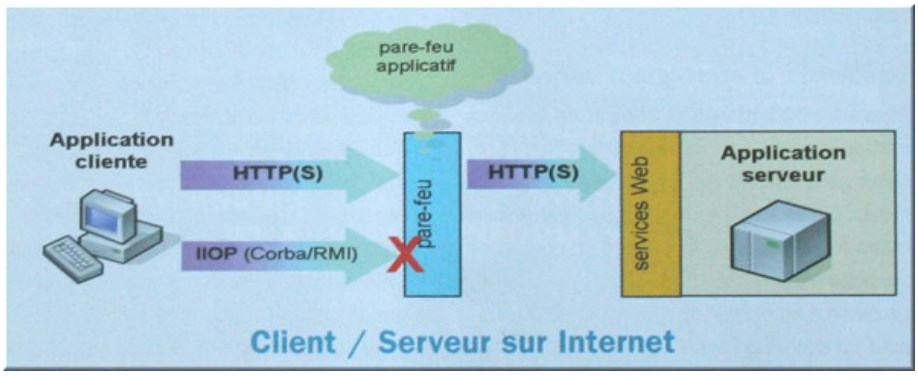
Durant cette étude, nous allons abordé les Services Web qui utilisent directement le protocole **HTTP** sans couche supplémentaire. Ces services sont dénommés **Service Web REST**. Cette approche, relativement simple à implémenter, correspondant à l'utilisation même du protocole **HTTP**, consiste à gérer des ressources distantes avec toutes les phases classiques : les phases de création, de récupération, de modification et de suppression (**CRUD**).

Présentation des services web avec des applications clientes riches

Avec la librairie intégrée de **Rust**, nous avons déjà élaboré de nombreux services différents grâce à des objets spécialisés comme **TcpListener** côté serveur et **TcpStream** côté client, qui sont également très faciles à développer avec un codage extrêmement simplifié. Par contre, pour que ce fonctionnement simple puisse s'établir, vous devez rester dans le réseau local de l'entreprise (les numéros de port des différents services vont être bloqués par le pare-feu de l'entreprise).



Le top du top serait de pouvoir faire comme en réseau local, c'est-à-dire de pouvoir utiliser une application fenêtrée, qui fait appel aux différents services, tout en étant sur Internet, et donc sans passer par un navigateur. Il existe une solution pour cela, il s'agit de la technique des services Web. En réalité, ces services Web communiquent comme une application Web, c'est-à-dire au travers du protocole HTTP (ce qui permet la communication par Internet).

Architecture d'un Service Web REST

REST est un type d'architecture reposant sur le fonctionnement même du web qu'il applique aux services web. Pour concevoir un **Service REST**, il faut bien connaître tout simplement le **protocole HTTP**, le principe des **URI** et respecter quelques règles. Il faut raisonner en terme de **ressources**.

Dans l'architecture REST, toute information est une ressource et chacune d'elles est désignée par une URI - généralement un lien sur le Web. Les ressources sont manipulées par un ensemble d'opérations simples et bien définies. L'architecture client-serveur de type REST est conçue pour utiliser un protocole de communication sans état. Ces principes encouragent la simplicité, la légèreté et l'efficacité des applications.

Termes	Explications
Ressources	***Les ressources jouent un rôle central dans les architectures REST. Une ressource est tout ce que peut désigner ou manipuler un client, tout information pouvant être référencée dans un lien hypertexte. Elle peut être stockée dans une base de données, un fichier, etc. Il faut éviter autant que possible d'exposer des concepts abstraits sous forme de ressources et privilégier plutôt les objets simples.***
URI	***Une ressource web est identifiée par une URI, qui est un identifiant unique formé d'un nom et d'une adresse indiquant où trouver la ressource. Il existe différents types d'URI : les adresses web, les UDI, les URI, et enfin les combinaisons d'URL et d'URN.***

Quelques exemples d'URI	`http://www.films.fr/catégories/aventure`
	`http://www.films.fr/catalogue/titres/films/123456`
	`http://www.météo.com/temps/2012?location=Aurillac,France`
	Les URI devraient être aussi descriptives que possible et ne désigner qu'une seule ressource.

Représentation	***Nous pouvons obtenir la représentation d'un objet sous forme de texte, de XML de PDF ou tout autre format. Un client traite toujours une ressource au travers de sa représentation ; la ressource elle-même reste sur le serveur. La représentation contient toutes les informations utiles à propos de l'état d'une ressource :***
	`http://www.presse.fr/livre/catalogue/cpp`
	`http://www.presse.fr/livre/catalogue/cpp.csv`
	`http://www.presse.fr/livre/catalogue/cpp.xml`

Le protocole HTTP – requête GET

La requête la plus simple du protocole **HTTP** est formé de **GET** suivi d'une **URL** qui pointe sur des données sous forme de fichiers statiques ou de documents issus d'un traitement dynamique.... Elle est envoyée par un navigateur quand nous saisissons directement une **URL** dans le champ d'adresse du navigateur. Le serveur **HTTP** répond en renvoyant les données demandées.

En tapant l'URL d'un site, l'internaute envoie via le navigateur une requête au serveur.

Une connexion s'établit entre le client et le serveur sur le port 80 port par défaut d'un serveur Web.

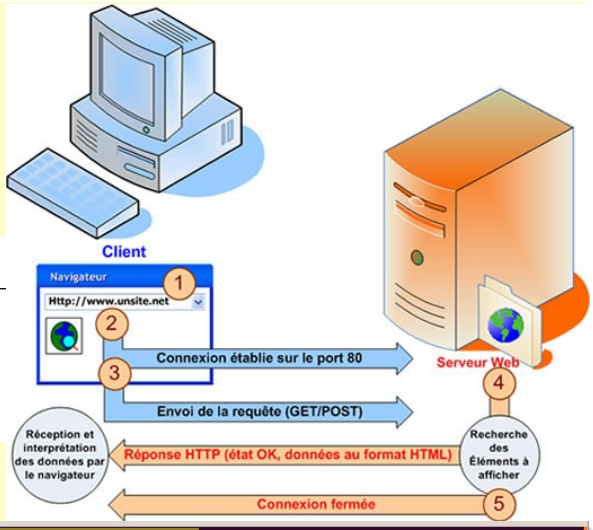
Le navigateur envoie une requête demandant l'affichage d'un document. La requête contient entre autres la méthode GET, POST, etc. qui précise comment l'information est envoyée.

Le serveur répond à la requête en envoyant une réponse HTTP composée de plusieurs parties, dont :

L'état de la réponse, à savoir une ligne de texte qui décrit le résultat du serveur (code 200 pour un accord, 400 pour une erreur due au client, 500 pour un erreur due au serveur) ;

Les données à afficher.

Une fois la réponse reçue par le client, la connexion est fermée. Pour afficher une nouvelle page du site, une nouvelle connexion doit être établie.



Requêtes et réponses du protocole HTTP

Un client **envoie une requête** à un serveur afin d'obtenir une réponse. Les messages utilisés pour ces échanges sont formés d'une enveloppe et d'un corps également appelé document ou représentation. Voici, par exemple, un type de requête envoyée à un serveur :

Cette requête contient plusieurs informations envoyées par le client :

- *La méthode HTTP GET ;*
- *Le chemin, ici la racine / ;*
- *Plusieurs autre en-têtes de requête.*
- *Vous remarquez que la requête n'a pas de corps un GET n'a jamais de corps. En réponse, le serveur renvoie sa réponse et elle est formée des parties suivantes :*
- *Un code de réponse : ici le code est 200 OK.*
- *Plusieurs en-têtes de réponse, notamment Date, Server, Content-Type. Ici, le type de contenu est « text/html », mais il pourrait s'agir de n'importe quel format comme du XML « application/json» ou une image « image/jpeg », etc.*
- *Un corps ou représentation. Ici, il s'agit du contenu de la page web renvoyée qui n'est pas visible sur la figure proposée ci-contre.*

```
manu@214-0:~$ curl -v http://www.google.fr
* About to connect() to www.google.fr port 80 (#0)
*   Trying 74.125.230.248... connected
* Connected to www.google.fr (74.125.230.248) port 80 (#0)
> GET / HTTP/1.1
> User-Agent: curl/7.21.6 (i686-pc-linux-gnu) libcurl/7.21.6 OpenSSL/1.0.0e zlib
/1.2.3.4 libidn/1.22 librtmp/2.3
> Host: www.google.fr
> Accept: */*
>
* HTTP 1.0, assume close after body
< HTTP/1.0 200 OK
< Date: Tue, 24 Apr 2012 15:15:13 GMT
< Expires: -1
< Cache-Control: private, max-age=0
< Content-Type: text/html; charset=ISO-8859-1
< Set-Cookie: PREF=ID=c6ea540480cc0b07:FF=0:TM=1335280513:LM=1335280513:S=JxT3BE
_X-XIaUnx4; expires=Thu, 24-Apr-2014 15:15:13 GMT; path=/; domain=.google.fr
< Set-Cookie: NID=59=fQpyGIyIYKAUR3uO49DI3p2MaRSgWBnJ48CI6hejGJQLrN0ueaU-UjJL2Yo
```

Soumettre la Requête.

