



Menu principal

Master IoT 1a (/index.php/master-iot-1a)	
Wiki mathématique (https://cours-info.iut-bm.univ-fcomte.fr/pmwiki/pmwiki.php)	Page d'accueil (/index.php)
Documentation (/index.php/menu-docs-sp-323795567)	S1 (/index.php/menu-cours-s1)
S2 (/index.php/menu-cours-s2)	S3 (/index.php/menu-cours-s3)
	S4 (/index.php/menu-cours-s4)
S5 (/index.php/menu-lpsil)	S6 (/index.php/s6)

normnext.com (http://normnext.com)

Connexion

 Identifiant

 Mot de passe

☐ Se souvenir de moi

Connexion

Identifiant oublié ? (/index.php/component/users/?view=remind&Itemid=125)

Mot de passe oublié ? (/index.php/component/users/?view=reset&Itemid=125)

R6.05 - Prog. Frameworks : Spring boot


TD n°2 : gestion d'erreur, mapping de requêtes, et plus encore

Détails



Écrit par stéphane Domas

Catégorie : R6.05 - Prog. Framework : Spring boot (/index.php/s6/objets-connectes)

 Publication : 31 décembre 2020

 Affichages : 784

Préambule

La démonstration 2 du TD 1 comporte des faiblesses au niveau du code, ainsi que des situations non traitées, avec notamment :

- le cas où un objet valant null est renvoyé par un service, ce qui conduit à une réponse http vide,
- les requêtes http avec un "body", par exemple dans le cas de requête POST, PUT, ...

L'objectif de ce cours est de présenter comment traiter ces deux points particuliers, tout en décrivant quelques fonctionnalités bien pratiques, à savoir la pagination des résultats de requête, le paramétrage "manuel" d'une réponse http, ou encore la lecture d'un fichier csv (soit intégré à l'application, soit uploadé) pour remplir la BdD

Le projet de démonstration des points abordés est téléchargeable [[ici \(/upload/supports/S6/SpringBoot/TD/td2/springboot-td2-src.tgz\)](#)]

1°/ Gestion d'erreurs

Par défaut, si une route demandée n'est pas valide, ou qu'il y a une erreur interne qui génère une exception, spring renvoie au client une page html basique. Dans le cas d'une API REST, c'est plutôt gênant puisqu'il faudrait plutôt renvoyer un objet JSON représentant l'erreur, avec au minimum un descriptif de l'erreur.

Il est relativement simple de changer ce comportement par défaut, en ajoutant quelques classes pour représenter les cas d'erreur ainsi que ces classes qui vont "intercepter" les exceptions non traitées pour en faire des réponses http comme on le désire. Plus en détail, il s'agit au minimum de :

- créer une classe d'exception (par exemple, héritant de `RuntimeException`) pour représenter des cas d'erreur lors des requêtes. Par exemple, cela peut être une classe `ResourceNotFoundException`, représentant le fait que l'on n'a rien trouvé en BdD par rapport aux paramètres de la requête.
- créer une classe permettant de représenter les objets JSON de type erreur que l'API va renvoyer au client. Par exemple, on peut créer une classe `ErrorDTO`, contenant un numéro et un message d'erreur.
- créer une classe annotée avec `@RestControllerAdvice`, qui va définir des méthodes de type "gestionnaires d'exception", celles-ci retournant un objet décrivant l'erreur. Par exemple, cette classe peut être nommée `ExceptionHandler`, qui définit un gestionnaire d'exception `resourceNotFoundHandler()`, qui retourne un `ErrorDTO`.

Au delà de ce minimum, il est conseillé de mettre en place une stratégie de gestion des erreurs plus complète, avec par exemple :

- Une (ou plusieurs) classe(s) listant les erreurs possibles grâce à une `HashTable<Integer, String>` dont la clé est le numéro d'une erreur et la valeur le message d'erreur.
- De faire plusieurs classes d'exception pour chacun des grands types d'exception, voire des sous-classes à celles-ci.
- D'ajouter un attribut indiquant le contexte qui a conduit à une erreur dans la classe servant à représenter les objets JSON d'erreur transmis aux client. Par exemple, si une ressource n'est pas trouvée, le contexte pourrait être une chaîne de caractères indiquant la requête qui a conduit à l'échec.
- D'ajouter des attributs aux classes d'exception pour que les gestionnaires puissent facilement retourner une instance d'objet représentant l'erreur. Avec les hypothèses ci-dessus, cela implique d'ajouter des attributs pour le numéro et le contexte, le message étant déjà par défaut dans toute classe d'exception.

