

Utilisation et conception d'interfaces API



Utilisation et conception d'interfaces API

Vincent Goulet

Professeur titulaire

École d'actuariat, Université Laval

Avec la collaboration de

Alexandre Coutu

Version 2025.07

© 2024-2025 par Vincent Goulet. *Utilisation et conception d'interfaces API* est mis à disposition selon le contrat **Attribution-Partage dans les mêmes conditions 4.0 International** de Creative Commons. En vertu de cette licence, vous êtes autorisé à :

- **partager** — copier, distribuer et communiquer le matériel par tous moyens et sous tous formats ;
- **adapter** — remixer, transformer et créer à partir du matériel pour toute utilisation, y compris commerciale.

L'Offrant ne peut retirer les autorisations concédées par la licence tant que vous appliquez les termes de cette licence.

Selon les conditions suivantes :



Attribution — Vous devez créditer l'œuvre, intégrer un lien vers la licence et indiquer si des modifications ont été effectuées à l'œuvre. Vous devez indiquer ces informations par tous les moyens raisonnables, sans toutefois suggérer que l'Offrant vous soutient ou soutient la façon dont vous avez utilisé son œuvre.



Partage dans les mêmes conditions — Dans le cas où vous effectuez un remix, que vous transformez, ou créez à partir du matériel composant l'œuvre originale, vous devez diffuser l'œuvre modifiée dans les mêmes conditions, c'est-à-dire avec la même licence avec laquelle l'œuvre originale a été diffusée.

Code source

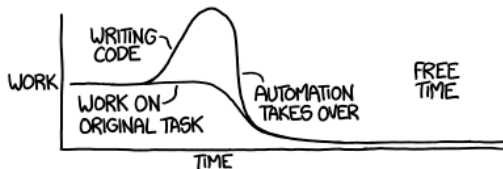
 [Voir sur GitLab](#)

Couverture

Spécimen mâle d'abeille pelle des montagnes rocheuses (*Anthophora montana*) récolté lors d'une étude menée dans le Parc national de Rocky Mountain (Colorado). Les anthophores sont des abeilles solitaires qui bâtissent leur nid dans le sol. Leur taille est similaire à celle d'un bourdon, mais les marques de couleur crème dans le visage leur sont uniques. Crédit photo : image du domaine public fournie par le *Bee Inventory and Monitoring Lab* du *United States Geological Survey* à Beltsville (Maryland); via [Wikimedia Commons](#), où l'image a été finaliste comme **photo de l'année 2015**.

"I SPEND A LOT OF TIME ON THIS TASK.
I SHOULD WRITE A PROGRAM AUTOMATING IT!"

THEORY:



REALITY:



Présentation générale

Utilisation d'une interface API avec curl

Réseautique 101

Concepts de base de la conception d'interfaces API

Analyse détaillée d'une API **plumber**

Déploiement sur un serveur Linux externe

Une dernière chose

- Utiliser une interface API avec l'utilitaire curl
- Concevoir une interface API simple dans R à l'aide du packaging **plumber**.

Cette formation requiert les outils logiciels suivants.

- Ligne de commande Unix (MSYS2 ou Git Bash dans Windows; Terminal dans macOS)
- Utilitaires curl et ssh (livrés avec MSYS2, Git Bash et macOS)
- Paquetages R **plumber** et **readr**



Les interfaces API de cette formation sont déployées sur le serveur act-2002.fsg.ulaval.ca.

- Serveur accessible seulement depuis le réseau de l'Université Laval ou après avoir établi une connexion sécurisée par VPN.
- Ouverture d'une session sur le serveur réservée aux étudiants inscrits au cours ACT-2002 Méthodes numériques en actuariat.

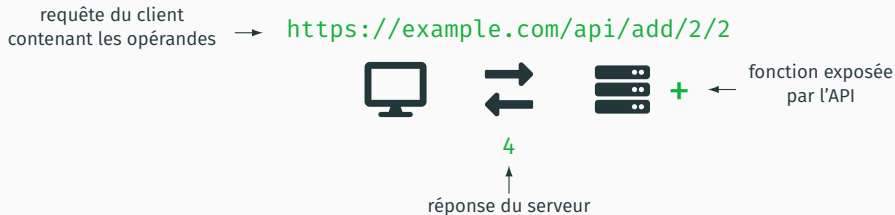
Présentation générale

Une interface API (ou interface de programmation d'application, de l'anglais *application programming interface*) est un ensemble de routines standards par lesquelles un logiciel, le **serveur**, offre des services à d'autres logiciels, les **clients**.



