

Tutoriel pour apprendre à créer une API REST, avec Java et Vert.x, en 5 minutes

Par [Thierry-Leriche-Dessirier](#)

Date de publication : 14 octobre 2019

Vert.x est une API asynchrone très proche du modèle d'acteurs. Vert.x est polyglotte, simple, scalable (élastique) et hautement concurrente. Vert.x est bien adapté aux architectures en microservices.

Dans cet article rapide, nous allons voir comment créer une API standard avec Vert.x. Et pour cela, on se donne 5 minutes... [Commentez](#)

I - Introduction.....	3
II - Domaine.....	3
III - Créer un serveur.....	4
III-A - Installation.....	4
III-B - Le verticle en charge de l'API.....	4
III-C - Ecouter un port.....	5
IV - Notre API naïve.....	6
IV-A - Charger une liste d'objets.....	6
IV-B - Charger un seul objet.....	9
IV-C - Envoyer une erreur.....	10
IV-D - Créer un objet.....	11
IV-E - Modifier un objet.....	13
IV-F - Supprimer un objet.....	14
V - Réorganiser le routeur.....	16
VI - Conclusions.....	17
VII - Remerciements.....	17
VIII - Annexes.....	17
VIII-A - Liens.....	17
VIII-B - Codes complets.....	17

I - Introduction

Créer une API en Java est un sujet déjà largement traité. Si vous utilisez Jersey et ses annotations, vous savez que c'est très simple et rapide à mettre en place. Dans cet article, je voudrais simplement vous montrer ce qu'on peut faire avec Vert.x en revenant aux fondamentaux. Et en vérité, la création d'une API sera avant tout un prétexte.

II - Domaine

Si vous lisez régulièrement mes articles, vous savez qu'ils parlent tous de chiens. C'est un domaine simple qui ne nécessite pas de longues explications. Le *bean* Dog suivant sera suffisant pour cette chronique.

Dog.java

```
1. public class Dog {
2.     private final String id;
3.     private final String name;
4.     private final String race;
5.     private final int age;
6.     // Constructeur
7.     public Dog(final String id, final String name, final String race, final int age) {
8.         super();
9.         this.id = id;
10.        this.name = name;
11.        this.race = race;
12.        this.age = age;
13.    }
14.    //
15.    // + getters
```

Je vous propose également DogService qui fera office de service, avec les opérations classiques. Un code complet naïf est donné en annexes.

DogService.java

```
1. public class DogService {
2.     // Charcher la liste des chiens
3.     public List<Dog> findAll() {
4.         ...
5.     }
6.     // Chercher un seul chien
7.     public Dog findById(final String id) {
8.         ...
9.     }
10.    // Mettre à jour un chien
11.    public Dog update(final Dog dog) {
12.        ...
13.    }
14.    // Effacer un chien
15.    public void remove(final String id) {
16.        ...
17.    }
18.    // Ajouter un chien
19.    public Dog add(final Dog dog) {
20.        ...
21.    }
```

Et finalement, je vous propose d'initialiser une liste d'exemples composée des chiens célèbres.

DogService.java

```
1. public class DogService {
2.     private final Map<String, Dog> dogs = new HashMap<String, Dog>();
3.     // Constructeur
4.     public DogService() {
5.         super();
6.         initDogs();
```

DogService.java

```
7.  }
8.  // Initialisation des chiens fictifs
9.  private void initDogs() {
10.     // Création de chiens fictifs
11.     final Dog lassie = new Dog("12345fh", "Lassie", "colley", 12);
12.     final Dog milou = new Dog("e7654", "Milou", "fox-terrier", 11);
13.     final Dog scoobydoo = new Dog("s93d78", "Scooby-Doo", "Danois", 7);
14.     final Dog idefix = new Dog("6222mk9p", "Idexif", "Bichon", 17);
15.     // Ajouts dans la map
16.     dogs.put(lassie.getId(), lassie);
17.     dogs.put(milou.getId(), milou);
18.     dogs.put(scoobydoo.getId(), scoobydoo);
19.     dogs.put(idefix.getId(), idefix);
20. }
```

III - Créer un serveur

III-A - Installation

Comme souvent, le plus simple va être de passer par Maven, Gradle, etc. Dans cet article, nous allons utiliser Maven, mais sentez-vous libre d'utiliser votre système préféré.

Nous allons donc ajouter une dépendance à Vert.x dans le fichier pom.xml. Vert.x est modulaire. Nous pouvons donc indiquer précisément les dépendances qui nous intéressent, à commencer par vertx-core. Et puisque nous allons développer pour le web, nous ajouterons également vertx-web, le module dédié.

pom.xml

```
1. <dependency>
2.   <groupId>io.vertx</groupId>
3.   <artifactId>vertx-core</artifactId>
4.   <version>${vertx.version}</version>
5. </dependency>
6. <dependency>
7.   <groupId>io.vertx</groupId>
8.   <artifactId>vertx-web</artifactId>
9.   <version>${vertx.version}</version>
10. </dependency>
```

III-B - Le verticle en charge de l'API

Contrairement à ce que vous verrez dans d'autres tutoriels, je préfère séparer mes *verticles* de mon main. Je vous encourage à faire de même dans vos applications. On va donc créer MyApiVerticle. Pour cela, il suffit d'hériter de AbstractVerticle et de réécrire les méthodes start() et stop(). Demandez à votre IDE de le faire pour vous.

MyApiVerticle.java

```
1. import io.vertx.core.AbstractVerticle;
2. public class MyApiVerticle extends AbstractVerticle {
3.     // Quand le verticle se lance
4.     @Override
5.     public void start() throws Exception {
6.     }
7.     // Quand le verticle s'arrête
8.     @Override
9.     public void stop() throws Exception {
10.    }
```

On va en profiter pour ajouter des logs. Vert.x fournit ses propres classes, mais rien ne vous empêche d'utiliser une autre bibliothèque, comme Flogger dont je parlais dans un précédent article, sobrement intitulé « Tutoriel pour logger facilement, en Java à l'aide de Flogger, en 5 minutes ».

MyApiVerticle.java

```

1. import io.vertx.core.logging.Logger;
2. import io.vertx.core.logging.LoggerFactory;
3. public class MyApiVerticle extends AbstractVerticle {
4.     private static final Logger LOGGER = LoggerFactory.getLogger(MyApiVerticle.class);
5.     // Quand le verticle se lance
6.     @Override
7.     public void start() throws Exception {
8.         LOGGER.info("Dans le start...");
9.     }
10.    // Quand le verticle s'arrête
11.    @Override
12.    public void stop() throws Exception {
13.        LOGGER.info("Dans le stop...");
14.    }

```

Il ne reste plus qu'à lancer le *verticle* depuis le main, de la même façon que dans mes autres articles dédiés à Vert.x.