Dans la démonstration, ces différents points sont utilisés, avec en premier lieu la classe `ErrorList` dont voici un extrait :

```
1 public final class ErrorList {
2
3     public static final Integer NOERROR = 0;
4     public static final Integer RESOURCE_NOT_FOUND = 100;
5     public static final Integer HERO_NOT_FOUND = 101;
6
7     public static Map<Integer, String> errorList;
8     static {
9         errorList = new HashMap<>();
10        errorList.put(NOERROR, "no error");
11        errorList.put(RESOURCE_NOT_FOUND, "resource not found");
12        errorList.put(HERO_NOT_FOUND, "hero not found");
13    }
14    ... // + methods to get/add a couple from/in the map
15 }
```

Comme on le constate, c'est une classe utilisée de façon purement statique, sans jamais créer d'instance. A noter que dans le cas de gros projets, avec plusieurs dizaines/certaines d'erreurs possibles, il serait judicieux de créer plusieurs fichiers de ce type pour chacun des grands type d'erreur.

La classe principale représentant une ressource non trouvée est `ResourceNotFoundException` :

```
1 public class ResourceNotFoundException extends RuntimeException {
2
3     protected Integer errorNumber;
4     protected String context;
5
6     public ResourceNotFoundException(Integer errorNumber, String context, String message) {
7         super(message);
8         this.errorNumber = errorNumber;
9         this.context = context;
10    }
11
12    public ResourceNotFoundException(String context) {
13        this(ErrorList.RESOURCE_NOT_FOUND, context, ErrorList.getMessage(ErrorList.RESOURCE_N
14    }
15    ...
16    public Integer getErrorNumber() { return errorNumber; }
17    public String getContext() { return context; }
18 }
```

Le constructeur normalement utilisé est celui qui n'a que le contexte en paramètre. On voit qu'il utilise `ErrorList` pour mettre la "bonne" valeur pour le numéro et le message. Le deuxième constructeur est fait pour être utilisé par les sous-classes, afin de changer le numéro et le message en fonction du type de la sous-classes. Par exemple, on peut créer une sous-classe `HeroNotFoundException` pour représenter le cas particulier d'une ressource de type héros non trouvée :

```
1 public class HeroNotFoundException extends ResourceNotFoundException {
2
3     public HeroNotFoundException(String context) {
4         super(ErrorList.HERO_NOT_FOUND, context, ErrorList.getMessage(ErrorList.HERO_NOT_FOUN
5     }
6 }
```

Pour "traduire" une exception en un objet JSON erreur renvoyé au client, on crée la classe `ErrorDTO` comme suivant :

```
1 public class ErrorDTO {
2
3     private Integer errorNumber;
4     private String context;
5     private String message;
6
7     public ErrorDTO(Integer number, String context, String message) {
8         this.errorNumber = number;
9         this.context = context;
10        this.message = message;
11    }
12    ... // + all possible getters/setter & toString()
13 }
```

Enfin, pour gérer les exceptions, on crée la classe `ControllerAdvice`, comme suivant :

```

1 | @RestControllerAdvice
2 | public class ControllerAdvice {
3 |
4 |     @ExceptionHandler(ResourceNotFoundException.class)
5 |     @ResponseStatus(HttpStatus.NOT_FOUND)
6 |     ErrorDTO resourceNotFoundHandler(ResourceNotFoundException ex) {
7 |         return new ErrorDTO(ex.getErrorNumber(), ex.getContext(), ex.getMessage());
8 |     }
9 | }

```

Remarques importantes :

- l'annotation `@ExceptionHandler` avant une méthode permet de spécifier la classe mère d'exception à traiter par celle-ci. Cela implique en l'occurrence que la méthode `resourceNotFoundHandler()` va être capable de traiter aussi bien les exception de type `ResourceNotFoundException` que de toutes ses sous-classes, donc aussi `HeroNotFoundException`.
- l'annotation `@ResponseStatus` permet de spécifier le status http de la réponse
- si la façon de renvoyer une erreur au client est unique quelle que soit l'erreur, c.a.d. même type d'objet et même status http (comme dans cette démo), on peut donc se contenter d'une seule méthode gestionnaire d'exception pour toute l'application.
- la méthode gestionnaire d'exception doit retourner le corps de la réponse. Si c'est un objet, comme dans le cas présent avec une instance de `ErrorDTO`, spring fera tout seul la traduction en JSON