En-têtes de requête.

Réponse envoyée par le serveur.

Les méthodes du protocole HTTP

Le web est formé de ressources bien identifiées, reliées ensembles et auxquelles accéder au moyen de requêtes **HTTP** simples. Les requêtes principales de **HTTP** sont de type **GET**, **POST**, **PUT** et **DELETE**. Ces types sont appelés verbes ou méthodes. **HTTP** définit quatre autres verbes plus rarement utilisés, **HEAD**, **TRACE**, **OPTIONS** et **CONNECT**.

Méthode	Description
GET	*GET est une méthode de lecture demandant une représentation d'une ressource. Elle doit être implémentée de sorte à ne pas modifier l'état de la ressource. En outre, GET doit être idempotente, ce qui signifie qu'elle doit laisser la ressource dans le même état, quel que soit le nombre de fois où elle est appelée. Ces deux caractéristiques garantissent une plus grande stabilité : si un client n'obtient pas de réponse à cause d'un problème réseau, par exemple, il peut renouveler sa requête et s'attendre à la même réponse que celle qu'il aurait obtenue initialement, sans corrompre l'état de la ressource sur le serveur.*
POST	*POST crée une nouvelle ressource subordonnée à une ressource principale identifiée par l'URI demandée. Des exemples d'utilisation de POST sont l'ajout d'un message à un fichier journal, d'un livre à une liste d'ouvrages, etc. POST modifie donc l'état de la ressource et n'est pas idempotente envoyer deux fois la même requête produit deux nouvelles ressources subordonnées. Si une ressource a été créée sur le serveur d'origine, le code de la réponse devrait être 201 Created.*
PUT	*La méthode PUT modifie toutes les représentations actuelles de la ressource visée avec le contenu de la requête.*
PATCH	*La méthode PATCH est utilisée pour appliquer des modifications partielles à une ressource.*
DELETE	*Une requête DELETE supprime une ressource. La réponse à DELETE peut être un message d'état dans le corps de la réponse ou aucun code du tout. DELETE est idempotente, mais elle modifie évidemment l'état de la ressource.*
HEAD	*HEAD ressemble à GET sauf que le serveur ne renvoie pas de corps dans sa réponse. HEAD permet par exemple de vérifier la validité d'un client ou la taille d'une entité sans avoir besoin de la transférer.*
TRACE	*TRACE retrace la requête reçue.*
OPTION	*OPTION est une demande d'information sur les options de communication disponibles pour la chaîne requête/réponse identifiée par l'URI. Cette méthode permet au client de connaître les options et/ou les exigences associées à une ressource, ou les possibilités d'un serveur sans demander d'action sur une ressource et sans récupérer aucune ressource.*
CONNECT	*CONNECT est utilisé avec un proxy pouvant se transformer dynamiquement en tunnel une technique grâce à laquelle le protocole HTTP sert d'enveloppe à différents protocoles réseau.*

Négociation du contenu

La négociation du contenu utilise entre autres les en-têtes **HTTP** : ***Accept, Accept-Charset, Accept-Encoding, Accept-Language et User-Agent***. Pour obtenir, par exemple, la représentation **CSV** de la liste des livres sur C++ publiés, l'application cliente ***« l'agent utilisateur »*** demandera **http://www.presse.com/livres/catalog** avec un en-tête ***Accept*** initialisé à ***« text/csv »***.

Type de contenu

HTTP utilise des types de supports Internet **« initialement appelés types MIME »** dans les en-têtes **Content-Type** et **Accept** afin de permettre un typage des données et une négociation de contenu ouverts et extensibles. Les types de support Internet sont divisés en cinq catégories : **« text, image, audio, video et application ».** Ces types sont à leur tour divisés en sous-types ***text/plain, text/html, text/xhtml, etc.***. Voici quelques-uns des plus utilisés :

Type de contenu	Description
text/html	***HTML est utilisé par l'infrastructure d'information du World Wide Web depuis 1990 et sa spécification a été décrite dans plusieurs documents informels. Le type de support « text/html » a été initialement défini en 1995. Il permet d'envoyer et d'interpréter les pages web classiques.***
text/plain	***Il s'agit du type de contenu par défaut car il est utilisé pour les messages textuels simples.***
image/gif image/jpeg image/png	***Le type de support image exige la présence d'un dispositif d'affichage, un écran ou une imprimante graphique, par exemple permettant de visualiser l'information.***
text/xml application/xml	***Envoi et réception de document XML.***
application/json	***JSON est un format textuel léger pour l'échange de données. Il est indépendant des langages de programmation.***

Code d'état

Un code **HTTP** est associé à chaque réponse. La spécification définit environ 60 codes d'états ; l'élément **Status-Code** est un entier de trois chiffres qui décrit le contexte d'une réponse et qui est intégré dans l'enveloppe de celle-ci. Le premier chiffre indique l'une des cinq classes de réponses possibles :

1xx : Information : La requête a été reçue et le traitement se poursuit.

2xx : Succès : L'action a bien été reçue, comprise et acceptée.

3xx : Redirection : Une autre action est requise pour que la requête s'effectue.

4xx : Erreur du client : La requête contient des erreurs de syntaxe ou ne peut pas être exécutée.

5xx : Erreur du serveur : Le serveur n'a pas réussi à exécuter une requête pourtant apparemment valide.

Voici quelques codes d'état que vous avez sûrement déjà dû rencontrer :

200 OK : La requête a réussi. Le corps de l'entité, si elle en possède un, contient la représentation de la ressource.

301 Moved Permanently : La ressource demandée a été affectée à une autre URI permanente et toute référence future à cette ressource devrait utiliser l'une des URI renvoyées.

404 Not Found : Le serveur n'a rien trouvé qui corresponde à l'URL demandée.

500 Internal Server Error : Le serveur s'est trouvé dans une situation inattendue qui l'a empêché de répondre à la requête.

Du Web aux Services Web

Nous savons comment fonctionne le Web : pourquoi les services web devraient-ils se comporter différemment ? Après tout, ils échangent souvent uniquement des ressources bien identifiées, liées à d'autres au moyen de liens hypertextes. L'architecture du Web ayant prouvé sa tenue en charge au cours du temps, pourquoi réinventer la roue ?

Pour créer, modifier et supprimer une ressource livre, pourquoi ne pas utiliser les verbes classiques de HTTP ? Par exemple :

- ***Utiliser POST sur des données au format XML, JSON ou texte afin de créer une ressource livre avec l'URI http://www.site.com/livres/. Le livre créé, la réponse renvoie l'URI de la nouvelle ressource http://www.site.com/livres/123456.***
- ***Utiliser GET pour lire la ressource et les éventuels liens vers d'autres ressources à partir du corps de l'entité à l'URI http://www.apress.com/books/123456.***
- ***Utiliser PUT pour modifier la ressource à l'URI http://www.site.com/livres/123456.***
- ***Utiliser DELETE pour supprimer la ressource à l'URI http://www.site.com/livres/123456.***

En se servant ainsi des verbes HTTP, nous pouvons donc effectuer toutes les actions CRUD sur une ressource à l'image des bases de données.

Sans état

La dernière fonctionnalité de **REST** est l'absence d'état, ce qui signifie que toute requête **HTTP** est totalement indépendante puisque le serveur ne mémorisera jamais les requêtes qui ont été effectuées.

Pour plus de clarté, l'état de la ressource et celui de l'application sont généralement différenciés : l'état de la ressource doit se trouver sur le serveur et être partagé par tous, tandis que celui de l'application doit rester chez le client et être sa seule propriété.

Si nous revenons à l'exemple des livres, l'état de l'application est que le client a récupéré par exemple une représentation du livre désiré, mais le serveur ne mémorisera pas cette information. L'état de la ressource, quant à lui, est l'information sur l'ouvrage : le serveur doit évidemment la mémoriser et le client peut le modifier. Si le panier virtuel est une ressource dont l'accès est réservé à un seul client, l'application doit stocker l'identifiant de ce panier dans la session du client.

L'absence d'état possède de nombreux avantages, notamment une meilleure adaptation à la charge : aucune information de session à gérer, pas besoin de router les requêtes suivantes vers le même serveur, gestion des erreurs, etc. Si vous devez mémoriser l'état, le client devra faire un travail supplémentaire pour le stocker.