Exemple trivial

Le serveur expose une fonction d'addition via une interface API.







Avantages et inconvénient d'une interface API

- **Flexibilité** : le client peut utiliser les résultats comme bon lui semble
- **Universalité** : le serveur et le client peuvent utiliser des langages de programmation différents — seul le langage de communication doit être commun
- **Automatisation** : tout se passe entre des machines — et les machines se prêtent bien à l'automatisation
- **Complexité** : l'utilisation et la conception d'interfaces API nécessite une certaine expertise informatique — mais pas nécessairement une expertise certaine

- **Utilisation** : utilitaire curl depuis la ligne de commande Unix
- **Conception** : packaging **plumber** dans R
- **Déploiement** : serveur Linux

Éléments de jargon additionnels

- Nous allons utiliser et créer des interfaces API d'architecture **REST**  — un standard *de facto* aujourd'hui
- L'échange de données, le cas échéant, se fera dans le format léger **JSON**  — un standard *de facto* aujourd'hui
- La communication avec le serveur s'effectue par le biais du protocole **HTTP**  (*hypertext transfert protocol*) — le standard qui a donné naissance au web
- Toute requête HTTP reçoit une réponse qui, notamment, contient un **code de réponse**  — la diapositive suivante dresse la liste des codes les plus courants

Codes de réponse HTTP courants

1xx Information

2xx Succès

200 OK

201 création réussie

3xx Redirection

4xx Erreur du côté client HTTP


400 syntaxe de requête erronée

404 ressource introuvable

5xx Erreur du côté serveur


500 erreur interne du serveur



L'accès aux sites web passe aujourd'hui à peu près exclusivement par le protocole **HTTPS**  (*hypertext transfert protocol secure*) qui est identique à HTTP, sinon qu'il ajoute une couche de chiffrement pour assurer l'intégrité et la confidentialité de la communication.

Utilisation d'une interface API avec curl

Présentation de curl

Nous allons effectuer nos requêtes aux interfaces API depuis la ligne de commande Unix à l'aide de l'utilitaire standard **curl** .

- Outil et bibliothèque de transfert de données à partir d'adresses URL
- Forme de degré zéro d'un navigateur web : l'utilitaire lance des requêtes à une adresse URL et affiche la réponse, toute la réponse, en texte brut; essayez

```
$ curl "https://example.com"
```



identifie l'invite de commande Unix

- Standard *de facto* : interfaces dans à peu près tous les langages de programmation

Parce que les traditions, c'est important : premier exercice avec une API de type *Hello, World!*

1. Effectuer la requête par défaut à l'API depuis la ligne de commande Unix (le drapeau `-s` masque les informations de communication).

```
$ curl -s "http://act-2002.fsg.ulaval.ca:4242/hello"
```

2. Effectuer des requêtes avec des paramètres (remplacer `<nom>` par votre prénom ; vous pouvez entrer la seconde requête sur une seule ligne en ayant soin de supprimer le symbole « `\` »).

```
$ curl -s "http://act-2002.fsg.ulaval.ca:4242/hello?name=<nom>"  
$ curl -s "http://act-2002.fsg.ulaval.ca:4242/hello?name=<nom>\&greeting=Bonjour"
```

3. Consulter la [documentation](#) de l'API générée automatiquement par [Swagger](#).



Vous pouvez aussi utiliser l'interface Swagger pour composer des requêtes de manière interactive et obtenir la commande curl correspondante. N'hésitez pas à en tirer profit... mais assurez-vous de savoir vous passer de cette béquille!

Vous pouvez reproduire l'exercice précédent sur votre poste de travail.

1. Si ce n'est déjà fait, décompresser le présent matériel de formation à un endroit facile d'accès sur votre disque (par ex. : `~/Documents`).
2. Si ce n'est déjà fait (bis), installer dans R le paquetage **plumber**.

```
> install.packages("plumber")
```

👉 identifie l'invite de commande de R

3. À la ligne de commande Unix, naviguer jusqu'au sous-répertoire `hello-api` du matériel de formation.

```
$ cd ~/Documents/utilisation-et-conception-interfaces-api/hello-api
```

👉 chemin d'accès à titre indicatif

⚙ Exercice (suite)

4. Lancer l'API avec R; le symbole `&` à la fin de la commande permet de lancer le programme en arrière-plan (*background*).

```
$ Rscript -e 'source("deploy-local.R")' &
```

5. Appuyer une autre fois sur `↵` pour récupérer l'invite de commande (`↵` identifie la touche **Entrée**).
6. Répéter les commandes de l'exercice précédent en remplaçant l'hôte et le port `act-2002.fsg.ulaval.ca:4242` dans les adresses URL par ceux affichés lors du lancement de l'API (de la forme `127.0.0.1:<nnnn>`).
7. Pour arrêter l'API, ramener d'abord le processus R à l'avant-plan (*foreground*).

```
$ fg
```

8. Appuyer ensuite sur `^C` pour stopper le processus et retourner à l'invite de commande (`^` identifie la touche **Contrôle**).