App.java

```

1. import io.vertx.core.Vertx;
2. public class App {
3.     public static void main(String[] args) {
4.         System.out.println("App...");
5.         final Vertx vertx = Vertx.vertx();
6.         vertx.deployVerticle(new MyApiVerticle());
7.     }

```

Il faut ensuite exécuter le programme et regarder les logs dans la console.

Console

```

1. App...
2. sept. 1, 2019 10:10:49 AM com.ice.articlevertxapi.MyApiVerticle
3. INFO: Dans le start...

```

III-C - Ecouter un port

On y est presque. Avant d'entrer dans le vif du sujet, je vous propose d'écrire un simple *Hello World* accessible depuis le navigateur. Il faut simplement créer un serveur, depuis la variable *vertx* fournie par *AbstractVerticle*. Dans le cas du *Hello World*, la *lambda* fournie au *RequestHandler* est très simple. Enfin, je vous propose d'écouter sur le port 8080.

MyApiVerticle.java

```

1. @Override
2. public void start() throws Exception {
3.     LOGGER.info("Dans le start...");
4.     vertx.createHttpServer()
5.         .requestHandler(routingContext -> routingContext.response().end("Hello World!"))
6.         .listen(8080);
7. }

```

Il ne reste plus à rechercher l'adresse `http://localhost:8080/` dans votre navigateur.



IV - Notre API naïve

IV-A - Charger une liste d'objets

On va maintenant écrire notre API. Dans un premier temps, commençons par l'opération qui me semble la plus simple : charger la liste avec tous les chiens.

Pour cela, je vous propose de définir la route « /api/v1/dogs » qui est composée de « api » pour préciser qu'on est sur l'URL de notre API, de « v1 » pour dire qu'on est en version 1, et de « dogs » au pluriel pour indiquer qu'on veut la liste des chiens.

Le code associé est relativement simple.

MyApiVerticle.java

```
1. @Override
2. public void start() throws Exception {
3.     LOGGER.info("Dans le start...");
4.     // Création du routeur
5.     Router router = Router.router.vertx();
6.     // Définition de la route
7.     router.get("/api/v1/dogs")
8.         .handler(this::getAllDogs);
9.     // Lancement du serveur
10.    vertx.createHttpServer()
11.        .requestHandler(router)
12.        .listen(8080);
```

Note : les tutoriels un peu plus vieux proposaient de faire appel à `router::accept` comme `RequestHandler`. Mais la méthode `accept` est désormais dépréciée. Dans les nouvelles versions, on peut passer directement le router, ce qui est plus simple.

A ce stade, le plus simple est de définir le code propre au traitement de la liste dans la méthode séparée `getAllDogs`, pour que ça reste lisible.

MyApiVerticle.java

```
1. private void getAllDogs(RoutingContext routingContext) {
2.     LOGGER.info("Dans getAllDogs...");
3. }
```

J'aimerais attirer votre attention sur le fait que la méthode `getAllDogs` renvoie `void` et non la liste de chiens. Cela peut vous déstabiliser en première approche, mais n'oubliez pas qu'on développe une API REST asynchrone. C'est Vert.x qui se chargera d'envoyer la réponse au client.

On peut relancer le programme se demander l'URL « `http://localhost:8080/api/v1/dogs` » depuis le navigateur. Celui-ci va tourner en boucle, puisqu'on n'a pas encore programmé l'envoi de la liste de chiens, mais on peut vérifier que la log s'affiche dans la console, signe que le routage a bien fonctionné.

Console

```
App...
sept. 1, 2019 10:10:19 AM com.ice.articlevertxapi.MyApiVerticle
INFOS: Dans le start...
sept. 1, 2019 10:10:33 AM com.ice.articlevertxapi.MyApiVerticle
INFOS: Dans getAllDogs...
```

Avant d'aller plus loin, je vous propose de compléter le code afin d'avoir naïvement accès au service de chiens présenté plus haut. Ici, je vais simplement utiliser l'instruction `new`, mais il est bien entendu possible de faire de l'injection. Pensez juste à choisir une bibliothèque rapide.

MyApiVerticle.java

```
1. public class MyApiVerticle extends AbstractVerticle {
2.     private final DogService dogService = new DogService();
3.     ...
4.     private void getAllDogs(RoutingContext routingContext) {
5.         LOGGER.info("Dans getAllDogs...");
6.         final List<Dog> dogs = dogService.findAll();
7.     }
```

On souhaite répondre au format JSON. On va donc créer l'objet correspondant et on y placera la liste de chiens. Et bien entendu, on peut ajouter autant d'attributs qu'on le souhaite.

MyApiVerticle.java

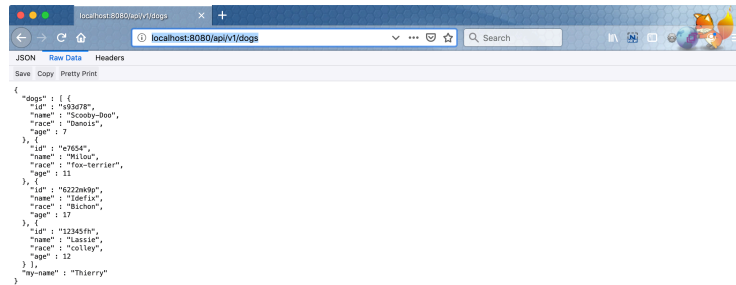
```
1. import io.vertx.core.json.JsonObject;
2. ...
3. private void getAllDogs(RoutingContext routingContext) {
4.     LOGGER.info("Dans getAllDogs...");
5.     // Recherche des chiens
6.     final List<Dog> dogs = dogService.findAll();
7.     // Création et remplissage de la réponse
8.     final JsonObject jsonResponse = new JsonObject();
9.     jsonResponse.put("dogs", dogs);
10.    jsonResponse.put("my-name", "Thierry");
11. }
```

Et finalement, on peut demander à Vert.x d'envoyer la réponse. Ici, je précise le statut 200 pour indiquer que tout s'est bien déroulé. En cas d'erreur, j'aurais indiqué un statut 500 par exemple. J'ajoute aussi un entête (*header*) pour préciser le `content-type` et indiquer qu'on renvoie du JSON. Et bien entendu, j'envoie la réponse.

MyApiVerticle.java

```
1. import io.vertx.core.json.Json;
2. ...
3. private void getAllDogs(RoutingContext routingContext) {
4.     LOGGER.info("Dans getAllDogs...");
5.     // Recherche des chiens
6.     final List<Dog> dogs = dogService.findAll();
7.     // Création et remplissage de la réponse
8.     final JsonObject jsonResponse = new JsonObject();
9.     jsonResponse.put("dogs", dogs);
10.    jsonResponse.put("my-name", "Thierry");
11.    // Envoi de la réponse
12.    routingContext.response()
13.        .setStatusCode(200)
14.        .putHeader("content-type", "application/json")
15.        .end(Json.encodePrettily(jsonResponse));
16. }
```

Vous pouvez rafraîchir votre navigateur sur l'adresse « `http://localhost:8080/api/v1/dogs` » et constater que vous recevez bien la liste des chiens au format JSON.