Format JSON

JSON est un format léger pour l'échange de données structurées complexes. Il est à l'image des documents **XML** en moins verbeux. Il est très utile lorsque vous devez transférer toutes les informations relatives à une structure ou à un objet persistant par exemple. Voici ci-dessous un exemple de document **JSON** représentant un objet de type **Personne** qui peut disposer de plusieurs numéros de téléphones :

```
{
  "id" : 51,
  "nom" : "PÊCHEUR",
  "prénom" : "Martin",
  "naissance" : 18/11/1969,
  "téléphones" : ["04-45-18-99-77", "06-89-89-87-23", "04-72-33-55-84"]
}
```

L'ossature du document ressemble à une structure dont le début et la fin sont désignés par des accolades. Chaque élément du document possède une clé et une valeur associée. L'ensemble des éléments sont séparés par des virgules. Pour la définition de la clé et de la valeur, vous devez l'écrire entre guillemets, sauf éventuellement pour les valeurs numériques. Enfin, si une clé possède plusieurs valeurs, vous devez les spécifier entre des crochets séparées par des virgules et toujours écrites entre guillemets.

*Dans **Rust**, l'implémentation du format **JSON** se fait au travers de la librairie « **serde** ». L'idée générale est de travailler avec le type **struct** dont la mise en œuvre doit donner une image précise du document **JSON** à soumettre ou à récupérer. La technique utilisée est une **sérialisation** (et une **dé-sérialisation** dans l'autre sens) où chacune des valeurs de chaque attribut de la structure est automatiquement transformé dans le bon type interne (là aussi dans les deux sens).*

Mise en œuvre du Service Web REST – Librairie Rocket

Après toute cette démarche explicative sur les Services Web de type **REST**, je vous propose maintenant de les mettre en œuvre grâce à la librairie **Rocket**. Vous devez régler les dépendances comme tout autre bibliothèque, comme suit :

Cargo.toml

```
[package]
name = "REST-Rocket"
version = "0.1.0"
edition = "2021"

[dependencies]
rocket = "0.5.0-rc.1"
```

main.rs

```
#[macro_use] extern crate rocket;

#[get("/")]
fn index() -> &'static str {
    "Bienvenue à tout le monde !"
}

#[launch]
fn rocket() -> _ {
    rocket::build().mount("/", routes![index])
}
```

Résultat

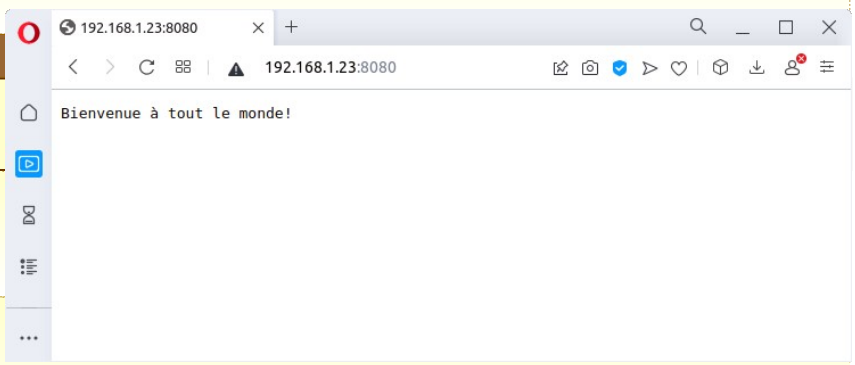
```
Configured for release.
   >> address: 127.0.0.1
   >> port: 8000
   >> workers: 8
   >> ident: Rocket
   >> keep-alive: 5s
   >> limits: bytes = 8KiB, data-form = 2MiB, file = 1MiB, form = 32KiB, json = 1MiB, msgpack = 1MiB, string = 8KiB
   >> tls: disabled
   >> temp dir: /tmp
   >> log level: critical
   >> cli colors: true
```

```
   >> shutdown: ctrlc = true, force = true, signals = [SIGTERM], grace = 2s, mercy = 3s
Routes:
   >> (index) GET /
Fairings:
   >> Shield (liftoff, response, singleton)
Rocket has launched from http://127.0.0.1:8000
```

*Par défaut, vous remarquez que le service Web est activé à **@IP 127.0.0.1** et sur le port **8000**. Si vous désirez modifier ces valeurs et offrir d'autres réglages possibles, vous pouvez les réaliser au travers d'un fichier de configuration spécifique du nom de « **Rochet.toml** ». Nous pouvons effectuer beaucoup plus de réglages personnalisés, ici nous proposons juste le choix de l'adresse IP et du port.*

Rocket.toml

```
[default]
address = "0.0.0.0"
port = 8080
```



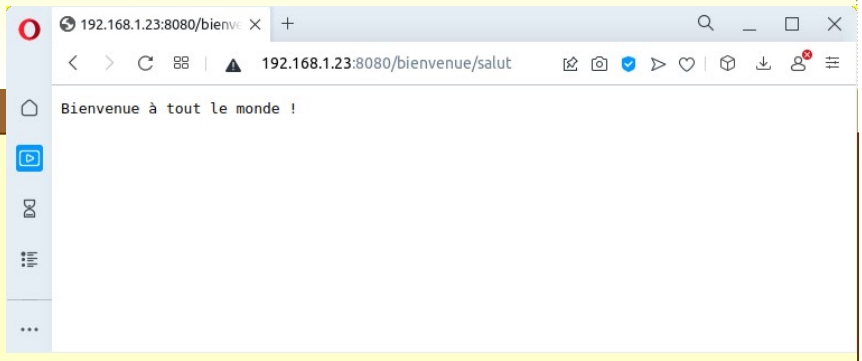
*Vous avez ci-contre l'exemple de l'utilisation de notre service Web qui prend en compte pour l'instant une seule fonction **GET**, utilisable avec un simple navigateur.*

*Lorsque nous consultons notre code source, avec cette API **Rocket**, vous remarquez que la mise en œuvre est vraiment très concise.*
*La première ligne spécifie nous pouvons utiliser des annotions (des directives) au dessus de chaque fonction faisant partie du service, comme par exemple **#[get(« / »)]**.*

*Grâce à ce modèle d'écriture, nous indiquons que la fonction **index()** répond automatiquement à une requête **HTTP GET** avec une **URL « / »**. On ne peut pas faire plus simple.*

*Il existe bien une fonction principale mais son nom est différent - **rocket()**. Il est nécessaire, pour qu'elle soit exécutée à la place de la fonction **main()**, qu'elle possède la directive **#[launch]**. Grâce à cette organisation le service est bien lancé après avoir monter – **mount()** - toutes les routes possibles de ce service. Ici, nous spécifions que seule la fonction **index()** fait partie du service.*

*Par ailleurs, le service est monté à la racine du serveur Web **« / »**. Si vous souhaitez intégrer plusieurs services Web, il peut être judicieux de proposer un nom particulier, pour le différencier des autres. À titre d'exemple, nous pouvons précisez que notre service Web se nomme **bienvenue** et que l'accès à la méthode **GET** se fait par l'URL « **salut** »*



main.rs

```
use rocket::*;

#[get("/salut")]
fn index() -> &'static str {
    "Bienvenue à tout le monde !"
}

#[launch]
fn service_web_rest() -> _ {
    build().mount("/bienvenue", routes![index])
}
```

Résultat

```
...
Routes:
   >> (index) GET /bienvenue/salut
Fairings:
   >> Shield (liftoff, response, singleton)
Rocket has launched from http://0.0.0.0:8080
```

*Dans ce code source, vous remarquez que vous pouvez choisir le nom de votre fonction principale puisque c'est la directive **#[launch]** qui la rend principale. Enfin, au lieu d'utiliser la macro au tout début du fichier source, vous pouvez faire un simple appel classique à l'utilisation du paquetage **rocket**. Il est également possible de proposer des alternatives d'**URL** pour une même route, comme suit :*

main.rs

```
use rocket::*;

#[get("/salut")]
fn index() -> &'static str {
    "Bienvenue à tout le monde !"
}
```

```rust
#[launch]
fn service_web_rest() -> _ {
    build()
      .mount("/bienvenue", routes![index])
      .mount("/bonjour", routes![index])
}
```

Résultat

```
...
Routes:
  >> (index) GET /bonjour/salut
  >> (index) GET /bienvenue/salut
Fairings:
  >> Shield (liftoff, response, singleton)
Rocket has launched from http://0.0.0.0:8080
```

Premier web service de conversion monétaire – utilisation de la méthode GET HTTP

Après cette phase de test, je vous propose maintenant de réaliser notre premier **service web** « utile »**.** Ce service doit calculer la conversion monétaire entre les **euros** et les **francs**. Nous proposons deux alternatives, soit calculer des **francs** à partir d'une valeur en **€uro**, soit l'inverse. À titre d'exemple, afin de bien montrer la souplesse de la syntaxe, je vous propose de faire en sorte de choisir le nombre de chiffres après la virgule pour le résultat.

