## Custom Validation

To create custom validation annotations in a Spring Boot application using the Spring Boot Starter Validation, you need to follow these steps:

1. **Create a Custom Annotation**: Define your custom validation annotation by creating a new Java annotation. This annotation will serve as a marker for fields or methods that need validation.

2. **Create a Validator**: Implement a validator class that performs the actual validation logic. This class should implement the `ConstraintValidator` interface provided by the Hibernate Validator framework.

3. **Wire Up the Validator**: Configure Spring to recognize and use your custom validator by registering it in the Spring context.

4. **Apply the Annotation**: Use your custom annotation to annotate fields or methods in your Spring components (like DTOs, entities, or request objects) that require validation.

Here's an example to illustrate these steps:

1. **Create a Custom Annotation**:

```java
import javax.validation.Constraint;
import javax.validation.Payload;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target({ElementType.FIELD, ElementType.METHOD, 
ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy = MyCustomValidator.class)
public @interface MyCustomValidation {
    String message() default "Invalid value";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};
}
```

2. **Create a Validator**:

```java
import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;

public class MyCustomValidator implements
ConstraintValidator<MyCustomValidation, String> {

    @Override
    public void initialize(MyCustomValidation constraintAnnotation) {
        // Initialization, if needed
    }

    @Override
    public boolean isValid(String value, ConstraintValidatorContext context) {
        // Your validation logic goes here
        // Return true if valid, false otherwise
        return value != null && value.startsWith("prefix_");
    }
}
```

3. **Wire Up the Validator**:

You can register the validator as a Spring bean in a configuration class:

```java
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import
org.springframework.validation.beanvalidation.LocalValidatorFactoryBean;

@Configuration
public class ValidatorConfig {

    @Bean
    public LocalValidatorFactoryBean validator() {
        LocalValidatorFactoryBean validatorFactoryBean = new
LocalValidatorFactoryBean();
        validatorFactoryBean.setValidationMessageSource(messageSource());
        return validatorFactoryBean;
    }

    @Bean
    public MessageSource messageSource() {
        ResourceBundleMessageSource messageSource = new
ResourceBundleMessageSource();
```

```
        messageSource.setBasename("messages");
        messageSource.setDefaultEncoding("UTF-8");
        return messageSource;
    }

    @Bean
    public MyCustomValidator myCustomValidator() {
        return new MyCustomValidator();
    }
}
```

4. **Apply the Annotation**:

```java
public class MyDTO {

    @MyCustomValidation
    private String customField;

    // Getter and setter methods
}
```

Now, `MyDTO.customField` will be validated according to the logic defined in `MyCustomValidator` whenever validation is triggered in your Spring application. Make sure to have the necessary error messages configured in your message properties file (`messages.properties` or similar) for custom validation messages.

In Java annotations, `payload` and `groups` are elements that can be used to provide additional metadata or configuration to the annotation. They are commonly used in conjunction with constraint annotations in the Java Bean Validation API (JSR 380), which is often used in Spring Boot applications for validation purposes.

Here's what each of them represents:

1. **groups**:
   - The `groups` element allows you to specify validation groups that this constraint belongs to. Validation groups are used to group constraints together and apply them selectively.
   - When you define a validation constraint with groups, it means that the constraint is only evaluated when the specified validation groups are active during validation. This provides a way to perform different sets of validations

depending on the context.
   - For example, you might have different validation groups for basic validation, advanced validation, or validation specific to certain use cases.

2. **payload**:
   - The `payload` element is a way to attach additional metadata to a constraint declaration. It's a more generic element and can be used to convey any kind of additional information.
   - The payload element typically represents a type (usually an interface) that can be used to carry additional information. It can be any arbitrary type.
   - In practice, the payload element is not commonly used directly in most custom constraint annotations. Instead, it's often used implicitly to attach additional metadata or information to the constraint.
   - For example, you could use the payload element to carry information about severity levels for constraint violations, error codes, or any other custom metadata relevant to your application.

Here's an example illustrating the usage of `groups` and `payload`:

```java
import javax.validation.Constraint;
import javax.validation.Payload;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target({ElementType.FIELD, ElementType.METHOD,
ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy = MyCustomValidator.class)
public @interface MyCustomValidation {
    String message() default "Invalid value";

    Class<?>[] groups() default {}; // Validation groups

    Class<? extends Payload>[] payload() default {}; // Additional metadata
payload
}
```

In this example, `groups` allows you to specify validation groups, and `payload` allows you to attach additional metadata to the constraint declaration.

Sure, here's an example illustrating the usage of `groups` and `payload` in a custom validation annotation:

Let's say we want to create a custom validation annotation `MyCustomValidation` that checks if a String starts with a specific prefix. Additionally, we want to define different validation groups for basic and advanced validation scenarios, and we want to attach a severity level to constraint violations using the `payload` element.

```java
import javax.validation.Constraint;
import javax.validation.Payload;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target({ElementType.FIELD, ElementType.METHOD,
ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy = MyCustomValidator.class)
public @interface MyCustomValidation {
    String message() default "Invalid value";

    Class<?>[] groups() default {}; // Validation groups

    Class<? extends Payload>[] payload() default {}; // Additional metadata
payload

    String prefix(); // Custom attribute for the annotation
}
```

And the corresponding validator class:

```java
import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;

public class MyCustomValidator implements
ConstraintValidator<MyCustomValidation, String> {

    private String prefix;

    @Override
    public void initialize(MyCustomValidation constraintAnnotation) {
        this.prefix = constraintAnnotation.prefix();
    }
    @Override
    public boolean isValid(String value, ConstraintValidatorContext context) {
        return value != null && value.startsWith(prefix);
    }
}
```

Now, let's use this custom annotation in a DTO class:

```java
public class MyDTO {

    @MyCustomValidation(prefix = "test", groups = BasicValidation.class, payload = Severity.Error.class)
    private String customField;

    // Getter and setter methods
}
```

In this example:

- We defined a custom validation annotation `MyCustomValidation`.
- The annotation has `groups` and `payload` elements.
- We defined a custom validation logic in the `MyCustomValidator` class.
- The annotation is applied to a field in the `MyDTO` class, specifying a prefix, validation groups (`BasicValidation`), and a severity level for payload (`Severity.Error`).

You can define the `BasicValidation` and `Severity` classes as follows:

```java
public interface BasicValidation {} // Validation group interface

public interface Severity {
    interface Error extends Payload {} // Payload interface for error severity
    interface Warning extends Payload {} // Payload interface for warning severity
    interface Info extends Payload {} // Payload interface for info severity
}
```

In this way, you can use the `groups` and `payload` elements to configure your custom validation annotation for different validation scenarios and attach additional metadata to constraint violations.