

Spring web app configuration.

chat.openai.com/chat

 vigneshkumareekpr@gmail.com

```
package com.kgisl.SpringJavaConfig; import
org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitializer;
public class AppInitializer extends
AbstractAnnotationConfigDispatcherServletInitializer { @Override protected Class <
? > [] getRootConfigClasses() { return new Class[] { ApplicationContext.class }; //return null;
} @Override protected Class < ? > [] getServletConfigClasses() { return new Class[] {
WebMvcConfig.class }; } @Override protected String[] getServletMappings() { return
new String[] { "/" }; } } explain this
```



This Java class is an initializer for a Spring web application that uses Java configuration instead of XML configuration. It extends the `AbstractAnnotationConfigDispatcherServletInitializer` class provided by Spring.

In this class, there are three methods:

1. `getRootConfigClasses()` method: This method returns an array of classes that define the root configuration classes. In this example, `ApplicationContext.class` is returned as the root configuration class. This class will define the beans that are available to the entire application context.
2. `getServletConfigClasses()` method: This method returns an array of classes that define the configuration classes for the `DispatcherServlet`. In this example, `WebMvcConfig.class` is returned as the configuration class for the `DispatcherServlet`. This class will define the beans that are specific to the `DispatcherServlet`.
3. `getServletMappings()` method: This method returns an array of `String`s that specify the mappings for the `DispatcherServlet`. In this example, the mapping is set to `/` which means that the `DispatcherServlet` will handle all requests coming to the root URL.

By extending `AbstractAnnotationConfigDispatcherServletInitializer`, this class enables the use of Java-based configuration for Spring MVC applications.

 vigneshkumareekpr@gmail.com

```
package com.kgisl.SpringJavaConfig; import java.util.Properties; import
javax.sql.DataSource; import
org.springframework.beans.factory.annotation.Autowired; import
org.springframework.context.annotation.Bean; import
org.springframework.context.annotation.ComponentScan; import
org.springframework.context.annotation.Configuration; import
org.springframework.context.annotation.PropertySource; import
org.springframework.core.env.Environment; import
org.springframework.jdbc.datasource.DriverManagerDataSource; import
org.springframework.orm.hibernate5.HibernateTransactionManager; import
org.springframework.orm.hibernate5.LocalSessionFactoryBean; import
org.springframework.transaction.annotation.EnableTransactionManagement;
@Configuration @PropertySource("classpath:database.properties")
@EnableTransactionManagement public class AppContext { @Autowired private
Environment environment; @Bean public LocalSessionFactoryBean sessionFactory()
{ LocalSessionFactoryBean sessionFactory = new LocalSessionFactoryBean();
sessionFactory.setDataSource(dataSource()); sessionFactory.setPackagesToScan(new
String[] { "com.kgisl.SpringJavaConfig" });
sessionFactory.setHibernateProperties(hibernateProperties()); return sessionFactory;
} @Bean public DataSource dataSource() { DriverManagerDataSource dataSource =
new DriverManagerDataSource();
dataSource.setDriverClassName(environment.getRequiredProperty("jdbc.driverClassName"));
dataSource.setUrl(environment.getRequiredProperty("jdbc.url"));
dataSource.setUsername(environment.getRequiredProperty("jdbc.username"));
dataSource.setPassword(environment.getRequiredProperty("jdbc.password")); return
dataSource; } private Properties hibernateProperties() { Properties properties = new
Properties(); properties.put("hibernate.dialect",
environment.getRequiredProperty("hibernate.dialect"));
properties.put("hibernate.show_sql",
environment.getRequiredProperty("hibernate.show_sql"));
properties.put("hibernate.format_sql",
environment.getRequiredProperty("hibernate.format_sql"));
properties.put("hibernate.hbm2ddl.auto",
environment.getRequiredProperty("hibernate.hbm2ddl.auto")); return properties; }
@Bean public HibernateTransactionManager getTransactionManager() {
```

```
HibernateTransactionManager transactionManager = new  
HibernateTransactionManager();  
transactionManager.setSessionFactory(sessionFactory().getObject()); return  
transactionManager; } } explain this code
```



This Java class is the configuration class for the application context and defines the beans required for Hibernate to work with the database. It is marked as a `@Configuration` class, which means it is a source of bean definitions.