*La particularité de ce projet, c'est que cette fois-ci, nous devons proposer une valeur numérique à soumettre dans nos **URL**, sachant que par défaut, le protocole **HTTP** ne fonctionne qu'avec une communication intrinsèque que sous forme de texte. Il s'agit donc de pouvoir transformer un texte en valeur numérique équivalente.*

*Le deuxième constat, c'est que nos **URL** sont **variables** ou paramétrables, puisque nous pouvons soumettre les conversions avec, bien entendu, des valeurs monétaires quelconques, et proposer ainsi plusieurs calculs consécutifs avec des valeurs différentes.*

*Pour mettre en œuvre une **URL paramétrable**, nous devons spécifier nos paramètres en donnant à chacun un nom bien précis et le placer entre les opérateurs **<** et **>**. Ce nom doit alors être un paramètre de la fonction associée à la méthode **HTTP** avec **l'URL** paramétrable. Vous choisissez le bon type de ce paramètre, la conversion se fera alors automatiquement.*

*La réponse à cette fonction peut être de type chaînes de caractères puisque le protocole **HTTP** propose souvent ce type de réponse par défaut. Nous verrons ultérieurement comment envoyer d'autres types d'information.*

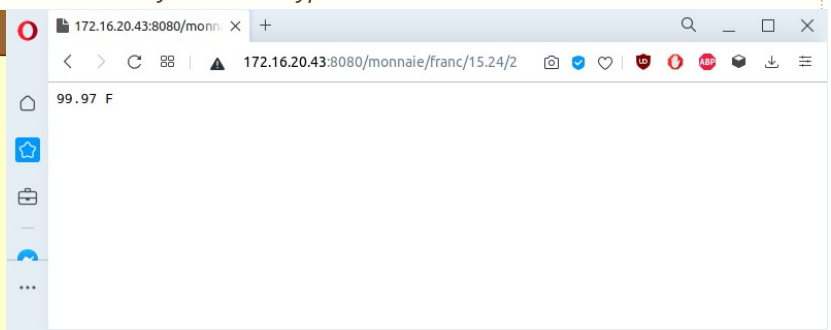
main.rs

```rust
use rocket::*;

#[get("/franc/<euro>/<chiffres>")]
fn franc(euro: f32, chiffres: usize) -> String {
    const TAUX: f32 = 6.55957;
    format!("{:.*} F", chiffres, euro * TAUX)
}

#[get("/euro/<franc>/<chiffres>")]
fn euro(franc: f32, chiffres: usize) -> String {
    const TAUX: f32 = 6.55957;
    format!("{:.*} €", chiffres, franc / TAUX)
}

#[launch]
fn monnaie() -> _ {
    build().mount("/monnaie", routes![euro, franc])
}
```



Résultat

```
...
Routes:
  >> (euro) GET /monnaie/euro/<franc>/<chiffres>
  >> (franc) GET /monnaie/franc/<euro>/<chiffres>
Fairings:
  >> Shield (liftoff, response, singleton)
Rocket has launched from http://0.0.0.0:8080
```

Le premier constat que nous pouvons faire, c'est que notre code source est encore une fois extrêmement concis. Ici, nous proposons deux possibilités de conversion, en spécifiant les deux routes possibles pour chacune des fonctions qui résout le calcul monétaire adéquat.

*Il n'est pas toujours facile de se souvenir de la syntaxe exacte à utiliser dans **l'URL**. Il est souvent plus judicieux de pouvoir spécifier le nom des paramètres à soumettre avec leurs valeurs respectives directement dans **l'URL**, avec une syntaxe que nous connaissons bien, de type **http://site:8080/service/question?param1=valeur1&param2=valeur2**.*

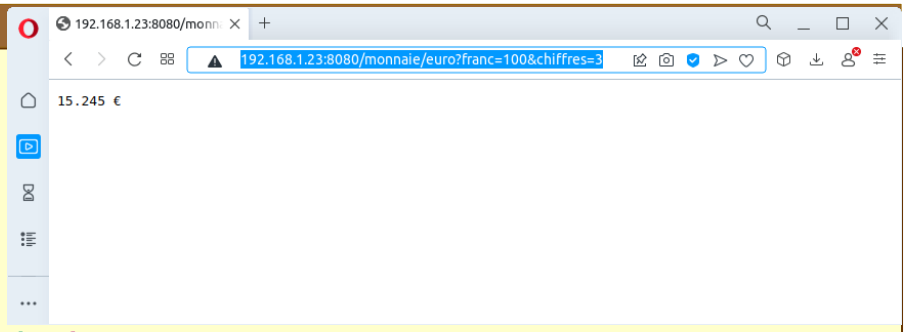
main.rs

```rust
use rocket::*;

#[get("/franc?<euro>&<chiffres>")]
fn franc(euro: f32, chiffres: usize) -> String {
    const TAUX: f32 = 6.55957;
    format!("{:.*} F", chiffres, euro * TAUX)
}

#[get("/euro?<franc>&<chiffres>")]
fn euro(franc: f32, chiffres: usize) -> String {
    const TAUX: f32 = 6.55957;
    format!("{:.*} €", chiffres, franc / TAUX)
}

#[launch]
fn monnaie() -> _ {
    build().mount("/monnaie", routes![euro, franc])
}
```



Résultat

```
...
Routes:
  >> (euro) GET /monnaie/euro?<franc>&<chiffres>
  >> (franc) GET /monnaie/franc?<euro>&<chiffres>
Fairings:
  >> Shield (liftoff, response, singleton)
Rocket has launched from http://0.0.0.0:8080
```

Plusieurs services Web - transferts de fichiers images

Toujours au travers des méthodes **GET HTTP**, nous allons rajouter un nouveau services Web qui permet de récupérer des photos dans un répertoire dédié dans le serveur et de connaître également toutes les photos disponibles.

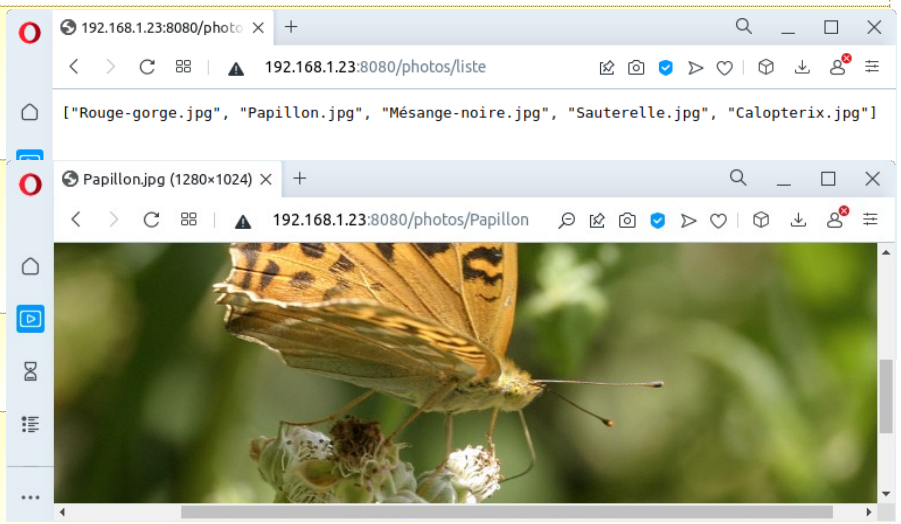
*Grâce aux requêtes paramétrées, nous pouvons récupérer n'importe quelle information, le tout étant de savoir quel est le type rattaché à ce paramètre. Dans les exemples précédents, nous avions pris des paramètres de type réel. Ici, nous pouvons décider que le paramètre correspond à un nom de fichier (type **PathBuf**).*

*La librairie **Rocket** possède une structure **NamedFile** représentant un fichier, que nous pouvons utiliser pour envoyer le flux d'octets associés en réponse à la requête **HTTP**.*

*Comme le flux d'octets peut prendre du temps suivant la taille du fichier, l'idéal est de proposer une fonction asynchrone (**async fn**) associé à la requête **HTTP GET**.*

Grâce à cette technique asynchrone, le flux d'octets peut être envoyé tout en étant opérationnel pour d'autres requêtes.

