API BEAN VALIDATION

DÉFINITION

Permet de définir des contraintes sur les attributs d'une classe en utilisant des annotations.

EXEMPLE

```java
public class Personne {

    @NotNull // Le champ ne doit pas être null
    private String nom;

    @Min(18) // Le champ doit être supérieur ou égal à 18
    private int age;

    @Email // Le champ doit être une adresse email valide
    private String email;

    // getters and setters
}
```

EXEMPLE

```java
public class Personne {

    @NotNull // Le champ ne doit pas être null
    private String nom;

    @Min(18) // Le champ doit être supérieur ou égal à 18
    private int age;

    @Email // Le champ doit être une adresse email valide
    private String email;

    // getters and setters
}
```

EXEMPLE

```java
public class Personne {

    @NotNull // Le champ ne doit pas être null
    private String nom;

    @Min(18) // Le champ doit être supérieur ou égal à 18
    private int age;

    @Email // Le champ doit être une adresse email valide
    private String email;

    // getters and setters
}
```

EXEMPLE

```java
public class Personne {

    @NotNull // Le champ ne doit pas être null
    private String nom;

    @Min(18) // Le champ doit être supérieur ou égal à 18
    private int age;

    @Email // Le champ doit être une adresse email valide
    private String email;

    // getters and setters
}
```

INSTALLATION

Comme pour les autres modules de Spring Boot, il suffit d'ajouter la dépendance suivante dans le fichier `pom.xml` :

```
<dependency>
    <groupId>org.hibernate.validator</groupId>
    <artifactId>hibernate-validator</artifactId>
</dependency>
```

INSTALLATION

Si vous utilisez `Gradle`, il faut ajouter la dépendance suivante dans le fichier `build.gradle` :

```
dependencies {
    implementation 'org.hibernate.validator:hibernate-validator:6.0.2.Final'
}
```

LES ANNOTATIONS

Il existe de nombreuses annotations pour définir des contraintes sur les attributs d'une classe.

- `@NotNull` : le champ ne doit pas être null
- `@NotBlank` : le champ ne doit pas être null ou vide
- `@Pattern` : Le champ correspond à une expression régulière spécifiée.
- `@Email` : Le champ contient une adresse email valide.

LES ANNOTATIONS

- `@Min` : le champ doit être supérieur ou égal à la valeur spécifiée
- `@Max` : le champ doit être inférieur ou égal à la valeur spécifiée
- `@Size` : le champ doit avoir une taille comprise entre les valeurs spécifiées

EXEMPLE

Prenons l'exemple d'une classe `Personne` :

Dans notre cas, le nom doit avoir une taille comprise entre 2 et 30 caractères, et ne peux pas être null ou vide.

```java
public class Personne {

    @NotBlank // Le champ ne doit pas être null ou vide
    @Size(min = 2, max = 30)
    // Le champ doit avoir une taille comprise entre 2 et 30
    private String nom;

    // constructeur, getters et setters
}
```

EXEMPLE

Prenons l'exemple d'une classe `Personne` :

Dans notre cas, le nom doit avoir une taille comprise entre 2 et 30 caractères, et ne peux pas être null ou vide.

```java
public class Personne {

    @NotBlank // Le champ ne doit pas être null ou vide
    @Size(min = 2, max = 30)
    // Le champ doit avoir une taille comprise entre 2 et 30
    private String nom;

    // constructeur, getters et setters
}
```

EXEMPLE

Prenons l'exemple d'une classe `Personne` :

Dans notre cas, le nom doit avoir une taille comprise entre 2 et 30 caractères, et ne peux pas être null ou vide.

```java
public class Personne {

    @NotBlank // Le champ ne doit pas être null ou vide
    @Size(min = 2, max = 30)
    // Le champ doit avoir une taille comprise entre 2 et 30
    private String nom;

    // constructeur, getters et setters
}
```

ERREUR

On peut également définir un message personnalisé, dans le cas d'une erreur de saisie par exemple :

```
1 public class Personne {
2
3     @NotBlank(message = "Le nom ne peut pas être vide")
4     @Size(min = 2, max = 30, message = "Le nom doit avoir une taille comp
5     private String nom;
6
7     // constructeur, getters et setters
8 }
```

ERREUR

On peut également définir un message personnalisé, dans le cas d'une erreur de saisie par exemple :

```
1 public class Personne {
2
3     @NotBlank(message = "Le nom ne peut pas être vide")
4     @Size(min = 2, max = 30, message = "Le nom doit avoir une taille comp
5     private String nom;
6
7     // constructeur, getters et setters
8 }
```

ERREUR

On peut également définir un message personnalisé, dans le cas d'une erreur de saisie par exemple :

```
1 public class Personne {
2
3     @NotBlank(message = "Le nom ne peut pas être vide")
4     @Size(min = 2, max = 30, message = "Le nom doit avoir une taille comp
5     private String nom;
6
7     // constructeur, getters et setters
8 }
```

ERREUR

A ce stade si on essaye de valider une instance de la classe `Personne` avec un nom vide, on obtient une erreur dans la console, car le nom ne respecte pas les contraintes définies.

Si on veut récupérer l'erreur en réponse, il faut utiliser un `Validator` dans notre controller.