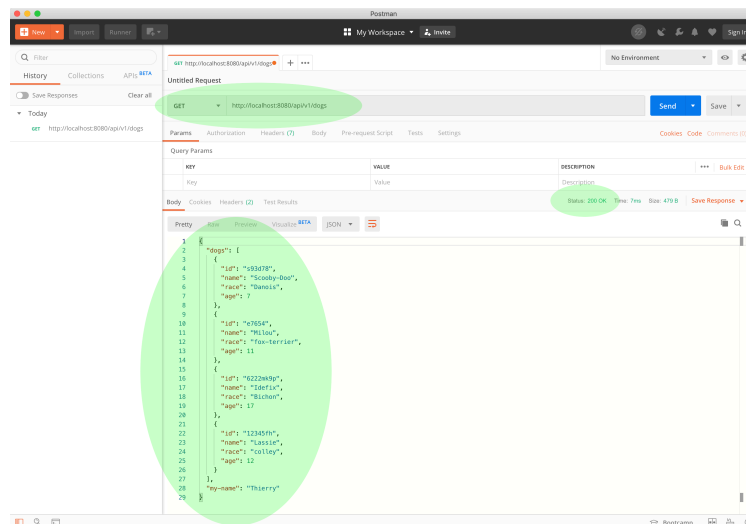
```

{
  "dogs": [
    {
      "id": "933678",
      "name": "Scobby-Doo",
      "race": "Danois",
      "age": 7
    },
    {
      "id": "e7654",
      "name": "Mélou",
      "race": "Fox-terrier",
      "age": 11
    },
    {
      "id": "6222mk9p",
      "name": "Léon",
      "race": "Bichon",
      "age": 17
    },
    {
      "id": "123456",
      "name": "Lassie",
      "race": "Colley",
      "age": 12
    }
  ],
  "my-name": "Thierry"
}
  
```

Note : Dans l'exemple, l'utilisation de `encodePrettily` est facultative. Je l'utilise uniquement pour que la réponse soit facile à lire. La plupart des clients proposent également cette facilité.

Pour la suite, je vous propose d'utiliser le logiciel **Postman** à la place du navigateur. Cela nous donnera plus de possibilités, en particulier pour les autres opérations *CRUD*.

Lancez donc une requête GET sur l'adresse « `http://localhost:8080/api/v1/dogs` » depuis Postman.



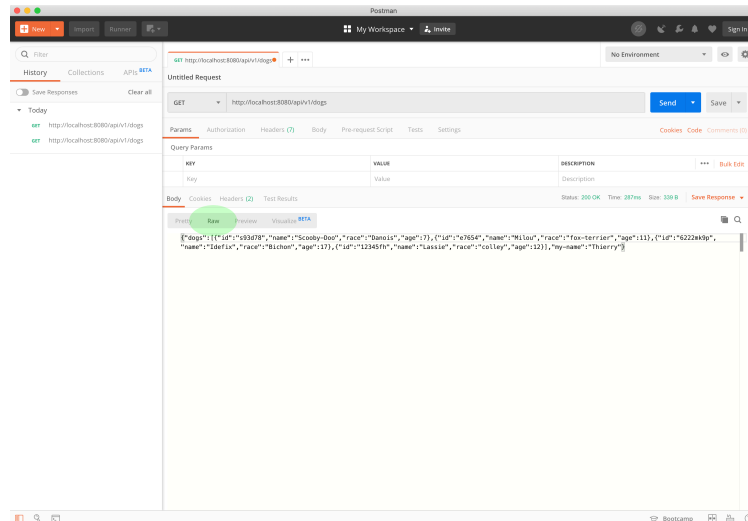
Vous constatez que la barre d'onglets vous permet de formater la réponse pour qu'elle soit lisible. Il n'est donc plus nécessaire de faire appel à `encodePrettily`.

MyApiVerticle.java

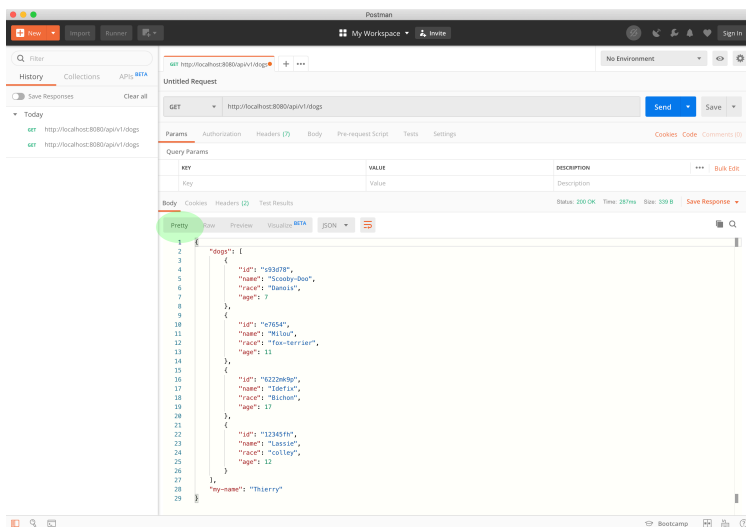
```

1. private void getAllDogs(RoutingContext routingContext) {
2.     LOGGER.info("Dans getAllDogs...");
3.     // Recherche des chiens
4.     final List<Dog> dogs = dogService.findAll();
5.     // Création et remplissage de la réponse
6.     final JsonObject jsonResponse = new JsonObject();
7.     jsonResponse.put("dogs", dogs);
8.     jsonResponse.put("my-name", "Thierry");
9.     // Envoi de la réponse
10.    routingContext.response()
11.        .setStatusCode(200)
12.        .putHeader("content-type", "application/json")
13.        .end(Json.encode(jsonResponse));
14. }
  
```

L'affichage en RAW n'est pas très lisible.



Contrairement à l'affichage « pretty ».



Mais je préfère que ce soit le client qui réalise la mise en forme, s'il en a vraiment besoin. Ce sera toujours quelques nanosecondes de gagnées sur le serveur et durant le transfert.

IV-B - Charger un seul objet

L'opération *CRUD* qu'on va naturellement développer ensuite est celle qui consiste à charger un chien, en particulier grâce à son id. Disons qu'on veuille charger les informations de Milou. Dans ce cas, il faudra appeler l'URL « http://localhost:8080/api/v1/dogs/e7654 » dans laquelle dogs est toujours écrit au pluriel, et où « e7654 » est l'id de Milou.

