**Ressource R605**
**Developpement Avancé**
**API REST**

**OBJECTIFS :**

- ✓ Développer une application REST – inspiré du blog **https://blog.miguelgrinberg.com**

## 1 RAPPELS : API REST (REPRESENTATIONAL STATE TRANSFER)

Un « **API** » – « **Application Programming Interface** » – correspond aux moyens pour mettre à disposition d'autres applications les données ou les fonctionnalités d'une application.

Chaque action est accessible via une URI spécifique qu'on appelle « **endpoint** ». Les informations échangées sont présentées dans un format structuré, généralement en « **JSON** » – « **JavaScript Object Notation** ». Hormis pour les APIs publics, il est généralement nécessaire de s'authentifier via une clé – « **API Key** » – pour obtenir l'accès aux données et aux fonctionnalités de l'API.

Une architecture « **REST** » – « **REpresentational State Transfer** » – définit des règles de communication entre applications. Une API compatible « **REST** », ou « **RESTful** », est une interface de programmation d'application utilisant les requêtes « **HTTP** » – « **HyperText Transfer Protocol** » – définissant une interface d'accès aux données « **CRUD** » (**Create**, **Read**, **Update**, **Delete**).

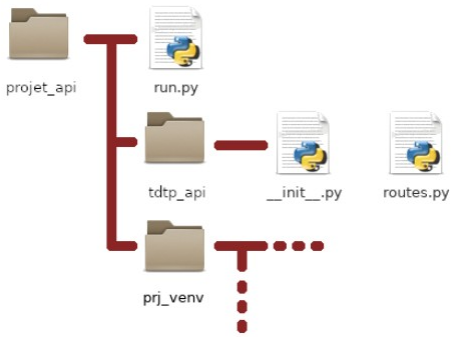
- ✓ *Create* → ***POST***
- ✓ *Read* → ***GET***
- ✓ *Update* → ***PUT***
- ✓ *Delete* → ***DELETE***

## 2 LE MICROFRAMEWORK PYTHON FLASK

Une application « **Flask** » est une application 100% « **WSGI** » – « **Web Server Gateway Interface** ». Elle repose sur un système d'organisation simple ou l'application reçoit directement du serveur WSGI les requêtes. Elle effectue un « **routage d'URL** » pour exécuter la fonction ou la classe chargée de retourner la réponse. Le site relatif au projet de développement de « **flask** » est disponible à l'adresse https://flask.palletsprojects.com/en/1.1.x/ .

### ***Arborescence de l'application « Flask »***

Lors de la création d'un projet « **Flask** », il faut créer un dossier à l'intérieur duquel nous allons enregistrer l'ensemble des fichiers qui constituent l'application.




*« fig1 – arborescence_projet_Flask »*

*Le dossier « projet_api » :* dossier d'accueil de l'ensemble du projet à créer préalablement.

*Le fichier « ./run.py » :* module contenant le programme d'exécution de l'application web « **Flask** » appelé depuis l'interpréteur python en mode développement.

*Le dossier « ./tdtp_api » :* Dossier représentant le paquet python de l'application.

*Le fichier « ./tdtp_api/__init__.py » :* définit le dossier « **tdtp_api** » comme un paquet python par sa seule présence à sa racine. Il initialise l'application et ses différents composants.

*Le fichier « ./tdtp_api/routes.py » :* contrôle le fonctionnement de l'application en routant les « **URIs** » – **Uniform Resource Identifiers** – vers les fonctions des vues qui doivent renvoyer la réponse.

***Exercice 1*** : *Vous devez définir un espace de travail pour collecter tous les éléments nécessaires à l'application.*

*Créez le dossier « **projet_api** » qui va accueillir votre application web. Ouvrez alors un terminal qui pointe à l'intérieur.*

### *Ajouter un environnement virtuel au projet*

On utilise alors un environnement virtuel pour cloisonner à l'intérieur toutes modifications de la base des bibliothèques nécessaire à votre application.

***Exercice 2*** : *Dans un terminal pointant à l'intérieur du dossier « **projet_api** », créez un environnement virtuel que vous nommerez « **prj_venv** » en utilisant l'outil « **venv** » de python conformément aux instructions ci-dessous.*

### *Installer les paquets dans l'environnement virtuel du projet*

On peut installer les paquets de l'application à l'aide de la commande « **pip** ». Le paquet « **flask** » est le paquet principal pour créer l'instance de l'application.