1. `@PropertySource("classpath:database.properties")` annotation: This annotation specifies the location of the `database.properties` file that contains the database connection details and other Hibernate configuration properties.
2. `@EnableTransactionManagement` annotation: This annotation enables transaction management for the application.
3. `@Autowired` annotation: This annotation is used to inject the `Environment` object, which is used to read the properties from the `database.properties` file.
4. `@Bean` annotation: This annotation is used to define a bean that can be injected into other parts of the application.
5. `sessionFactory()` method: This method defines a `LocalSessionFactoryBean` bean that is used to create a Hibernate `SessionFactory`. It sets the data source, the packages to scan for entity classes, and the Hibernate properties.
6. `dataSource()` method: This method defines a `DriverManagerDataSource` bean that is used to create a data source for the application. It sets the database connection details by reading them from the `database.properties` file.
7. `hibernateProperties()` method: This method defines a `Properties` object that is used to set the Hibernate properties. It reads the properties from the `database.properties` file.
8. `getTransactionManager()` method: This method defines a `HibernateTransactionManager` bean that is used to manage transactions for the application. It sets the `SessionFactory` object created by the `sessionFactory()` method.

In summary, this configuration class sets up the necessary beans required for Hibernate to work with the database, and enables transaction management for the application.

□vigneshkumareekpr@gmail.com

what is HibernateTransactionManager



HibernateTransactionManager is a Spring Framework class that provides a transaction management abstraction for Hibernate-based database access. It is responsible for managing transactions between the application and the database by wrapping database operations in transactions and ensuring their atomicity, consistency, isolation, and durability (ACID) properties.

It works with the **SessionFactory** object, which is responsible for creating database connections and executing Hibernate queries. The **HibernateTransactionManager** object manages the transactional state of the **SessionFactory** object, ensuring that all database operations executed through the **SessionFactory** object are performed within a transaction. It also provides a simple API for the application to begin, commit, or rollback transactions.

In summary, **HibernateTransactionManager** is an important component in a Spring-Hibernate integration that provides a transactional boundary to database operations, ensuring data consistency and reliability.

□vigneshkumareekpr@gmail.com

```
package com.kgisl.SpringJavaConfig; import java.util.List; import
javax.persistence.Query; import javax.persistence.criteria.CriteriaBuilder; import
javax.persistence.criteria.CriteriaQuery; import javax.persistence.criteria.Root;
import org.hibernate.Session; import org.hibernate.SessionFactory; import
org.springframework.beans.factory.annotation.Autowired; import
org.springframework.stereotype.Repository; @Repository public class
CustomerDAOImpl implements CustomerDAO { @Autowired private SessionFactory
sessionFactory; @Override public List < Customer > getCustomers() { Session session
= sessionFactory.getCurrentSession(); CriteriaBuilder cb =
```

```

session.getCriteriaBuilder(); CriteriaQuery < Customer > cq =
cb.createQuery(Customer.class); Root < Customer > root = cq.from(Customer.class);
cq.select(root); Query query = session.createQuery(cq); return query.getResultList();
} @Override public void deleteCustomer(int id) { Session session =
sessionFactory.getCurrentSession(); Customer book =
session.byId(Customer.class).load(id); session.delete(book); } @Override public void
saveCustomer(Customer theCustomer) { Session currentSession =
sessionFactory.getCurrentSession(); currentSession.saveOrUpdate(theCustomer); }
@Override public Customer getCustomer(int theId) { Session currentSession =
sessionFactory.getCurrentSession(); Customer theCustomer =
currentSession.get(Customer.class, theId); return theCustomer; } } Explain this code

```



This is the implementation of a DAO (Data Access Object) interface for performing CRUD (Create, Read, Update, and Delete) operations on `Customer` objects in a database.