Options et drapeaux de curl

curl compte une très longue liste d'options et de drapeaux. Certains seront utiles dans le cadre de la formation. Pour chacun il existe une version courte (une lettre) et une version longue (un mot).

-d, --data *<données>*
données à transmettre avec la requête

-F, --form *<nom=contenu>*
données à transmettre sous forme
nom=contenu

-H, --header *<entête>*
information additionnelle à passer en
entête de la requête

-o, --output *<fichier>*
écrire les résultats dans *<fichier>* plutôt
qu'à la sortie standard

-X, --request *<méthode>*
méthode à utiliser pour la requête (**GET**,
POST, **PUT**, **DELETE**)

-s, --silent
mode silencieux (masque des
informations de communication)

-u, --user *<utilisateur>*
nom d'utilisateur pour la connexion
avec authentification

-v, --verbose
mode bavard (utile pour le débogage)

Méthodes pour les requêtes

Dans le protocole HTTP, une méthode spécifie un type de requête. Il y a quatre principales méthodes.

- GET** méthode la plus courante; récupérer une ressource sans rien modifier
- POST** transmettre des données en vue d'un traitement; le résultat peut être la création de nouvelles ressources ou la modification de ressources existantes
- PUT** remplacer ou ajouter une ressource (non utilisée dans cette formation)
- DELETE** supprimer une ressource (non utilisée dans cette formation)



Le type de requête par défaut dans curl est **GET**, c'est pourquoi il n'est pas nécessaire de spécifier cette méthode avec l'option **-X**.

Structure d'une requête GET

Une requête GET peut contenir ou non des paramètres. Lorsqu'il y en a, les paramètres sont spécifiés en suffixe de l'adresse URL sous forme d'une chaîne de requête [↗](#).

- Séparés de l'adresse par le symbole « ? »
- Spécifiés sous la forme clé=valeur
- Séparés les uns des autres par le symbole « & »

Forme générale d'une requête GET :

```
$ curl "http://<adresse>?<clé>=<valeur>&<clé>=<valeur>[...]"
```

Structure d'une requête POST

Les requêtes **POST** sont généralement utilisées pour transmettre des données à un serveur. Avec curl, il faut alors fournir les informations suivantes :

- type de requête avec l'option **-X**;
- format des données avec l'option **-H**;
- données avec les options **-d** ou **-F**.

Forme générale d'une requête **POST** avec des données — ou **corps de requête** — en format JSON :

```
$ curl -X POST -H "Content-Type: application/json" \  
-d '{"<clé>": "<valeur>", "<clé>": "<valeur>", [...]}' "http://<adresse>"
```




Bien que devant normalement servir à transmettre des données, les requêtes **POST** sont aussi couramment utilisées en lieu et place des requêtes **GET** pour récupérer de l'information. Imaginez, par exemple, un scénario où l'on doit spécifier 400 paramètres à l'API pour recevoir l'information voulue : il est beaucoup plus simple de les placer dans un objet JSON que dans l'adresse URL!



Les requêtes **GET** sont également moins sécuritaires, car susceptibles d'être mises en cache (sauvegardées pour être ensuite récupérées) par le serveur. Ce n'est pas le cas pour les requêtes **POST**.

Cet exercice nécessite un accès au référentiel Bitbucket de la Faculté des sciences et de génie. Vous devrez vous authentifier à l'aide de votre IDUL et de votre NIP. Votre NIP ne s'affichera pas lors de la saisie; c'est une mesure de sécurité.

Comme tous les référentiels Git, Bitbucket permet d'effectuer une **foule d'opérations**  par le biais de son interface API.

1. Obtenir la liste, en format JSON, des projets auxquels vous avez accès dans le référentiel (remplacer `<IDUL>` par votre IDUL).

