managing the configuration settings needed by Hibernate. It acts as a central registry for various services required by Hibernate during the bootstrapping process and throughout the lifetime of the application.

Here's a breakdown of its key features and responsibilities:

1. **Centralized Configuration**: The `StandardServiceRegistry` holds configuration settings for Hibernate, such as database connection properties, caching options, dialect settings, and other environment-specific parameters.

2. **Service Management**: It manages various services required by Hibernate, such as connection pooling, transaction management, metadata sources, and caching. These services are typically registered and configured during the initialization phase of Hibernate.

3. **Builder Pattern**: Hibernate provides a `StandardServiceRegistryBuilder`

class to construct and configure the `StandardServiceRegistry`. This builder pattern allows you to specify and customize various settings and services before building the registry.

4. **Lifecycle Management**: The `StandardServiceRegistry` has lifecycle management capabilities, allowing it to be created, configured, and closed gracefully. It ensures that resources are properly initialized and released to prevent memory leaks or resource leaks.

5. **Immutable Configuration**: Once built, the `StandardServiceRegistry` is immutable, meaning its configuration cannot be modified after creation. This ensures consistency and stability of the Hibernate environment throughout the application lifecycle.

Overall, the `StandardServiceRegistry` plays a crucial role in managing the configuration and services required by Hibernate, providing a unified and standardized approach to bootstrapping Hibernate in different environments and applications.

More about @Cacheable and @Cache

Yes, you're correct. In addition to configuring the second level cache at the session factory level, Hibernate also provides the `@Cacheable` annotation at the entity level to enable caching for individual entities. This annotation allows you to specify which entities should be cached and how they should be cached.

Here's an overview of how `@Cacheable` works:

1. **Entity-Level Caching**: By annotating an entity class with `@Cacheable`, you indicate to Hibernate that instances of this entity should be cached in the second level cache.

   ```java
   import javax.persistence.Cacheable;
   import javax.persistence.Entity;

   @Entity
   @Cacheable
   public class YourEntity {
       // Entity attributes and methods
   }
   ```

   With this annotation, Hibernate will cache instances of `YourEntity` in the second level cache.

2. **Fine-Grained Control**: The `@Cacheable` annotation provides options for fine-grained control over caching behavior. For example, you can specify the cache region name and various caching options using attributes of the `@Cacheable` annotation.

   ```java
   @Cacheable(region = "entityCache", cacheConcurrencyStrategy =
CacheConcurrencyStrategy.READ_WRITE)
   ```

   This allows you to customize caching settings for each entity based on your application's requirements.

3. **Cache Concurrency Strategy**: You can also specify the cache concurrency strategy for the entity cache. Hibernate provides several built-in cache concurrency strategies, such as `READ_ONLY`, `NONSTRICT_READ_WRITE`, and `READ_WRITE`, each with different characteristics for handling concurrent access to cached data.

   ```java
   import org.hibernate.annotations.CacheConcurrencyStrategy;

   @Cacheable
   @org.hibernate.annotations.Cache(usage =
CacheConcurrencyStrategy.READ_WRITE)
   public class YourEntity {
       // Entity attributes and methods
   }
   ```

   This annotation configuration allows you to control how Hibernate manages concurrent access to cached entity data.

By using the `@Cacheable` annotation, you can selectively enable caching for specific entities in your Hibernate application, providing more control and flexibility over the caching behavior at the entity level.

In Hibernate, the `region` attribute in the `@Cacheable` annotation is used to specify the name of the cache region where instances of the annotated entity will be cached.

A cache region is a logical grouping of cached data within the second level cache. By specifying a region name, you can organize cached data into separate regions based on your application's requirements. This allows you to control caching behavior at a more granular level and provides flexibility in managing cached data.

Here's how the `region` attribute is typically used in the `@Cacheable` annotation:

```java
import javax.persistence.Cacheable;
import org.hibernate.annotations.Cache;
import org.hibernate.annotations.CacheConcurrencyStrategy;

@Entity
@Cacheable
@Cache(region = "entityCache", usage =
CacheConcurrencyStrategy.READ_WRITE)
public class YourEntity {
    // Entity attributes and methods
}
```

In this example:

- `region = "entityCache"`: Specifies the name of the cache region as "entityCache". This indicates that instances of the `YourEntity` class will be cached in the "entityCache" cache region.

- `usage = CacheConcurrencyStrategy.READ_WRITE`: Specifies the cache concurrency strategy for the cache region. In this case, the `READ_WRITE` strategy is used, which allows concurrent read and write access to cached data.

By specifying a cache region name, you can control caching behavior for individual entities or groups of entities, such as setting different expiration policies, eviction policies, or other caching settings specific to that region. This allows you to optimize caching performance and behavior based on your application's requirements and usage patterns.