The class implements the `CustomerDAO` interface, which defines the method signatures for performing database operations. The `@Repository` annotation is used to mark this class as a Spring bean that performs data access functions.

The `SessionFactory` object is injected using Spring's dependency injection feature, which provides a connection to the database.

The `getCustomers()` method retrieves all `Customer` objects from the database using Hibernate's Criteria API. It creates a `CriteriaQuery` object to specify the selection criteria and returns a list of `Customer` objects.

The `deleteCustomer(int id)` method deletes a `Customer` object from the database by its `id`. It uses the `Session` object to get a reference to the `Customer` object and then deletes it.

The `saveCustomer(Customer theCustomer)` method saves a new `Customer` object to the database or updates an existing one. It uses the `Session` object to save or update the `Customer` object.

The `getCustomer(int theId)` method retrieves a single `Customer` object from the database by its `id`. It uses the `Session` object to get a reference to the `Customer` object and returns it.

In summary, this code defines a DAO implementation that performs database operations on `Customer` objects using the Hibernate API and Spring's dependency injection mechanism. It provides a layer of abstraction between the application and the database, making it easier to manage database access and maintain code consistency.

□vigneshkumareekpr@gmail.com

why we use @Repository in spring



In Spring, `@Repository` is a type of stereotype annotation used to indicate that the annotated class is a repository, which is responsible for data access and persistence logic. It is a specialization of the `@Component` annotation and serves to identify classes at the persistence layer.

The `@Repository` annotation is primarily used for two purposes:

1. It indicates to the Spring container that the annotated class should be instantiated as a bean and added to the container's application context.
2. It provides a convenient way to apply persistence-related aspects, such as transactions and exception translation, to classes that access the database.

By using the `@Repository` annotation, Spring can automatically detect the annotated class and add it to the container's application context. This makes it easier to manage and inject dependencies into the repository, as well as provide other services such as transaction management and exception handling.

□vigneshkumareekpr@gmail.com

what is Hibernate's Criteria API with example



Hibernate's Criteria API is a powerful and flexible tool for building type-safe queries. It provides a programmatic way to build SQL-like queries using Java code, without having to write SQL statements directly.

Here is an example of how to use Hibernate's Criteria API to perform a simple query:

Suppose we have a `Customer` entity with the following properties:

```
java

public class Customer {
    private int id;
    private String name;
    private String email;
    // getters and setters
}
```

We want to retrieve all customers whose name starts with "John". Here is how we can use Hibernate's Criteria API to build this query:

```
java

CriteriaBuilder cb = session.getCriteriaBuilder();
CriteriaQuery<Customer> cq = cb.createQuery(Customer.class);
Root<Customer> root = cq.from(Customer.class);
cq.select(root).where(cb.like(root.get("name"), "John%"));
Query<Customer> query = session.createQuery(cq);
List<Customer> customers = query.getResultList();
```

In this code, we first obtain a `CriteriaBuilder` instance from the `Session` object. The `CriteriaBuilder` is a factory for building `CriteriaQuery` objects.

Next, we create a `CriteriaQuery` object and specify the entity type to be queried (`Customer` in this case). We also obtain a `Root` object representing the entity's root element.

We then use the `CriteriaBuilder` to build a `Predicate` that specifies the search criteria. In this case, we use the `like()` method to create a predicate that matches all customers whose name starts with "John".

Finally, we create a `Query` object from the `CriteriaQuery` and execute the query using the `getResultList()` method. This returns a `List` of `Customer` objects that match the search criteria.

Overall, Hibernate's Criteria API provides a flexible and expressive way to build type-safe queries using Java code, without having to write SQL statements directly.