Windows (**winpty** nécessaire pour **contourner un bogue de Git Bash** )

```
$ winpty curl -u <IDUL> -s \  
"https://projets.fsg.ulaval.ca/git/rest/api/1.0/projects"
```

macOS

```
$ curl -u <IDUL> -s \  
"https://projets.fsg.ulaval.ca/git/rest/api/1.0/projects"
```

2. Modifier votre image de profil (avatar) dans Bitbucket directement par le biais de l'API. Vous devrez choisir sur votre poste de travail une image dont les dimensions sont d'au plus 1024×1024 pixels (vous pouvez utiliser le fichier `avatar.png` livré avec cette formation). Remplacer `<IDUL>` à deux endroits par votre IDUL et `<fichier>` par le nom du fichier contenant l'image.

Windows

```
winpty curl -u <IDUL> -s -X POST \  
-H "X-Atlassian-Token: no-check" -F avatar=@<fichier> \  
"https://projets.fsg.ulaval.ca/git/rest/api/1.0/users/<IDUL>/avatar.png"
```

macOS

```
curl -u <IDUL> -s -X POST \  
-H "X-Atlassian-Token: no-check" -F avatar=@<fichier> \  
"https://projets.fsg.ulaval.ca/git/rest/api/1.0/users/<IDUL>/avatar.png"
```

Réseautique 101

Adresses IP et noms de domaine

- Chaque appareil faisant partie d'un réseau informatique dispose d'un numéro d'identification unique nommé **adresse IP** (*Internet Protocol*)
- Les adresses IP (de la version 4) sont composées de quatre nombres compris entre 0 et 255 et séparés par des points : 132.203.190.200
- Afin de faciliter leur repérage, les adresses IP sont souvent masquées par un **nom de domaine** : **act-2002.fsg.ulaval.ca**
- L'adresse IP 127.0.0.1 et le nom de domaine **localhost** sont réservés pour identifier l'ordinateur (ou hôte) local

- Un serveur — ou n'importe quel ordinateur — peut exécuter des dizaines de processus qui, chacun, écoutent ou émettent des informations à des endroits spécifiques : les **ports** (par analogie, si l'adresse IP identifie une maison, le port identifie une porte de chambre)
- S'il faut préciser le port précis d'un service, son numéro est placé après l'adresse IP ou le nom de domaine, séparé par un deux-points : `127.0.0.1:5342`, `act-2002.fsg.ulaval.ca:4242`
- L'utilisation des ports inférieurs à 1023 requiert des droits d'accès administrateur
- Quelques services et leurs ports usuels :

22	SSH (accès sécurisé)	123	NTP (horloge)
25	SMTP (envoi de courriels)	443	HTTPS (sites web sécurisés)
80	HTTP (sites web)	25565	Minecraft

Concepts de base de la conception d'interfaces API

Composantes d'une interface API

La conception d'une interface API nécessite la définition de quelques éléments incontournables.

- Points d'accès (*endpoints*) qui identifient où se trouvent les ressources pour les clients
- Format de sortie des ressources
- Fonctions de création de ressources et leurs paramètres pour chacun des points d'accès
- Fonctions de gestion des erreurs, le cas échéant
- Adresse URL du serveur (adresse IP et port)

plumber est un paquetage qui permet de créer facilement des interfaces API directement dans R en se chargeant de... la tuyauterie :

- création des points d'accès;
- conversion des résultats vers le format de sortie;
- déploiement du serveur.

Vous conservez la responsabilité des éléments qui circulent dans la tuyauterie :

- création des ressources;
- gestion des erreurs.



Avertissement

Comme son logo le laisse présager, **plumber** fait partie de la nébuleuse de paquets Tidyverse.

Le principal élément de dialecte que vous rencontrerez dans l'utilisation et dans la documentation de **plumber** est l'opérateur de tuyau « **%>%** ».



- Analogue pour R de l'opérateur « | » de la ligne de commande Unix
- Remplace le paradigme de programmation fonctionnelle par un de création de résultats par l'applications successives de « filtres »

```
> f(g(x))
```



```
> x %>% g %>% f
```

Syntaxe des fichiers plumber

Le code source de création d'interfaces API avec **plumber** est placé dans des fichiers de script R standards, à deux caractéristiques près.

1. Configuration du serveur (« tuyauterie ») placée derrière des balises spéciales « `#*` » (concept inspiré de **roxygen2** [↗](#))
2. Fonctionnalités des points d'accès sous forme de fonctions anonymes



Examiner le code source de l'API *Hello, World!* qui se trouve dans le fichier `hello-api/api.R`. Identifier les éléments de configuration du serveur et la fonction (anonyme) responsable de fournir la salutation.

