Spring Boot Starter Web : exposer et consommer des APIs REST

Contexte & Objectif

- Construire des APIs REST rapidement et proprement
- Comprendre les outils à disposition : `@RestController`, `RestTemplate`, `WebClient`
- Savoir capturer et gérer les erreurs élégamment

Slide d'introduction : Pourquoi ce sujet ? Qu'allons-nous voir ensemble ?

Pourquoi `spring-boot-starter-web` ?

Dès que votre application doit dialoguer avec l'extérieur via HTTP, ce starter est votre point de départ.

- Inclut Tomcat en embedded
- Fournit Spring MVC
- Simplifie la création d'APIs REST

Exposer une API REST : `@RestController`

```
@RestController
@RequestMapping("/books")
public class BookController {

    @GetMapping("/{id}")
    public Book findById(@PathVariable Long id) {
        return service.getBookById(id);
    }

    @PostMapping
    public Book create(@RequestBody Book book) {
        return service.save(book);
    }
}
```

Expose automatiquement en JSON via Jackson

Mapping d'URLs

- `@GetMapping("/path")` → GET
- `@PostMapping("/path")` → POST
- `@PutMapping("/path")` → PUT
- `@DeleteMapping("/path")` → DELETE

Validation des entrées : les bases

- Permet de vérifier automatiquement les données entrantes
- Utilise la spécification `Bean Validation` (JSR-380)
- Fonctionne avec l'annotation `@Valid`

Validation automatique des requêtes JSON entrantes dans vos contrôleurs.

Ajouter la dépendance Validation

Si non incluse automatiquement (ex: projet sans starter complet) :

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

- `jakarta.validation-api` → spécification
- `hibernate-validator` → implémentation de référence

Utiliser les annotations de validation

Exemple de DTO avec validation standard :

```
public class BookRequest {

    @NotNull
    private String title;

    @NotBlank
    private String author;
```

```
    @Min(1)
    private int pageCount;

    @PastOrPresent
    private LocalDate publicationDate;

    // getters et setters
}
```

Et dans le Controller :

```
@PostMapping("/books")
public Book createBook(@Valid @RequestBody BookRequest bookRequest) {
    return service.save(bookRequest);
}
```

Validation personnalisée : besoin spécifique

Quand les contraintes standard ne suffisent pas (ex : ISBN valide).

1. Créer une annotation custom

```
@Documented
@Constraint(validatedBy = IsbnValidator.class)
@Target({ ElementType.FIELD })
@Retention(RetentionPolicy.RUNTIME)
public @interface ValidIsbn {
    String message() default "ISBN invalide";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}
```

2. Implémenter la logique métier

```
public class IsbnValidator implements ConstraintValidator<ValidIsbn, String> {

    @Override
    public boolean isValid(String isbn, ConstraintValidatorContext context) {
        if (isbn == null) return false;
        return isbn.matches("\\d{10}|\\d{13}");
    }
}
```

- `ConstraintValidator<AnnotationType, ChampType>`
- Implémenter la méthode `isValid`

Utilisation d'une validation custom

```
public class BookRequest {

    @ValidIsbn
    private String isbn;

    // autres champs...
}
```

- Transparent pour les développeurs
- Déclenche automatiquement la validation
- Message personnalisé dans la réponse

Best practices de validation

- **Toujours valider** les entrées extérieures (POST/PUT/PATCH)
- **Séparer** entités persistantes (Entity) et objets d'API (DTO)
- **Personnaliser** les messages d'erreurs pour être compréhensibles
- **Centraliser** la gestion des erreurs via `@ControllerAdvice`

La validation

- **Facile** avec les annotations standards (`@NotNull`, `@Size`, etc.)
- **Extensible** avec vos propres règles
- **Indispensable** pour la robustesse et la sécurité

Slide pour conclure sur l'importance de la validation dans toute API REST professionnelle.

Consommer une API REST : `RestTemplate` (legacy)

```
RestTemplate restTemplate = new RestTemplate();
Book book = restTemplate.getForObject(
    "http://localhost:8080/books/1", Book.class
);
```

- Simple à utiliser
- Synchronisé (bloquant)
- Déprécié au profit de WebClient

Consommer une API REST : `WebClient` (réactif)

```
WebClient webClient = WebClient.create();

Book book = webClient.get()
    .uri("http://localhost:8080/books/1")
    .retrieve()
    .bodyToMono(Book.class)
    .block();
```

- Basé sur Project Reactor
- Non-bloquant
- S'intègre dans une approche réactive

Pourquoi gérer les erreurs proprement ?

- Donner des réponses claires aux clients d'API
- Eviter les fuites d'informations sensibles
- Faciliter le debugging et la maintenance

Transition avant de parler de `@ControllerAdvice`.

Gérer les erreurs avec `@ControllerAdvice`

- Centraliser la gestion des erreurs
- Attraper les exceptions pour renvoyer des réponses propres

```
@RestControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(BookNotFoundException.class)
    public ResponseEntity<String> handleBookNotFound(BookNotFoundException ex) {
        return ResponseEntity.status(HttpStatus.NOT_FOUND)
                             .body(ex.getMessage());
    }
}
```

Isole la gestion d'erreur en un seul point = code plus propre et plus maintenable.

Exemple plus complet : Erreur standardisée

```
public record ApiError(
    String message,
    int status,
    Instant timestamp
) {}

@ExceptionHandler(Exception.class)
public ResponseEntity<ApiError> handleException(Exception ex) {
    ApiError error = new ApiError(
        ex.getMessage(),
        HttpStatus.INTERNAL_SERVER_ERROR.value(),
        Instant.now()
    );
    return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(error);
}
```

- Réponses JSON structurées
- Timestamp utile pour le suivi

Best practices pour `@ControllerAdvice`

- Créer des classes d'exceptions spécifiques (ex: `BookNotFoundException`)
- Ne pas exposer de stacktrace
- Toujours renvoyer un corps JSON clair
- Ajouter un `RequestId` si besoin pour le traçage

Résumé

- `spring-boot-starter-web` simplifie la création d'APIs
- `WebClient` pour les consommations modernes
- `@ControllerAdvice` pour capturer proprement toutes les erreurs
- Objectif : des APIs robustes, claires et professionnelles

Questions ?

"Un bon design d'API commence par une bonne gestion de ses erreurs."

Slide final pour laisser place aux questions.