Pour que les méthodes du gestionnaire d'exception soient utilisées, il suffit que les méthodes de contrôle associées aux routes renvoient un exception au lieu d'un objet réponse. Pour ce faire, il vaut mieux que ce soient les services qui créent l'exception et que le contrôle se contente de la propager (-> meilleur découplage). Par exemple, dans la méthode de recherche de héros par id dans `HeroesService`, on écrit :

```

1 | @Service
2 | public class HeroesService {
3 |     ...
4 |     public Hero findHeroById(Long id) throws ResourceNotFoundException {
5 |         Optional<Hero> hero = heroesRepository.findById(id);
6 |         if (hero.isEmpty()) throw new HeroNotFoundException("searching for hero with id = "+id);
7 |         return hero.get();
8 |     }
9 |     ...
10 | }

```

Grâce à `throws ResourceNotFoundException`, cette méthode va propager une éventuelle exception à la méthode appelante. On remarque qu'en fait, elle peut générer une instance de `HeroNotFoundException`, ce qui ne pose aucun problème puisque c'est une sous-classe de `ResourceNotFoundException`.

Dans le contrôleur `HeroesController`, on écrit :

```
1  @RestController
2  public class HeroesController {
3      ...
4      // get hero by id
5      @GetMapping("/heroes/{id}")
6      public Hero getHeroById(@PathVariable Long id) throws ResourceNotFoundException {
7          Hero hero = heroesService.findHeroById(id);
8          return hero;
9      }
10     ...
11 }
```

Il n'y a aucun try/catch lors de l'appel au service, ce qui est un des deux conditions pour que la méthode propage l'exception. L'autre est d'indiquer `throws ResourceNotFoundException` dans son entête (comme dans le service)

Démonstration :

- tester l'URL : `http://localhost:8080/heroes/10` (`http://localhost:8080/heroes/10`). La réponse est : `{"errorNumber":101,"context":"searching for hero with id = 10","message":"hero not found"}`. Toute la chaîne de gestion d'exception a donc bien fonctionné comme prévu, notamment la méthode gestionnaire d'exception malgré le fait que le type d'instance est `HeroNotFoundException` et pas `ResourceNotFoundException`.
- tester l'URL : `http://localhost:8080/heroes/getbypublicname/aaa` (`http://localhost:8080/heroes/getbypublicname/aaa`). La réponse est cette fois `{"errorNumber":100,"context":"searching for hero with publicName = aaa","message":"resource not found"}`. Normal car la méthode de service génère une instance de `ResourceNotFoundException` (juste pour l'exemple car pas cohérent)

2°/ mappings

On utilise généralement des requête de type POST pour envoyer des données impliquant la création de données en BdD, et PUT (ou PATCH) pour mettre à jour. Dans les deux cas, les données sont dans le corps de la requête et spring sait extraire ces données pour les représenter par un objet. Pour cela, il suffit de :

- créer une classe représentant l'objet JSON à recevoir,
- utiliser l'annotation `@RequestBody` devant un paramètre de la méthode associée à un mapping de type POST ou PUT.

La seule subtilité consiste à bien définir la classe, pour notamment tenir compte qu'une requête PUT permet de mettre à jour seulement certaines colonnes en BdD. C'est pourquoi, il est conseillé que cette classe n'utilise que des attributs objet, même pour les types primaires. On peut ainsi facilement vérifier si ils sont null, ce qui implique qu'il n'y a pas leur équivalent dans le JSON reçu.

Dans la démonstration, on définit ainsi la classe `HeroDTO` comme suivant :

```
1 public class HeroDTO {
2     private Long id;
3     private String publicName;
4     private String realName;
5     private String power;
6     private Integer powerLevel;
7
8     public HeroDTO(Long id, String publicName, String realName, String power, Integer powerLevel) {
9         this.id = id;
10        this.publicName = publicName;
11        this.realName = realName;
12        this.power = power;
13        this.powerLevel = powerLevel;
14    }
15    ... // + all getters/setters & toString()
16 }
```

ATTENTION : quand le client enverra un objet JSON à l'API, il faudra que les champs aient le même nom que les attributs de `HeroDTO`.

