

# Consommer des web service REST (réaliser des clients REST)

Le projet contrôleur REST 2<sup>ème</sup> version du cours précédent (p12) est utilisé comme fournisseur de l'api rest utilisé dans ce cours.

## Consommer un service REST avec Spring Boot

► Démarrer l'application contrôleur REST 2<sup>ème</sup> version du cours précédent (p12).

La plupart des langages de programmation propose des API pour consommer des web services. On présente ici une façon de faire en Java qui utilise Spring Boot pour consommer les web service du projet 2<sup>ème</sup> version démarré ci-dessus.

► Créez un nouveau **Spring Starter Project**, donnez lui un nom, par exemple **client-1**, cochez uniquement **Spring Web** sur le formulaire des dépendances.

Il faut cocher Spring Web car ces applications client REST font appel à des classes définies dans le framework Spring Web.

► On crée ici une application en ligne de commande, l'exécution d'un serveur Tomcat est inutile.

Modifier la méthode **main()** comme il est indiqué ci-dessous pour ne pas démarrer le serveur Tomcat.

```
package greta78.cda;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.WebApplicationType;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.web.client.RestTemplateBuilder;
import org.springframework.context.annotation.Bean;
import org.springframework.web.client.RestTemplate;

@SpringBootApplication
public class Client1Application {

    public static void main(String[] args) {
        SpringApplication springApplication =
            new SpringApplication(Client1Application.class);
        springApplication.setWebApplicationType(WebApplicationType.NONE);
        springApplication.run(args);
    }
}
```

## La classe RestTemplate

La classe **RestTemplate** est la classe de base nécessaire pour créer des applications clientes de web services. Elle propose un grand nombre de méthodes pour exécuter les opérations http

habituelles (GET, POST, PUT, DELETE) afin de consommer des web services. Voir la documentation Spring.

## Une application client de base

L'application client est créée dans un **CommandLineRunner**.

```
package greta78.cda;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.boot.CommandLineRunner;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Component;
import org.springframework.web.client.RestClientResponseException;
import org.springframework.web.client.RestTemplate;

@Component
public class ClientCommandLineRunner implements CommandLineRunner {

    private static final Logger log =
        LoggerFactory.getLogger(ClientApplication.class);

    @Override
    public void run(String... args) throws Exception {
        // TODO Auto-generated method stub
        RestTemplate restTemplate = new RestTemplate();
        try {
            Invite uninvite = restTemplate
                .getForObject("http://localhost:8080/dupond",
                    Invite.class);
            log.info(uninvite.toString());

            String invites = restTemplate
                .getForObject("http://localhost:8080/all",
                    String.class);
            log.info(invites);

            ResponseEntity<Invite[]> responseEntity = restTemplate
                .getForEntity("http://localhost:8080/all",
                    Invite[].class);
            Invite[] lesinvites = responseEntity.getBody();
            for (Invite i : lesinvites)
                System.out.print(i.getNom()+" ");
        }
        catch (RestClientResponseException e) {
            System.out.println(e.getRawStatusCode());
        }
    }
}
```

L'instruction suivante

```
Invite uninvite=
restTemplate.getForObject("http://localhost:8080/dupond",Invite.class);
```

retourne la ressource Invite accessible à l'URL <http://localhost:8080/dupond>, c'est-à-dire ici un objet Invite.

L'instruction suivante

```
String invites =  
restTemplate.getForObject("http://localhost:8080/all",String.class);
```

retourne la liste de tous les invités dans une chaîne de caractères.

L'instruction suivante

```
ResponseEntity<Invite[]> responseEntity =  
    restTemplate.getForEntity("http://localhost:8080/all",Invite[].class);
```

retourne tous les objets Invite dans un tableau d'objets Invite.

### ► Déclenchement de l'exception `RestClientResponseException`

Tester avec l'URL suivant:

```
Invite  
uninvite=restTemplate.getForObject("http://localhost:8080/dupond1",Invite.class);
```

L'exception est levée, elle est attrapée dans le bloc `catch()`, le code de retour **404** est affiché.

## Une application client plus complète

Cette application a pour but de montrer l'utilisation de quelques méthodes de la classe `RestTemplate` pour exécuter des requêtes http GET, POST, PUT et DELETE.

On se propose d'organiser le code et donc de définir l'accès aux ressources dans des méthodes séparées pour bien isoler chaque exemple.

## [La méthode pour exécuter un web service GET et la nouvelle classe `ClientCommandLineRunner`](#)

```
package greta78.cda;  
  
import java.net.URI;  
import org.slf4j.Logger;  
import org.slf4j.LoggerFactory;  
import org.springframework.boot.CommandLineRunner;  
import org.springframework.http.HttpEntity;  
import org.springframework.http.HttpHeaders;  
import org.springframework.http.MediaType;  
import org.springframework.http.ResponseEntity;  
import org.springframework.stereotype.Component;  
import org.springframework.util.LinkedMultiValueMap;  
import org.springframework.util.MultiValueMap;  
import org.springframework.web.client.RestClientException;  
import org.springframework.web.client.RestClientResponseException;  
import org.springframework.web.client.RestTemplate;  
  
@Component  
public class ClientCommandLineRunner implements CommandLineRunner {  
  
    RestTemplate restTemplate = new RestTemplate();  
  
    private static final Logger log =  
LoggerFactory.getLogger(ClientApplication.class);  
    ///////////////////////////////////////  
    public void doGet() {  
        try {
```

```

        Invite uninvite =
restTemplate.getForObject("http://localhost:8080/dupond", Invite.class);
        log.info(uninvite.toString());

        String invites =
restTemplate.getForObject("http://localhost:8080/all", String.class);
        log.info(invites);

        ResponseEntity<Invite[]> responseEntity =
restTemplate.getForEntity("http://localhost:8080/all",
                Invite[].class);
        Invite[] lesinvites = responseEntity.getBody();
        for (Invite i : lesinvites)
            System.out.print(i.getNom() + " ");
    } catch (RestClientResponseException e) {
        System.out.println(e.getRawStatusCode());
    }
}
////////////////////////////////////
@Override
public void run(String... args) throws Exception {
    // TODO Auto-generated method stub
    doGet();
}
}

