# @ConfigurationProperties

`@ConfigurationProperties` is an annotation in Spring Boot used to bind external configuration properties to a Java bean. It allows you to map properties defined in external configuration files (such as `application.properties` or `application.yml`) to fields of a Java bean.

Here's an explanation of how `@ConfigurationProperties` works:

1. **Binding External Properties**: Spring Boot allows you to externalize configuration properties from your application code. You can define properties in various formats like properties files (`application.properties`) or YAML files (`application.yml`). These properties can represent configuration settings for your application, such as database connection details, server ports, or custom settings.

2. **Mapping to Java Bean**: With `@ConfigurationProperties`, you can map these external properties to fields of a Java bean. This bean serves as a container for the configuration properties. Each property from the external configuration file is mapped to a corresponding field in the Java bean based on its name.

3. **Binding Rules**: By default, Spring Boot uses relaxed binding rules to map properties to Java bean fields. This means that property names are case-insensitive, and certain variations of property names are accepted (e.g., dashed, underscored, or camel-cased). You can customize the binding rules using the `prefix` attribute of `@ConfigurationProperties`.

4. **Type Conversion**: Spring Boot automatically performs type conversion when binding properties to Java bean fields. It converts property values from their string representation in the configuration files to the appropriate data types of the bean fields.

5. **Validation**: `@ConfigurationProperties` supports property validation using JSR-303/JSR-380 bean validation annotations. You can annotate bean fields with validation constraints to enforce constraints on the property values.

6. **Spring Boot Auto-Configuration**: When you use `@ConfigurationProperties` to bind properties related to Spring Boot's auto-configuration, Spring Boot automatically wires the configuration properties into the auto-configured beans. This allows you to customize the behavior of auto-configured components using external configuration.

Overall, `@ConfigurationProperties` provides a convenient and flexible way to externalize configuration properties in Spring Boot applications and bind them to Java bean fields, enabling better separation of configuration from application logic and promoting maintainability and reusability.

If you don't specify a prefix in `@ConfigurationProperties`, Spring Boot will attempt to bind the properties directly to the fields of the annotated class based on the exact property names.

Here's what happens when you omit the prefix:

1. **Exact Property Names**: Spring Boot will attempt to bind each property in the configuration files to a corresponding field in the class based on the exact property name. For example, a property named `myapp.greeting` would be mapped to a field named `myapp.greeting` in the class.

2. **Case Sensitivity**: The binding process is case-sensitive by default. This means that the property names in the configuration files must match the field names in the class exactly, including case.

3. **Direct Mapping**: Without a prefix, there is a direct mapping between the property names and the field names in the class. This can lead to potential conflicts if the property names in the configuration files collide with existing field names in the class.

4. **Limited Flexibility**: Omitting the prefix limits the flexibility of mapping properties to fields. You lose the ability to group related properties under a common prefix and to customize the binding process using relaxed binding rules or custom prefixes.

5. **Namespace Pollution**: If the configuration properties are not namespaced under a prefix, there is a risk of namespace pollution, especially in larger applications with many configuration properties. This can make it harder to manage and maintain the configuration files.

In summary, while omitting the prefix in `@ConfigurationProperties` can simplify the binding process in some cases, it may lead to potential conflicts, reduced flexibility, and namespace pollution. It's generally recommended to use a prefix to group related properties and to provide a clear namespace for configuration properties.

https://www.baeldung.com/configuration-properties-in-spring-boot

Sure, here's an example of how to use `@ConfigurationProperties` in a Spring Boot application:

1. Define a configuration properties class:

```java
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.stereotype.Component;

@Component
@ConfigurationProperties(prefix = "myapp")
public class MyAppProperties {

    private String greeting;
    private int timeout;

    // Getters and setters

    public String getGreeting() {
        return greeting;
    }

    public void setGreeting(String greeting) {
        this.greeting = greeting;
    }

    public int getTimeout() {
        return timeout;
    }

    public void setTimeout(int timeout) {
        this.timeout = timeout;
    }
}
```

2. Configure properties in `application.properties`:

```properties
myapp.greeting=Hello, World!
myapp.timeout=5000
```

3. Use the configuration properties in your application:

```java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
```

```
@SpringBootApplication
public class MyApp implements CommandLineRunner {

    @Autowired
    private MyAppProperties myAppProperties;

    public static void main(String[] args) {
        SpringApplication.run(MyApp.class, args);
    }

    @Override
    public void run(String... args) throws Exception {
        System.out.println(myAppProperties.getGreeting());
        System.out.println("Timeout: " + myAppProperties.getTimeout() + " ms");
    }
}
```

In this example:

- We define a `MyAppProperties` class with `@ConfigurationProperties` annotation, specifying the prefix `"myapp"`. This class contains fields for `greeting` and `timeout` properties.
- We configure the properties in the `application.properties` file using the specified prefix.
- In the `MyApp` class, we autowire `MyAppProperties` and use its getters to access the configured properties.
- When the application starts, it prints the configured greeting message and timeout value to the console.

https://www.geeksforgeeks.org/spring-boot-configurationproperties/

If you don't use `@ConfigurationProperties` and directly write configuration properties in the `application.properties` file (or `application.yml`), you can still access those properties in your Spring Boot application. However, there are some differences and considerations to keep in mind:

1. **Manual Property Binding**: Without `@ConfigurationProperties`, you'll need to manually bind the configuration properties to fields in your Java classes. This typically involves using the `@Value` annotation or accessing the properties directly from the environment.

   Example using `@Value` annotation:
   ```java
   @Value("${myapp.property}")
   private String property;
   ```

```

2. **Type Conversion**: When using `@ConfigurationProperties`, Spring Boot automatically converts configuration values to the appropriate data types based on the target field's type. Without it, you may need to perform type conversion manually.

3. **No Validation**: `@ConfigurationProperties` supports validation of configuration values using standard validation annotations. Without it, you'll need to perform validation manually if needed.

4. **No Grouping or Prefixing**: `@ConfigurationProperties` allows you to group related properties under a common namespace by specifying a prefix. This can help organize and manage configuration properties effectively. Without it, you'll need to handle property grouping and prefixing manually.

5. **Easier to Manage**: Using `@ConfigurationProperties` provides a more structured and organized approach to accessing configuration properties. It's especially useful for larger applications with many configuration properties, as it promotes maintainability and readability.

In summary, while it's possible to directly access configuration properties without `@ConfigurationProperties`, using this annotation provides a more convenient, type-safe, and structured way to manage configuration properties in Spring Boot applications. It's generally recommended to use `@ConfigurationProperties` for better maintainability and ease of use, especially in larger and more complex projects.

https://mkyong.com/spring-boot/spring-boot-configurationproperties-example/
```