Pour créer des méthodes associées à des requête POST et PUT, on utilise les annotations `@PostMapping` et `@PutMapping`. Par exemple, pour ajouter ou mettre à jour un héros, on écrit dans `HeroesController` :

```
1 // create a hero
2 @PostMapping("/heroes")
3 public Hero createHero(@RequestBody HeroDTO hero) {
4     return heroesService.createHero( hero.getPublicName(), hero.getRealName(), hero.getPower());
5 }
6
7 // update hero by id
8 @PutMapping("/heroes")
9 public Hero updateHero(@RequestBody HeroDTO hero) throws ResourceNotFoundException {
10     return heroesService.updateHero(hero.getId(), hero.getPublicName(), hero.getRealName(), hero.getPower());
11 }
```

On remarque que ces méthodes n'ont qu'un seul paramètre qui est un objet représentant tout l'objet JSON reçu, en l'occurrence un objet `HeroDTO`. A noter que si par exemple le JSON reçu n'a pas de champ `power`, alors l'attribut `power` de l'objet passé en paramètre sera `null`.

Il suffit enfin d'écrire les méthodes de service associées, pour créer ou mettre à jour en BdD. Par exemple, dans `HeroesService`, on écrit :

```
1      public Hero createHero(String publicName, String realName, String power, Integer powerLevel) {
2          Hero hero = new Hero(publicName, realName, power, powerLevel);
3          heroesRepository.save(hero);
4          return hero;
5      }
6
7      public Hero updateHero(Long id, String publicName, String realName, String power, Integer
8          Optional<Hero> heroOpt = heroesRepository.findById(id);
9          if (heroOpt.isEmpty()) throw new HeroNotFoundException("updating hero with id = "+id);
10         Hero hero = heroOpt.get();
11         if (publicName != null) hero.setPublicName(publicName);
12         if (realName != null) hero.setRealName(realName);
13         if (power != null) hero.setPower(power);
14         if (powerLevel != null) hero.setPowerLevel(powerLevel);
15         heroesRepository.save(hero);
16         return hero;
17     }
```

On remarque que pour `updateHero()`, il est très facile de ne mettre à jour que les champs fournis dans le JSON, simplement en testant si les paramètres sont `null` ou pas.

Démonstration : utiliser postman pour ajouter puis modifier un héros.

3° manipuler les réponses http

Dans les exemples précédents, les méthodes de contrôle se contentent de directement renvoyer un objet `Hero` ou bien `List<Hero>`, qui seront transformés automatiquement en JSON par spring, avec normalement un status = 200 (c.a.d. OK). Cependant, il est parfois utile de pouvoir manipuler plus finement la réponse http, notamment pour spécifier un autre status, ou pour ajouter des entêtes de réponse.

Pour cela, on modifie la valeur de retour des contrôleurs en utilisant la classe `ResponseEntity<T>`, où `T` est le type d'objet qui est utilisé pour construire le corps de la réponse. Ensuite, dans le contrôleur, on crée une instance avec `new` ou avec ses méthodes statiques puis on retourne cette instance. Parmi les méthodes disponibles, il y a notamment :

- `status(HttpStatus status)`, permettant de spécifier un status bien précis,
- `ok()` : pour utiliser le status 200,
- `body(T t)` : pour "donner" l'objet `t` servant de corps de la réponse.
- `header(String header, String value)` : pour ajouter des entêtes.

Exemple d'utilisation dans `HeroesController` :

```

1 // example with response manipulation
2 @GetMapping("/heroes/getbypower")
3 public ResponseEntity<List<Hero>> getHeroesByPower(@RequestParam String power, @RequestPa
4     List<Hero> list = null;
5     if (pattern == null) pattern = false;
6     list = heroesService.findAllByPower(power,pattern);
7
8     // if list is empty, do not send the empty list but instead a NO_CONTENT status code
9     if (list.isEmpty()) return ResponseEntity.status(HttpStatus.NO_CONTENT).body(null);
10    // if list is not empty, send it, together with 2 customs response headers
11    return ResponseEntity
12        .ok()
13        .header("myheader","toto")
14        .header("otherheader","hello")
15        .body(list);
16    }

```

Démonstration :

- tester l'URL : <http://localhost:8080/heroes/getbypower?power=qqqq> (<http://localhost:8080/heroes/10>). Dans l'inspecteur chrome, on remarque que la réponse est bien vide et que le status est 204 (pas de contenu)
- tester l'URL : <http://localhost:8080/heroes/getbypower?power=pai> (<http://localhost:8080/heroes/10>). Dans l'inspecteur chrome, on remarque que le status de la réponse est 200 (ok), et qu'il y a bien 2 entêtes de réponse `myheader` et `otherheader`.