***Exercice 3*** : *Dans un terminal pointant à l'intérieur du dossier « **projet_api** », démarrez l'environnement virtuel et installez le paquet « **flask** ».*

## 3 CREER L'APPLICATION « FLASK »

### *Préparer le paquet contenant l'application*

***Exercice 4*** : *Dans un terminal pointant à l'intérieur du dossier « **projet_api** », créez le dossier « **tdtp_api** » qui correspond au paquet représentant l'application. Placez-vous dans ce dossier et créez le fichier « **__init__.py** ». Cela suffit pour initialiser le dossier comme un paquet python.*

*Créez également le fichier « **routes.py** » qui représentant le module au cœur de l'organisation fonctionnelle de l'application. Il réalise le routage d'URI vers la fonction relative à la requête reçue.*

Le fichier « **__init__.py** » doit initialiser l'application et faire le lien vers le module « **routes.py** »

**Exercice 5** : Implémentez le contenu du fichier « **__init__.py** » en important la classe « **Flask** » du paquet « **flask** » et instanciez-la en initialisant la variable « **app** » en utilisant l'argument « **__name__** » pour lier l'instance au paquet accéder à ses ressources. La dernière instruction importe le module « **routes.py** »

Le fichier « **routes.py** » déclare les fonctions de vue et leur associe une route en prenant en compte le type de requête HTTP utilisée

***Exercice 6*** : *Implémentez le contenu du module « **routes.py** » en important l'application « **app** » créée dans le paquet « **tdtp_api** » par le module d'initialisation « **__init__.py** » conformément au code ci-après.*

```
from projet_api import app
```

## *Créer le module d'exécution en ligne de commande de l'application*

Permet de tester notre application lors de la phase de développement en l'exécutant directement avec l'interpréteur python de l'environnement virtuel.

***Exercice 7*** : *Dans un terminal pointant à l'intérieur du dossier « **projet_api** », créez le fichier « **run.py** » pour obtenir l'arborescence ci-dessous*

*Implémentez le en important l'instance de l'application initialisée dans le paquet « **tdtp_api** ». Ajoutez l'instruction de lancement de l'application en lui passant en argument l'expression « **debug=True** » pour charger le mode débogage et ne pas la réinitialiser à chaque modification de code.*

Le contenu du fichier run.py est le suivant:

```
from tdtp_api import app
if __name__ == "__main__":
  app.run(debug=True)
```

***Exercice 8*** : *Dans un terminal pointant à l'intérieur du dossier « **projet_api** », vérifiez que l'environnement virtuel que vous avez préparé est activé, si ce n'est pas le cas, activez-le comme vu précédemment.*

*Lancez l'application à partir de l'interpréteur python: "**python run.py**"*

*Le serveur doit démarrer normalement mais si vous utiliser un navigateur pour le consulter, vous obtenez une erreur 404 – « Not Found »*

## *Transformer l'application pour la rendre RESTful*

L'application à construire permet de connaitre le statut d'un module dans le cadre du Contrôle en Cours de Formation, « **CCF** », du DUT réseaux et télécommunications. Chaque module est représenté par un **identifiant unique,** une **référence**, une **description** et un **coefficient**.

Le stockage de données le plus simple consiste à mettre en œuvre liste python de dictionnaires représentant respectivement chaque module enregistré.

*Cette technique ne fonctionnera que lorsque le serveur Web qui exécute l'application est à un seul processus et à un seul thread. Cela convient pour le serveur Web de développement de Flask.*
*Elle n'est pas envisageable sur un serveur Web de production, une configuration de base de données appropriée doit être utilisée dans ce cas. Nous étudierons cela plus tard.*

***Exercice 9*** : *Modifiez le fichier « **__init__.py** » en déclarant une liste que vous nommerez « **modules** » contenant quatre dictionnaires.*

*Chaque dictionnaire représente un module spécifique comme défini dans le tableau ci-dessous.*

| | | | |
|---|---|---|---|
| 1 | M1101 | Initiation aux réseaux d'entreprise | 3 |
| 2 | M1102 | Initiation à la téléphonie d'entreprise | 2 |
| 3 | M1103 | Architecture des équipements informatiques | 1.5 |
| 4 | M1104 | Principes et architectures des réseaux | 2 |

*« tab1 – table des modules du DUT RT »*

Pour se conformer à l'architecture « **REST** », notre application doit répondre aux requêtes POST, GET, UPDATE et DELETE du protocole HTTP afin de répondre aux objectif d'une interface CRUD.

*Le routage d'URI de Flask permet de lier une méthode HTTP à une résolution de route. A l'invocation du décorateur de routage de la vue, en plus « **endpoint** » – premier argument représentant la route – il faut préciser en second argument nommé « method » la liste des requêtes acceptés*

*« **@app.route("endpoint", method=['POST', 'GET', 'UPDATE', 'DELETE'])** » – ne lister que les requêtes HTTP souhaitées.*

Par soucis de clarté, nous utiliserons des URI commençant par « **/api/modules/** ». La réponse renvoyée par la fonction ne doit pas être du type « **text/html** » mais un objet « **application/json** » pour répondre aux exigences d'une API REST.

