1. Rappels généraux

Tous les TD et TP se dérouleront sur **Linux**. Nous allons travailler avec plusieurs outils qui dépendent de **Docker**.

N'oubliez pas de déposer votre travail sur Moodle à *chaque* séance. Vous pouvez le continuer chez vous après la séance et le re-déposer.

Ce TP est prévu pour durer deux semaines, car les outils sont complexes. Il y aura deux zones de dépôt, une pour chaque semaine.

2. Présentation générale

Le but du TP est de tester automatiquement une API REST. Voici des explications.

2.1. Requêtes, ressources et collections

Une API REST est un service de type HTTP. Un client de cette API fait une requête, c'est à dire une demande ou une modification, sur un document, appelé *ressource*, présent sur le serveur à un emplacement désigné par l'URL : c'est le chemin du document concerné. Chaque ressource est identifiée par son chemin. Ce chemin est composé un peu comme un chemin Unix pour désigner la ressource. Les chemins sont appelés *terminaisons* (*endpoints*).

Exemple, sur un serveur de livres, on utilisera `http://serveur/livres/4265`, pour accéder au livre n°4265 et `http://serveur/genres` pour obtenir la liste des genres de livres.

Les ressources peuvent avoir des propriétés, auxquelles on accède par un chemin comme `http://serveur/livres/4265/titre`. Et, toujours selon ce qui est programmé dans le serveur, on peut ajouter des paramètres comme `http://serveur/livres/4265/résumé?lang=fr`, pour préciser la réponse voulue.

En général, un chemin comme `http://serveur/livres/4265` désigne une seule ressource, tandis que `http://serveur/livres` désigne la totalité des ressources de ce type, c'est une *collection*.

Tout comme les n-uplets d'un SGBD, les ressources peuvent être consultées (`GET`), créées (`POST`), remplacées (`PUT`), modifiées (`PATCH`) et supprimées (`DELETE`). Ces codes entre parenthèses sont appelés les *méthodes* dans le protocole HTTP. Avec un navigateur internet, on n'utilise que la méthode `GET` pour consulter des pages web, mais dans le modèle REST, les autres méthodes sont aussi utilisées.

Exemple, faire une requête HTTP sur `http://serveur/livres/4265` avec la méthode `DELETE` supprime ce livre, et `PATCH` sur `http://serveur/livres/4265/titre` modifie son titre.

Les méthodes `POST`, `PUT` et `PATCH` demandent une *charge utile* (*body* ou *payload*) en plus de l'URL. C'est ce qu'il faut créer ou modifier dans la ressource. Et d'autre part, il faut généralement spécifier la représentation *MIME* des données qu'on veut recevoir ou envoyer. Par exemple, il faut indiquer qu'on veut du XML en ajoutant un entête HTTP `Accept: application/xml` et pour préciser qu'on envoie de l'XML, on ajoute l'entête `Content-Type: application/xml`.

Quand une requête réussit, le serveur retourne les données avec un code d'état en $2xx$, par exemple 200 pour une réponse à un `GET`, 201 pour des `POST` et `PUT` réussis. Quand il y a une erreur dans la

demande du client, le serveur retourne généralement un court message explicatif avec un code d'état en $4xx$. Par exemple, 409 pour une donnée en double. Les codes sont décrits sur cette page.

2.2. Test des terminaisons, requêtes et réponses

Le logiciel `PostMan` permet de créer ces requêtes HTTP. On indique la méthode, l'URL, les entêtes et la charge utile. Et il affiche les données reçues en résultat. Avec `PostMan`, on peut automatiser l'envoi de requêtes sur les différentes terminaisons, mais on ne peut pas effectuer des tests approfondis sur les réponses obtenues, comme valider les données XML et les inspecter avec XPath (idem avec du JSON). Donc on ne peut pas l'utiliser pour automatiser ces tests.

En ligne de commande, il y a `cURL` qui permet de communiquer avec un serveur REST, et récupérer entêtes et charge utile, pour les soumettre à tous les tests qu'on veut avec `xmllint`. Par contre, c'est du `bash`, avec des tubes, des fichiers temporaires, etc.