Déploiement d'une interface

Le déploiement (ou la mise en service) d'une interface API avec **plumber** se déroule en deux étapes.

1. Création d'un objet contenant toute la logique de l'API avec la fonction `plumb`; cette fonction prend en argument :
 - `file` nom du script contenant la définition de l'API
2. Déploiement de l'API avec la fonction `pr_run`; cette fonction prend en arguments :
 - `pr` objet créé par `plumb`
 - `host` adresse IP du serveur (par défaut `127.0.0.1`)
 - `port` port du serveur auquel l'interface écoutera (déterminé automatiquement si non spécifié)

Déploiement local et sur un serveur

Le déploiement d'une API peut s'effectuer directement depuis la ligne de commande R.

- Déploiement sur l'ordinateur local à un port déterminé automatiquement (et donc susceptible de changer chaque fois)

```
> pr_run(plumb("api.R"))
```

- Déploiement sur un serveur à une adresse IP et un port fixes

```
> pr_run(plumb(file = "api.R"),  
+         host = "132.203.190.200", port = 4242)
```



Examiner et comparer le code des scripts de déploiement
[hello-api/deploy.R](#) et [hello-api/deploy-local.R](#).



En démarrant une API depuis la ligne de commande R ou depuis RStudio, le processus R demeure bloqué tant et aussi longtemps que l'API est en fonction. Une meilleure approche consiste à lancer un processus R uniquement dédié à l'interface API. C'est ce que vous avez fait dans les exercices précédents avec `Rscript` depuis la ligne de commande Unix.

Analyse détaillée d'une API plumber

Présentation de l'API

Nous allons maintenant plonger plus avant dans la conception d'interfaces API avec **plumber** en étudiant une API comportant :

- plusieurs points d'accès et autant de ressources;
- divers formats de sortie;
- gestion des erreurs.

L'API est basée sur les données ouvertes de **BIXI Montréal** . Elle est déployée aux points d'accès suivants :

<http://act-2002.fsg.ulaval.ca:4243/data>

<http://act-2002.fsg.ulaval.ca:4243/summary>

<http://act-2002.fsg.ulaval.ca:4243/revenues>


<http://act-2002.fsg.ulaval.ca:4243/plot>

Vous êtes en mesure de déchiffrer le fonctionnement global de l'API à partir de sa documentation et de son code source.

1. Identifier les points d'accès de l'API à partir de son **interface Swagger** [↗](#).
2. Effectuer des requêtes simples à l'aide de Swagger, puis directement depuis la ligne de commande avec curl.
3. Étudier le code source de l'API dans le répertoire **bixi-api** en portant une attention toute particulière au fichier **api.R**.



L'interface API BIXI a recours à la programmation orientée objet de R. La fonction `cost` retourne un objet de classe `"cost"`. La fonction `plot.cost` est une méthode de la fonction générique `plot` pour de tels objets. Consultez la [section 10.9](#) de *An Introduction to R* pour une brève explication de ces concepts.

Le code source de l'interface API a recours à diverses **annotations**  placées après les balises spéciales « `#*` ».

- `@apiTitle, @apiDescription, @apiVersion`
- `@plumber`
- `@get, @post`
- `@param`
- `@serializer`

Annotations globales

Les annotations globales `@apiTitle`, `@apiDescription`, `@apiVersion` permettent de fournir des informations générales sur l'API.

- Peuvent apparaître n'importe où dans le code source
- Ne nécessitent pas d'être suivies d'une expression R
- Retours à la ligne `interdits` dans les annotations

```
/* @apiTitle Analyse des données de BIXI Montréal  
/* @apiDescription Analyse sommaire des données [...]  
/* @apiVersion 2025.07
```

Modification du routeur

Dans le jargon de **plumber** le routeur est l'application qui administre l'API.

L'annotation `@plumber` permet de modifier la configuration par défaut du routeur.

- Doit être suivie d'une expression R
- Expression habituellement une définition de fonction anonyme qui modifie le routeur

```
## @plumber
function(pr)
{
  pr %>%
    pr_set_error(error_handler) %>%
    pr_set_serializer(serializer_unboxed_json())
}
```


Modification du routeur (suite)

- Fonction prend un objet `pr` (*plumber router*) et le modifie à l'aide de deux « filtres » (des fonctions, vraiment)
- `pr_set_error` indique la fonction responsable de traiter les erreurs (nous y reviendrons)
- `pr_set_serializer` indique le « sérialiseur » par défaut, soit le filtre responsable de la conversion des données vers le format de sortie (nous y reviendrons aussi)

Définition et configuration des points d'accès

Les annotations `@get` et `@post` définissent des points d'accès utilisant les méthodes HTTP correspondantes.