```

L'exécution du GET avec un paramètre peut être avantageusement améliorée avec la méthode `getForObject(...)` suivante:

```

String nom="dupond";
Invite uninvite = restTemplate
    .getForObject("http://localhost:8080/{nom}", Invite.class, nom);

```

Le paramètre de l'URL est ici passé dans une variable comme 3<sup>ème</sup> argument.

S'il y a plusieurs paramètres, on peut utiliser la méthode:

```
getForObject(String url, Class<T> responseType, Map<String,?> uriVariables)
```

## 1<sup>ère</sup> méthode pour exécuter un web service POST

Un grand nombre de méthodes POST sont proposées. Il faut sélectionner celle qui correspond au web service utilisé.

Un 1<sup>er</sup> POST de notre web service reçoit les paramètres d'un objet invité (nom, prénom et email) au format **x-www-form-urlencoded**, retourne l'URL de l'objet persisté dans le champ Location de l'entête de la réponse avec le code 201 ou uniquement le code 204 si l'invité n'a pas été persisté.

Les import à ajouter

```

import org.springframework.http.HttpHeaders;
import org.springframework.http.MediaType;
import org.springframework.util.MultiValueMap;
import org.springframework.web.client.RestClientException;
import org.springframework.http.HttpEntity;

```

La méthode

```

public void doPost1() {
    Invite invite = new Invite("Duchemin", "Jean",
    "j.duchemin@orange.fr");
}

```

```

try {
    HttpHeaders headers = new HttpHeaders();
    headers.setContentType(MediaType.APPLICATION_FORM_URLENCODED);

    MultiValueMap<String, String> body=
        new LinkedMultiValueMap<String, String>();

    body.add("nom", invite.getNom());
    body.add("prenom", invite.getPrenom());
    body.add("email", invite.getEmail());

    HttpEntity<MultiValueMap<String, String>> entity=
        new HttpEntity<MultiValueMap<String,
String>>(body,headers);

    URI uri=
restTemplate.postForLocation("http://localhost:8080/ajoute",
entity);

    if (uri == null)
        Log.info("Erreur ajout");
    else
        Log.info(uri.toString());
} catch (RestClientResponseException e) {
    System.out.println(e.getRawStatusCode());
}
}

```

Il faut utiliser un objet `HttpHeader` pour préciser le type **x-www-form-urlencoded**.

```

HttpHeaders headers = new HttpHeaders();
headers.setContentType(MediaType.APPLICATION_FORM_URLENCODED);

```

Il faut utiliser un objet `LinkedMultiValueMap` pour créer des paires **nom:valeur** pour les 3 paramètres à passer. Cet objet sera le body de la requête.

```

MultiValueMap<String, String> body=
    new LinkedMultiValueMap<String, String>();

body.add("nom", invite.getNom());
body.add("prenom", invite.getPrenom());
body.add("email", invite.getEmail());

```

On instancie un objet de type `HttpEntity` avec comme arguments le body et le header créés précédemment.

```

HttpEntity<MultiValueMap<String, String>> entity =
    new HttpEntity<MultiValueMap<String,
String>>(body,headers);

```

On exécute la requête POST avec comme argument l'URL de la ressource et comme 2<sup>ème</sup> l'objet `HttpEntity` créé.

```

URI uri= restTemplate.postForLocation("http://localhost:8080/ajoute", entity);

```

L'URL retourné est la valeur du champ Location de l'entête de la réponse. Il est **null** si le nouvel invité n'a pas été persisté.

Résultat observable à l'exécution de la méthode `testPost1()` pour un nouvel invité:

2021-05-19 14:51:24 ----- http://localhost:8080/ajoute/47
---

## 2<sup>ème</sup> méthode pour exécuter un web service POST

```
public void doPost2() {
    Invite invite = new Invite("Granjean", "Alain",
"a.granjean@orange.fr");
    try {

        HttpHeaders headers = new HttpHeaders();
        headers.setContentType(MediaType.APPLICATION_FORM_URLENCODED);

        MultiValueMap<String, String> body=
            new LinkedMultiValueMap<String, String>();
        body.add("nom", invite.getNom());
        body.add("prenom", invite.getPrenom());
        body.add("email", invite.getEmail());

        HttpEntity<MultiValueMap<String, String>> entity=
            new HttpEntity<MultiValueMap<String,
String>>(body,
                headers);

        ResponseEntity<Invite> reponse=
            restTemplate
                .postForEntity("http://localhost:8080/ajoute",
entity, Invite.class);
        System.out.println("Status de la réponse: "
            + reponse.getStatusCodeValue());
        System.out.println("Valeur champ Location: "
            + reponse.getHeaders().getFirst("Location"));

    } catch (RestClientResponseException e) {
        System.out.println(e.getRawStatusCode());
    }
}
```

On utilise ici la méthode **postForEntity** qui retourne un **ResponseEntity**.

```
ResponseEntity<Invite> reponse=restTemplate
    .postForEntity("http://localhost:8080/ajoute",
entity, Invite.class);
```

Elle présente l'avantage de pouvoir accéder à l'entête de la réponse et donc au code retourné.

```
System.out.println("Status de la réponse: "+ reponse.getStatusCodeValue());
```