ERREUR

```
@PostMapping("/personnes")
// On utilise l'annotation @Validated pour indiquer à Spring qu'il fau
// On récupère les erreurs de validation dans un objet BindingResult
  public ResponseEntity<Object> nouvellePersonne(@Validated @RequestBo
    // On parcourt les erreurs et on les concatène dans un StringBuild
      if (bindingResult.hasErrors()) {
          StringBuilder errors = new StringBuilder();
          // getFieldErrors() extrait les erreurs de validation
          bindingResult.getFieldErrors().forEach(error -> {
          // getDefaultMessage() récupère le message d'erreur
              errors.append(error.getField()).append(": ").append(erro
          });
          // On retourne une réponse avec le code 400 et le message d'
          return new ResponseEntity<>(errors.toString(), HttpStatus.BA
      }
      // On sauvegarde la personne
```

ERREUR

```
@PostMapping("/personnes")
// On utilise l'annotation @Validated pour indiquer à Spring qu'il fau
// On récupère les erreurs de validation dans un objet BindingResult
  public ResponseEntity<Object> nouvellePersonne(@Validated @RequestBo
    // On parcourt les erreurs et on les concatène dans un StringBuild
      if (bindingResult.hasErrors()) {
          StringBuilder errors = new StringBuilder();
          // getFieldErrors() extrait les erreurs de validation
          bindingResult.getFieldErrors().forEach(error -> {
          // getDefaultMessage() récupère le message d'erreur
              errors.append(error.getField()).append(": ").append(erro
          });
          // On retourne une réponse avec le code 400 et le message d'
          return new ResponseEntity<>(errors.toString(), HttpStatus.BA
      }
      // On sauvegarde la personne
```

ERREUR

```
@PostMapping("/personnes")
// On utilise l'annotation @Validated pour indiquer à Spring qu'il fau
// On récupère les erreurs de validation dans un objet BindingResult
  public ResponseEntity<Object> nouvellePersonne(@Validated @RequestBo
    // On parcourt les erreurs et on les concatène dans un StringBuild
      if (bindingResult.hasErrors()) {
          StringBuilder errors = new StringBuilder();
          // getFieldErrors() extrait les erreurs de validation
          bindingResult.getFieldErrors().forEach(error -> {
          // getDefaultMessage() récupère le message d'erreur
              errors.append(error.getField()).append(": ").append(erro
          });
          // On retourne une réponse avec le code 400 et le message d'
          return new ResponseEntity<>(errors.toString(), HttpStatus.BA
      }
      // On sauvegarde la personne
```

ERREUR

```
@PostMapping("/personnes")
// On utilise l'annotation @Validated pour indiquer à Spring qu'il fau
// On récupère les erreurs de validation dans un objet BindingResult
  public ResponseEntity<Object> nouvellePersonne(@Validated @RequestBo
    // On parcourt les erreurs et on les concatène dans un StringBuild
      if (bindingResult.hasErrors()) {
          StringBuilder errors = new StringBuilder();
          // getFieldErrors() extrait les erreurs de validation
          bindingResult.getFieldErrors().forEach(error -> {
          // getDefaultMessage() récupère le message d'erreur
              errors.append(error.getField()).append(": ").append(erro
          });
          // On retourne une réponse avec le code 400 et le message d'
          return new ResponseEntity<>(errors.toString(), HttpStatus.BA
      }
      // On sauvegarde la personne
```

ERREUR

```
@PostMapping("/personnes")
// On utilise l'annotation @Validated pour indiquer à Spring qu'il fau
// On récupère les erreurs de validation dans un objet BindingResult
  public ResponseEntity<Object> nouvellePersonne(@Validated @RequestBo
    // On parcourt les erreurs et on les concatène dans un StringBuild
      if (bindingResult.hasErrors()) {
          StringBuilder errors = new StringBuilder();
          // getFieldErrors() extrait les erreurs de validation
          bindingResult.getFieldErrors().forEach(error -> {
          // getDefaultMessage() récupère le message d'erreur
              errors.append(error.getField()).append(": ").append(erro
          });
          // On retourne une réponse avec le code 400 et le message d'
          return new ResponseEntity<>(errors.toString(), HttpStatus.BA
      }
      // On sauvegarde la personne
```

ERREUR

```
@PostMapping("/personnes")
// On utilise l'annotation @Validated pour indiquer à Spring qu'il fau
// On récupère les erreurs de validation dans un objet BindingResult
  public ResponseEntity<Object> nouvellePersonne(@Validated @RequestBo
    // On parcourt les erreurs et on les concatène dans un StringBuild
      if (bindingResult.hasErrors()) {
          StringBuilder errors = new StringBuilder();
          // getFieldErrors() extrait les erreurs de validation
          bindingResult.getFieldErrors().forEach(error -> {
          // getDefaultMessage() récupère le message d'erreur
              errors.append(error.getField()).append(": ").append(erro
          });
          // On retourne une réponse avec le code 400 et le message d'
          return new ResponseEntity<>(errors.toString(), HttpStatus.BA
      }
      // On sauvegarde la personne
```

ERREUR

```
@PostMapping("/personnes")
// On utilise l'annotation @Validated pour indiquer à Spring qu'il fau
// On récupère les erreurs de validation dans un objet BindingResult
  public ResponseEntity<Object> nouvellePersonne(@Validated @RequestBo
    // On parcourt les erreurs et on les concatène dans un StringBuild
      if (bindingResult.hasErrors()) {
          StringBuilder errors = new StringBuilder();
          // getFieldErrors() extrait les erreurs de validation
          bindingResult.getFieldErrors().forEach(error -> {
          // getDefaultMessage() récupère le message d'erreur
              errors.append(error.getField()).append(": ").append(erro
          });
          // On retourne une réponse avec le code 400 et le message d'
          return new ResponseEntity<>(errors.toString(), HttpStatus.BA
      }
      // On sauvegarde la personne
```

ERREUR

- On utilise l'annotation `@Validated` pour indiquer à Spring qu'il faut valider l'objet `Personne` passé en paramètre.
- On récupère les erreurs de validation dans un objet `BindingResult`.
- On parcourt les erreurs et on les concatène dans un `StringBuilder`.
- On retourne une réponse avec le code `400` et le message d'erreur.

suite