Nous allons plutôt utiliser un autre outil, très polyvalent, `Robot Framework`. Il permet de simuler les actions d'un humain sur une interface web, mais aussi de dérouler des tests sur une API REST. Cet aspect n'a pas été présenté en cours, mais sera utilisé dans ce TP. Par exemple, un test consistera à émettre une requête POST avec une charge utile précise et vérifier que le serveur a répondu ce qui était attendu, par exemple que cette charge utile était invalide.

Tous ces outils, `PostMan`, `cURL` et `Robot Framework` sont pour le côté client, c'est à dire pour émettre des requêtes et vérifier les réponses du serveur. Côté serveur, on va utiliser un autre outil, `WireMock Studio`. Il permet de simuler un serveur REST, de répondre des contenus factices pour mettre les tests au point, en attendant que le vrai serveur soit programmé.

On va commencer par étudier `WireMock Studio` et après on verra `Robot Framework`.

3. WireMock Studio

`WireMock Studio` est un éditeur et serveur d'API REST simulées à l'aide de règles appelées « *mappings* » ou « *stubs* ». Il y a aussi un outil sans interface, `WireMock` pour uniquement servir ces règles aux clients.

NB: `WireMock Studio` est maintenant déprécié, au profit de WireMock Cloud qui est payant.

1. Ouvrez un shell et tapez les commandes suivantes : 

```
mkdir -p /Docker/$USER/tp6/serveur
cd /Docker/$USER/tp6/serveur
docker pull wiremock-studio
echo "docker run --tty --interactive --rm --memory=2G \
--volume ${PWD}:/home/wiremock -p 9000:9000 -p 8000-8100:8000-8100 \
wiremock-studio" > run_wiremock_studio.sh
chmod u+x run_wiremock_studio.sh
```

Ces commandes créent un sous-dossier `tp6/serveur` et font venir l'image Docker pour lancer `WireMock Studio`, et créent un script de lancement.

Si vous travaillez sur votre PC personnel, adaptez le dossier de travail en respectant l'arborescence dans `tp6`. Vous pouvez récupérer l'image avec `docker pull melnibon/wiremock-studio`.

2. On continue avec la commande suivante qui lance `WireMock Studio` :

```
./run_wiremock_studio.sh
```

Vous pouvez ouvrir un nouvel onglet du terminal (menu Fichier, Ouvrir un onglet). Il vous place dans le même dossier pour d'autres commandes.

3. Dans un navigateur, ouvrez http://localhost:9000. C'est l'URL de l'interface utilisateur.
4. Cliquez sur le bouton `Create new mock API`.

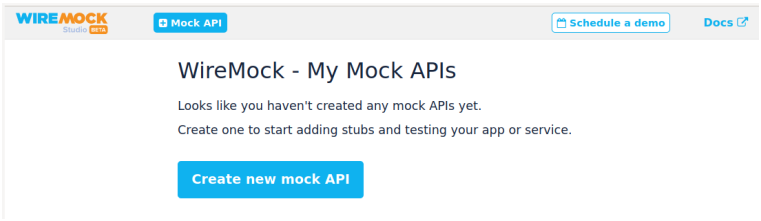

Figure 1: Bouton Create new mock API

Par défaut, le modèle d'API `blank` est sélectionné. C'est ce qu'il faut.

5. Donnez un nom à votre API : `livres`. Puis cliquez sur le bouton `Save`. Ça vous envoie sur la page de la figure 2.

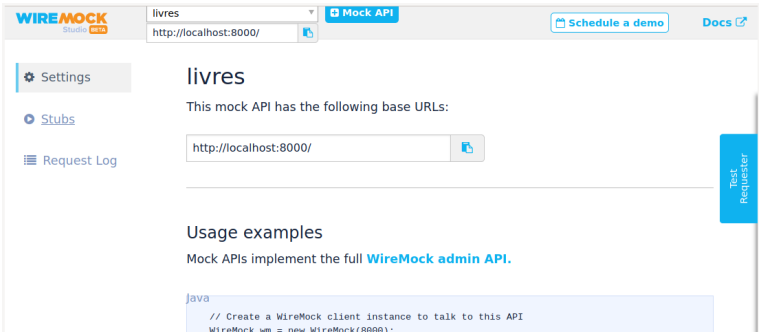

Figure 2: Accueil Settings

6. Cliquez sur `Stubs` dans le volet en haut à gauche pour voir la liste des *mappings*. Il n'y en a aucun au début.

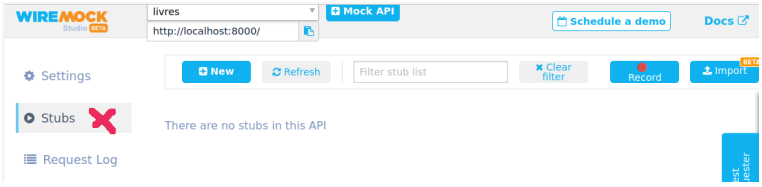

Figure 3: Accueil Stubs

On va en rajouter, mais avant, il faut comprendre la structure de l'interface.