Accès au champ Location de l'entête:

```
System.out.println("Valeur champ Location:
"+reponse.getHeaders().getFirst("Location"));
```

Résultat observable à l'exécution de la méthode testPost2() pour un nouvel invité:

Status de la réponse: 201 Valeur champ Location: http://localhost:8080/ajoute/48
---

### 3<sup>ème</sup> méthode pour exécuter un web service POST, données transmises en json

Un 2<sup>ème</sup> POST de notre web service reçoit les paramètres d'un objet invité (nom, prénom et email) au format **json**, retourne l'URL de l'objet persisté dans le champ Location de l'entête de la réponse avec le code 201 ou uniquement le code 204 si l'invité n'a pas été persisté.

```
public void doPost3() {
    Invite invite = new Invite("Maurel", "Alain", "a.maurel@orange.fr");
    try {

        HttpHeaders headers = new HttpHeaders();
        headers.setContentType(MediaType.APPLICATION_JSON);

        HttpEntity<Invite> entity=
            new HttpEntity<Invite>(invite,headers);

        ResponseEntity<Invite> reponse=
            restTemplate
                .postForEntity("http://localhost:8080/ajoutejson",
                    entity,Invite.class);
        System.out.println("Status de la réponse: "
            + reponse.getStatusCodeValue());
        System.out.println("Valeur champ Location: "
            + reponse.getHeaders().getFirst("Location"));

    } catch (RestClientResponseException e) {
        System.out.println(e.getRawStatusCode());
    }
}
```

Il faut utiliser un objet HttpHeaders pour préciser le type **APPLICATION\_FORM\_URLENCODED**.

```
HttpHeaders headers = new HttpHeaders();
headers.setContentType(MediaType.APPLICATION_FORM_URLENCODED);
```

On instancie un objet HttpEntity en lui passant comme paramètre l'invité à créer et le header.

```
HttpEntity<Invite> entity=new HttpEntity<Invite>(invite,headers);
```

On exécute la requête comme dans l'exemple précédent.

```
ResponseEntity<Invite> reponse=restTemplate
    .postForEntity("http://localhost:8080/ajoutejson",
        entity,Invite.class);
```

### Une méthode pour exécuter un web service PUT

La classe **RestTemplate** propose 3 méthodes **void put(...)** qui ne sont pas suffisamment intéressantes dans notre exemple car elles ne retournent rien.

La classe **RestTemplate** propose des méthodes **exchange(...)** de type **ResponseEntity<T>** utilisables dans notre exemple.

```
public void doPut() {
    Invite invite = new Invite("Maurel", "Alain", "a.maurel@free.fr");
    try {
```

```

        ResponseEntity<Void> reponse=restTemplate
            .exchange(
                "http://localhost:8080/modifie?nom=Lambrun&prenom=Lucien&email=l.lambrun@free.com"
                ,
                HttpMethod.PUT,
                null,
                Void.class);

        System.out.println("Status de la réponse: "
            + reponse.getStatusCodeValue());
        System.out.println("Valeur champ Location: "
            + reponse.getHeaders().getFirst("Location"));

    } catch (RestClientResponseException e) {
        System.out.println(e.getRawStatusCode());
    }
}

```

Le type de la réponse est **Void** car le **body** dans la réponse est vide. On aurait également pu utiliser **Void** dans les méthodes `doPost()` précédentes.

L'exécution de cette fonction donne:

Status de la réponse: 201 Valeur champ Location: http://localhost:8080/51
--

On modifie maintenant l'email et on exécute à nouveau:

```

"http://localhost:8080/modifie?nom=Lambrun&prenom=Lucien&email=l.lambrun@orange.co
m",

```

L'exécution donne:

Status de la réponse: 200 Valeur champ Location: null
--

## Autre solution pour passer les paramètres pour exécuter un web service PUT

Cet exemple montre comment construire l'URL de requête avec la possibilité d'utiliser des variables pour les paramètres.

```

public void doPut2() {
    try {
        UriComponents uri = UriComponentsBuilder
            .fromUriString("http://localhost:8080/modifie")
            .queryParams("nom", "Carton")
            .queryParams("prenom", "Lucien")
            .queryParams("email", "l.carton@sfr.fr")
            .build();

        ResponseEntity<Void> reponse=
            restTemplate
                .exchange(uri.toUriString(),
                    HttpMethod.PUT,
                    null,
                    Void.class
                );

        System.out.println("Status de la réponse: "
            + reponse.getStatusCodeValue());
    }
}

```



```

        System.out.println("Valeur champ Location: "
            + reponse.getHeaders().getFirst("Location"));

    } catch (RestClientResponseException e) {
        System.out.println(e.getRawStatusCode());
    }
}

```

## Une méthode pour exécuter un web service DELETE

Comme pour le PUT, les 3 méthodes **void delete(...)** de la classe RestTemplate ne retournent rien. On utilise ici la méthode **ResponseEntity<T> exchange(...)**.

```

public void doDelete() {
    try {
        UriComponents uri = UriComponentsBuilder
            .fromUriString("http://localhost:8080/")
            .queryParams("nom", "Carton")
            .queryParams("prenom", "Lucien")
            .build();

        ResponseEntity<Void> reponse=
            restTemplate
                .exchange(uri.toUriString(),
                    HttpMethod.DELETE,
                    null,
                    Void.class
                );

        System.out.println("Status de la réponse: "
            + reponse.getStatusCodeValue());

    } catch (RestClientResponseException e) {
        System.out.println(e.getRawStatusCode());
    }
}

```

Tester dans la méthode run() toutes les méthodes proposées dans le cours. Essayer ces méthodes avec divers URL. Justifier les résultats obtenus.