4°/ paginer des résultats de requête

Même si les capacités des réseaux vont s'accroissant, il est toujours bien de faire attention de ne pas échanger des volumes de données trop importants. C'est pourquoi il est fréquent de recourir à la pagination pour récupérer des données potentiellement volumineuses.

La pagination consiste simplement à demander des données par morceaux en indiquant la taille d'un morceau et le numéro du morceau. Mais au lieu de parler de morceau, on utilise plutôt le terme historique de page car les données étaient généralement ensuite affichées comme sur une page. De plus, il est souvent possible de récupérer les données en les triant selon certains champs, par ordre croissant ou décroissant.

Cette fonctionnalité étant centrale et importante pour la performance d'une grosse applicaiton, spring propose un mécanisme qui automatise quasiment tout. Pour cela, il faut :

- utiliser un repository de type `PagingAndSortingRepository`, ce qui est le cas du `JpaRepository` utilisé dans la démonstration. Cela permet d'avoir accès à un méthode `Page<T> findAll(Pageable p)` qui permet de récupérer toutes les données de façon paginée.
- si besoin déclarer dans ce repository des méthodes plus précises, en leur ajoutant un paramètre `Pageable`. Ces méthodes doivent retourner un objet de type `Page<T>` ou `T` est le type d'objet géré par le repository. Par exemple `Page<Hero>indAllByRealName(String realName, Pageable p)`
- créer des méthodes de service qui prend en paramètre un objet `Pageable` et qui font appel à celles du repository.
- créer des méthodes contrôleurs qui prennent en paramètre un objet `Pageable` et qui font appel à celles du service.

Quand spring voit un contrôleur avec un paramètre `Pageable`, il récupère automatiquement dans les paramètre de requêtes les valeurs des champs `page`, `size` et `sort` pour créer un objet `Pageable`. Le problème est que la requête ne contient pas forcément tous ces champs. Il est donc possible de spécifier à spring des valeurs par défaut, grâce aux annotation `@PageableDefault` et `@SortDefault`.

Voici un exemple de pagination pour récupérer des héros. Dans le classe `HeroesService`, on ajoute :


```

1      public Page<Hero> findAllWithPagination(Pageable pageable) {
2          return heroesRepository.findAll(pageable);
3      }

```

Remarque : pas besoin de déclarer `findAll()` dans `HeroesRepository` puisqu'elle fait partie des méthodes héritées.

Dans `HeroesController`, on ajoute :

```

1  @GetMapping("/heroes/paging")
2      public Page<Hero> getAllHeroesWithPagination(@PageableDefault(page = 0, size = 5)
3                                                     @SortDefault.SortDefaults({
4                                                         @SortDefault(sort = "id", direction = Sort.Direction.ASC)
5                                                     })Pageable pageable) {
6      return heroesService.findAllWithPagination(pageable);
7  }

```

Et c'est tout !

Démonstration :

- tester l'URL : `http://localhost:8080/heroes/paging?page=0&size=3` (`http://localhost:8080/heroes/10`). On obtient un objet complexe, avec notamment le champ `content` qui est une liste de 3 héros triés par id croissant. Il y a aussi des champs très utiles tels que `totalPages` (= nombre total de pages), `last` (si dernière page ou non), `offset` (= `page*size`)
- tester l'URL : `http://localhost:8080/heroes/paging?page=1&size=3&sort=powerLevel,desc` (`http://localhost:8080/heroes/10`). On obtient cette fois une liste de héros triés par niveau de pouvoir décroissant.
- tester l'URL : `http://localhost:8080/heroes/paging?page=5` (`http://localhost:8080/heroes/10`). On obtient normalement une liste vide, puisqu'il n'y a pas suffisamment de héros pour avoir une page 5.

5° Utiliser des fichiers CSV

Le TD n°1 a présenté comment remplir les tables de la BdD à la fin de l'initialisation de l'application, de deux façons : via un fichier ressource `data.sql` ou bien via des instructions dans un Runner. Il y a bien entendu d'autres solutions, dont la plus courante consiste à utiliser des fichiers csv.

Selon l'application, ce type de fichier peut être "intégré" aux ressources de l'application ou bien peut être uploadé via une route. Dans les deux cas, l'objectif est de lire le contenu de ce fichier via un service pour créer des instances d'entités, que l'on sauve ensuite dans le BdD via un repository.