*Si le fichier est une image, le client voit la photo apparaître progressivement sans qu'il faille attendre le transfert complet des octets. Pour que tout ce système fonctionne parfaitement et en étant en adéquation avec la fonction asynchrone, vous devez toujours la placer en attente d'interaction grâce à l'opérateur **await**.*



main.rs

```rust
#[macro_use] use rocket::*;
use std::path::{PathBuf, Path};
use rocket::fs::NamedFile;

#[get("/franc?<euro>&<chiffres>")]
fn franc(euro: f32, chiffres: usize) -> String {
    const TAUX: f32 = 6.55957;
    format!("{:.*} €", chiffres, euro * TAUX)
}

#[get("/euro?<franc>&<chiffres>")]
fn euro(franc: f32, chiffres: usize) -> String {
```

```
    const TAUX: f32 = 6.55957;
    format!("{:.*} €", chiffres, franc / TAUX)
}

#[get("/<fichier>")]
async fn photo(fichier: PathBuf) -> Option<NamedFile> {
    NamedFile::open(Path::new("./Images/").join(fichier)).await.ok()
}

#[get("/liste")]
fn liste() -> String {
    let mut photos = Vec::new();
    if let Ok(fichiers) = Path::new("./Images/").read_dir() {
        for fichier in fichiers {
            let nom = fichier.unwrap().file_name().to_string_lossy().to_string();
            photos.push(nom);
        }
    }
    format!("{:?}", photos)
}

#[launch]
fn principal() -> _ {
    build()
      .mount("/monnaie", routes![euro, franc])
      .mount("/photos", routes![photo, liste])
}
```

Résultat

```
...
Routes:
  >> (liste) GET /photos/liste
  >> (photo) GET /photos/<fichier>
  >> (euro) GET /monnaie/euro?<franc>&<chiffres>
  >> (franc) GET /monnaie/franc?<euro>&<chiffres>
Fairings:
  >> Shield (liftoff, response, singleton)
Rocket has launched from http://0.0.0.0:8080
```

Web service qui exploite toutes les méthodes de gestion du HTTP – Stockage de photos

Le plus gros intérêt des **web services** de type **REST**, c'est de transmettre les requêtes et les réponses tout simplement à l'aide du **protocole HTTP**, ce qui nous permet de communiquer par Internet, puisque le pare-feu n'a pas besoin d'une configuration particulière et nous protège en laissant passer le port **80** ou **8080**.

> *Le projet que nous allons mettre en œuvre maintenant consiste à poursuivre l'étude précédente en rajoutant les autres méthodes **HTTP** enfin de gérer complètement l'archivage de photos pour les récupérer ultérieurement en temps voulu.*

Je rappelle qu'il s'agit ici de services web de type REST, c'est-à-dire des services qui permettent de gérer des ressources à distance, avec donc les méthodes principales du protocole HTTP de gestion de ces ressources :

GET : cette méthode permet de récupérer une ressource depuis le serveur pour l'avoir sur le poste local.

POST : cette méthode est l'inverse de la précédente, cette fois-ci nous envoyons une ressource depuis le poste local. Elle permet ainsi de sauvegarder à distance une ressource que nous possédons sur notre ordinateur.

PUT : cette méthode permet de modifier une ressource distante.

DELETE : cette méthode permet, comme son nom l'indique de supprimer une ressource du serveur.

main.rs

```
use rocket::*;
use rocket::fs::{NamedFile, TempFile};
use std::path::{PathBuf, Path};
use std::fs::{remove_file, rename};
use std::io::Result;

#[get("/lire/<fichier>")]
async fn photo(fichier: PathBuf) -> Option<NamedFile> {
    NamedFile::open(Path::new("./Images/").join(fichier)).await.ok()
}

#[get("/liste")]
fn liste() -> String {
    let mut photos = Vec::new();
    if let Ok(fichiers) = Path::new("./Images/").read_dir() {
        for fichier in fichiers {
            let nom = fichier.unwrap().file_name().to_string_lossy().to_string();
            photos.push(nom);
        }
```

```
    }
    format!("{:?}\n", photos)
}

#[post("/ajout/<fichier>", data = "<octets>")]
async fn ajouter(fichier: PathBuf, mut octets: TempFile<'_>) -> Result<()> {
    octets.persist_to(Path::new("./Images/").join(fichier)).await
}

#[delete("/supprime/<fichier>")]
fn supprimer(fichier: PathBuf) -> Result<()> {
    remove_file(Path::new("./Images/").join(fichier))
}

#[put("/change/<ancien>/<nouveau>")]
fn changer(ancien: PathBuf, nouveau: PathBuf) -> Result<()> {
    let avant = Path::new("./Images/").join(ancien);
    let apres = Path::new("./Images/").join(nouveau);
    rename(avant, apres)
}

#[launch]
fn photos() -> _ {
    build().mount("/photos", routes![photo, liste, ajouter, supprimer, changer])
}
```

Résultat

```
...
Routes:
  >> (liste) GET /photos/liste
  >> (photo) GET /photos/lire/<fichier>
  >> (ajouter) POST /photos/ajout/<fichier>
  >> (supprimer) DELETE /photos/supprime/<fichier>
  >> (changer) PUT /photos/change/<ancien>/<nouveau>
Fairings:
  >> Shield (liftoff, response, singleton)
Rocket has launched from http://0.0.0.0:8080
```

*La première remarque que nous pouvons faire, c'est que ce code est encore une fois extrêmement concis. Très peu de lignes sont utilisées pour exprimer les quatre fonctions principales du protocole **HTTP**. Nous retrouvons les deux fonctions **liste()** et **photo()** que nous avons implémenté dans la chapitre précédent.*

*Nous trouvons ensuite la fonction **ajouter()** qui réalise l'opération inverse de **photo()** et qui permet d'archiver une nouvelle image dans le répertoire « **Images** » du serveur. Pour stocker une nouvelle ressource, nous devons donc passer par la méthode **HTTP POST**.*

*Pour envoyer la ressource, vous devez préciser le nom de votre paramètre à prendre en compte au niveau de la déclaration **#[post()]** dans la rubrique **data**, ici il s'agit du paramètre **octets** : **#[post(« /ajout/<fichier>, data=<octets>)]**.*

*La fonction **ajouter()** prend deux paramètres. Le premier spécifie le nom du fichier à sauvegarder. Le deuxième permet de récupérer l'ensemble complet des octets du fichier image, grâce à une structure spécialement définie pour cela dans **Rocket** qui se nomme **TempFile<_>**.*

*Cette structure possède la méthode **persist_to()** qui réalise l'enregistrement sur le serveur de la photo envoyée, dans le bon répertoire, en précisant juste le nom du fichier récupéré dans la requête. La fonction **ajouter()** est définie **asynchrone** puisque le temps de transfert des octets peut être relativement long.*

*Comme son nom l'indique, la fonction **supprimer()** permet de supprimer définitivement une photo dans le répertoire de stockage du serveur. Vous devez juste spécifier le nom du fichier image dans **l'URL** de votre requête. La fonction suivante **changer()** associée à la méthode **PUT HTTP**, permet uniquement de changer le nom du fichier image stocké dans le répertoire du serveur, en précisant dans l'**URL** l'ancien nom suivi du nouveau nom à prendre en compte.*

*Pour ces trois fonctions, vous remarquez le retour systématique de type **Result<()>** qui permet de générer automatiquement une page **HTML** dans le cas où la requête n'aboutit pas. Cette page Web nous renseigne sur le mauvais fonctionnement.*

Test sur le service Web de gestion de photos

Pour vérifier le bon fonctionnement de toutes ces fonctions, le navigateur ne suffit plus puisqu'il ne s'intéresse qu'à la méthode **HTTP GET**, il récupère systématiquement une ressource qu'elle quel soit. Pour tester les autres méthodes **HTTP**, vous devez installer un plugin sur votre navigateur ou utiliser l'API **CURL** qui s'utilise en ligne de commande.

*La plupart des plugins utilisant la technologie **REST** ne permette pas « **l'upload** » de fichier. Je préfère utiliser plutôt **CURL**. Cette API dispose de beaucoup de commandes et d'options qui vont nous permettre de vérifier l'ensemble des informations renvoyées par les fonctions du Web service d'archivage de photos.*

*Je vous propose de vérifier le fonctionnement de l'ensemble des fonctionnalités de notre service Web en commençant par les méthodes **HTTP GET**. Nous en profiterons pour découvrir certaines options de commande.*

Commande CURL

```
curl -X GET http://192.168.1.23:8080/photos/liste
```

Réponse

```
["Rouge-gorge.jpg", "Papillon.jpg", "Sauterelle.jpg", "Calopterix.jpg"]
```

*À la suite de la commande **curl**, vous devez toujours préciser l'adresse **URL** complète associée à la fonction désirée. Vous spécifiez également le type de méthode **HTTP** souhaité grâce à l'option **-X**. Dans le cas de la méthode **HTTP GET**, cette option n'est pas obligatoire.*

Commande CURL

```
curl -X GET http://192.168.1.23:8080/photos/liste --head
```

Réponse

```
HTTP/1.1 200 OK
content-type: text/plain; charset=utf-8
server: Rocket
permissions-policy: interest-cohort=()
x-frame-options: SAMEORIGIN
x-content-type-options: nosniff
content-length: 72
date: Tue, 26 Oct 2021 14:42:11 GMT
```

*Lorsque nous rajoutons l'option **–head**, nous ne voulons pas le résultat en tant que tel, mais uniquement vérifier le contenu complet de l'en-tête de la réponse.*

