Hibernate

https://docs.spring.io/spring-framework/reference/data-access/orm/hibernate.html
https://docs.jboss.org/hibernate/orm/6.2/userguide/html_single/Hibernate_User_Guide.html#architecture
https://thorben-janssen.com/entity-mappings-introduction-jpa-fetchtypes/
https://thorben-janssen.com/5-ways-to-initialize-lazy-relations-and-when-to-use-them/
https://vladmihalcea.com/a-beginners-guide-to-jpa-and-hibernate-cascade-types/
https://vladmihalcea.com/jpa-default-fetch-plan/
https://vladmihalcea.com/the-best-way-to-map-a-onetoone-relationship-with-jpa-and-hibernate/
https://www.baeldung.com/hibernate-lazy-eager-loading
https://vladmihalcea.com/hibernate-lazytoone-annotation/

- **Spring Configuration**:
  In your Spring configuration class, you can enable JPA support by using @EnableJpaRepositories and specifying the base package where your JPA repositories are located. Additionally, configure the LocalContainerEntityManagerFactoryBean to specify the persistence unit name and JPA properties.
  java

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;
import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;

import javax.sql.DataSource;

@Configuration
@EnableJpaRepositories(basePackages = "com.example.repository")
public class JpaConfig {

    @Bean
    public LocalContainerEntityManagerFactoryBean
entityManagerFactory(DataSource dataSource) {
        LocalContainerEntityManagerFactoryBean entityManagerFactory = new
LocalContainerEntityManagerFactoryBean();
        entityManagerFactory.setDataSource(dataSource);
        entityManagerFactory.setPersistenceUnitName("myPersistenceUnit");
        // Additional JPA properties can be set here
        return entityManagerFactory;
    }
}
```

DataSource Configuration:
Configure a DataSource bean to provide database connectivity. You can use any supported DataSource implementation, such as BasicDataSource, DriverManagerDataSource, or a connection pool like HikariDataSource.
java

```
import org.apache.commons.dbcp2.BasicDataSource;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class DataSourceConfig {

    @Bean
    public DataSource dataSource() {
        BasicDataSource dataSource = new BasicDataSource();
        dataSource.setDriverClassName("com.mysql.cj.jdbc.Driver");
        dataSource.setUrl("jdbc:mysql://localhost:3306/mydatabase");
        dataSource.setUsername("username");
        dataSource.setPassword("password");
        return dataSource;
    }
}
```

- **Entity Classes**:
  Define your JPA entity classes annotated with @Entity, @Table, @Id, and other JPA annotations as needed.
- **Repository Interfaces**:
  Create repository interfaces extending JpaRepository or CrudRepository for data access operations. These interfaces provide methods for common CRUD operations and can be customized as required.

That's it! With these configurations in place, Hibernate ORM will be used as the JPA provider in your Spring application, and you can perform database operations using JPA repositories and entity managers.

In the context of Spring applications, `HibernateJpaVendorAdapter` is commonly used when configuring JPA-related beans programmatically. It allows you to specify various Hibernate-specific properties and settings for the

JPA EntityManagerFactory.

Here's where `HibernateJpaVendorAdapter` is typically used:

1. **JPA Configuration**:

   When configuring the `LocalContainerEntityManagerFactoryBean`, you can set the `JpaVendorAdapter` to specify the JPA provider (in this case, Hibernate) and provide additional vendor-specific properties.

   ```java
   import org.springframework.context.annotation.Bean;
   import org.springframework.context.annotation.Configuration;
   import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;
   import org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter;

   import javax.sql.DataSource;

   @Configuration
   public class JpaConfig {

       @Bean
       public LocalContainerEntityManagerFactoryBean entityManagerFactory(DataSource dataSource) {
           LocalContainerEntityManagerFactoryBean entityManagerFactory = new LocalContainerEntityManagerFactoryBean();
           entityManagerFactory.setDataSource(dataSource);
           entityManagerFactory.setPersistenceUnitName("myPersistenceUnit");

           // Set JPA vendor adapter (HibernateJpaVendorAdapter)
           HibernateJpaVendorAdapter jpaVendorAdapter = new HibernateJpaVendorAdapter();
           jpaVendorAdapter.setGenerateDdl(true);
           jpaVendorAdapter.setShowSql(true);
           entityManagerFactory.setJpaVendorAdapter(jpaVendorAdapter);

           return entityManagerFactory;
       }
   }
   ```

   In this example, we're creating a `HibernateJpaVendorAdapter` instance, setting properties like `generateDdl` and `showSql`, and then assigning it to the `LocalContainerEntityManagerFactoryBean`.

2. **Additional Properties**:

You can also configure additional properties specific to Hibernate through the `HibernateJpaVendorAdapter`.

```java
HibernateJpaVendorAdapter jpaVendorAdapter = new HibernateJpaVendorAdapter();
jpaVendorAdapter.setDatabase(Database.MYSQL);
jpaVendorAdapter.setDatabasePlatform("org.hibernate.dialect.MySQLDialect");
// Set other Hibernate-specific properties
```

This allows you to customize Hibernate behavior such as the database dialect, batch size, and other properties.

In summary, `HibernateJpaVendorAdapter` is used to configure Hibernate-specific properties and settings when setting up the JPA EntityManagerFactory in a Spring application. It provides a convenient way to customize Hibernate behavior without directly manipulating the EntityManagerFactory's properties.

Before the widespread adoption of `HibernateJpaVendorAdapter`, developers often configured Hibernate-specific properties directly on the `LocalContainerEntityManagerFactoryBean` without using a dedicated adapter class. This approach required setting Hibernate properties individually, which could be more verbose and less standardized across applications.

For example, instead of using `HibernateJpaVendorAdapter` to set properties like `generateDdl` and `showSql`, developers would configure these properties directly on the `LocalContainerEntityManagerFactoryBean`:

```java
LocalContainerEntityManagerFactoryBean entityManagerFactory = new LocalContainerEntityManagerFactoryBean();
entityManagerFactory.setDataSource(dataSource);
entityManagerFactory.setPersistenceUnitName("myPersistenceUnit");

// Set Hibernate-specific properties directly
Properties jpaProperties = new Properties();
jpaProperties.setProperty("hibernate.show_sql", "true");
jpaProperties.setProperty("hibernate.hbm2ddl.auto", "create");
entityManagerFactory.setJpaProperties(jpaProperties);
```

While this approach is still valid, using `HibernateJpaVendorAdapter` offers a more standardized and concise way to configure Hibernate-specific properties, especially as it encapsulates many common settings into a single adapter class. Additionally, it provides better support for switching between different JPA providers if needed.

`hibernate.hbm2ddl.auto` and `hibernate.show_sql` are Hibernate-specific properties used to control aspects of database schema generation and SQL logging during application runtime.

1. **hibernate.hbm2ddl.auto**:
   - This property controls the automatic generation of database schema based on the mapping metadata provided by Hibernate.
   - Possible values include:
     - `create`: Hibernate creates the database schema from scratch every time the application starts. This involves dropping existing tables and recreating them based on the entity mappings.
     - `update`: Hibernate updates the database schema to match the entity mappings. It adds new tables or columns and updates existing ones, but it does not drop any tables.
     - `validate`: Hibernate validates the database schema against the entity mappings. It checks if the schema matches the mappings but does not make any changes to the database.
     - `none`: Hibernate does not perform any automatic schema generation or validation.
   - It's important to be cautious when using `hibernate.hbm2ddl.auto`, especially in production environments, as it can lead to data loss or corruption if not used properly.

2. **hibernate.show_sql**:
   - This property determines whether Hibernate logs SQL statements executed by the application.
   - When set to `true`, Hibernate prints SQL statements to the console or logging framework. This can be useful for debugging and understanding the interaction between the application and the database.
   - In production environments, it's typically set to `false` to reduce log verbosity and improve performance.

These properties are typically configured as part of the Hibernate configuration either in the `hibernate.cfg.xml` file or through properties passed programmatically when configuring the `SessionFactory` or `EntityManagerFactory`.

In Hibernate, the `@Column` annotation is used to specify the column mapping details such as name, type, length, etc. Within the `@Column` annotation, the `columnDefinition` attribute allows you to specify the SQL column definition, including the data type.

For example:

```java
@Column(name = "age", columnDefinition = "INTEGER")
private int age;
```

In this example, the `columnDefinition` attribute specifies that the column named "age" should be of type `INTEGER` in the underlying database.

The `columnDefinition` attribute accepts a string value representing the SQL data type. This value should be compatible with the SQL dialect of the underlying database system. Hibernate then translates this SQL data type to the appropriate column type for the chosen database.

It's important to note that using `columnDefinition` provides flexibility, but it can also make your application less portable across different database systems, as you're specifying SQL-specific details. If you need your application to work with multiple databases, it's often better to rely on Hibernate's default mapping mechanisms and let it handle the translation of Java types to database-specific column types.

@JoinColumn

The `referencedColumnName` attribute in the `@JoinColumn` annotation is used to specify the name of the column in the referenced (target) entity's table that is being used as the foreign key.

Here's how it works:

```java
@Entity
public class Education {
    @Id
    @GeneratedValue
    private long id;
    private int sscPercentage;
    private int hsscPercentage;

    // Establishing a one-to-one relationship with the Student entity
    @OneToOne
    @JoinColumn(name = "student_id", referencedColumnName = "student_id")
    private Student student;
}
```

In this example, `@JoinColumn` is used to specify that the `student` attribute in the `Education` entity should be mapped to the `student_id` column in the database table. The `referencedColumnName` attribute specifies that the foreign key in the `Education` table should reference the `student_id` column in the `Student` table.

So, when Hibernate creates the database schema, it will create a foreign key constraint in the `Education` table that references the `student_id` column in the `Student` table.

Using `referencedColumnName` allows you to explicitly specify the column name in the referenced entity's table that corresponds to the foreign key relationship, providing more flexibility in mapping the relationship between entities.

@EmbeddedId

The `@EmbeddedId` annotation in JPA is used to represent a composite primary key in an entity. It is typically used when the primary key of an entity consists of multiple columns.

Here's how it works:

```java
@Embeddable
public class StudentId implements Serializable {
    private Long studentId;
    private String departmentId;

    // Constructors, getters, setters, equals, and hashCode methods omitted for
brevity
}

@Entity
public class Student {
    @EmbeddedId
    private StudentId id;

    private String name;

    // Constructors, getters, setters, etc. omitted for brevity
}
```

In this example, `StudentId` is an embeddable class annotated with `@Embeddable`, which indicates that it can be embedded within other entities. It represents the composite primary key for the `Student` entity and consists of two fields: `studentId` and `departmentId`.

The `Student` entity uses the `@EmbeddedId` annotation to specify that the `id` attribute represents the composite primary key of the entity. This tells JPA to treat the `id` attribute as a composite key composed of the fields defined in the `StudentId` class.

Using `@EmbeddedId` allows you to encapsulate the composite primary key fields within a separate class, improving code organization and readability. It also allows you to reuse the composite key definition in multiple entities if needed.