@EnableAutoConfiguration and @Configuration

https://www.javatpoint.com/spring-boot-auto-configuration

To see the auto configuration class, we can see ->> auto-configuration jar -> specific library -> meta-inf -> spring

To register our configuration as one of auto configuration we have to put it under resources/META-INF/spring/ org.springframework.boot.autoconfigure.AutoConfiguration.imports file and put the fully qualified name of the auto-configuration class which we have made so that it loads the the beans only on the basis of some @Conditional.
Now the @AutoConfiguration class will be detected without any issue.
It enables the custom auto configuration class.

Now we can mention on the class like @ConditionalOnClass(name="fully qualified name of the class to be present in classpath") so that the beans in the configuration will be loaded.

If we write debug=true in application.properties we can see which auto configured beans we have got while loading.

`@EnableAutoConfiguration` is a Spring Boot annotation that enables automatic configuration of Spring beans and the application context based on the dependencies present in the classpath and the configuration provided by Spring Boot.

When you add `@EnableAutoConfiguration` to your Spring Boot application, Spring Boot automatically attempts to configure the Spring application context by analyzing the project's dependencies and the configuration properties present in the project. It automatically configures beans for various features such as data sources, web servers, security, messaging, etc., based on the detected dependencies and the specified configuration.

Here are some key points regarding the usage and benefits of `@EnableAutoConfiguration`:

1. **Simplifies Configuration**:
   `@EnableAutoConfiguration` simplifies the configuration of Spring Boot applications by automatically configuring beans and components without requiring explicit configuration from the developer.

2. **Convention over Configuration**:
   Spring Boot follows the principle of convention over configuration, where

sensible defaults and conventions are applied to reduce the amount of boilerplate configuration code required. `@EnableAutoConfiguration` plays a key role in applying these conventions by automatically configuring beans based on predefined rules and conditions.

3. **Reduced Development Time**:
   By leveraging auto-configuration, developers can quickly bootstrap Spring Boot applications without having to manually configure every component. This reduces development time and allows developers to focus on writing application logic rather than dealing with intricate configuration details.

4. **Customization and Overrides**:
   While auto-configuration provides defaults and conventions, developers can still customize and override the auto-configured beans by providing their own configuration. By placing custom configuration classes or properties in the application context, developers can influence the auto-configuration process and tailor it to their specific requirements.

5. **Integration with Spring Boot Starters**:
   `@EnableAutoConfiguration` works seamlessly with Spring Boot starters, which are curated sets of dependencies and auto-configuration rules designed to provide ready-to-use features for common use cases. By including starters in your project's dependencies, you automatically inherit the required configuration and dependencies for those features.

Overall, `@EnableAutoConfiguration` is a powerful annotation provided by Spring Boot that streamlines the configuration process, promotes best practices, and accelerates the development of Spring Boot applications. It enables developers to build production-ready applications with minimal effort and maximum flexibility.

The `@EnableAutoConfiguration` annotation in Spring Boot does not directly accept attribute classes. It is a meta-annotation, which means it is used to enable auto-configuration by annotating another configuration class.

Typically, you would apply `@EnableAutoConfiguration` at the top of your main Spring Boot application class or a configuration class. Here's an example:

```java
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class MyApplication {
    public static void main(String[] args) {
```

```
        SpringApplication.run(MyApplication.class, args);
    }
}
```

In this example, `@SpringBootApplication` is a composed annotation that includes `@EnableAutoConfiguration` among others. It enables auto-configuration for the Spring Boot application.

While `@EnableAutoConfiguration` itself doesn't accept attribute classes, it does consider various conditions and configuration classes to determine which beans to auto-configure based on the dependencies and properties present in the application's classpath and configuration.

If you need to customize the auto-configuration behavior, you can provide additional configuration classes or exclude certain auto-configuration classes using other annotations such as `@Import`, `@ComponentScan`, or `@ExcludeAutoConfiguration`. These annotations allow you to include or exclude specific configuration classes or packages from the auto-configuration process.

In Spring Framework, `@Conditional` is a meta-annotation that allows you to conditionally include or exclude components, configuration classes, or beans based on certain conditions. It provides a way to dynamically control the bean registration process and the application context configuration.

When you apply `@Conditional` to a component or configuration class, Spring evaluates the specified condition(s) associated with the annotation. If the condition(s) are met, the component or configuration class is registered with the application context. Otherwise, it is not registered, effectively excluding it from the application context.