3.1. Présentation de l'interface

Le bouton `+ New` ouvre le dialogue d'édition d'un *stub*, figures 4 et 5.

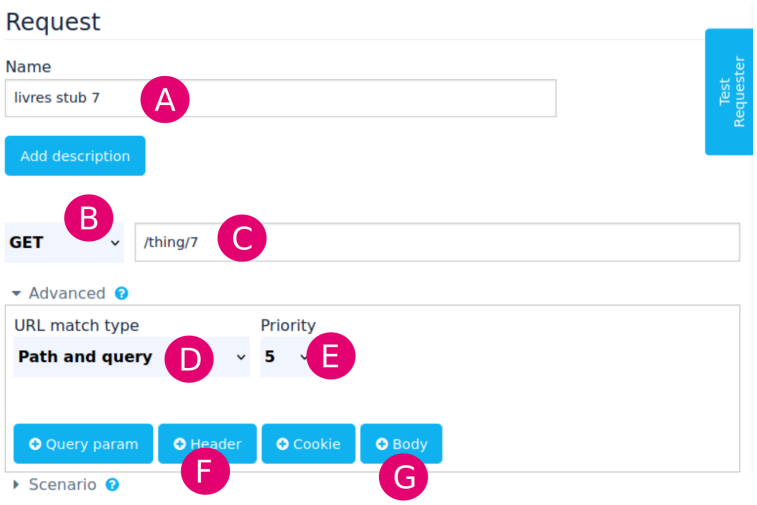

Figure 4: Édition d'un stub, partie Request

Ce dialogue est structuré en deux grandes parties. Il y a la partie `Request`, figure 4, où on spécifie quelles sont les conditions qui activent ce *stub*, par exemple sur quel URL et quelle méthode il s'active.

Pour commencer, on définit le nom du *stub* (A), puis (B) la méthode qu'il va gérer : `GET`, `POST`... Le dernier choix, `ANY` attrape toutes les méthodes sur le même URL.

À côté de la méthode, il y a le chemin (C). On peut l'écrire sous la forme d'une expression régulière comme `/livres/[0-9]+` ou en chaîne constante. Le chemin doit toujours commencer par un `/`.

En dessous, on doit choisir (D) comment faire la comparaison de l'URL reçu avec celui qui est dans (C). Si l'URL est fixe, on met `Path`. S'il y a une expression régulière, c'est `Path regex`.

En (E), on choisit la priorité. Laisser 5 par défaut. Il faut comprendre qu'on va définir plusieurs *stubs*, chacun chargé d'un URL et méthode. Chaque *stub* va donc examiner l'URL de la requête, la méthode de la requête, les paramètres (ce qui est après un `?` dans l'URL), les entêtes, les cookies et le corps, pour savoir si c'est lui qui la traite ou non. Les *stubs* sont rangés par priorités, mais c'est mal nommé, car plus cette priorité est faible, plus le *stub* est prioritaire. En fait, cette priorité, c'est l'ordre d'examen des *stubs* ; ceux de priorité 1 en premier. Donc 5 est une priorité moyenne.

En (F), vous pouvez ajouter des filtres sur les entêtes présents dans la requête, par exemple exiger qu'il y ait un entête `Accept` valant `application/xml`.

En (G), vous pouvez configurer un contenu fixe ou avec des jokers (*placeholders*).

Ensuite, il y a la partie `Response`, figure 5, où on indique ce que le stub renvoie quand la requête correspond aux conditions précédentes.