La fonction « **jsonify()** » du paquet « **flask** » permet de générer un **objet JSON** à partir d'un dictionnaire Python passé en argument.

*Python met à disposition par défaut la bibliothèque « **json** » et sa fonction « **json.dumps()** » pour sérialiser une donnée au format JSON.*

*En fait, la fonction « **flask.jsonify()** » utilise la fonction « **json.dumps()** », mais en plus, elle adapte l'entête HTTP de la réponse en initialisant la propriété « **Content-Type** » avec la valeur « **application/json** » alors que « json.dumps » retournerait la valeur « **text/html** ».*

*Il est donc préférable d'utiliser « **flask.jsonify()** » pour les échanges « **HTTP REST** ». Vous devez importer cette fonction depuis le module « **flask** ».*

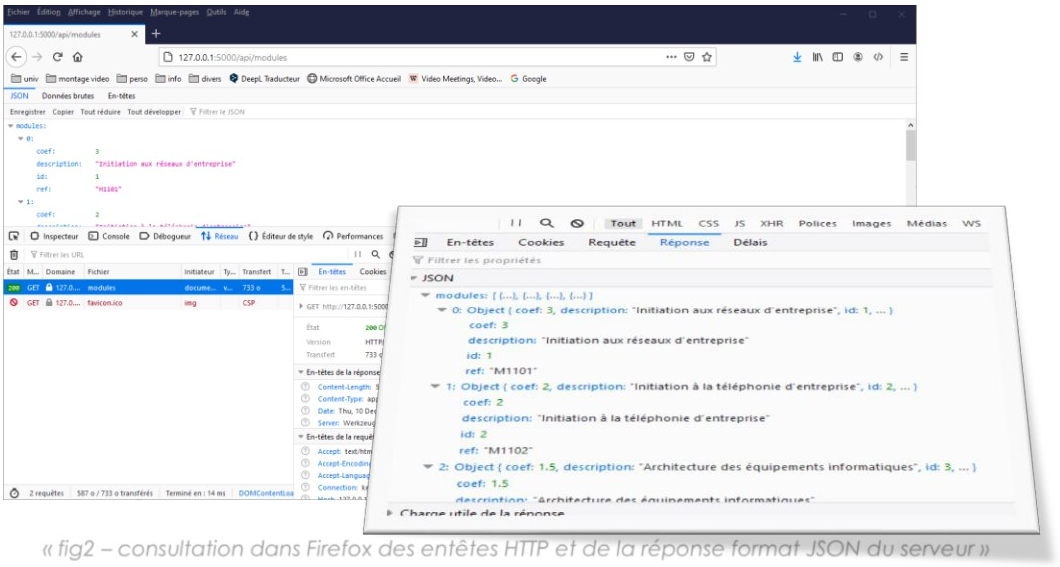
***Exercice 10** : Modifiez le fichier « **routes.py** » pour importer la liste « **modules** » déclarés à l'initialisation du paquet « **tdtp_api** » et la fonction « **jsonify** » depuis le module « **flask** ».*

*Supprimez la fonction « **index()** » et ajoutez la fonction « **get_modules()** » qui sera ()appelée pour une méthode **GET** sur l'URI « **/api/ modules** ».*

*La valeur retournée est un objet JSON construit à partir de la liste « **modules** » conformément aux instructions ci-après.*

```
@app.route("/api/modules/", methods=['GET'])
def get_modules():
    return jsonify({'modules': modules})
```