- Généralement accompagnées d'une ou plusieurs annotations `@param`
- Suivies de la fonction (anonyme) de création de ressources au point d'accès

```
[...]
/* @param start:string  Date de début du déplacement
/* @param end:string    Date de fin du déplacement
[...]
/* @get /data
function(start = NA, end = NA, code = NA,
          duration_min = NA, duration_max = NA, status = NA, res)
{
    [...]
}
```

Définition et configuration des points d'accès (suite)

Les annotations `@param` définissent les paramètres du point d'accès sous la forme

```
#* @param <nom>:<type> <description>
```

- `<nom>` est le nom du paramètre; on le retrouve normalement dans les arguments de la fonction de création de ressources du point d'accès
- `<type>` est le **type**  du paramètre (`numeric`, `integer`, `logical`, `character`, `object`, etc.)
- `<description>` est une courte description du paramètre qui est affichée dans la documentation de l'API





Les valeurs des paramètres passés dans l'URL sont toujours transmis à R en mode `character`, et ce, peu importe le `<type>` mentionné dans l'annotation `@param`. Vous devez donc vous assurer de les convertir dans le bon mode dans vos fonctions R.



Dans l'API BIXI, ce sont les fonctions `check.*` qui se chargent de cette conversion.


La fonction de création de ressources est une fonction R **anonyme** qui est appelée à chaque requête à un point d'accès.

- Point d'accès **@get** : fonction prend en arguments tous les paramètres potentiels de la chaîne de requête (prévoir des valeurs par défaut pour les paramètres optionnels, comme pour toute fonction R)
- Point d'accès **@post** : fonction prend en argument l'**objet de requête**  **req** contenant les informations sur la requête — notamment le corps de requête JSON dans la liste **req\$body**
- Point d'accès effectuant une gestion des erreurs : fonction prend aussi en argument l'**objet de réponse**  **res** contenant la réponse de l'API — notamment le corps de la réponse dans la liste **res\$body** et le code de réponse HTTP dans **res\$status**




plumber ne permet pas d'attribuer une valeur par défaut **NULL** aux paramètres optionnels d'une chaîne de requête. Pour indiquer que des arguments de la fonction de création de ressources peuvent n'avoir aucune valeur, il faut plutôt utiliser une valeur par défaut **NA**.

Identifier et comparer les fonctions de création de contenu des points d'accès `/data`, `/summary`, `/revenues` et `/plot` de l'interface API BIXI.

1. Déterminer le type de chacun des paramètres des points d'accès.
2. JSON ne comporte que deux **structures de données**  : le **dictionnaire** (collection de couples `"clé": "valeur"`) et la **liste de valeurs ordonnées** (un vecteur dans la terminologie de R). Identifier comment ces deux structures sont utilisées dans le corps de requête JSON du point d'accès `/plot`.
3. Déterminer le rôle de la fonction auxiliaire `getRentals`.
4. Obtenir le sommaire statistique des déplacements effectués jusqu'au 15 juin 2021 par des membres BIXI et impliquant la station 75.

Conversion vers le format de sortie

L'annotation optionnelle `@serializer` permet de spécifier un **convertisseur de données**  — ou **sérialiseur** — différent pour un point d'accès.

- Sérialiseur par défaut de **plumber** est `json`
- Sérialiseur par défaut de l'API déterminé avec `pr_set_serializer` dans la configuration du routeur
- Conversion prise en charge par les fonctions de **plumber**

1. Identifier les sérialiseurs utilisés par les points d'accès `/data`, `/summary`, `/revenues` et `/plot` de l'interface API BIXI.
2. Le sérialiseur `png` accepte des options additionnelles. Déterminer leur rôle à partir de la documentation des `sérialiseurs` [🔗](#) et de celle de la fonction `png`.
3. Obtenir un graphique de la répartition des revenus pour les déplacements de moins de 30 minutes effectués entre les 6 et 22 juin 2021 (inclusivement). Utiliser d'abord une couleur unique `lightblue2`, puis la palette `Okabe-Ito`. Effectuer la requête dans l'interface Swagger et avec curl. Depuis la ligne de commande, vous devrez soit spécifier le fichier de sortie avec l'option `-o` de curl, soit rediriger le résultat dans un fichier avec l'opérateur « `>` ».