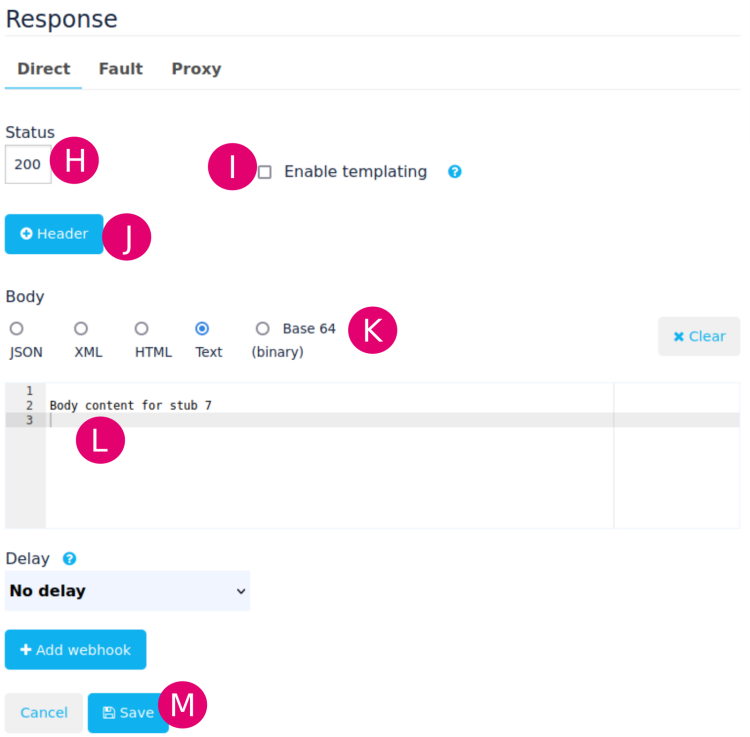

Figure 5: Édition d'un stub, partie Response

C'est nettement plus simple. On indique le code de réponse (H), en général 200 pour une réussite, et des codes en $4xx$ pour les erreurs.

En (I), on indique si la réponse contient des jokers ou non. En (J), on peut ajouter des entêtes de réponse. En (K), on choisit le format de la réponse, et donc en général, on ajoute un entête qui correspond au format, par exemple, `Content-type: application/xml` pour un corps XML. La réponse est à mettre en (L).

On valide avec le bouton d'enregistrement (M) qu'il ne faut surtout pas oublier. Quand un *stub* existe, on peut le cloner en cliquant sur le bouton `Clone` qui apparaît à côté de `Save`. C'est pour faire des variantes.

3.2. Création d'un premier *stub*

Voici maintenant la mise en pratique concrète. On veut créer un *stub* qui répond par un code 404 à toute requête qui n'a pas été traitée par un autre *stub*.

Voici ses spécifications :

- Nom : `Default404`
- Priorité : la plus faible, 10
- Requête
  - méthode: n'importe laquelle
  - chemin: n'importe lequel
- Réponse
  - code 404

– texte avec jokers : `Error: endpoint {{request.path}} or method {{request.method}} not found`

Comme ce *stub* a la plus faible priorité, 10, il ne sera examiné que si aucun des autres *stubs* n'a été reconnu. Son message affichera le chemin et la méthode qui sont incriminés.

Voici comment le créer.

1. Cliquez sur le bouton `+ New`
2. Changez toutes les informations comme ceci, voir le résultat figure 6 :
   a. Name: "`Default404`"
   b. Juste en dessous, sélectionnez `ANY` au lieu de `GET`, c'est pour dire que vous interceptez toutes les méthodes,
   c. À côté de `ANY` mettez `/.*`, pour intercepter tous les URLs,
   d. Dépliez `> Advanced` (ça sera toujours à faire)
   e. Sous `URL match type`, choisissez `Path regex` et mettez 10 pour Priority,
   f. Dans la partie `Response`, mettez `404` à la place de `200` pour Status,
   g. Cochez la case `Enable templating`
   h. Dans la partie `Body`, mettez "`Error: endpoint {{request.path}} or method {{request.method}} not found`" sur la ligne 1, et ne laissez aucune ligne vide en dessous (ça complique un peu les tests).
   i. Cliquez sur `Save` tout en bas.