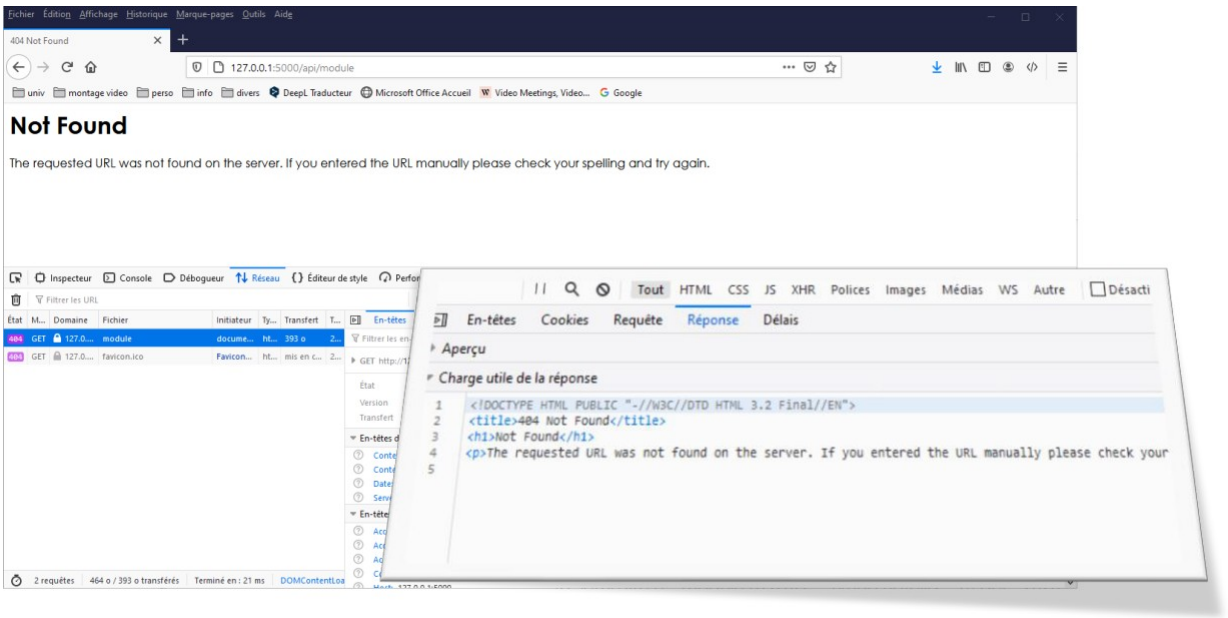
***Exercice 11** : Testez vos modifications à l'aide d'un navigateur et observez le résultat obtenu. Vous utiliserez pour cela l'URI http://127.0.0.1:5000/api/modules, associée au « **endpoint** » de **l'API** servant à renvoyer au format JSON l'ensemble des données représentant les modules enregistrées. Utilisez l'inspecteur de votre navigateur pour analyser la réponse obtenue et les entêtes « **HTTP** » de la requête et de la réponse.*



« fig2 – consultation dans Firefox des entêtes HTTP et de la réponse format JSON du serveur »

***Exercice 12*** *: Testez à présent votre API pour provoquer une erreur en omettant par exemple la lettre «* ***s*** *» à la fin dans l'URI. Observez le résultat avec l'URI http://127.0.0.1:5000/api/module.*

*Utilisez l'inspecteur de votre navigateur pour analyser la réponse obtenue et les entêtes «* ***HTTP*** *» de la requête et de la réponse.*




« fig3 – consultation dans Firefox des entêtes HTTP et la réponse HTML en cas d'erreur »

Le résultat obtenu est une page **erreur 404** – « **Not Found** ». Si l'on veut se conformer aux normes de notre « **API REST** », il faut retourner l'erreur au format JSON. Pour cela, il faut capturer l'erreur associée au code « HTTP » correspondant à l'aide du décorateur « **@app.errorhandler(404)** ».

La fonction à activer doit retourner une réponse HTTP conforme encapsulant un message JSON caractérisant l'erreur. La fonction « **make_response()** » de « **flask** » permet de générer une telle réponse en lui fournissant en premier argument le message. Elle peut prendre en second argument le code statut de la réponse. Voir dans la documentation de **flask**

https://flask.palletsprojects.com/en/1.1.x/api/?highlight=errorhandler#flask.make_response

***Exercice 13*** *: Dans le fichier « **routes.py** », ajoutez la fonction « **custom_error()** » qui sera appelée en cas de capture d'une erreur 404.*

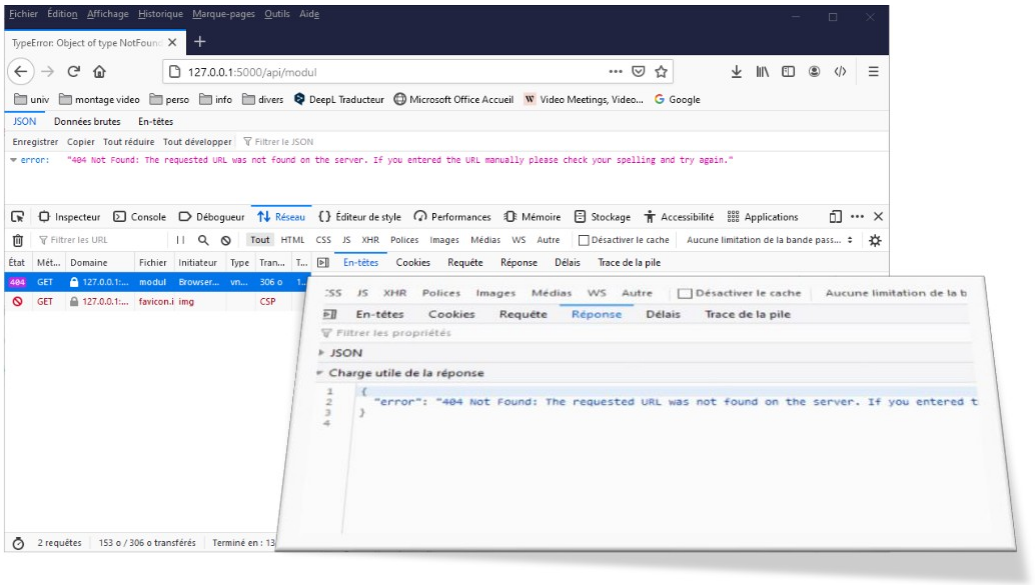
*La valeur retournée est construite à l'aide de la méthode « **make_response()** » conformément au code ci-après à partir d'un objet JSON et du code erreur HTTP.*

```
@app.errorhandler(404)
def custom_error(error):
    return make_response(jsonify({'error': str(error)}), 404)
```