La syntaxe pour préciser un paramètre dans le routage de l'URL est :param

MyApiVerticle.java

```

1. @Override
2. public void start() throws Exception {
3.   LOGGER.info("Dans le start...");
4.   // Création du routeur
5.   Router router = Router.router.vertx;
6.   // Définition des routes
7.   router.get("/api/v1/dogs").handler(this::getAllDogs);
8.   router.get("/api/v1/dogs/:id").handler(this::getOneDog);

```

Il faudra bien entendu récupérer cet id depuis la requête dans la méthode.

MyApiVerticle.java

```
1. private void getOneDog(RoutingContext routingContext) {
2.     LOGGER.info("Dans getOneDog...");
3.     final String id = routingContext.request().getParam("id");
```

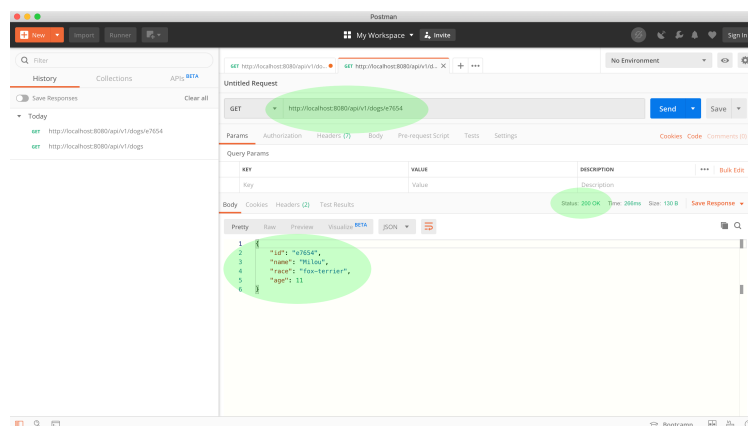
La suite est relativement simple. Il faut récupérer le chien à l'aide du service et l'envoyer au client.

MyApiVerticle.java

```
1. private void getOneDog(RoutingContext routingContext) {
2.     LOGGER.info("Dans getOneDog...");
3.     // Param id de la requête
4.     final String id = routingContext.request().getParam("id");
5.     // Recherche du chien correspondant à l'id
6.     final Dog dog = dogService.findById(id);
7.     // Envoi de la réponse
8.     routingContext.response()
9.         .setStatusCode(200)
10.        .putHeader("content-type", "application/json")
11.        .end(Json.encode(dog));
12. }
```

Note : Ici je n'utilise pas d'objet JSON intermédiaire, car je souhaite renvoyer uniquement le chien, sans information complémentaire.

Il ne reste plus qu'à tester l'adresse `http://localhost:8080/api/v1/dogs/e7654` dans Postman.



IV-C - Envoyer une erreur

La gestion d'erreur devrait monopoliser une grande partie de votre temps. Dans ce tutoriel, on va se concentrer sur le cas simple, correspondant aux identifiants non trouvés, mais on pourrait parler de nombreux types d'erreurs : sécurité, restrictions, problème de connexion aux ressources (comme la base de données), bugs, etc.

Dans l'exemple, on pourrait renvoyer une erreur si le chien n'est pas trouvé.

MyApiVerticle.java

```
1. private void getOneDog(RoutingContext routingContext) {
2.     LOGGER.info("Dans getOneDog...");
3.     // Param id de la requête
4.     final String id = routingContext.request().getParam("id");
5.     // Recherche du chien correspondant à l'id
6.     final Dog dog = dogService.findById(id);
7.     // Gestion d'erreur si le chien n'est pas trouvé...
8.     if (dog == null) {
9.         // TODO chien non trouvé...
```

MyApiVerticle.java

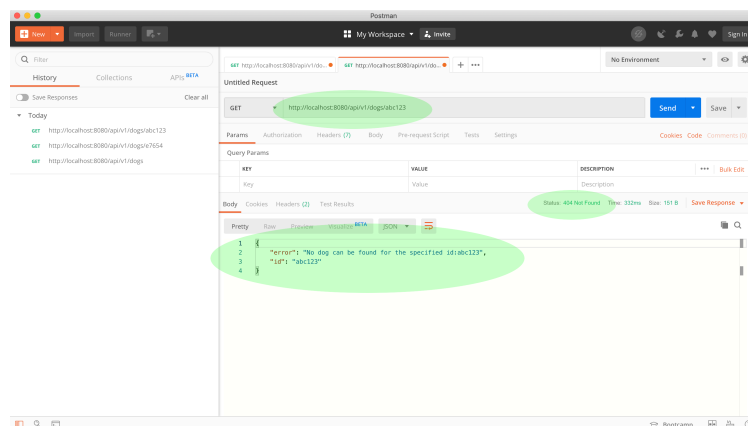
```
10. }
11. // Envoi de la réponse
12. routingContext.response()
13.     .setStatusCode(200)
14.     .putHeader("content-type", "application/json")
15.     .end(Json.encode(dog));
16. }
```

Il suffit de créer un message JSON contenant les informations que vous jugerez utiles et de préciser un statut 404 indiquant que la ressource n'a pas été trouvée.

MyApiVerticle.java

```
1. if (dog == null) {
2.     final JsonObject errorJsonResponse = new JsonObject();
3.     errorJsonResponse.put("error", "No dog can be found for the specified id:" + id);
4.     errorJsonResponse.put("id", id);
5.     // Envoi de la réponse avec erreur 404
6.     routingContext.response()
7.         .setStatusCode(404)
8.         .putHeader("content-type", "application/json")
9.         .end(Json.encode(errorJsonResponse));
10.    return;
11. }
```

Pour pouvez tester que vous recevez bien un message d'erreur quand vous appelez l'adresse `http://localhost:8080/api/v1/dogs/abc123` depuis Postman, sachant qu'aucun chien n'a l'id « abc123 ». Vous pouvez vérifier (sur la droite dans l'interface de Postman) que le statut est bien 404 Not Found.



IV-D - Créer un objet

Habituellement, on utilise le verbe POST pour créer un nouvel objet, ce qui est très simple à configurer dans le router.

MyApiVerticle.java

```
1. @Override
2. public void start() throws Exception {
3.     ...
4.     // Définition des routes
5.     router.get("/api/v1/dogs").handler(this::getAllDogs);
6.     router.get("/api/v1/dogs/:id").handler(this::getOneDog);
7.     router.post("/api/v1/dogs").handler(this::createOneDog);
8. }
```

Il faut ensuite récupérer les informations transmises dans la requête. Pour charger un objet par son id en *GET*, on avait récupéré la valeur de l'id dans l'URL. En *POST*, on trouvera les informations dans le corps (*body*). Je peux ensuite créer un nouveau chien à partir de ces informations et demander au service de l'ajouter à la liste.