Here's how you can use `@Conditional` in Spring:

1. **Condition Interfaces**:
   Spring provides several built-in condition interfaces that you can use with `@Conditional`. These interfaces include:
   - `Condition`: The base interface for defining custom conditions.
   - `ConditionContext`: Provides contextual information for evaluating conditions.
   - `ConfigurationCondition`: A condition interface for conditions applied to configuration classes.

2. **Custom Conditions**:
   You can create custom conditions by implementing the `Condition` interface

or extending existing condition classes. Custom conditions allow you to define complex logic to determine whether a component or configuration class should be included or excluded.

3. **Applying @Conditional**:
   You can apply `@Conditional` to:
   - Component classes (`@Component`, `@Service`, `@Repository`, etc.).
   - Configuration classes (`@Configuration`).
   - Bean declaration methods (`@Bean`).
   - Import selectors (`@Import`).

4. **Example**:
   Here's an example of how to use `@Conditional` to conditionally register a bean based on a custom condition:

   ```java
   import org.springframework.context.annotation.Bean;
   import org.springframework.context.annotation.Conditional;
   import org.springframework.context.annotation.Configuration;

   @Configuration
   public class MyConfiguration {

       @Bean
       @Conditional(MyCondition.class)
       public MyBean myBean() {
           return new MyBean();
       }
   }
   ```

   In this example, the `MyBean` bean will be registered with the application context only if the condition specified by the `MyCondition` class is met.

5. **Predefined Conditions**:
   Spring Boot provides several predefined conditions that you can use out of the box, such as `ConditionalOnClass`, `ConditionalOnProperty`, `ConditionalOnBean`, etc. These conditions allow you to conditionally configure beans based on class availability, property values, bean existence, and other criteria.

`@Conditional` is a powerful mechanism in Spring that enables flexible and dynamic bean registration and application context configuration. It allows you to control the application's behavior based on various conditions, making your Spring applications more adaptable and configurable.

You can apply `@Conditional` to the `@Import` annotation in Spring to

conditionally import configuration classes based on certain conditions. This allows you to dynamically control which configuration classes are imported into your application context based on runtime conditions.

Here's how you can apply `@Conditional` to `@Import`:

1. **Define a Condition Class**:
   First, define a condition class that implements the `Condition` interface. This class will contain the logic to determine whether the configuration class should be imported or not. For example:

   ```java
   import org.springframework.context.annotation.Condition;
   import org.springframework.context.annotation.ConditionContext;
   import org.springframework.core.type.AnnotatedTypeMetadata;

   public class MyCondition implements Condition {
       @Override
       public boolean matches(ConditionContext context,
AnnotatedTypeMetadata metadata) {
           // Your condition logic here
           return true; // Return true if condition is met, false otherwise
       }
   }
   ```

2. **Apply @Conditional to @Import**:
   Apply `@Conditional` to the `@Import` annotation along with your condition class. This will conditionally import the configuration class based on the result of the condition evaluation. For example:

   ```java
   import org.springframework.context.annotation.Configuration;
   import org.springframework.context.annotation.Import;
   import org.springframework.context.annotation.Conditional;

   @Configuration
   @Conditional(MyCondition.class)
   @Import(MyOtherConfiguration.class)
   public class MyConfiguration {
       // Your configuration class code here
   }
   ```

   In this example, the `MyOtherConfiguration` class will be imported into the application context only if the condition specified by `MyCondition` class is met.

By applying `@Conditional` to `@Import`, you can dynamically control which configuration classes are imported into your application context based on runtime conditions. This provides flexibility and allows you to adapt your application's configuration based on various factors, such as environment variables, system properties, or external conditions.

If you apply `@Conditional` after `@Import`, the condition specified by `@Conditional` will not have any effect on the import process. The `@Conditional` annotation must be applied directly to the element (such as `@Configuration` or `@Import`) that determines whether a bean or configuration class should be included or excluded based on certain conditions.

In Spring's annotation-based configuration, the order of annotations matters. Annotations are processed in the order they appear, and later annotations typically override or augment the behavior specified by earlier annotations.

Consider the following example:

```java
@Configuration
@Import(MyOtherConfiguration.class)
@Conditional(MyCondition.class)
public class MyConfiguration {
    // Your configuration class code here
}
```

In this example, `MyOtherConfiguration` will always be imported into the application context regardless of the condition specified by `MyCondition`. This is because `@Import` is applied before `@Conditional`, so the import process occurs before the condition is evaluated.