***Exercice 14*** *: Testez à présent votre API et provoquez une erreur comme lors de l'exercice 12.*

*Observez le résultat obtenu avec l'URI http://127.0.0.1:5000/api/modules.*

*Utilisez l'inspecteur de votre navigateur pour analyser la réponse obtenue et les entêtes « **HTTP** » de la requête et de la réponse.*




*« fig4 – consultation dans Firefox des entêtes HTTP et de la réponse format JSON en cas d'erreur »*

## ***Méthode GET – Lire une donnée spécifique à partir d'une API REST***

Nous venons de voir que la méthode « **HTTP GET** » est associée à une lecture de données pour satistifaire l'action « **Read** » de « **CRUD** ». Pour récupérer une donnée particulière, il faut un critère de recherche à passer en argument de la fonction de recherche.

***Exercice 15*** *: Dans le fichier « **routes.py** », ajoutez la fonction « **get_moduleById(module_id)** » qui sera appelée lors d'une recherche à partir d'un entier représentant l'identifiant du module. Cette méthode sera invoquée pour le endpoint « **/api/modules/module_id** ». Utilisez pour cela le décorateur suivant :*

```
@app.route("/api/modules/<int:module_id>", methods=['GET'])
```

*La méthode doit retrouver dans la liste « **modules** » le dictionnaire ayant comme valeur pour la clé « **id** » l'argument « **module_id** ». Le dictionnaire correspondant est retourné après mise en forme par la méthode « **jsonify()** ».*

*Si aucun module ne correspond à l'identifiant transmis, vous devrez lancer une erreur avec le code 404. Utilisez pour cela la méthode « **abort()** » de « **flask** » que vous devrez importer préalablement. Voir dans la documentation de **flask** :*

*https://flask.palletsprojects.com/en/1.1.x/api/?highlight=abort#flask.abort*

Très souvent, le seul élément connu du module est sa référence. Il est donc nécessaire d'effectuer la recherche de ses caractéristiques à partir de sa référence et non son identifiant.

***Exercice 16*** *: Dans le fichier « **routes.py** », ajoutez la fonction « **get_moduleByRef(module_ref)** » qui sera appelé lors d'une recherche à partir d'une chaîne de caractères représentant la référence du module. Cette méthode sera invoquée pour le endpoint «**/api/modules/module_ref** ». Utilisez pour cela le décorateur suivant :*

```
@app.route("/api/modules/<module_ref>", methods=['GET'])
```

*La méthode doit retrouver dans la liste « **modules** » le dictionnaire ayant comme valeur pour la clé « **ref** » l'argument « **module_ref** ». Le dictionnaire correspondant est retourné après mise en forme par la méthode « **jsonify()** ».*

*Si aucun module ne correspond, lancez une erreur avec le code 404.*

***Exercice 17*** *: Testez à présent votre API à l'aide d'un navigateur pour un module existant puis pour un module non référencé dans les deux modes de recherche, c'est-à-dire depuis l'identifiant ou la référence.*

*Observez les réponses obtenues. Utilisez l'inspecteur de otre navigateur pour analyser les entêtes « **HTTP** » de la requête et de la réponse.*

## ***Méthode POST – Création d'une donnée spécifique à partir d'une API REST***

La méthode « HTTP POST » est dédiée à la création de nouvelles données, action **Create** de **CRUD**. Pour enregistrer une nouvelle donnée, il faut transmettre un objet JSON dont chaque attribut est une propriété de la donnée. Dans notre cas, l'objet JSON à transmettre est du type :

```
{
   "ref": "M1105",
   "coef": 2,
   "description": "Bases des systèmes d'exploitation"
}
```

On ne communique pas l'identifiant numérique car il doit être implémenter automatiquement par l'API au moment de l'enregistrement pour éviter tout risque de doublon.