Développer coté contrôleur 2 méthodes:

- Une méthode PUT dont les données de l'invité à modifier/créer sont fournies dans un objet au format json. Tester avec postman.
- Une méthode DELETE dont les données de l'invité à supprimer {nom, prénom} sont fournies dans un objet au format json. Tester avec postman.

Ecrire les méthodes correspondantes qui consomment ces 2 web services (en suivant le modèle fourni dans ce chapitre).

# Consommer des services REST en JavaScript

On utilise l'application REST 2<sup>ème</sup> version développée dans le cours précédent (p12).

## Préparation du projet SpringBoot

SpringBoot doit publier les fichiers HTML lors de l'exécution de l'application.  
Les fichiers html et javascript peuvent être placés dans le dossier **static** créé dans le dossier **src/main/resources**.

Si on veut les placer dans un dossier différent, suivre la procédure indiquée ci-dessous.

► Ajouter la ligne suivante dans le fichier **application.properties**  
`spring.mvc.static-path-pattern=/**`

► Créer un dossier **public** dans **src/main/resources**.

► Spring Boot publiera par défaut les fichiers HTML contenus dans le dossier **src/main/resources/public**

► Créer un dossier **site** dans le dossier **public** précédent.

Les fichiers HTML seront créés par la suite dans le dossier **src/main/resources/public/site** si on a suivi la procédure ci-dessus ou dans le dossier **src/main/resources/static/site** en utilisant le dossier publié par défaut.

## Installation du plugin Eclipse Web Developer Tools

Ce plugin contient notamment des éditeurs HTML, CSS, JSON et JavaScript.

Menu **Help>Eclipse Marketplace**

Taper web (ou html) dans le champ de saisie **Find**

Faire défiler les résultats et sélectionner **Eclipse Web Developer Tools**

Cliquer sur **Install** puis **Confirm**

Cocher pour accepter la licence puis **Finish**

## Test élémentaire du fonctionnement

On crée 2 fichiers html simples.

➔ **Clic droit sur le dossier static > File**

Nommer le fichier **index.html** puis valider

➔ Taper le contenu suivant :

```
<html>
<head>
</head>
<body>
  <h1>This is a test</h1>
</body>
</html>
```

➔ **Clic droit sur le dossier site > File**

Nommer le fichier **index.html** puis valider

➔Taper le contenu suivant :

```
<html>
<head>
</head>
<body>
    <h1>This is a site test</h1>
</body>
</html>
```

➔Démarrer l'application par : clic droit sur le projet > Run As > Spring Boot App

➔Ouvrir un navigateur et tester les url localhost:8080 et localhost:8080/site/index.html

## L'API JavaScript Fetch

On présente ici l'utilisation de l'API **Fetch** et notamment de la fonction **fetch()**.

La fonction **fetch(url,options)** reçoit 2 paramètres :

- le 1<sup>er</sup> est l'URL de la ressource à atteindre; elle exécute par défaut la **méthode http GET**,
- le 2<sup>ème</sup> paramètre est optionnel, il est inutile pour l'instant.

Elle retourne un objet **Promise**: la "**promesse**" de contenir la réponse à la requête effectuée.

On exécute ensuite **une action** sur cette promesse par **promesse.then()**.

L'enchaînement **.then(...)** prend jusqu'à 2 arguments dont le 1<sup>er</sup> obligatoire est une fonction pour traiter la promesse (donc la réponse) qui sera passée en paramètre de cette fonction. Après l'exécution de cette fonction, cet enchaînement retourne une nouvelle "promesse" qui peut également exécuter un enchaînement **.then(...)** et ainsi de suite.

Exemple :

URL= '.....';

**fetch(URL).then(response=>...).then(response=>...);**

L'objet **Promise** retourné par **fetch()** est du type **Response**.

Un objet **Response** propose plusieurs attributs, notamment:

- L'attribut **status** qui contient le code http de la réponse.
- L'attribut **ok** booléen à true si le code de retour est compris entre 200 et 299.

Un objet **Response** propose plusieurs méthodes pour  récupérer le corps de la réponse, notamment:

- La méthode **text()** retourne la réponse sous la forme d'une chaîne de caractères;
- La méthode **json()** renvoie la réponse en tant qu'objet JavaScript;
- La méthode **formData()** retourne la réponse en tant qu'objet FormData;
- La méthode **arrayBuffer()** retourne la réponse en tant qu'objet ArrayBuffer ;
- La méthode **blob()** retourne la réponse en tant qu'objet Blob;

## GET: un 1<sup>er</sup> exemple REST

On exécute le web point <https://api.binance.com/api/v3/ticker/24hr?symbol=BTCEUR> qui renvoie au format json des informations sur la conversion du bitcoin en euros.