Lors du traitement des requêtes, une API peut rencontrer des erreurs provenant d'appels à la fonction **stop** (exceptions prévue dans le code) ou de l'évaluation même du code (bogues).

- Sans traitement spécial, toute erreur cause un code de retour HTTP 500 (erreur interne du serveur) peu informatif
- Pour les **exceptions prévues**, préférable de retourner un code HTTP 400 (syntaxe de requête erronée)
- Code 500 demeure adéquat pour les **bogues**

Gestion des erreurs (suite)

La configuration du routeur peut indiquer avec `pr_set_error` une fonction à appeler lorsque l'API rencontre une erreur.

- Fonction compte trois arguments :
 1. objet de requête `req`
 2. objet de réponse `res` contenant notamment le code de retour HTTP dans `res$status`
 3. objet d'erreur `err` contenant notamment le texte du message d'erreur dans `err$message`
- Si `err$message` provient d'une exception prévue dans le code : fonction fixe `res$status` à 400 et retourne le message d'erreur qui sera transmis dans la réponse
- Autrement : erreur considérée comme un bogue, la fonction fixe `res$status` à 500 et retourne le message d'erreur qui sera transmis dans la réponse



Retranscrire les messages complets des exceptions dans la fonction de gestion des erreurs est propice aux... erreurs. C'est pourquoi l'interface API BIXI a plutôt recours à des « codes d'erreur » compacts dans `stop`. Les véritables messages d'erreur sont ainsi regroupés dans la fonction `error_handler`.

Étudier la fonction de gestion des erreurs `error_handler` de l'interface API BIXI définie dans le fichier du même nom.

1. Identifier de quelles fonctions du code source proviennent les divers messages d'erreur.
2. Identifier les codes de réponse HTTP retournés par la fonction.
3. Effectuer des requêtes à l'API qui produisent les messages d'erreur de type « invalide ».

Déploiement sur un serveur Linux externe

Déploiement local vs déploiement externe

Le déploiement d'une interface API sur un serveur Linux comporte quelques différences mineures par rapport à un déploiement local depuis la ligne de commande Unix.

- Accéder au serveur depuis la ligne de commande
- Utiliser l'adresse IP du serveur et un port spécifique dans la fonction `pr_run`
- Garder le processus R actif après la déconnexion du serveur

Les instructions de déploiement qui suivent supposent que le code source de votre interface API est hébergé dans un dépôt Git.

1. Créer dans le projet un fichier `deploy.R` basé sur celui livré avec les interfaces `hello-api` et `bixi-api`
2. Utiliser `host = "132.203.190.200"` dans la fonction `pr_run` pour un déploiement sur le serveur `act-2002.fsg.ulaval.ca`
3. Si vous connaissez le port à utiliser, indiquez-le avec l'argument `port` de la fonction `pr_run`
4. Publier le fichier dans le dépôt

Pour effectuer des opérations sur un serveur Linux externe, il faut d'abord y obtenir un accès par le biais de l'utilitaire standard ssh (*secure shell*).

- Accéder au serveur de la formation depuis la ligne de commande Unix (remplacer `<IDUL>` par votre IDUL)

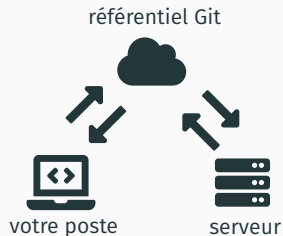
```
$ ssh <IDUL>@act-2002.fsg.ulaval.ca
```

- Utiliser la ligne de commande du serveur comme celle de Git Bash (Windows) ou du Terminal (macOS)
- Pour mettre fin à la session, utiliser en temps et lieu les commandes `exit` ou `logout`, ou encore le raccourci clavier `^D`

Transférer le code source de l'API vers le serveur

Une copie à jour du code source de votre interface API doit se trouver sur le serveur.

- Git est votre ami pour transférer le code vers le serveur
- Cloner le code source de votre interface sur le serveur Linux comme à l'habitude
- Changer le répertoire courant pour celui contenant le fichier `deploy.R`






En disposant dorénavant de deux copies du projet — une sur votre poste de travail et une sur le serveur — vous collaborez en quelque sorte avec vous-même via Git. Si vous effectuez des modifications au code source dans une copie, vous devrez les publier dans le référentiel avec `git push` et les récupérer dans l'autre copie avec `git pull`.

Déployer l'interface API

Tout processus lancé depuis une session ssh doit normalement être interrompu au moment de quitter la session. Or, le processus R qui administre l'interface API doit continuer de s'exécuter de manière autonome.