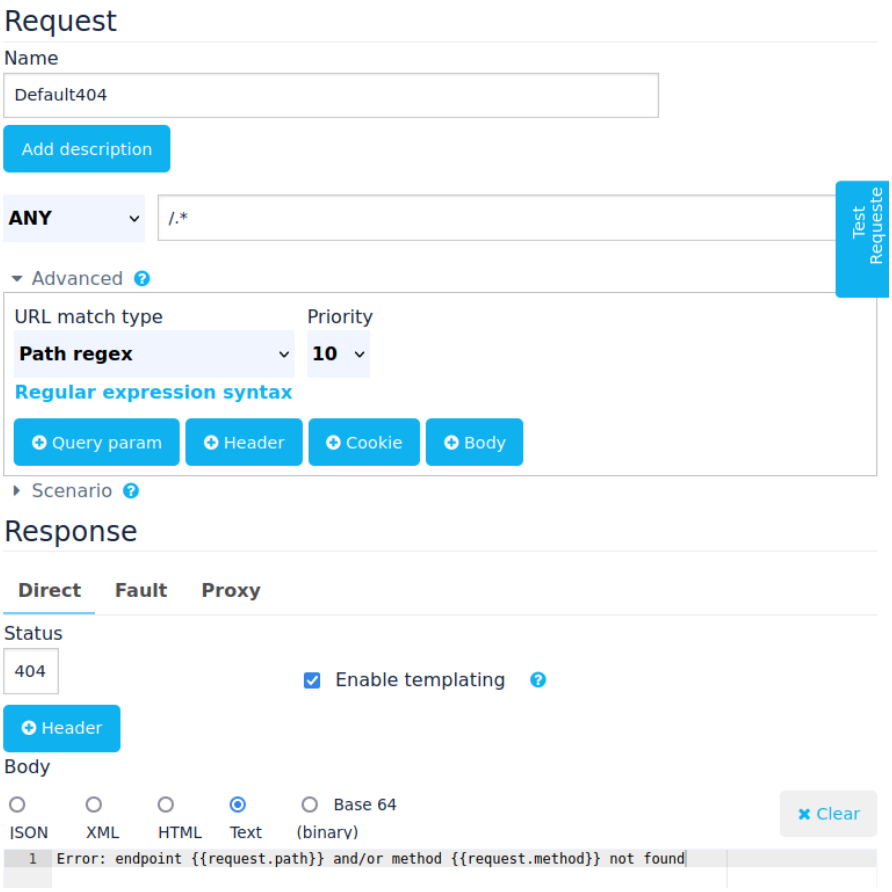

Figure 6: Stub Default404 (tronquée)

On va pouvoir le tester directement car `WireMock Studio` intègre un serveur pour ces *stubs*.

3. D'abord, il y a une petite configuration à faire à l'IUT pour que `cURL` n'utilise pas le *proxy*. Tapez la commande suivante dans un terminal, une seule fois suffit :

```
echo 'noproxy = localhost,127.0.0.1' > ~/.curlrc
```

3. Pour vérifier l'API, tapez les commandes suivantes, une par une, dans un terminal :

```
curl http://localhost:8000/nimportequoi
curl -X POST http://localhost:8000/vraimentnimportequoi
curl -X DELETE http://localhost:8000/encorenimportequoi
```

Vérifiez que vous récupérez chaque fois le code 404 avec le message prévu.

Vous constatez que le message est adapté à l'URL et la méthode demandée. C'est parce que ce message, spécifié dans la partie *Response body*, est un *template*. Vous avez coché *Enable templating*, donc les `{{request.champ}}` sont remplacées par leurs valeurs. Jetez un coup d'œil à la documentation des *templates* des réponses. Ce sont les variables qu'on peut mettre entre `{{}}` dans une réponse.

4. Dans `WireMock Studio`, ouvrez l'onglet `Request Log` du volet gauche et dépliez les requêtes pour voir comment ça s'est passé. Figure 7, on voit la demande initiale (URL, méthode, IP du client), ses entêtes HTTP, et la réponse du *stub* concerné.

