Validation des données et Gestion des exceptions

20. Validation des données (Data Validation)

La règle d'or en développement backend est la suivante : **ne faites jamais confiance aux données envoyées par l'utilisateur**. Avant d'enregistrer une information dans votre base de données, vous devez vous assurer qu'elle respecte vos critères (par exemple, un nom ne doit pas être vide, une adresse email doit avoir un format valide, etc.).

Spring Boot simplifie cela grâce au module de validation (implémenté via Hibernate Validator). Si vous ne l'avez pas déjà, vous devez ajouter la dépendance `spring-boot-starter-validation` à votre projet.

Étape 1 : Ajouter les annotations de validation sur le Modèle (ou DTO)

Nous allons modifier notre classe qui reçoit les données (généralement un DTO - Data Transfer Object) pour y ajouter des contraintes grâce aux annotations de la spécification *Jakarta Bean Validation*.

```
package com.studiogames.mysterybackend.dto;

import jakarta.validation.constraints.Max;
import jakarta.validation.constraints.Min;
import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.Size;

public class SuspectRequest {

    // @NotBlank vérifie que la chaîne n'est ni nulle, ni vide, ni composée uniquement
d'espaces
    @NotBlank(message = "Le nom du suspect est obligatoire")
    @Size(min = 2, max = 50, message = "Le nom doit contenir entre 2 et 50 caractères")
    private String nom;

    @NotBlank(message = "L'alibi ou la description ne peut pas être vide")
    private String description;

    // @Min et @Max valident les bornes d'un nombre
    @Min(value = 0, message = "Le niveau de suspicion ne peut pas être négatif")
    @Max(value = 100, message = "Le niveau de suspicion maximum est de 100")
    private int niveauSuspicion;

    // Getters et Setters...
}
```

Étape 2 : Activer la validation dans le Contrôleur (@Valid)

Pour que Spring vérifie ces contraintes lors d'une requête HTTP, il suffit d'ajouter l'annotation `@Valid` devant l'objet à valider dans votre contrôleur.

```
@RestController
@RequestMapping("/api/suspects")
public class SuspectController {

    @PostMapping
    public ResponseEntity<String> ajouterSuspect(@Valid @RequestBody SuspectRequest requete)
{
        // Si les données sont invalides, Spring bloque la requête AVANT même d'entrer dans
cette méthode.
        // Si on arrive ici, c'est que toutes les données (nom, description, niveau) sont
valides.

        return ResponseEntity.ok("Suspect ajouté avec succès !");
    }
}
```

21. Gestion globale des exceptions (Exception Handling)

Que se passe-t-il si un utilisateur envoie un nom vide à notre contrôleur ci-dessus ? Par défaut, Spring renvoie une erreur **400 Bad Request** avec un message JSON très long, complexe et illisible pour le client (comme une application mobile ou un site web).

Pour offrir une meilleure *User Experience (UX)* et faciliter le travail des développeurs frontend, nous devons intercepter ces erreurs et renvoyer une réponse JSON propre et structurée. C'est le rôle de **`@RestControllerAdvice`**.

Création du gestionnaire d'exceptions global (GlobalExceptionHandler)

Cette classe agit comme un filet de sécurité qui attrape toutes les erreurs (exceptions) lancées par n'importe quel contrôleur de votre application.

```
package com.studiogames.mysterybackend.exception;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.validation.FieldError;
import org.springframework.web.bind.MethodArgumentNotValidException;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.RestControllerAdvice;

import java.util.HashMap;
```

```
import java.util.Map;

// @RestControllerAdvice indique que cette classe intercepte les exceptions et renvoie du
JSON
@RestControllerAdvice
public class GlobalExceptionHandler {

    /**
     * Intercepte spécifiquement les erreurs de validation déclenchées par @Valid.
     */
    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<Map<String, String>>
handleValidationExceptions(MethodArgumentNotValidException ex) {

        Map<String, String> erreurs = new HashMap<>();

        // On parcourt toutes les erreurs de validation trouvées par Spring
        ex.getBindingResult().getAllErrors().forEach((error) -> {
            // On extrait le nom du champ qui a échoué (ex: "nom")
            String nomChamp = ((FieldError) error).getField();
            // On extrait le message d'erreur défini dans notre classe (ex: "Le nom du
suspect est obligatoire")
            String messageErreur = error.getDefaultMessage();

            erreurs.put(nomChamp, messageErreur);
        });

        // On renvoie un code 400 Bad Request avec notre map propre
        return new ResponseEntity<>(erreurs, HttpStatus.BAD_REQUEST);
    }

    /**
     * Vous pouvez ajouter d'autres méthodes pour gérer vos propres erreurs.
     * Exemple : Intercepter une exception personnalisée quand un élément n'est pas trouvé.
     */
    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<String> handleResourceNotFound(ResourceNotFoundException ex) {
        return new ResponseEntity<>(ex.getMessage(), HttpStatus.NOT_FOUND); // Code 404
    }
}
```

Le résultat magique :

Désormais, si un client envoie une requête POST avec un niveau de suspicion de `150` et un nom vide, au lieu d'une erreur serveur incompréhensible, il recevra ce JSON propre et précis :

```
{
  "nom": "Le nom du suspect est obligatoire",
  "niveauSuspicion": "Le niveau de suspicion maximum est de 100"
}
```

> C'est exactement ce dont un client front-end a besoin pour afficher des messages d'erreur en rouge sous ses formulaires !

Rédigé et structuré par : ***Med Khalil Kribi***

Source : https://apcpedagogie.com/validation-donnees-et-gestion-exceptions-spring · téléchargé le 07/11/2026 · © APC Pédagogie · apcpedagogie.com

apcpedagogie.com