*La méthode HTTP GET envoie au serveur un entête HTTP en joignant à l'URL les données à transmettre. Le corps de requête est vide.*

*L'inconvénient de la méthode GET est le manque de protection des données qui sont visibles sans chiffrement dans la barre d'URL du navigateur, son cache, les historiques de navigation et les journaux du serveur, fichiers « .log ».*

*La longueur d'une URL est limitée à 2000 caractères. Cela restreint la taille des données nécessairement en caractères ASCII et ne peuvent être binaires (son, vidéo, …)*

*La méthode HTTP POST envoie au serveur un entête et un corps de message qui intègre les données. Les données transmises peuvent être de tout type et taille « illimitée ». Dans le corps de la requête, elles demeurent confidentielles.*

*Elles ne sont pas mises en cache et n'apparaissent pas dans les historiques*

L'API doit lire le corps de la requête pour récupérer les données. Dans une application Python « **flask** », la requête est représentée par un objet global nommée « **request** », instance de la classe « **Request** » – https://werkzeug.palletsprojects.com/en/1.0.x/wrappers/#werkzeug.wrappers.Request

Il faut importer l'objet « **request** » du paquet « **flask** » pour extraire les données transmises. En utilisant sa méthode « **request.get_json()** », vous pouvez vérifier que les données receptionnées sont bien du type mime « **application/json** » pour les analyser – voir la documentation :

https://flask.palletsprojects.com/en/1.1.x/api/?highlight=request%20get_json#flask.Request.get_json

Avant tout enregistrement, il faut vérifier que les données transmises correspondent bien aux champs représentant les propriétés à enregistrer. Il faut vérifier que l'enregistrement n'existe pas déjà.

***Exercice 18*** *: Dans le fichier « **routes.py** », ajoutez la fonction « **create_module()** » chargée d'enregistrer un module à partir des données transmises. Cette méthode est invoqué pour le endpoint « **/api/modules** » par une méthode « **POST** ». Utilisez pour cela le décorateur « `@app.route("/api/modules", methods=['POST'])` »*

*La fonction vérifie la conformité des données puis, si l'enregistrement n'existe pas, elle ajoute un nouveau module dans la liste, initialisé à partir des données transmises. Dans le cas contraire, elle lance une erreur avec le code 400 – voir les codes de statut HTTP https://developer.mozilla.org/fr/docs/Web/HTTP/Status .*

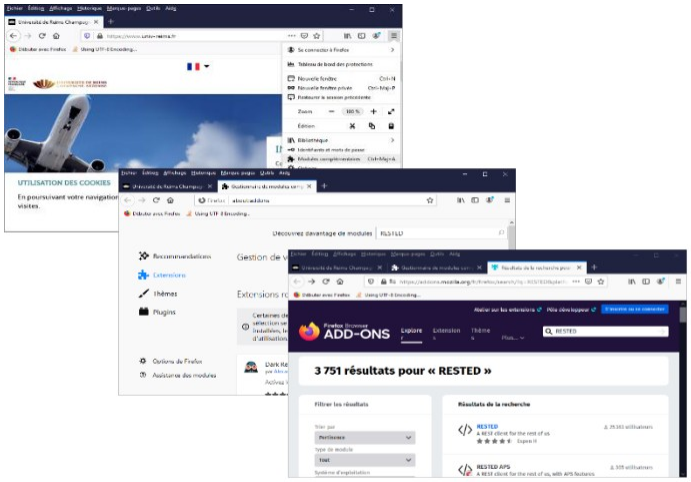
***Envoyer une requête HTTP POST à un serveur d'API REST***

Nous souhaitons tester notre fonction en enregistrant le module M1105.

```
{
   "ref": "M1105"
   "coef": 3,
   "description": "Bases des systèmes d'exploitation",
}
```

Pour tester une insertion, vous devez générer une requête HTTP POST. Par défaut, le navigateur ne peut envoyer que des méthodes GET. Il existe différents clients REST pour effectuer vos tests. La commande en ligne « **cURL** » existe pour la plupart des systèmes d'exploitation.

Il existe aussi d'autres clients REST comme le plugin « **RESTED** ». Il est proposé dans les principaux navigateurs. Il est moins complet que « **curl** » mais plus simple d'usage car il possède une interface graphique.
Pour l'installer sur firefox, dans le menu latéral, choisir « **Modules complémentaires** » ; dans la page, il faut sélectionner « **Extensions** » et saisir « **RESTED** » dans la zone de recherche avant de valider. Choisir alors dans la liste l'extension « **RESTED** » et l'« **Ajouter à Firefox** ».