La première étape consiste donc à écrire un service capable d'une telle chose. Pour cela, le plus simple consiste à utiliser le package Apache de manipulation de csv. Pour les inclure au projet, il suffit d'éditer le `pom.xml` en ajoutant dans les dépendances :

```

1  <dependency>
2      <groupId>org.apache.commons</groupId>
3      <artifactId>commons-csv</artifactId>
4      <version>1.8</version>
5  </dependency>

```

Ce package permet notamment d'utiliser les classes `CSVParser` et `CSVRecord` pour lire un csv et en extraire des enregistrements.

La classe représentant le service n'est pas très compliquée à écrire et l'on peut la réutiliser dans différents projets. Sa structuration est assez simple, avec :

- une méthode "générale" permettant de lire un flux d'octet (`InputStream`) venant d'un fichier csv, et d'obtenir une liste

de données de la forme `Iterable<CSVRecord>`.

- des méthodes pour convertir des `CSVRecord` en une liste d'entité d'un certain type,
- des méthodes qui vont ouvrir un fichier csv pour obtenir un flux d'octet, appeler les méthodes précédentes pour obtenir des liste d'entités, pour enfin sauver celles-ci en BdD.

Les dernières méthodes peuvent au choix prendre en entrée un fichier csv sous la forme :

- d'une instance de la classe `Resource`, lorsque le fichier csv se trouve dans le répertoire `resources` du projet,
- d'une instance de `MultipartFile`, lorsque le fichier est reçu via une requête à l'application.

La deuxième étape consiste à créer utiliser ce service :

- soit dans un contrôleur avec une route pour uploader un fichier, en écrivant quelque chose du genre :

```
1  @RestController
2  public class HeroesController {
3      private final CSVService csvService;
4      ... // other attributes
5
6      @Autowired
7      HeroesController(CSVService csvService, ...) {
8          this.csvService = csvService;
9          ...
10     }
11     @PostMapping("/heroes/upload")
12     public void uploadCSVForHeroes(@RequestParam("file") MultipartFile file) throws CSVConversionException {
13         csvService.saveCSVToHeroes(file);
14     }
15 }
```

- soit dans un runner, qui utilise un fichier ressource, avec par exemple :

```
1  @Component
2  public class MyRunner implements CommandLineRunner {
3      private final CSVService csvService;
4      @Value("classpath:heroes.csv")
5      Resource resourceFile;
6
7      @Autowired
8      public MyRunner(CSVService csvService) {
9          this.csvService = csvService;
10     }
11     @Override
12     @Transactional
13     public void run(String... args) throws Exception {
14         csvService.saveCSVToHeroes(resourceFile);
15     }
16 }
```

Démonstration :

- lancer l'application. Normalement, dans le terminal, on constate que les 5 héros du fichier ressource `heroes.csv` ont bien été ajoutés. On peut également le constater via la console h2.
- tester avec postman l'URL : `http://localhost:8080/heroes/upload`, (`http://localhost:8080/heroes/upload`), en utilisant des paramètres suivants, comme indiqué dans la figure qui suit :


- type de body = form-data
- paramètres du contenu : clé = file, type = file.
- valeur du contenu : un fichier csv présent sur le disque. Celui indiqué ci-dessous se trouve dans le répertoire demo2 de l'archive de démonstration.

request for csv / **upload a csv with toher heroes**

POST localhost:8080/heroes/upload

Params Authorization Headers (8) **Body** Scripts Tests Settings

☐ none ☒ form-data ☐ x-www-form-urlencoded ☐ raw ☐ binary ☐ GraphQL

	Key		Value
<input checked="" type="checkbox"/>	<u>file</u>	File ▾	 otherheroes.csv
	Key	Text ▾	Value

- récupérer ensuite la liste de tous les héros (<http://localhost:8080/heroes> (<http://localhost:8080/heroes>)) pour constater que l'insertion a eu lieu correctement.
- modifier le fichier csv pour qu'il soit invalide, par exemple en changeant le nom d'une entête, puis relancer la requête dans postman. Cette fois, on obtient un JSON représentant une erreur.

◀ Précédent (/index.php/s6/objets-connectes/2580-td-n-6-fonctionnalites-avancees-acces-a-un-api-avec-axios-4)

Suivant ▶ (/index.php/s6/objets-connectes/2584-td-n-6-fonctionnalites-avancees-acces-a-un-api-avec-axios-7)