To conditionally import configuration classes based on certain conditions, you should apply `@Conditional` directly to the `@Import` annotation or to the `@Configuration` annotation that contains the `@Import` annotation, as shown in the previous response. This ensures that the condition is evaluated before the import process takes place, allowing you to dynamically control which configuration classes are imported based on runtime conditions.

If you directly apply `@Conditional` before a `@Configuration` class, Spring will evaluate the condition specified by `@Conditional` before considering the annotated class as a configuration class. If the condition evaluates to `true`, the class will be registered as a configuration class, and its bean definitions will

be processed by the Spring container. If the condition evaluates to `false`, the class will not be registered as a configuration class, and its bean definitions will be ignored.

Here's how you can directly apply `@Conditional` before a `@Configuration` class:

```java
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Conditional;

@Conditional(MyCondition.class)
@Configuration
public class MyConfiguration {
    // Your configuration class code here
}
```

In this scenario, the `MyConfiguration` class will be conditionally registered as a configuration class based on the condition specified by `MyCondition`. If the condition is met (i.e., evaluates to `true`), `MyConfiguration` will be registered as a configuration class, and its bean definitions will be processed. If the condition is not met (i.e., evaluates to `false`), `MyConfiguration` will not be registered as a configuration class, and its bean definitions will be ignored.

This approach allows you to control the registration of configuration classes based on runtime conditions using `@Conditional`, providing flexibility in configuring your Spring application context.

To disable certain auto-configurations

To debug autoconfiguration on application startup in a Spring Boot application, you can enable debug logging for the `org.springframework.boot.autoconfigure` package. This will provide detailed information about the autoconfiguration process, including which configurations are being applied, the conditions being evaluated, and any conditions that are not met.

Here's how you can enable debug logging for autoconfiguration:

1. Open your `application.properties` file.

2. Add the following line to enable debug logging for autoconfiguration:

```properties
logging.level.org.springframework.boot.autoconfigure=DEBUG
```

Alternatively, you can enable debug logging for autoconfiguration only during application startup by specifying the logging level as DEBUG in your application startup script or command:

```bash
java -jar -Dlogging.level.org.springframework.boot.autoconfigure=DEBUG your-application.jar
```

With debug logging enabled for autoconfiguration, you will see detailed information in the console or log files during application startup. This information includes which autoconfiguration classes are being processed, which conditions are being evaluated, and the outcome of those conditions.

To disable autoconfiguration based on certain conditions, you can use the `spring.autoconfigure.exclude` property. This property allows you to specify a comma-separated list of fully qualified class names to exclude from autoconfiguration.

For example, let's say you have an autoconfiguration class named `MyAutoConfiguration` and you want to disable it based on a condition. You can create a condition class and conditionally exclude `MyAutoConfiguration` using the `spring.autoconfigure.exclude` property:

```java
import org.springframework.context.annotation.Condition;
import org.springframework.context.annotation.ConditionContext;
import org.springframework.core.type.AnnotatedTypeMetadata;

public class MyCondition implements Condition {

    @Override
    public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {
        // Implement your condition logic here
        // Return true to exclude the autoconfiguration, false to include it
        return false; // For example, conditionally exclude based on some property
    }
}
```

Then, annotate `MyAutoConfiguration` with `@Conditional` and specify

`MyCondition`:

```java
import org.springframework.context.annotation.Conditional;

@Conditional(MyCondition.class)
public class MyAutoConfiguration {
    // Autoconfiguration logic here
}
```

Finally, configure `spring.autoconfigure.exclude` in your `application.properties` or `application.yml`:

```properties
spring.autoconfigure.exclude=com.example.MyAutoConfiguration
```

With this configuration, `MyAutoConfiguration` will be excluded from autoconfiguration based on the condition implemented in `MyCondition`.

https://www.baeldung.com/spring-data-disable-auto-config

https://docs.spring.io/spring-boot/docs/1.4.x/reference/html/using-boot-auto-configuration.html#:~:text=If%20you%20find%20that%20specific,of%20%40EnableAutoConfiguration%20to%20disable%20them.&text=If%20the%20class%20is%20not,the%20fully%20qualified%20name%20instead

We can also exclude using @EnableAutoConfiugration(exclude = {....})