- Lancer le processus R en arrière-plan à l'aide de la commande **nohup**  afin que le processus reste actif après la déconnexion

```
$ nohup Rscript -e 'source("deploy.R")' &
```

- Vérifier que le déploiement s'est déroulé correctement en faisant afficher les dernières lignes du fichier **nohup.out**

```
$ tail nohup.out
```

- Quitter la session ssh
- Tester votre API déployée sur le serveur

Interrompre l'interface API

L'interruption d'une interface API déployée sur un serveur nécessite de récupérer le numéro du processus R qui l'administre.

- Accéder au serveur avec ssh
- Obtenir la liste de tous les processus vous appartenant

```
$ ps -cux
```

- Identifier le numéro du processus (**PID**) de R
- Mettre fin au processus R (remplacer *<PID>* par le numéro de processus identifié à l'étape précédente)

```
$ kill <PID>
```

Pour tester le déploiement d'une interface sur un serveur Linux, vous allez déployer l'API *Hello, World!* de la présente formation à partir de son code source.

1. Accéder au serveur de la formation.

```
$ ssh <IDUL>@act-2002.fsg.ulaval.ca
```

2. Cloner le dépôt Git de la formation.

```
$ git clone https://gitlab.com/vigou3/\  
utilisation-et-conception-interfaces-api.git
```

3. Changer le répertoire courant pour celui contenant le code source de la formation.

```
$ cd utilisation-et-conception-interfaces-api
```

Étape unique à cet exercice!

4. Le code source de la formation est rédigé en programmation lettrée. Il faut donc extraire le code des interfaces API du fichier `.Rnw`.

```
$ make api
```

5. Changer le répertoire courant pour celui contenant l'API.

```
$ cd hello-api
```

6. Comparer le contenu des fichiers de script et `deploy.R` et `deploy-exercice.R`.
Le port dans le premier étant déjà occupé par l'API en production, le second ne spécifie aucun port afin de laisser **plumber** en déterminer un automatiquement.

```
$ cat deploy.R  
$ cat deploy-exercice.R
```

7. Déployer l'API avec le script `deploy-exercice.R`.

```
$ nohup Rscript -e 'source("deploy-exercice.R")' &
```


8. Vérifier que le déploiement s'est déroulé correctement et identifier le port utilisé en faisant afficher les dernières lignes du fichier `nohup.out`.

```
$ tail nohup.out
```

9. Quitter la session ssh avec `^D`.
10. Tester votre API déployée sur le serveur (remplacer `<nnnn>` par le port mentionné dans le fichier `nohup.out`).

```
$ curl http://act-2002.fsg.ulaval.ca:<nnnn>/hello
```

11. Accéder de nouveau au serveur.

```
$ ssh <IDUL>@act-2002.fsg.ulaval.ca
```

12. Mettre fin à l'interface API (remplacer *<PID>* par le numéro de processus identifié avec la commande *ps*).

```
$ ps -cux  
$ kill <PID>
```

Une dernière chose

Et les routes dynamiques dans tout ça ?

Dans la présentation générale, nous avons donné l'exemple d'une interface API d'addition de deux nombres dont les paramètres étaient passés à même l'adresse URL, plutôt que dans une chaîne de requête :

`https://example.com/api/add/2/2`

Dans le jargon, cela s'appelle une **route dynamique** (puisque l'adresse change selon les valeurs des paramètres).

Création d'une route dynamique

La création des routes dynamiques passe par les annotations d'un point d'accès.

- Format général de l'annotation d'un point d'accès **GET** avec une route dynamique :

```
#* @get/<{param1}<type>>/<{param2}<type>>[ / ... ]
```

- *<param1>*, *<param2>*, sont les arguments de la fonction de création de ressource
- *<type>* est le type du paramètre, comme précédemment

Étudier et déployer l'interface API d'addition de deux nombres dans le répertoire `add-api`.

1. Identifier la définition de route dynamique dans le fichier `api.R`.
2. Déployer l'API localement et y effectuer des requêtes via l'interface Swagger et directement depuis la ligne de commande avec curl.

Ce document a été produit par le système de mise en page $\text{X}_{\text{Y}}\text{L}_{\text{A}}\text{T}_{\text{E}}\text{X}$ avec la classe **beamer** et le thème Metropolis. Les titres et le texte sont composés en Fira Sans, le code informatique en Fira Mono. Les icônes proviennent de la police Font Awesome. Les illustrations ont été entièrement réalisées avec $\text{L}_{\text{A}}\text{T}_{\text{E}}\text{X}$.