MyApiVerticle.java

```
1. private void createOneDog(RoutingContext routingContext) {
2.     LOGGER.info("Dans createOneDog...");
3.     final JsonObject body = routingContext.getBodyAsJson();
4.     final String name = body.getString("name");
5.     final String race = body.getString("race");
6.     final Integer age = body.getInteger("age");
```

Pour pouvoir récupérer le corps (*body*), on doit spécifier un *BodyHandler* sur les routes `"/api/v1/dogs*"`, où l'étoile sert de joker, car il n'y en a pas par défaut.

MyApiVerticle.java

```
1. import io.vertx.ext.web.handler.BodyHandler;
2. ...
3. public class MyApiVerticle extends AbstractVerticle {
4.     ...
5.     @Override
6.     public void start() throws Exception {
7.         LOGGER.info("Dans le start...");
8.         // Création du routeur
9.         Router router = Router.router(vertx);
10.        // Body handler
11.        router.route("/api/v1/dogs*").handler(BodyHandler.create());
12.        // Définition des routes
13.        router.get("/api/v1/dogs").handler(this::getAllDogs);
14.        router.get("/api/v1/dogs/:id").handler(this::getOneDog);
15.        router.post("/api/v1/dogs").handler(this::createOneDog);
```

Il est temps d'ajouter un nouveau chien dans la liste à partir des informations reçues. Je crée donc d'abord (*immutable*) un chien sans id. C'est le service qui aura la charge de le créer. En retour, le service me renvoie le chien créé, avec maintenant un id.

MyApiVerticle.java

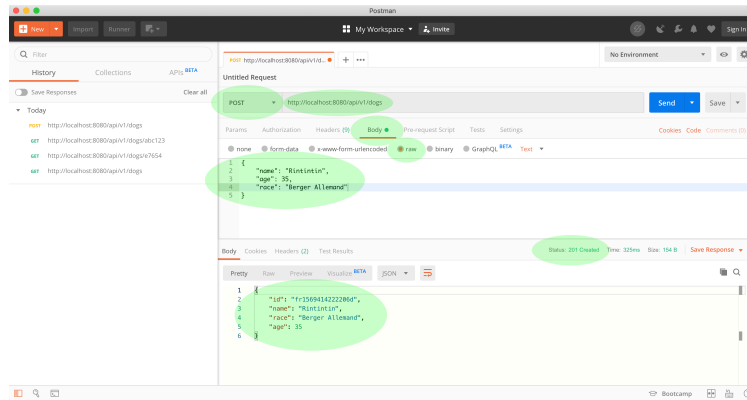
```
1. private void createOneDog(RoutingContext routingContext) {
2.     ...
3.     final Dog dog = new Dog(null, name, race, age);
4.     final Dog createdDog = dogService.add(dog);
```

Il ne reste plus qu'à notifier Vert.x que la réponse peut être envoyée. Elle contient le nouveau chien. Et bien entendu, puisqu'il s'agit d'une création, on indique le statut `201`.

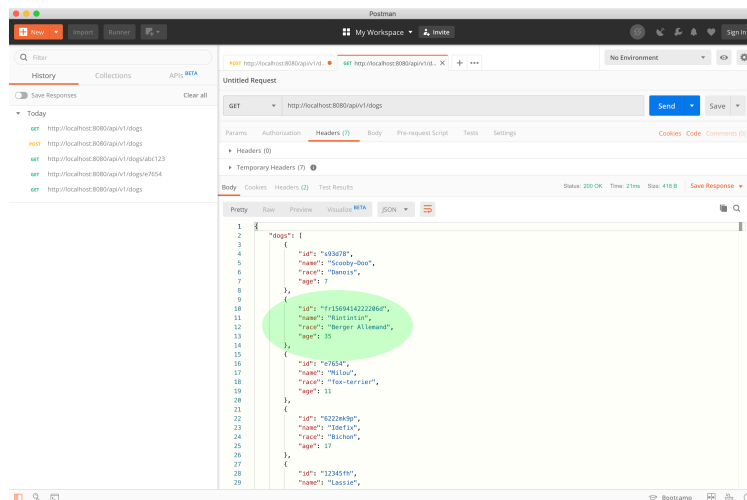
MyApiVerticle.java

```
1. private void createOneDog(RoutingContext routingContext) {
2.     ...
3.     // TODO Vérification des champs...
4.     ...
5.     // Création d'un chien
6.     final Dog dog = new Dog(null, name, race, age);
7.     final Dog createdDog = dogService.add(dog);
8.     // Envoi de la réponse
9.     routingContext.response()
10.        .setStatusCode(201)
11.        .putHeader("content-type", "application/json")
12.        .end(Json.encode(createdDog));
13. }
```

Dans Postman, on peut maintenant appeler l'URL `http://localhost:8080/api/v1/dogs` en *POST* en précisant un body en JSON. On peut vérifier que le statut de retour est bien `201 created` et qu'un id a été choisi pour notre nouveau chien.



Quand on redemande la liste des chiens, on voit que celui que nous venons d'ajouter est bien présent.



IV-E - Modifier un objet

La mise à jour d'un chien va maintenant être relativement simple, car on a déjà tous les éléments. On va donc aller plus vite. Habituellement, on utilise le verbe *PUT* et on précise l'id dans l'URL.

MyApiVerticle.java

```

1. @Override
2. public void start() throws Exception {
3.   ...
4.   // Définition des routes
5.   router.get("/api/v1/dogs").handler(this::getAllDogs);
6.   router.get("/api/v1/dogs/:id").handler(this::getOneDog);
7.   router.post("/api/v1/dogs").handler(this::createOneDog);
8.   router.put("/api/v1/dogs/:id").handler(this::updateOneDog);

```

On prend donc l'id dans l'URL et les champs dans le body.

MyApiVerticle.java

```

1. private void updateOneDog(RoutingContext routingContext) {
2.   LOGGER.info("Dans updateOneDog...");
3.   // Param id de la requête
4.   final String id = routingContext.request().getParam("id");
5.   // Params pris dans le body
6.   final JsonObject body = routingContext.getBodyAsJson();
7.   final String name = body.getString("name");
8.   final String race = body.getString("race");
9.   final Integer age = body.getInteger("age");

```

Il ne reste plus qu'à faire appel au service et renvoyer l'objet mis à jour.