Commande CURL

```
curl -X GET http://192.168.1.23:8080/photos/listes
```

Réponse

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="utf-8">
   <title>404 Not Found</title>
</head>
<body align="center">
   <div role="main" align="center">
      <h1>404: Not Found</h1>
      <p>The requested resource could not be found.</p>
      <hr />
   </div>
   <div role="contentinfo" align="center">
      <small>Rocket</small>
   </div>
</body>
```

*Lorsque l'URL n'est pas correcte, vous remarquez que **Rocket** génère automatiquement une page Web nous renseignant sur une mauvaise écriture de notre requête avec la fameuse erreur avec le code **404**.*

Commande CURL

```
curl -X GET http://192.168.1.23:8080/photos/listes --head
```

Réponse

```
HTTP/1.1 404 Not Found
content-type: text/html; charset=utf-8
server: Rocket
permissions-policy: interest-cohort=()
x-frame-options: SAMEORIGIN
x-content-type-options: nosniff
content-length: 383
date: Tue, 26 Oct 2021 14:50:34 GMT
```

*Nous pouvons bien entendu vérifier ce code d'erreur en demandant uniquement l'en-tête de la réponse comme ci-dessus. Vous pouvez avoir encore plus d'informations en prenant l'option habituelle **-v** (**verbose**) :*

Commande CURL

```
curl -X GET http://192.168.1.23:8080/photos/liste -v
```

Réponse

```
Note: Unnecessary use of -X or --request, GET is already inferred.
*   Trying 192.168.1.23:8080...
* TCP_NODELAY set
```

```
* Connected to 192.168.1.23 (192.168.1.23) port 8080 (#0)
> GET /photos/liste HTTP/1.1
> Host: 192.168.1.23:8080
> User-Agent: curl/7.68.0
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< content-type: text/plain; charset=utf-8
< server: Rocket
< permissions-policy: interest-cohort=()
< x-frame-options: SAMEORIGIN
< x-content-type-options: nosniff
< content-length: 72
< date: Tue, 26 Oct 2021 14:52:53 GMT
<
["Rouge-gorge.jpg", "Papillon.jpg", "Sauterelle.jpg", "Calopterix.jpg"]
* Connection #0 to host 192.168.1.23 left intact
```

*Nous voyons les deux en-têtes du protocole **HTTP**, celle de la requête et celle de la réponse. Par ailleurs, vous remarquez qu'on nous informe que l'option **-X GET** n'est pas obligatoire et que donc, nous pouvons nous en passer.*

Commande CURL

```
curl -X GET http://192.168.1.23:8080/photos/lire/Rouge-gorge.jpg -o rouge-gorge.jpg
```

Réponse

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  408k  100  408k    0     0  79.7M      0 --:--:-- --:--:-- --:--:-- 79.7M
```

*Pour récupérer une photo, vous devez spécifier le nom du fichier qui va contenir l'ensemble des octets récupérés grâce à l'option **-o**. À la suite de la commande, vous pouvez remarquer le taux de transfert de l'ensemble des octets de l'image et votre photo apparaît dans le répertoire où se situe votre commande **curl**.*

Commande CURL

```
curl -X POST http://192.168.1.23:8080/photos/ajout/chien.jpg -T chien.jpg
```

*Pour transférer une nouvelle photo, puisqu'il s'agit d'envoyer une nouvelle ressource, nous devons passer par la méthode **HTTP POST**. Vous spécifiez alors la fichier photo à soumettre grâce à l'option **-T** (**transfert**). Vous avez ci-dessous les commandes **curl** pour respectivement : renommer un fichier image sur le serveur où supprimer définitivement une photo.*

Commandes CURL

```
curl -X PUT http://192.168.1.23:8080/photos/change/papillon.jpg/Papillon.jpg
curl -X DELETE http://192.168.1.23:8080/photos/supprime/chien.jpg
```

Réponses au format JSON

J'aimerais revenir sur le projet monétaire. La réponse que nous donnions pour les deux méthodes **GET HTTP** étaient systématiquement une chaîne de caractères avec la valeur numérique du résultat suivi du symbole monétaire. Il serait peut-être plus judicieux de formater la réponse avec une structure **JSON**.

*Ce format **JSON** est très souvent utilisé lorsque nous désirons échanger plusieurs informations en même temps. C'est devenu le standard des échanges. Nous avons largement abordé ce sujet lors de l'étude précédente sur la communication réseau, et donc je n'y reviendrai pas. Souvenez-vous juste qu'il est nécessaire d'intégrer la librairie « **serde** ».*

*L'implémentation est toujours simple puisqu'il s'agit de sérialiser une structure à votre convenance en proposant tous les attributs nécessaires à votre réponse. Par contre, pour que cela fonctionne correctement dans **Rocket**, vous devez compléter votre fichier de configuration **Cargo.toml**.*

Cargo.toml

```
[package]
name = "REST-Rocket"
version = "0.1.0"
edition = "2018"

[dependencies]
rocket = { version = "0.5.0-rc.1", features = ["json"] }
serde = "1.0.130"
```

main.rs

```
use rocket::*;
use rocket::serde::{Serialize, json::Json};

#[derive(Serialize)]
struct Resultat {
```

```
    demande: String,
    franc: f32,
    euro: f32
}

#[get("/franc?<euro>")]
fn franc(euro: f32) -> Json<Resultat> {
    const TAUX: f32 = 6.55957;
    Json( Resultat {
        demande: "franc".to_string(),
        euro,
        franc: euro*TAUX
    })
}

#[get("/euro?<franc>")]
fn euro(franc: f32) -> Json<Resultat> {
    const TAUX: f32 = 6.55957;
    Json( Resultat {
        demande: "euro".to_string(),
        franc,
        euro: franc/TAUX
    })
}

#[launch]
fn monnaie() -> _ {
    build().mount("/monnaie", routes![euro, franc])
}
```

Résultat

```
...
Routes:
  >> (euro) GET /monnaie/euro ?<franc>
  >> (franc) GET /monnaie/franc ?<euro>
Fairings:
  >> Shield (liftoff, response, singleton)
Rocket has launched from http://0.0.0.0:8080
```

Commande CURL

```
curl http://192.168.1.23:8080/monnaie/euro?franc=100
```

Réponse

```
{"demande":"euro","franc":100.0,"euro":15.244902}
```

Communication complète au format JSON

Maintenant que nous maîtrisons bien le format **JSON**, je vous propose de faire en sorte que la requête soit elle aussi dans ce format là. Pour cela, nous devons passer par une méthode **HTTP** de type **POST** pour que nous puissions envoyer nos données (nos ressources) dans ce format **JSON**.

> *Dans l'en-tête d'une méthode **HTTP POST** ou **PUT**, vous devez spécifier le type de contenu correspondant aux données qui vont être transitées de telle sorte que le service Web soit capable de les déchiffrer. L'attribut associé est très connu et se nomme « **Content-Type** » et vous devez lui associer la valeur « **application/json** ».*
>
> *De son côté, le service doit être capable de reconnaître ce type de contenu, ce qui se fait simplement par l'attribut « **format** » qui doit alors être présent dans la déclaration de la fonction associée à la méthode **HTTP POST**.*
>
> *Nous allons expérimenter cette approche au travers du projet de financement que nous avons abordé lors de l'étude précédente sur la mise en œuvre de la communication réseau. Encore une fois, vous remarquerez la concision du code.*

main.rs

```
use rocket::*;
use rocket::serde::{Serialize, Deserialize, json::Json};

#[derive(Deserialize)]
struct Requete {
    capital: f32,
    annees: u16,
    taux: f32
}

#[derive(Serialize)]
struct Reponse {
    mois: u16,
    mensualites: f32,
```

```
    total: f32,
    interets: f32
}

#[post("/", format = "json", data="<requete>")]
fn calcul(requete: Json<Requete>) -> Json<Reponse> {
    let capital = requete.capital;
    let taux = requete.taux / 100.;
    let ans = requete.annees;
    let mois = ans * 12;
    let mensualites = capital*taux/12./(1.-f32::powf(1.+taux/12., -(ans as f32) * 12.));
    let total = mois as f32 * mensualites;
    let interets = total - capital;
    Json(Reponse {mois, mensualites, total, interets})
}

#[launch]
fn financement() -> _ {
    build().mount("/financement", routes![calcul])
}
```

*Vous remarquez que le paramètre **requete** qui est normalement de type **Json<Requete>** est en fait utilisé comme si le type était finalement **Requete**, ce qui permet d'avoir une utilisation très simple grâce à cet **inférence de type automatique**.*

Résultat

```
...
Routes:
  >> (calcul) POST /financement/ application/json
Fairings:
  >> Shield (liftoff, response, singleton)
Rocket has launched from http://0.0.0.0:8080
```

Commande CURL

```
curl -X POST -H 'Content-Type: application/json' -d '{"capital": 5000, "annees": 5, "taux": 3}'
http://192.168.1.23:8080/financement
```

Réponse

```
{"mois":60.0,"mensualites":89.84153,"total":5390.4917,"interets":390.4917}
```

*Dans votre commande, vous devez renseigner le type de données envoyées grâce à l'option **-H** (**Header**). Ensuite, vos données sont spécifiées à la suite de l'option **-d** (**data**). Puisque le contenu des données comporte des **guillemets**, vous devez localiser l'ensemble de vos données entre **apostrophes**.*

Capturer les erreurs dues aux mauvaises requêtes

Je vous propose de rajouter de nouvelles fonctionnalités au projet précédent. En parallèle à la méthode **HTTP POST**, je propose une autre méthode **HTTP GET** pour aboutir au même calcul, en plaçant cette fois-ci les paramètres directement dans **l'URL**, sans passer par un format **JSON**. Si les différentes requêtes sont mal formées, je propose également de capturer les erreurs pour proposer un affichage circonstancié.