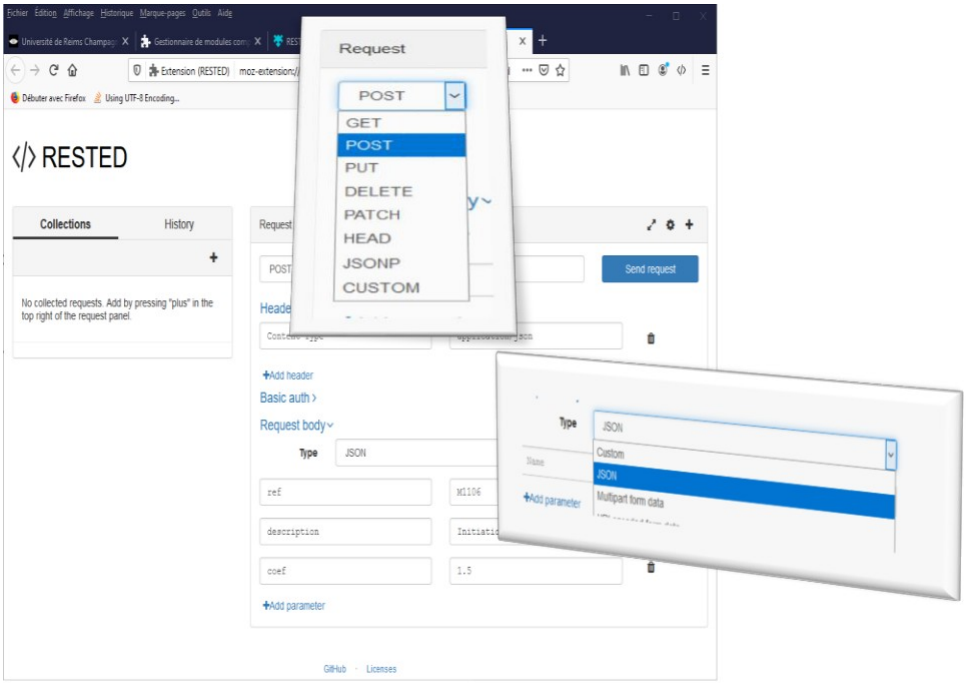



« fig9 – Firefox installer l'extension RESTED »

On lance l'application en cliquant sur l'icône qui apparaît dans la barre de raccourcis.


L'interface de l'extension est très intuitive et permet de :

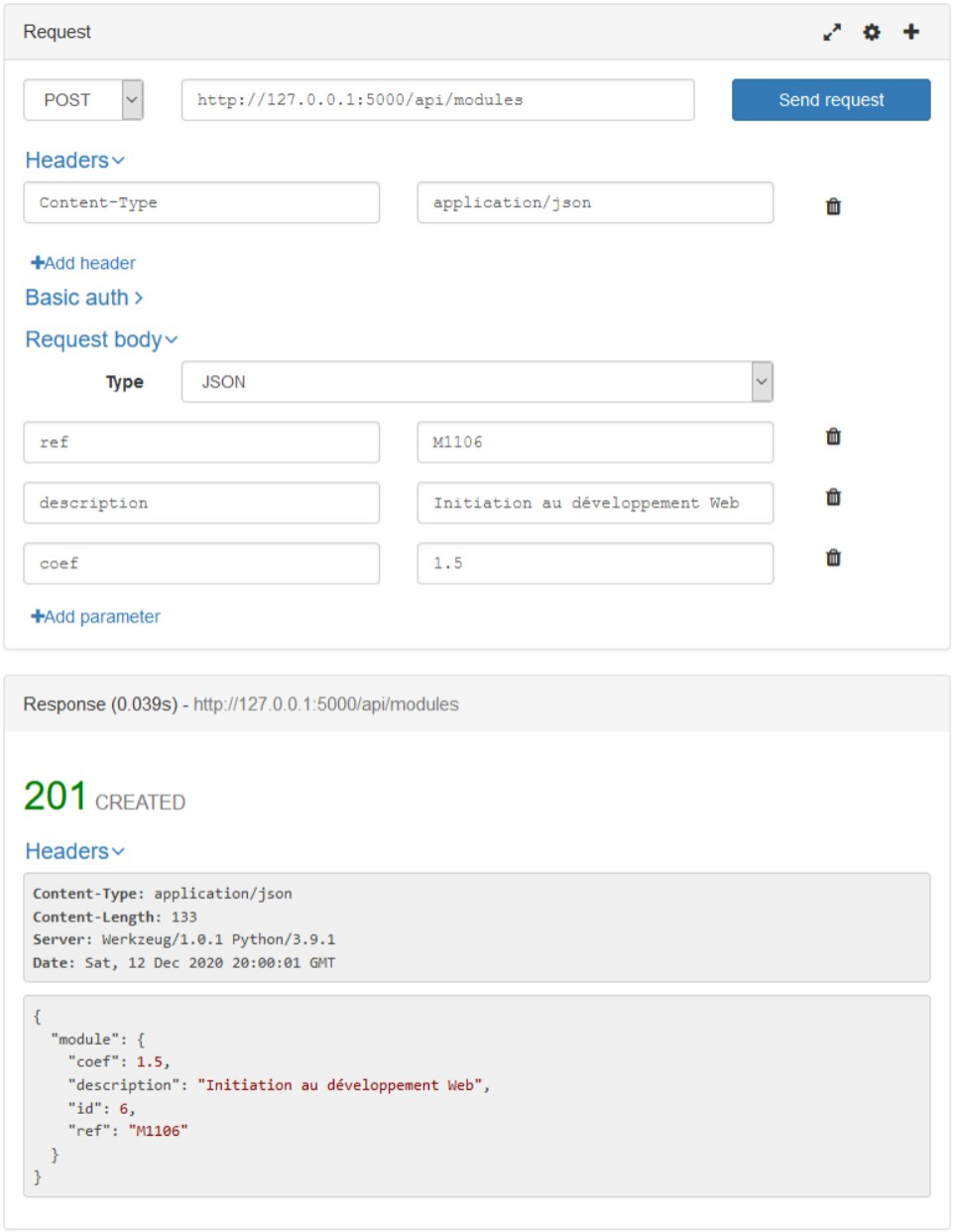
- ✓ Choisir la méthode HTTP,
- ✓ Saisir l'url du « **endpoint** » appelé,
- ✓ Ajouter des propriété d'entête avec le bouton « **+Add header** » de la section « **Headers** »,
- ✓ Définir le « Type » de données dans le corps de la requête, section « **Request body** »
- ✓ Ajouter des données à l'aide du bouton « **+Add parameter** » pour les requêtes qui le nécessite