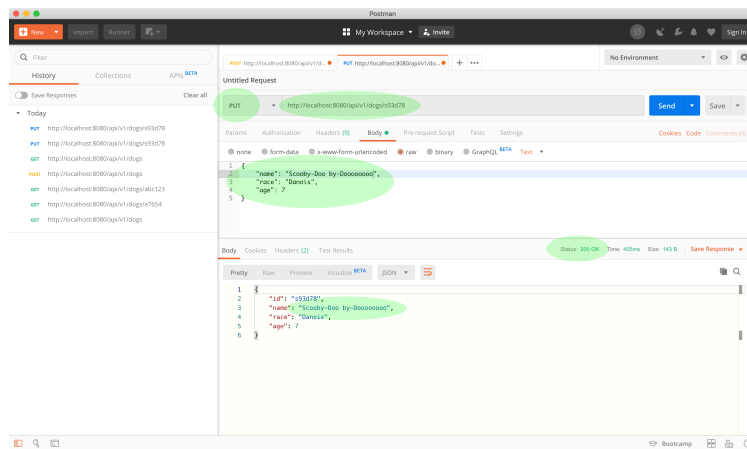
MyApiVerticle.java

```

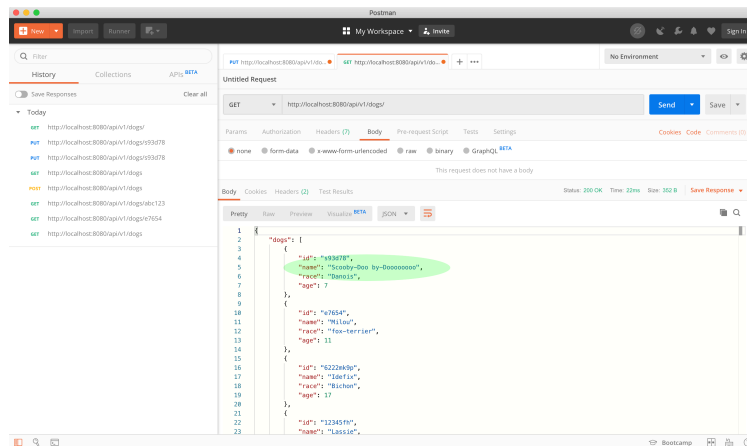
1. private void updateOneDog(RoutingContext routingContext) {
2.   ...
3.   // Création/maj d'un chien
4.   final Dog dog = new Dog(id, name, race, age);
5.   final Dog updatedDog = dogService.update(dog);
6.   // Envoi de la réponse
7.   routingContext.response()
8.     .setStatusCode(200)
9.     .putHeader("content-type", "application/json")
10.    .end(Json.encode(updatedDog));

```

Dans Postman, il faudra simplement choisir le verbe *PUT* sur l'URL `http://localhost:8080/api/v1/dogs/s93d78`, où « s93d78 » est l'id de Scooby-Doo.



On peut vérifier que Scooby-Doo a bien été changé.



IV-F - Supprimer un objet

On arrive à la dernière opération *CRUD*, consistant à supprimer un chien. À ce stade, cela devrait être une formalité. On commence en indiquant une route sur le verbe *DELETE*.

MyApiVerticle.java

```

1. @Override
2. public void start() throws Exception {
3.   ...
4.   // Définition des routes

```

MyApiVerticle.java

```

5.  router.get("/api/v1/dogs").handler(this::getAllDogs);
6.  router.get("/api/v1/dogs/:id").handler(this::getOneDog);
7.  router.post("/api/v1/dogs").handler(this::createOneDog);
8.  router.put("/api/v1/dogs/:id").handler(this::updateOneDog);
9.  router.delete("/api/v1/dogs/:id").handler(this::deleteOneDog);

```

Le contenu de la méthode est ensuite très simple.

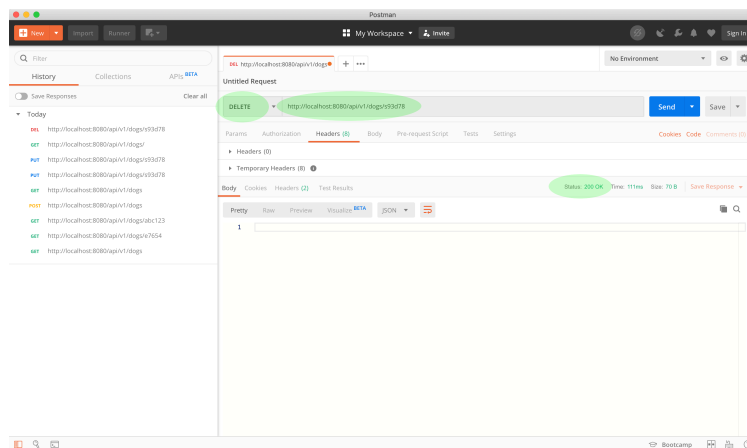
MyApiVerticle.java

```

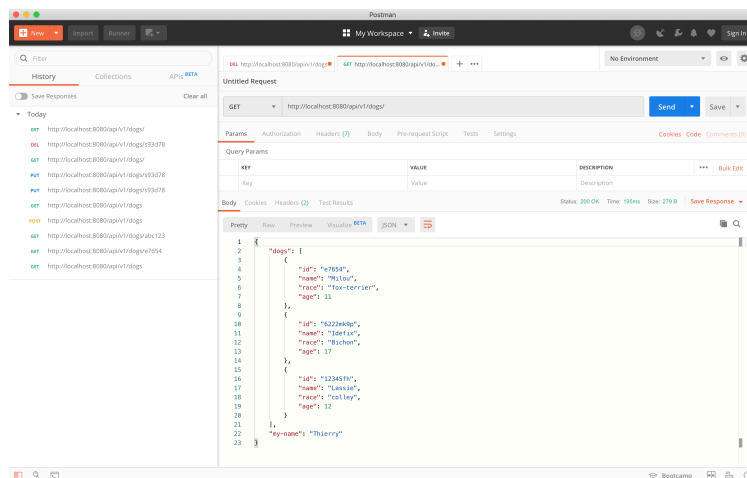
1. private void deleteOneDog(RoutingContext routingContext) {
2.   LOGGER.info("Dans deleteOneDog...");
3.   // Param id de la requête
4.   final String id = routingContext.request().getParam("id");
5.   // Suppression du chien
6.   dogService.remove(id);
7.   // Envoi de la réponse
8.   routingContext.response()
9.     .setStatusCode(200)
10.    .putHeader("content-type", "application/json")
11.    .end();
12. }

```

Dans Postman, on va supprimer Scooby-Doo en appelant `http://localhost:8080/api/v1/dogs/s93d78` sur le verbe **DELETE**, où « s93d78 » est l'id de Scooby-Doo.



Et bien entendu, on vérifie que Scooby-Doo a bien été retiré de la liste.



V - Réorganiser le routeur

Maintenant qu'on a fait l'essentiel de l'API pour nos chiens, on constate que la partie routage commence à être en pagaille. Il faut imaginer que vous aurez probablement plusieurs domaines complexes dans votre programme. Je vous propose donc d'organiser le code par ressource.

On commence par créer la classe `DogResource` et on y déplace tout le code relatif à l'API chien.