Modifier le fichier **index.html** comme suit :

```

<html>
<head>
<script src="script.js" defer></script>
</head>
<body>
  <h1>This is a site test</h1>
</body>
</html>

```

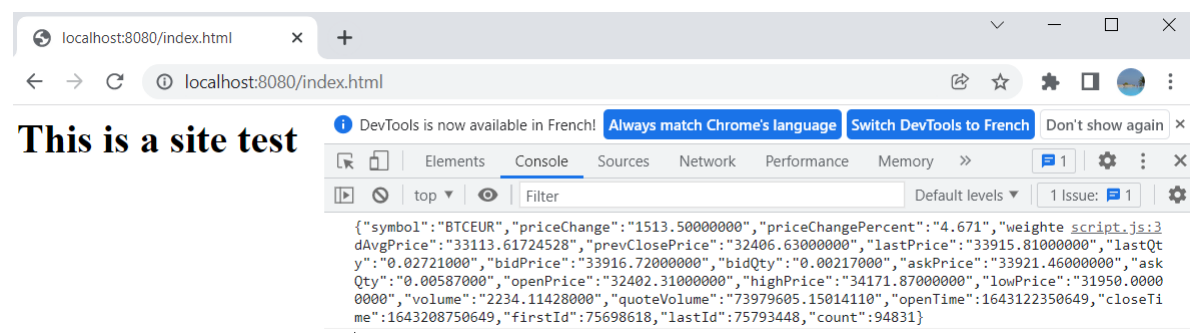
Ecrire le code suivant du **script js script.js** dans le dossier **static** :

```

fetch("https://api.binance.com/api/v3/ticker/24hr?symbol=BTCEUR")
  .then(reponse1=>reponse1.text())
  .then(reponse2=>console.info(reponse2))

```

Après exécution, la réponse «reponse1» est convertie en une chaîne de caractères «reponse2» puis «reponse2» est affichée dans la console du navigateur.



## GET: un 1<sup>er</sup> consommateur REST de l'URL localhost:8080/default

Le fichier getdefault1.html est créé dans le dossier **src/main/resources/public**

```

<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<script type="text/javascript">
function send() {
const url = 'http://localhost:8080/default';
fetch(url)
.then(response => {
  document.getElementById("responsestatus").innerHTML = response.status;
  response.text().then(response => {
    document.getElementById("responsetexte").innerHTML = response;
  })
})
.catch(error => alert("Erreur : " + error));
}
</script>
<title>Page de test getdefault1.html</title>
</head>
<body id='bod'>
  <p>
    <button type="submit" onclick="javascript:send();">Exécution</button>
  </p>
  <p>
    <div id='responsestatus'></div>
  </p>
  <p>
    <div id='responsetexte'></div>
  </p>
</body>
</html>

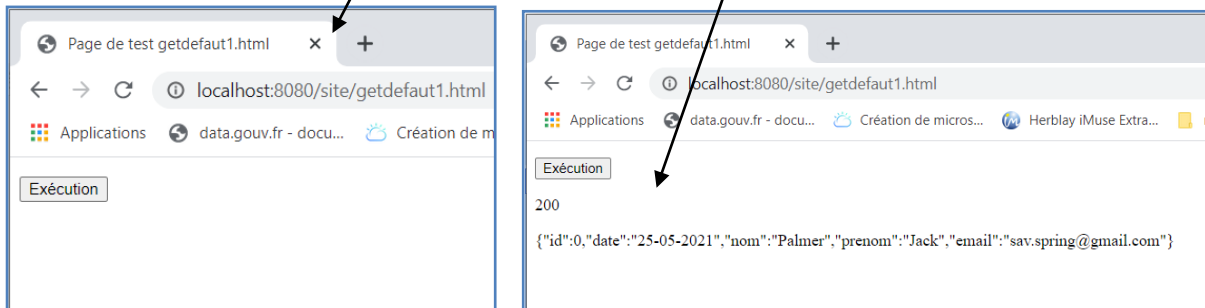
```

Le **fetch** étant exécuté, la promesse est rangée dans **response**.

Puis l'instruction `document.getElementById("responsestatus").innerHTML = response.status;` est exécutée,

Puis l'instruction `response.text()` qui retourne une nouvelle réponse sur laquelle l'instruction suivante `document.getElementById("responsetexte").innerHTML = response` est exécutée.

L'exécution de la page web donne, puis après avoir cliqué sur le bouton



Cet exemple montre bien que le corps de la réponse retourné par **response.text()** est composé d'un string JSON correspondant à un objet invite.

On peut ensuite utiliser la méthode **JSON.parse()** pour analyser le string JSON et retourner un objet JavaScript.

```
var obj = JSON.parse(response);
document.getElementById("responsejson").innerHTML =obj.nom+" "+obj.prenom+" "+
obj.email+" "+obj.date ;
```

Le code complet de la page web.

```
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<script type="text/javascript">
function send() {
const url = 'http://localhost:8080/default';
fetch(url)
.then(response => {
    document.getElementById("responsestatus").innerHTML = response.status;
    response.text().then(response => {
        document.getElementById("responsetexte").innerHTML = response;
        var obj = JSON.parse(response);
        document.getElementById("responseobjet").innerHTML =obj.nom+"
"+obj.prenom+" "+obj.email+" "+obj.date ;
    })
})
.catch(error => alert("Erreur : " + error));
}
</script>
<title>Page de test getdefault1.html</title>
</head>
<body id='bod'>
    <p>
        <button type="submit" onclick="javascript:send();">Exécution</button>
    </p>
    <p>
        <div id='responsestatus'></div>
    </p>
    <p>
```

```

<div id='responsetexte'></div>
</p>
<p>
<div id='responseobjet'></div>
</p>
</body>
</html>

```

## Résultat



## ► Créer une erreur, par exemple avec l'URL localhost:8080/default1

On doit voir le code de retour **404** affiché dans le navigateur.

## ► Gestion d'une erreur avec un code de retour différent de 200

```

<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<script type="text/javascript">
function send() {
const url = 'http://localhost:8080/default1';
fetch(url)
.then(response => {
    document.getElementById("responsestatus").innerHTML = response.status;
    if(response.status==200) {
        response.text().then(response => {
            document.getElementById("responsetexte").innerHTML = response;
            var obj = JSON.parse(response);
            document.getElementById("responseobjet").innerHTML =obj.nom+"
"+obj.prenom+" "+obj.email+" "+obj.date ;
        })
    }
    else
        document.getElementById("responsetexte").innerHTML = "Erreur, code de
retour incorrect!";
    })

```

```

.catch(error => alert("Erreur : " + error));
}
</script>
<title>Page de test getdefault1.html</title>
</head>
<body id='bod'>
  <p>
    <button type="submit" onclick="javascript:send();">Exécution</button>
  </p>
  <p>
    <div id='responsestatus'></div>
  </p>
  <p>
    <div id='responsetexte'></div>
  </p>
  <p>
    <div id='responseobjet'></div>
  </p>
</body>
</html>

```

Tester.