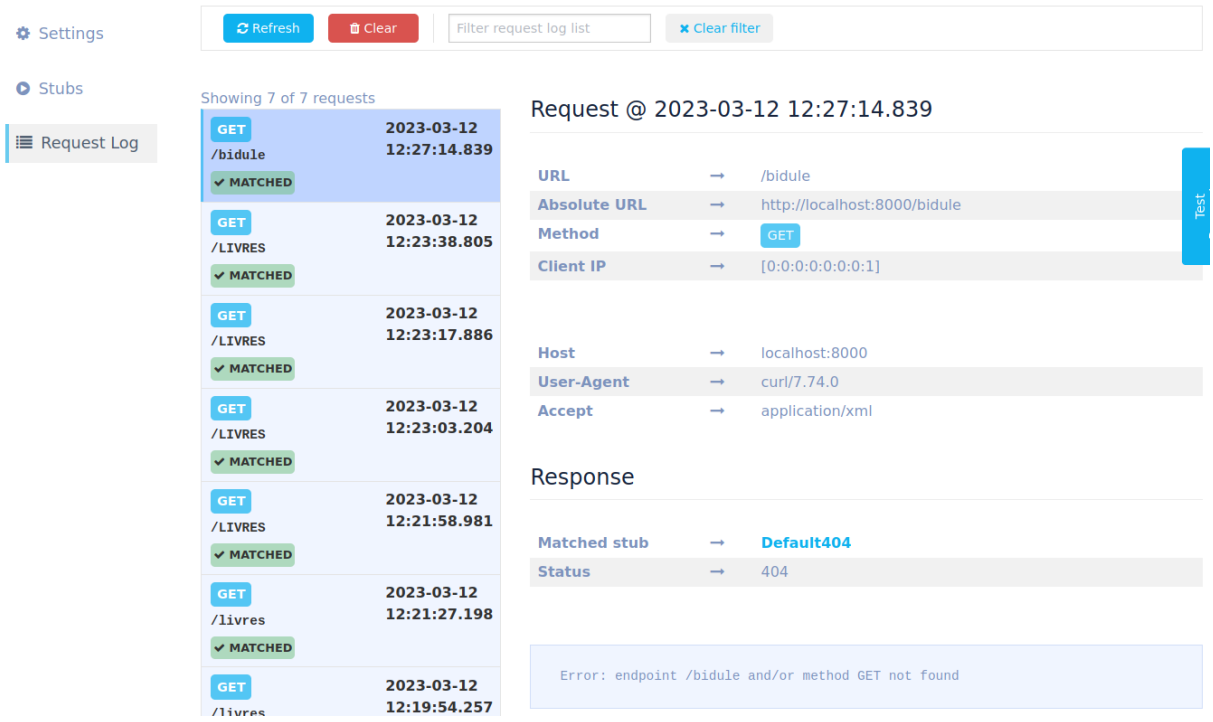

Figure 7: Requêtes sur le stub Default404

3.3. *Stub* `Get livres`

On va maintenant faire quelque chose de plus informatif, créer une terminaison pour `/livres` qui, au final, devra retourner une liste de livres en XML.

Voici ses spécifications (ce qui n'est pas spécifié doit rester dans l'état par défaut, par exemple la priorité à 5, le code à 200).

- Nom : `GET /livres`
- Requête
  - méthode: `GET`
  - chemin: `/livres`
  - header: `Accept: application/xml`
- Réponse
  - entête `Content-Type: application/xml`
  - corps XML, le contenu d'un fichier XML, voir ci-dessous.

Voici comment le créer.

1. Revenez dans la partie *Stubs*
2. Bouton `+ New` pour créer ceci, figure 8 :
   a. Name: "`GET /livres`"
   b. À côté de `GET`, mettez `/livres`
   c. Dépliez `> Advanced`, cliquez sur `+ Header`, et mettez `Accept` contains `application/xml`
   d. Dans la partie `Response`, cliquez sur `+ Header`, et mettez `Content-Type` et `application/xml`
   e. Dans `Body`, cochez `XML`
   f. Dans la zone de texte du bas, collez le contenu de livres.xml (à télécharger).
   g. Cliquez sur `Save` tout en bas.