DogResource.java

```
1. public class DogResource {
2.     ...
3.     private void getAllDogs(final RoutingContext routingContext) {
4.         ...
5.     }
6.     private void getOneDog(final RoutingContext routingContext) {
7.         ...
8.     }
9.     private void createOneDog(final RoutingContext routingContext) {
10.        ...
11.    }
12.    private void createOneDog(final RoutingContext routingContext) {
13.        ...
14.    }
15.    private void deleteOneDog(final RoutingContext routingContext) {
16.        ...
17.    }
```

Je voudrais également que toute la logique de routage spécifique aux chiens soit déportée vers cette classe. Et je voudrais que le routage se fasse plus particulièrement sur la fin de `/api/v1/dogs/`.

DogResource.java

```
1. public Router getSubRouter(final Vertx vertx) {
2.     // Création du sous-routeur
3.     final Router subRouter = Router.router(vertx);
4.     // Body handler
5.     subRouter.route("/*").handler(BodyHandler.create());
6.     // Définition des routes
7.     subRouter.get("/").handler(this::getAllDogs);
8.     subRouter.get("/:id").handler(this::getOneDog);
9.     subRouter.post("/").handler(this::createOneDog);
10.    subRouter.put("/:id").handler(this::updateOneDog);
11.    subRouter.delete("/:id").handler(this::deleteOneDog);
12.    return subRouter;
13. }
```

Au niveau du `verticle`, qui a été nettoyé, je peux définir un `SubRouter` et mapper toutes les URL en `/api/v1/dogs` dessus.

MyApiVerticle.java

```
1. @Override
2. public void start() throws Exception {
3.     LOGGER.info("Dans le start...");
4.     // Création du routeur
5.     final Router router = Router.router(vertx);
6.     // Création de la ressource
7.     final DogResource dogResource = new DogResource();
8.     // Définition des routes et sous-routes
9.     final Router dogSubRouter = dogResource.getSubRouter(vertx);
10.    router.mountSubRouter("/api/v1/dogs", dogSubRouter);
11.    // Lancement du serveur
12.    vertx.createHttpServer()
13.        .requestHandler(router)
14.        .listen(8080);
15. }
```


Bien entendu, je peux décomposer `/api/v1/dogs` en trois sous routers « imbriqués ». L'étape d'après serait de créer la ressource `CatResource` et les sous-routeurs associés.

VI - Conclusions

Aurons-nous réussi à écrire une API naïve avec Vert.x en 5 minutes chrono ? On ne doit pas en être loin. On a pu voir que c'est relativement simple.

On a tout de même fait l'impasse sur un ensemble de préoccupations, notamment en ce qui concerne la sécurité.

Pour aller plus loin, je vous recommande le **module `vertx-web-api-service`** qui va grandement simplifier la création d'une API standardisée. Ce module vous permet notamment de définir les routes et les contrats en Yaml.



Vos retours et remarques nous aident à améliorer les publications de [Developpez.com](#). N'hésitez donc pas à commenter cet article. [Commentez](#)

VII - Remerciements

D'abord, j'adresse mes remerciements à l'équipe Vert.x et aux créateurs de modules complémentaires, pour avoir développé une bibliothèque aussi utile et pour la maintenir. Je n'oublie pas tous les contributeurs qui participent, notamment sur le forum.

Plus spécifiquement en ce qui concerne cet article, je tiens à remercier l'équipe de [Developpez.com](#) et plus particulièrement à Mickael Baron et Bruno Barthel.

VIII - Annexes

VIII-A - Liens

Vert.x : <https://vertx.io/>

Documentation du module `vertx-web` : <https://vertx.io/docs/vertx-web/java/>

Documentation du module `vertx-web-api-service` : <https://vertx.io/docs/vertx-web-api-service/java/>

Postman : <https://www.getpostman.com/>

Série dédiée à Vert.x : <https://thierry-leriche-dessirier.developpez.com/tutoriels/java/vertx/creer-lancer-tester-verticle/>

Autres articles de la série « en 5 minutes » : <https://thierry-leriche-dessirier.developpez.com/#5min>

VIII-B - Codes complets

`pom.xml`

```
1. <project xmlns="http://maven.apache.org/POM/4.0.0"
2.   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3.   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
4.   <modelVersion>4.0.0</modelVersion>
5.   <groupId>com.ice</groupId>
6.   <artifactId>article-vertx-api</artifactId>
```

pom.xml

```
7. <version>0.0.1-SNAPSHOT</version>
8. <packaging>jar</packaging>
9. <name>article-vertx-api</name>
10. <url>http://maven.apache.org</url>
11. <properties>
12.   <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
13.   <maven.compiler.source>1.8</maven.compiler.source>
14.   <maven.compiler.target>1.8</maven.compiler.target>
15.   <vertx.version>3.8.1</vertx.version>
16. </properties>
17. <dependencies>
18.   <dependency>
19.     <groupId>io.vertx</groupId>
20.     <artifactId>vertx-core</artifactId>
21.     <version>${vertx.version}</version>
22.   </dependency>
23.   <dependency>
24.     <groupId>io.vertx</groupId>
25.     <artifactId>vertx-web</artifactId>
26.     <version>${vertx.version}</version>
27.   </dependency>
28. </dependencies>
29. </project>
```

Dog.java

```
1. package com.ice.articlevertxapi.bean;
2. public class Dog {
3.   private final String id;
4.   private final String name;
5.   private final String race;
6.   private final int age;
7.   public Dog(final String id, final String name, final String race, final int age) {
8.     super();
9.     this.id = id;
10.    this.name = name;
11.    this.race = race;
12.    this.age = age;
13.  }
14.  public String getId() {
15.    return id;
16.  }
17.  public String getName() {
18.    return name;
19.  }
20.  public String getRace() {
21.    return race;
22.  }
23.  public int getAge() {
24.    return age;
25.  }
26.  @Override
27.  public String toString() {
28.    return "Dog [id=" + id + ", name=" + name + ", race=" + race + ", age=" + age + "]";
29.  }
30. }
```

DogService.java

```
1. package com.ice.articlevertxapi.service;
2. import java.util.HashMap;
3. import java.util.List;
4. import java.util.Map;
5. import java.util.stream.Collectors;
6. import com.ice.articlevertxapi.bean.Dog;
7. public class DogService {
8.   private final Map<String, Dog> dogs = new HashMap<String, Dog>();
9.   public DogService() {
10.     super();
11.     initDogs();
12.   }
```