## GET: un 2<sup>ème</sup> consommateur REST de l'URL localhost:8080/default

Le fichier getdefault2.html est créé dans le dossier **src/main/resources/public**

La méthode **response.text()** peut être ici avantageusement remplacée par **response.json()** car celle-ci retourne directement un objet JavaScript.

```

<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<script type="text/javascript">
function send() {
const url = 'http://localhost:8080/default1';
fetch(url)
.then(response => {
  document.getElementById("responsestatus").innerHTML = response.status;
  if(response.status==200) {
    response.json().then(obj => {
      document.getElementById("responseobjet").innerHTML =obj.nom+"
"+obj.prenom+" "+obj.email+" "+obj.date ;
    })
  }
  else
    document.getElementById("responsetexte").innerHTML = "Erreur, code de
retour incorrect!";
})
.catch(error => alert("Erreur : " + error));
}
</script>
<title>Page de test getdefault2.html</title>
</head>
<body id='bod'>
  <p>
    <button type="submit" onclick="javascript:send();">Exécution</button>
  </p>

```

```

    </p>
    <p>
    <div id='responsestatus'></div>
    </p>
    <p>
    <div id='responsetexte'></div>
    </p>
    <p>
    <div id='responseobjet'></div>
    </p>
</body>
</html>

```

## GET: Le consommateur REST de l'URL localhost:8080/all

La méthode `response.json().then(data=> {...` retourne dans `data` un tableau d'objets JavaScript. On obtient le nombre d'objet par `data.length`. Il faut ensuite parcourir ce tableau d'objets avec une boucle et créer dynamiquement le contenu de la page web, chaque objet étant accessible par `data[i]`.

```

<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<script type="text/javascript">
function send() {
const url = 'http://localhost:8080/all';
fetch(url)
.then(response=> {
    document.getElementById("responsestatus").innerHTML = response.status;
    console.log(response);
    if (response.status==200) {
        response.json().then(data=> {
            document.getElementById("responsenombre").innerHTML = "Nombre
d'invités: "+data.length ;
            var parag = document.createElement('p');
            parag.textContent='Liste des invités: ';
            var list = document.createElement('ul');
            for (var i=0; i<data.length ;i++) {
                var listItem = document.createElement('li');
                listItem.textContent = data[i].nom+" "+data[i].prenom+"
"+data[i].email+" "+data[i].date;
                list.appendChild(listItem);
            }
            parag.appendChild(list);
            var node = document.createElement('div');
            node.appendChild(parag);
            document.body.appendChild(node);
        })
    }
})
.catch(error=>console.log("Erreur grave: "+error));
}
</script>
<title>Page de test</title>
</head>
<body id='bod'>
    <p>
    <button type="submit" onclick="javascript:send();">Exécution</button>

```



```

    </p>
    <p>
    <div id='responsestatus'></div>
    </p>
    <p>
    <div id='responseenombre'></div>
    </p>
</body>
</html>

```

## GET: Le consommateur REST de l'URL localhost:8080/{invite}

Le paramètre variable de l'URL correspondant au nom de l'invité est fourni dans un champ de saisie:

```
<input type="text" id="nom" name="user_name">
```

On lit cette valeur dans la fonction **send()** et on l'ajoute à l'URL de base:

```

var nom=document.getElementById('nom').value;
const url = 'http://localhost:8080/'+nom;

```

### Le code de la page HTML

```

<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<script type="text/javascript">
function send() {
var nom=document.getElementById('nom').value;
const url = 'http://localhost:8080/'+nom;
fetch(url)
.then(response=> {
    document.getElementById("responsestatus").innerHTML = response.status;
    console.log(response);
    if (response.status==200) {
        response.json().then(obj => {
            document.getElementById("responseobjet").innerHTML =obj.nom+"
"+obj.prenom+" "+obj.email+" "+obj.date ;
        })
    }
    else document.getElementById("responsetexte").innerHTML="Erreur";
})
.catch(error=>console.log("Erreur grave: "+error));
}
</script>
<title>Page de test</title>
</head>
<body id='bod'>
    <p>
    <label for="name">Nom :</label>
    <input type="text" id="nom" name="user_name">
    <br><br>
    <button type="submit" onclick="javascript:send();">Exécution</button>
    </p>
    <p>
    <div id='responsestatus'></div>
    </p>
    <p>

```

```

    <div id='responsetexte'></div>
  </p>
  <div id='responseobjet'></div>
</body>
</html>

```

Tester avec un invité présent dans la base et un qui n'existe pas.