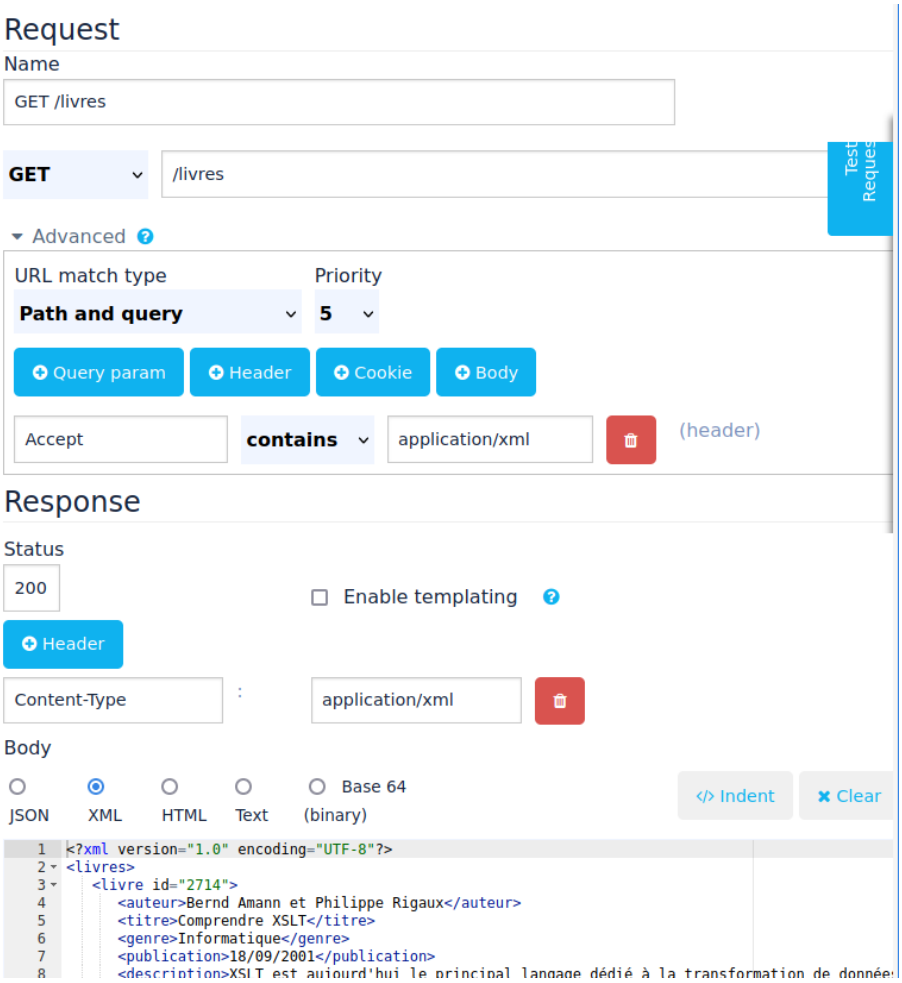

Figure 8: Requêtes sur le stub GET /livres (tronquée)

Tout ça signifie que le *stub* va accepter un `GET` sur l'URL `/livres` seulement si c'est du XML, et qu'on va lui répondre le document XML prévu.

3. Pour vérifier, tapez les commandes suivantes, une par une, dans un terminal. Seule la dernière doit passer, car elle valide toutes les conditions attendues par le *stub* :

```
curl http://localhost:8000/livres
curl -H 'Accept: application/xml' http://localhost:8000/LIVRES
curl -H 'Accept: application/xml' http://localhost:8000/livres
```

On voit arriver la bonne réponse quand tout est correct, méthode, URL et entête. Mais il manque quelque chose à ces tests manuels : la vérification détaillée de ce qu'on a reçu en réponse.

4. Robot Framework

Maintenant, on va se placer du côté du client et on va tester le serveur, en faisant comme si on ne savait pas que c'est une maquette. On va utiliser `Robot Framework` et une librairie pour faire des requêtes HTTP. Le cours en amphi a montré un autre usage de `Robot Framework`, pour tester une interface Web, mais ici, ça sera des réponses de requêtes sans interface graphique.

4.1. Mise en place du cadre et premier test

1. Ouvrez un autre shell, ou un autre onglet, et tapez les commandes suivantes :

```
mkdir -p /Docker/$USER/tp6/client/tests/files
cd /Docker/$USER/tp6/client
docker pull robot-selenium
echo "docker run --tty --interactive --rm --volume ${PWD}:/projet \
--add-host host.docker.internal:host-gateway robot-selenium" > run_robot.sh
chmod u+x run_robot.sh
```

Si vous travaillez sur votre PC personnel, respectez l'arborescence dans `tp6`. Vous pouvez récupérer l'image avec `docker pull melnibon/robot-selenium`.