*Les **services Web** que nous proposons lors de cette étude seront utilisés normalement par des applications clientes qui vont soumettre des requêtes parfaitement étudiées. Il peut malgré tout être intéressant de voir comment capturer des erreurs éventuelles puisque **Rocket** le permet.*

*L'étude n'est pas exhaustive puisque différents types d'erreur peuvent apparaître, mais cela nous permet découvrir comment implémenter simplement quelques erreurs fréquentes, notamment l'erreur de type (**404**). Il existe pour cela la déclaration spécifique **#[catch(n° erreur)]** au dessus de la fonction qui doit traiter l'erreur.*

*Pour résoudre certains manques éventuels dans votre requête, vous devez proposer un paramètre de type **Request** dans votre fonction de traitement afin de connaître l'élément manquant par exemple.*

*Pour que ces différentes fonctions de traitement d'erreur soient prises en compte, vous devez les enregistrer dans dans la fonction principale au même moment que la mise en place des différentes « **routes** » grâce à la méthode **register()**. S'il existe plusieurs alternatives, vous devez les enchaîner.*

main.rs

```
use rocket::*;
use rocket::serde::{Serialize, Deserialize, json::Json};

#[derive(Deserialize)]
struct Requete {
    capital: f32,
    annees: u16,
    taux: f32
}
```

```rust
#[derive(Serialize)]
struct Reponse {
    mois: u16,
    mensualites: f32,
    total: f32,
    interets: f32
}

#[post("/json", format = "json", data="<requete>")]
fn post(requete: Json<Requete>) -> Json<Reponse> {
    let capital = requete.capital;
    let annees = requete.annees;
    let taux = requete.taux;
    calcul(Requete{capital, annees, taux})
}

#[get("/requete?<capital>&<annees>&<taux>")]
fn get(capital: f32, annees: u16, taux: f32) -> Json<Reponse> {
    calcul(Requete{capital, annees, taux})
}

fn calcul(requete: Requete) -> Json<Reponse> {
    let capital = requete.capital;
    let taux = requete.taux / 100.;
    let ans = requete.annees;
    let mois = ans * 12;
    let mensualites = capital*taux/12./(1.-f32::powf(1.+taux/12., -(ans as f32) * 12.));
    let total = mois as f32 * mensualites;
    let interets = total - capital;
    Json(Reponse {mois, mensualites, total, interets})
}

#[catch(404)]
fn erreur_generale() -> &'static str {
    "(Erreur 404) Requête non valide : Il existe uniquement deux requêtes :\n\
    Méthode POST : http:192.168.1.23:8080/financement/json + {\"capital\": somme, \"annees\": nombre,
\"taux\": valeur}\n\
    Méthode GET : http:192.168.1.23:8080/financement/requete?capital=somme&annees=nombre&taux=valeur"
}


#[catch(404)]
fn erreur_requete(requete: &Request) -> String {
    let mut resultat = String::from(format!("Erreur (404) requête = http:192.168.1.23:8080{}\n", requete.uri()));
    match requete.query_value::<f32>("capital") {
        Some(capital) => resultat.push_str(format!("capital={}\n", capital.unwrap()).as_str()),
        None => resultat.push_str(format!("capital=?\n").as_str())
    }
    match requete.query_value::<f32>("annees") {
        Some(annees) => resultat.push_str(format!("annees={}\n", annees.unwrap()).as_str()),
        None => resultat.push_str(format!("annees=?\n").as_str())
    }
    match requete.query_value::<f32>("taux") {
        Some(taux) => resultat.push_str(format!("taux={}\n", taux.unwrap()).as_str()),
        None => resultat.push_str(format!("taux=?\n").as_str())
    }
    resultat
}

#[catch(422)]
fn erreur_json(requete: &Request) -> String {
    format!("Erreur (422) : requête = http://192.168.1.23:8080{}\n\
    Votre format JSON n'est pas correct !\n", requete.uri())
}

#[launch]
fn financement() -> _ {
    build()
      .mount("/financement", routes![get, post])
      .register("/financement", catchers![erreur_generale])
      .register("/financement/requete", catchers![erreur_requete])
      .register("/financement/json", catchers![erreur_json])
}
```

Résultat

```
...
Routes:
  >> (post) POST /financement/json application/json
```

```
   >> (get) GET /financement/requete?<capital>&<annees>&<taux>
Catchers:
   >> (erreur_generale) /financement 404
   >> (erreur_json) /financement/json 422
   >> (erreur_requete) /financement/requete 404
Fairings:
   >> Shield (liftoff, response, singleton)
Rocket has launched from http://0.0.0.0:8080
```

Commande CURL

```
curl 'http://192.168.1.23:8080/financement/requete?capital=5000&annees=5&taux=3'
```

Réponse

```
{"mois":60.0,"mensualites":89.84153,"total":5390.4917,"interets":390.4917}
```

Commande CURL

```
curl 'http://192.168.1.23:8080/financement/requete?capital=5000&annees=5'
```

Réponse

```
Erreur (404) requête = http:192.168.1.23:8080/financement/requete?capital=5000&annees=5
capital=5000
annees=5
taux=?
```

Commande CURL

```
curl 'http://192.168.1.23:8080/financement'
```

Réponse

```
Erreur 404) Requête non valide : Il existe uniquement deux requêtes :
Méthode POST : http:192.168.1.23:8080/financement/json + {"capital": somme, "annees": nombre, "taux": valeur}
Méthode GET : http:192.168.1.23:8080/financement/requete?capital=somme&annees=nombre&taux=valeur
```

Commande CURL

```
curl -X POST -H 'Content-Type: application/json' -d '{"capital": 5000, "annees": 5, "taux": 3}'
http://192.168.1.23:8080/financement/json
```

Réponse

```
{"mois":60.0,"mensualites":89.84153,"total":5390.4917,"interets":390.4917}
```

Commande CURL

```
curl -X POST -H 'Content-Type: application/json' -d '{"capital": 5000, "annees": 5}'
http://192.168.1.23:8080/financement/json
```

Réponse

```
Erreur (422) : requête = http://192.168.1.23:8080/financement/json
Votre format JSON n'est pas correct !
```

> *Vous remarquez encore une fois que l'implémentation de la capture d'erreur est relativement simple. Toutefois tous les types d'erreurs ne sont pas traités ici. Déjà beaucoup de code est utilisé que pour cet aspect du traitement. Personnellement, j'utiliserai assez rarement ces captures sachant que **Rocket** propose déjà par défaut un affichage circonstancié.*

Application cliente pour communiquer avec le service Web REST de financement

Jusqu'à présent, tous les chapitres traités se cantonnés sur les **service Web REST**. Nous allons maintenant découvrir comment réaliser une application cliente qui communique avec ces services Web. Dans ce chapitre, nous allons mettre en œuvre une application cliente qui propose des requêtes qui communique avec le service de financement du projet précédent.