## GET: Le consommateur REST de l'URL localhost:8080

Les paramètres de recherche passés à l'url de base **nom=...&prénom=...** sont fournis dans 2 champs de saisie:

```

var n=document.getElementById('nom').value;
var p=document.getElementById('prenom').value;

```

Une 1<sup>ère</sup> solution serait d'effectuer une concaténation des chaînes de caractères pour obtenir l'url final, par exemple:

```
url = 'localhost:8080?nom='+n+'&prenom='+p
```

On propose ici une autre solution qui utilise la classe **URL**.

```
const url = new URL('http://localhost:8080');
```

L'objet **URL** ainsi obtenu possède la propriété **searchParams** qui permet de récupérer/modifier/ajouter les paramètres de recherche de l'url de la requête à exécuter.

```
var params = url.searchParams;
```

On positionne les paramètres de la recherche:

```

params.append('nom',n);
params.append('prenom',p);

```

On exécute le fetch() avec l'url modifié:

```

fetch(url)
.then(response=> {----

```

### Code de la page html

```

<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<script type="text/javascript">
function send() {
var n=document.getElementById('nom').value;
var p=document.getElementById('prenom').value;
const url = new URL('http://localhost:8080');
var params = url.searchParams;
params.append('nom',n);
params.append('prenom',p);
fetch(url)
.then(response=> {
    document.getElementById("responsestatus").innerHTML = response.status;
    console.log(response);
    if (response.status==200) {
        response.json().then(obj => {
            document.getElementById("responseobjet").innerHTML =obj.nom+"
"+obj.prenom+" "+obj.email+" "+obj.date ;

```

```

    })
  }
  else document.getElementById("responsetexte").innerHTML="Erreur";
}
}.catch(error=>console.log("Erreur grave: "+error));
}
</script>
<title>Page de test</title>
</head>
<body id='bod'>
  <p>
    <label for="name">Nom :</label>
    <input type="text" id="nom" name="user_name">
    <br><br>
    <label for="surname">Prénom :</label>
    <input type="text" id="prenom" name="user_surname">
    <br><br>
    <button type="submit" onclick="javascript:send();">Exécution</button>
  </p>
  <p>
    <div id='responsestatus'></div>
  </p>
  <p>
    <div id='responsetexte'></div>
  </p>
  <div id='responseobjet'></div>
</body>
</html>

```

Tester avec un invité présent dans la base et un qui n'existe pas.

## POST: un 1<sup>er</sup> consommateur REST de l'URL localhost:8080/ajoutejson

La fonction **fetch()** reçoit comme 1<sup>er</sup> argument l'URL de la ressource à atteindre et comme 2<sup>ème</sup> argument un paramètre qui contient toutes les paramètres spécifiques à la requête à exécuter: la méthode POST, le type de données dans le header et le string json correspondant à l'objet à persister.

La méthode JSON.stringify(invite) retourne le string JSON correspondant à l'objet invité passé en argument.

```

<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<script type="text/javascript">
function send() {
const invite = {
  nom:'Michelin',
  prenom:'Roland',
  email:'r.michelin@gmail.com'
};
const parametres = {
  method:'POST',
  body:JSON.stringify(invite),
  headers:{
    'Content-Type':'application/json'
  }
};

```

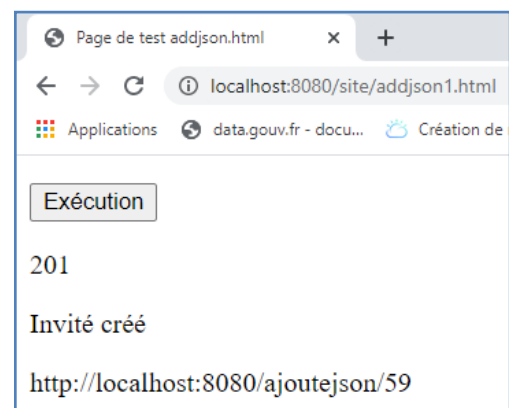
```

const url = 'http://localhost:8080/ajoutejson';
fetch(url, parametres)
.then(response => {
    document.getElementById("responsestatus").innerHTML = response.status;
    if(response.status==201) {
        document.getElementById("responsetexte").innerHTML ="Invité créé" ;
        document.getElementById("responseobjet").innerHTML
=response.headers.get('Location');
    }
    else
        document.getElementById("responsetexte").innerHTML = "Erreur, code de
retour incorrect!";
    })
.catch(error => alert("Erreur : " + error));
}
</script>
<title>Page de test addjson.html</title>
</head>
<body id='bod'>
    <p>
        <button type="submit" onclick="javascript:send();">Exécution</button>
    </p>
    <p>
        <div id='responsestatus'></div>
    </p>
    <p>
        <div id='responsetexte'></div>
    </p>
    <p>
        <div id='responseobjet'></div>
    </p>
</body>
</html>

```

► L'exécution donne :

► Exécuter une 2<sup>ème</sup> fois, le code 204 doit être affiché.



## POST: un 2<sup>ème</sup> consommateur REST de l'URL localhost:8080/ajoutejson

Les données sont fournies à l'aide d'un formulaire.

L'objet invité est initialisé à partir des données du formulaire:

```

var invite = {
    nom:document.getElementById("nom").value,
    prenom:document.getElementById("prenom").value,
    email:document.getElementById("mail").value
};