« fig10 – Firefox utiliser l'extension RESTED »

La réponse du serveur est affichée et on peut visualiser l'entête.

***Exercice 19*** *: Tester votre API avec le client « **RESTED** » pour saisir le module « **M1106** » comme proposé dans la figure « **fig16 – Firefox extension RESTED ajouter un module** »*




*« fig11 – Firefox extension RESTED ajouter un module »*

*Méthode PUT – Mettre à jour une donnée spécifique à partir d'une API REST*

La méthode « HTTP PUT » est dédiée à la mise à jour d'une donnée, action **Update** de **CRUD**. Avant de modifier une donnée, il faut transmettre un objet JSON dont chaque attribut est une propriété de la donnée comme dans le cas d'une méthode POST de création d'un enregistrement. Par contre, il est important de passer l'identifiant dans l'URI pour bien identifier l'enregistrement à modifier. Le « **endpoint** » est celui retourné dans sa représentation publique.

La première action à réaliser par la fonction est de vérifier que l'objet transmis correpond bien à un objet enregistré. Il faut également que le type de chaque attribut de l'objet transmis soit conforme à celui atttendu.

***Exercice 20*** *: Dans le fichier « **routes.py** », ajoutez la fonction « **update_module(module_id)** » chargée de modifier un module à partir des données transmises. Cette méthode sera invoquée ar le endpoint « **/api/modules/id** » pour une méthode « **PUT** ». Utilisez pour cela le décorateur «* `@app.route("/api/modules/<int:module_id>", methods=['PUT'])` *».*

*La fonction vérifie l'existence de l'enregistrement grâce à son identifiant et la conformité des données, leur type. Dans ce cas, elle modifie le module dans la liste à partir des données transmises. Sinon, elle lance une erreur avec le code 400.*

***Exercice 21*** *: Tester votre API avec le client « **RESTED** » d'un navigateur pour redonner au module « **M1105** » son coefficient initial.*

*Méthode DELETE – supprimer une donnée spécifique à partir d'une API REST*

La méthode « HTTP DELETTE » est dédiée à la suppression d'une donnée, action **DELETE** de **CRUD**. Avant de supprimer une donnée, il faut transmettre l'identifiant du module dans l'URI, le « **endpoint** » est celui retourné dans sa représentation publique. Il permet de cibler le module à supprimer dans la liste.

***Exercice 22*** *: Dans le fichier « **routes.py** », ajoutez la fonction « **delete_module(module_id)** » chargé de supprimer un module à partir des données transmises. Cette méthode sera invoqué pour le endpoint « **/api/modules/id** » pour une méthode « **DELETE** ». Utilisez pour cela le décorateur* `@app.route("/api/modules/<int :module_id", methods=['DELETE'])`

*La fonction vérifie si l'enregistrement existe à partir de son identifiant ou de sa référence, la conformité des données, leur type. Dans ce cas, elle supprime le module de la liste. Sinon, elle lance une erreur avec le code 400.*

***Exercice 23*** *: Tester votre API avec le client « **RESTED** » d'un navigateur pour supprimer le module « **M1106** ».*