> *Comme d'habitude dans **Rust**, il existe un certain nombre d'alternatives qui laisse une grande liberté de codage. Par soucis de simplicité, je préfère utiliser la librairie **isahc** qui, un peu à l'image de **Rocket**, propose toujours des solutions plus simples et concises que d'autres librairies.*

> *Encore une fois, pour intégrer cette librairie **isahc**, vous devez compléter votre fichier de configuration **Cargo.toml**. Dans beaucoup de services Web, le format **JSON** est souvent le moyen de communication par excellence. Comme beaucoup d'autres librairies, vous devez intégrer également la librairie **serde**.*

Cargo.toml

```
[package]
name = "client-http"
version = "0.1.0"
edition = "2021"
```

```
[dependencies]
isahc = { version = "1.5", features = ["json"] }
serde = { version = "1.0.130", features = ["derive"] }
serde_json = "1.0.68"
openssl = { version = "0.10", features = ["vendored"] }
```

main.rs

```
use isahc::{prelude::*, get, Request};
use serde::{Serialize, Deserialize};

#[derive(Debug, Serialize)]
struct Requete {
    capital: f32,
    annees: u16,
    taux: f32
}

#[derive(Debug, Deserialize)]
struct Reponse {
    mois: u16,
    mensualites: f32,
    total: f32,
    interets: f32
}

fn methode_get(capital: f32, annees: u16, taux: f32) -> Reponse {
  let uri = format!("http://192.168.1.23:8080/financement/requete?capital={}&annees={}&taux={}",
      capital, annees, taux);
  get(uri).unwrap().json::<Reponse>().unwrap()
}

fn methode_post(capital: f32, annees: u16, taux: f32) -> Reponse {
  let uri = "http://192.168.1.23:8080/financement/json";
  let donnees = Requete { capital, annees, taux };
  Request::post(uri)
    .header("Content-Type", "application/json")
    .body(serde_json::to_string(&donnees).unwrap())
    .unwrap().send().unwrap().json::<Reponse>().unwrap()
}

fn methode_post_chaine() -> Reponse {
  let uri = "http://192.168.1.23:8080/financement/json";
  Request::post(uri)
    .header("Content-Type", "application/json")
    .body(r#"{"capital": 5000, "annees": 5, "taux": 3}"#)
    .unwrap().send().unwrap().json::<Reponse>().unwrap()
}

fn main() {
  println!("{:?}", methode_get(5000., 5, 3.));
  println!("{:?}", methode_post(20000., 4, 1.5));
  println!("{:?}", methode_post_chaine());
}
```

Résultat

```
Reponse { mois: 60, mensualites: 89.84153, total: 5390.4917, interets: 390.4917 }
Reponse { mois: 48, mensualites: 429.54227, total: 20618.03, interets: 618.0293 }
Reponse { mois: 60, mensualites: 89.84153, total: 5390.4917, interets: 390.4917 }
```

*Cette bibliothèque **isahc** propose toutes les fonctions **get()**, **post()**, **put()** et **delete()** relatives aux différentes méthodes classiques du **HTTP**. En parallèle avec ces fonctions, la librairie dispose également de structures comme **Request** et **Response** qui vont nous permettre de faire des réglages plus précis, en proposant par exemple des en-têtes spécifiques grâce à la méthode **header()**.*

*Bien sûr, la structure **Request** possède également toutes les méthodes du protocole **HTTP**, savoir **get()**, **post()**, **put()** et **delete()**. Dans les fonctions, les requêtes sont automatiquement soumises au serveur. Dans le cas où vous passer par la structure **Request**, vous devez envoyer votre requête en exécutant explicitement la méthode **send()**.*

*Enfin, pour manipuler le format **JSON**, vous pouvez utiliser les compétences de la librairie **serde** en sérialisant et désérialisant les structures correspondantes. Pour récupérer une information dans ce format là, il existe la méthode **json()**. Vous devez juste préciser le type voulu pour réaliser la transcription automatique.*

Application cliente pour communiquer avec le service Web REST d'archivage de photos

Je propose de clôturer notre étude avec une autre application cliente qui va utiliser l'ensemble des méthodes **HTTP** et nous profiterons de l'occasion pour voir comment transférer des fichiers. Nous nous servons pour cela du service Web d'archivage de photos. Je propose de changer le service pour qu'il soit en parfaite adéquation avec notre application cliente.

*L'objectif **côté serveur** est de vérifier que le nom du fichier proposé dans **l'URL** correspond bien à une photo existante dans le répertoire du serveur. Les méthodes **supprimer()** et **changer()** renvoient une chaîne de caractères donnant le résultat effectif de l'opération.*

***Côté client**, nous développons des fonctions correspondantes à chacune des méthodes **HTTP**. Pour chacune de ces fonctions, nous vérifions systématiquement si l'opération s'est bien déroulée avec un message circonstancié.*

main.rs (source modifié côté service web REST)

```
use rocket::*;
use rocket::fs::{NamedFile, TempFile};
use std::path::{PathBuf, Path};
use std::fs::{remove_file, rename};
use std::io::Result;

#[get("/lire/<fichier>")]
async fn photo(fichier: PathBuf) -> Option<NamedFile> {
   NamedFile::open(Path::new("./Images/").join(fichier)).await.ok()
}

#[get("/liste")]
fn liste() -> String {
   let mut photos = Vec::new();
   if let Ok(fichiers) = Path::new("./Images/").read_dir() {
      for fichier in fichiers {
         let nom = fichier.unwrap().file_name().to_string_lossy().to_string();
         photos.push(nom);
      }
   }
   format!("{:?}\n", photos)
}

#[post("/ajout/<fichier>", data = "<octets>")]
async fn ajouter(fichier: PathBuf, mut octets: TempFile<'_>) -> Result<()> {
   octets.persist_to(Path::new("./Images/").join(fichier)).await
}

#[delete("/supprime/<fichier>")]
fn supprimer(fichier: PathBuf) -> String {
   match remove_file(Path::new("./Images/").join(&fichier)) {
      Ok(_) => format!("'{}' définitivement supprimée", fichier.to_string_lossy()),
      Err(_) => format!("'{}' n'existe pas dans le serveur", fichier.to_string_lossy())
   }
}

#[put("/change/<ancien>/<nouveau>")]
fn changer(ancien: PathBuf, nouveau: PathBuf) -> String {
   let avant = Path::new("./Images/").join(&ancien);
   let apres = Path::new("./Images/").join(nouveau);
   match rename(avant, apres) {
      Ok(_) => "Changement effectué".to_string(),
      Err(_) => format!("'{}' n'existe pas dans le serveur", ancien.to_string_lossy())
   }
}

#[launch]
fn photos() -> _ {
   build().mount("/photos", routes![photo, liste, ajouter, supprimer, changer])
}
```

main.rs (côté client)

```
use isahc::{prelude::*, get, delete, post, Request};
use std::io::{Read, stdin, stdout, Write};
use std::fs::File;

fn liste() {
  match get("http://192.168.1.23:8080/photos/liste") {
    Ok(mut reponse) => println!("Photos : {}", reponse.text().unwrap()),
    Err(_) => println!("ATTENTION ! Service non démarré...")
  }
}

fn recuperer() {
  let photo = saisie("Nom du fichier image à récupérer");
  let uri = format!("http://192.168.1.23:8080/photos/lire/{}", &photo);
  let reponse = get(uri).unwrap();
  let mut octets = Vec::new();
  reponse.into_body().read_to_end(&mut octets).unwrap();
```

```
  if octets.len() > 1000 {
    let mut fichier = File::create(format!("/home/manu/Images/{}", &photo)).unwrap();
    fichier.write_all(octets.as_slice()).unwrap();
    println!("La photo '{}' est récupérée dans le répertoire local", photo)
  }
  else {  println!("La photo '{}' est inexistante dans le serveur distant !", photo)  }
}

fn ajouter() {
  let photo = saisie("Nom du fichier image à envoyer");
  match  File::open(format!("/home/manu/Images/{}", &photo)) {
    Ok(fichier) => {
      let uri = format!("http://192.168.1.23:8080/photos/ajout/{}", &photo);
      let _reponse = post(uri, fichier).unwrap();
      println!("Transfert de la photo '{}' effectué", photo)
    }
    Err(_) => println!("La photo '{}' est inexistante dans le répertoire local !", photo)
  }
}

fn changer() {
  let ancien = saisie("Ancien nom");
  let nouveau = saisie("Nouveau nom");
  let uri = format!("http://192.168.1.23:8080/photos/change/{}/{}", ancien, nouveau);
  let mut reponse = Request::put(uri).body(()).unwrap().send().unwrap();
  println!("{}", reponse.text().unwrap());
}

fn supprimer() {
  let photo = saisie("Nom du fichier image à supprimer");
  let uri = format!("http://192.168.1.23:8080/photos/supprime/{}", photo);
  let mut reponse = delete(uri).unwrap();
  println!("{}", reponse.text().unwrap());
}

fn saisie(message: &str) -> String {
  let mut saisie = String::new();
  print!("{} : ", message);
  stdout().flush().unwrap();
  stdin().read_line(&mut saisie).unwrap();
  saisie.trim().to_string()
}

fn saisie_choix() -> u16 { saisie("Votre choix").parse().unwrap() }

fn menu() {
  println!("Récupérer une photo ..... 1");
  println!("Ajouter une photo ....... 2");
  println!("Renommer une photo ...... 3");
  println!("Supprimer une photo ..... 4");
  println!("Quitter le programme .... 0");
}

fn main() {
  println!("Gestion de photos à distance");
  loop {
    liste();
    menu();
    match saisie_choix() {
      1 => recuperer(),
      2 => ajouter(),
      3 => changer(),
      4 => supprimer(),
      0 | _ => break
    } }
}
```