```

### Le fichier html complet

```

<!DOCTYPE html>
<html>
<head>

```

```

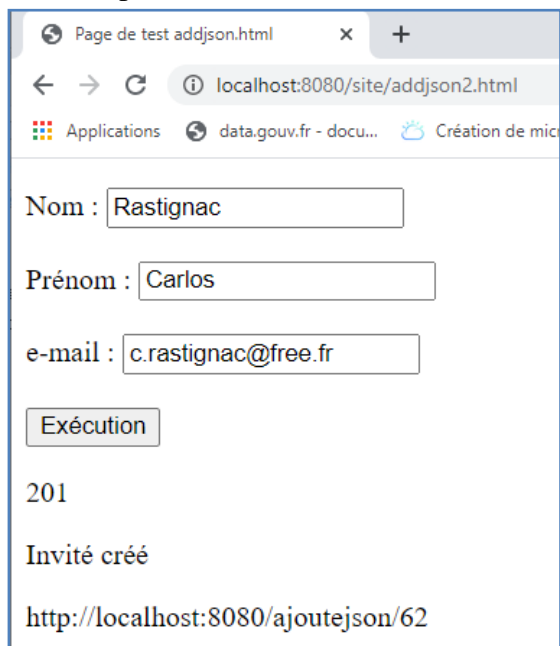
<meta charset="ISO-8859-1">
<script type="text/javascript">
function send() {
var invite = {
  nom:document.getElementById("nom").value,
  prenom:document.getElementById("prenom").value,
  email:document.getElementById("mail").value
};
const parametres = {
  method:'POST',
  body:JSON.stringify(invite),
  headers:{
    'Content-Type':'application/json'
  }
};
const url = 'http://localhost:8080/ajoutejson';
fetch(url,parametres)
.then(response => {
  document.getElementById("responsestatus").innerHTML = response.status;
  if(response.status==201) {
    document.getElementById("responsetexte").innerHTML ="Invit&eacute;
cr&eacute;&eacute;";
    document.getElementById("responseobjet").innerHTML
=response.headers.get('Location');
  }
  else
    document.getElementById("responsetexte").innerHTML = "Erreur, code de
retour incorrect!";
  })
.catch(error => alert("Erreur : " + error));
}
</script>
<title>Page de test addjson.html</title>
</head>
<body id='bod'>
  <p>
    <form action=javascript:send() method="post">
      <label for="name">Nom :</label>
      <input type="text" id="nom" name="user_name">
    <br><br>
      <label for="surname">Pr&eacute;nom :</label>
      <input type="text" id="prenom" name="user_surname">
    <br><br>
      <label for="mail">e-mail :</label>
      <input type="email" id="mail" name="user_mail">
    <br><br>
      <button type="submit">Ex&eacute;cution</button>
    </form>
  </p>
  <p>
    <div id='responsestatus'></div>
  </p>
  <p>
    <div id='responsetexte'></div>
  </p>
  <p>
    <div id='responseobjet'></div>
  </p>
</body>

```

</html>

### ► Les résultats obtenus:

Après une 1<sup>ère</sup> exécution



Page de test addjson.html x +

localhost:8080/site/addjson2.html

Nom : Rastignac

Prénom : Carlos

e-mail : c.rastignac@free.fr

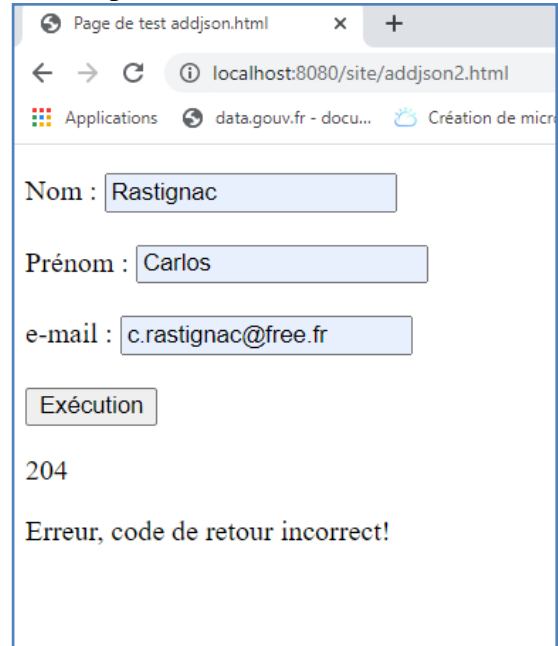
Exécution

201

Invité créé

http://localhost:8080/ajoutejson/62

Après une 2<sup>ème</sup> exécution



Page de test addjson.html x +

localhost:8080/site/addjson2.html

Nom : Rastignac

Prénom : Carlos

e-mail : c.rastignac@free.fr

Exécution

204

Erreur, code de retour incorrect!

## DELETE: un consommateur REST de l'URL localhost:8080/deljson

Le end-point **DELETE localhost:8080/deljson** reçoit dans le body de la requête le string json correspondant à l'invité à supprimer. Ce string json contient le nom et le prénom de l'invité à supprimer.

```
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<script type="text/javascript">
function send() {
var invite = {
  nom:document.getElementById("nom").value,
  prenom:document.getElementById("prenom").value
};
const parametres = {
  method:'DELETE',
  body:JSON.stringify(invite),
  headers:{
    'Content-Type':'application/json'
  }
};
const url = 'http://localhost:8080/deljson';
fetch(url,parametres)
.then(response => {
  document.getElementById("responsestatus").innerHTML = response.status;
  if(response.status==204) {
    document.getElementById("responsetexte").innerHTML ="Invité
supprimé;" ;
```

```

        document.getElementById("responseobjet").innerHTML
=response.headers.get('Location');
    }
    else
        document.getElementById("responsetexte").innerHTML = "Erreur, code de
retour incorrect!";
    })
    .catch(error => alert("Erreur : " + error));
}
</script>
<title>Page de test addjson.html</title>
</head>
<body id='bod'>
    <p>
        <form action=javascript:send() method="post">
            <label for="name">Nom :</label>
            <input type="text" id="nom" name="user_name">
        <br><br>
            <label for="surname">Prénom :</label>
            <input type="text" id="prenom" name="user_surname">
        <br><br>
            <button type="submit">Exécuter</button>
        </form>

        </p>
        <p>
            <div id='responsestatus'></div>
        </p>
        <p>
            <div id='responsetexte'></div>
        </p>
        <p>
            <div id='responseobjet'></div>
        </p>
    </body>
</html>

```

► Effectuer plusieurs tests.