DogService.java

```
13. private void initDogs() {
14.     final Dog lassie = new Dog("12345fh", "Lassie", "colley", 12);
15.     final Dog milou = new Dog("e7654", "Milou", "fox-terrier", 11);
16.     final Dog scoobydoo = new Dog("s93d78", "Scooby-Doo", "Danois", 7);
17.     final Dog idefix = new Dog("6222mk9p", "Idefix", "Bichon", 17);
18.     dogs.put(lassie.getId(), lassie);
19.     dogs.put(milou.getId(), milou);
20.     dogs.put(scoobydoo.getId(), scoobydoo);
21.     dogs.put(idefix.getId(), idefix);
22. }
23. public List<Dog> findAll() {
24.     return dogs.values().stream()
25.         .collect(Collectors.toList());
26. }
27. public Dog findById(final String id) {
28.     return dogs.get(id);
29. }
30. public Dog update(final Dog dog) {
31.     dogs.put(dog.getId(), dog);
32.     return dog;
33. }
34. public void remove(final String id) {
35.     dogs.remove(id);
36. }
37. public Dog add(final Dog dog) {
38.     final String id = "fr" + System.currentTimeMillis() + "d";
39.     final Dog newdog = new Dog(id,
40.         dog.getName(),
41.         dog.getRace(),
42.         dog.getAge());
43.     dogs.put(id, newdog);
44.     return newdog;
45. }
46. }
```

MyApiVerticle.java

```
1. package com.ice.articlevertxapi;
2. import com.ice.articlevertxapi.resource.DogResource;
3. import io.vertx.core.AbstractVerticle;
4. import io.vertx.core.logging.Logger;
5. import io.vertx.core.logging.LoggerFactory;
6. import io.vertx.ext.web.Router;
7. public class MyApiVerticle extends AbstractVerticle {
8.     private static final Logger LOGGER = LoggerFactory.getLogger(MyApiVerticle.class);
9.     @Override
10.    public void start() throws Exception {
11.        LOGGER.info("Dans le start...");
12.        final Router router = Router.router(vertx);
13.        final DogResource dogResource = new DogResource();
14.        final Router dogSubRouter = dogResource.getSubRouter(vertx);
15.        router.mountSubRouter("/api/v1/dogs", dogSubRouter);
16.        vertx.createHttpServer()
17.            .requestHandler(router)
18.            .listen(8080);
19.    }
20.    @Override
21.    public void stop() throws Exception {
22.        LOGGER.info("Dans le stop...");
23.    }
24. }
```

DogResource.java

```
1. package com.ice.articlevertxapi.resource;
2.
3. import java.util.List;
4.
5. import com.ice.articlevertxapi.bean.Dog;
6. import com.ice.articlevertxapi.service.DogService;
7. 
```

DogResource.java

```
8. import io.vertx.core.Vertx;
9. import io.vertx.core.json.Json;
10. import io.vertx.core.json.JsonObject;
11. import io.vertx.core.logging.Logger;
12. import io.vertx.core.logging.LoggerFactory;
13. import io.vertx.ext.web.Router;
14. import io.vertx.ext.web.RoutingContext;
15. import io.vertx.ext.web.handler.BodyHandler;
16.
17. public class DogResource {
18.
19.     private static final Logger LOGGER = LoggerFactory.getLogger(DogResource.class);
20.
21.     private final DogService dogService = new DogService();
22.
23.     public Router getSubRouter(final Vertx vertx) {
24.         final Router subRouter = Router.router(vertx);
25.
26.         // Body handler
27.         subRouter.route("/*").handler(BodyHandler.create());
28.
29.         // Routes
30.         subRouter.get("/").handler(this::getAllDogs);
31.         subRouter.get("/:id").handler(this::getOneDog);
32.         subRouter.post("/").handler(this::createOneDog);
33.         subRouter.put("/:id").handler(this::updateOneDog);
34.         subRouter.delete("/:id").handler(this::deleteOneDog);
35.
36.         return subRouter;
37.     }
38.
39.     private void getAllDogs(final RoutingContext routingContext) {
40.         LOGGER.info("Dans getAllDogs...");
41.
42.         final List<Dog> dogs = dogService.findAll();
43.
44.         final JsonObject jsonResponse = new JsonObject();
45.         jsonResponse.put("dogs", dogs);
46.         jsonResponse.put("my-name", "Thierry");
47.
48.         routingContext.response()
49.             .setStatusCode(200)
50.             .putHeader("content-type", "application/json")
51.             .end(Json.encode(jsonResponse));
52.     }
53.
54.     private void getOneDog(final RoutingContext routingContext) {
55.         LOGGER.info("Dans getOneDog...");
56.
57.         final String id = routingContext.request().getParam("id");
58.
59.         final Dog dog = dogService.findById(id);
60.
61.         if (dog == null) {
62.             final JsonObject errorJsonResponse = new JsonObject();
63.             errorJsonResponse.put("error", "No dog can be found for the specified id:" + id);
64.             errorJsonResponse.put("id", id);
65.
66.             routingContext.response()
67.                 .setStatusCode(404)
68.                 .putHeader("content-type", "application/json")
69.                 .end(Json.encode(errorJsonResponse));
70.             return;
71.         }
72.         routingContext.response()
73.             .setStatusCode(200)
74.             .putHeader("content-type", "application/json")
75.             .end(Json.encode(dog));
76.     }
77.     private void createOneDog(final RoutingContext routingContext) {
78.         LOGGER.info("Dans createOneDog...");
```

DogResource.java

```
79.     final JsonObject body = routingContext.getBodyAsJson();
80.     final String name = body.getString("name");
81.     final String race = body.getString("race");
82.     final Integer age = body.getInteger("age");
83.     // TODO Vérification des champs...
84.     final Dog dog = new Dog(null, name, race, age);
85.     final Dog createdDog = dogService.add(dog);
86.     routingContext.response()
87.         .setStatusCode(201)
88.         .putHeader("content-type", "application/json")
89.         .end(Json.encode(createdDog));
90. }
91. private void updateOneDog(final RoutingContext routingContext) {
92.     LOGGER.info("Dans updateOneDog...");
93.     final String id = routingContext.request().getParam("id");
94.     final JsonObject body = routingContext.getBodyAsJson();
95.     final String name = body.getString("name");
96.     final String race = body.getString("race");
97.     final Integer age = body.getInteger("age");
98.     // TODO Vérification des champs...
99.     final Dog dog = new Dog(id, name, race, age);
100.    final Dog updatedDog = dogService.update(dog);
101.    routingContext.response()
102.        .setStatusCode(200)
103.        .putHeader("content-type", "application/json")
104.        .end(Json.encode(updatedDog));
105. }
106. private void deleteOneDog(final RoutingContext routingContext) {
107.     LOGGER.info("Dans deleteOneDog...");
108.     final String id = routingContext.request().getParam("id");
109.     dogService.remove(id);
110.     routingContext.response()
111.         .setStatusCode(200)
112.         .putHeader("content-type", "application/json")
113.         .end();
114. }
115. }
```

App.java

```
1. package com.ice.articlevertxapi;
2. import io.vertx.core.Vertx;
3. public class App {
4.     public static void main(String[] args) {
5.         System.out.println("App...");
6.         final Vertx vertx = Vertx.vertx();
7.         vertx.deployVerticle(new MyApiVerticle());
8.     }
9. }
```