2. Dans `tp6/client/tests`, créez un fichier appelé `TestBonjour.robot` contenant ceci :

```
*** Test Cases ***
Bonjour du robot
    Log  Bien le bonjour chez toi !
    Log to console  Vas voir dans le dossier reports
```

Attention à bien respecter la mise en page pour ceux qui s'obstinent encore à copier-coller depuis le pdf. Il y a exactement deux espaces entre le mot-clé `Log` et ses paramètres sur la 3e ligne, et exactement un entre les mots du message. Et sur la 4e ligne, ce sont des espaces uniques dans le mot-clé `Log to console` et deux pour le séparer du texte.

3. On va maintenant exécuter ce test :

```
./run_robot.sh
```

Le script `run_robot.sh` contient la commande et toutes les options pour lancer l'image `robot-selenium` facilement, et vous devriez voir à peu près ceci :

```
.../tp6/client$ ./run_robot.sh
==============================================================================
Tests
==============================================================================
Tests.TestBonjour
==============================================================================
Bonjour du robot                                .Vas voir dans le dossier reports
Bonjour du robot                                                        | PASS |
------------------------------------------------------------------------------
Tests.TestBonjour                                                       | PASS |
1 test, 1 passed, 0 failed
==============================================================================
Tests                                                                   | PASS |
1 test, 1 passed, 0 failed
==============================================================================
Output:  /projet/reports/output.xml
```

```
Log:     /projet/reports/log.html
Report:  /projet/reports/report.html
```

4. Ouvrez le fichier `reports/report.html` avec un navigateur, c'est là que `Robot Framework` a déposé son compte-rendu. En bas de la page, sous `Test Details`, cliquez sur l'onglet `All`, puis sur le nom `Bonjour du robot`. Vous aurez le détail des étapes qui ont réussi et celles qui ont échoué, et les raisons. Relisez le script de test pour tout comprendre et vous imprégner de la syntaxe.

4.2. Test du service REST livres

On va maintenant programmer de vrais tests sur l'API des livres. `Robot Framework` va émettre des requêtes sur l'API, à l'aide de la librairie `RequestsLibrary`, et comparer les réponses avec ce qui est attendu.

4.2.1. Réponses aux URLs incorrects

1. Dans `tp6/client/tests`, ajoutez un nouveau fichier `TestsDefault404.robot` : 

```
*** Settings ***
# doc: https://marketsquare.github.io/robotframework-requests/doc/RequestsLibrary.html
Library  RequestsLibrary

# session sur le serveur WireMock
Suite Setup  Create Session  SERVEUR  http://host.docker.internal:8000

*** Test Cases ***

GET sur un URL inconnu doit retourner 404
    ${response}=  GET on Session  SERVEUR  /nimportekoi  expected_status=404
    Status Should Be  404  ${response}
    Should Be Equal  ${response.text}
    ...  Error: endpoint /nimportekoi or method GET not found

POST sur un URL inconnu doit retourner 404
    ${response}=  POST on Session  SERVEUR  /bidule  expected_status=404
    Status Should Be  404  ${response}
    Should Be Equal  ${response.text}
    ...  Error: endpoint /bidule or method POST not found
```

Les `...` suivis de 2 espaces en début de ligne signifient que la ligne précédente continue là. C'est parce que la ligne est trop longue pour ce sujet de TP. L'indentation doit être respectée, donc ces points et deux espaces aussi.

2. Exécutez les tests avec `run_robot.sh`. Ils doivent réussir.
3. Consultez la documentation de RequestsLibrary. Regardez :
   - `Create Session` : sert à créer une connexion permamente (*keep-alive*) pour permettre une rafale de requêtes sur le même serveur,

- `GET on Session` : effectue un `GET` sur l'URL, dans le cadre de la session. Regardez bien ses paramètres. Ils sont séparés par deux espaces. On trouve le nom de la session puis le chemin, et enfin une option `expected_status` qu'on met seulement en cas de code en $4xx$ attendu. Il y a toutes les autres méthodes, comme `POST on Session`. Ces mots-clés retournent un objet qu'on met dans la variable `${response}`. Cet objet est documenté sur cette page.
- `Status Should Be` : pour vérifier le code de réponse, on donne le code attendu et ensuite la variable qui contient la réponse du `GET`.

4. Consultez la documentation des fonctions prédéfinies. Regardez :
   - `Should Be Equal` qui compare deux chaînes.
5. Ajoutez au moins deux autres tests, l'un pour `PUT`, l'autre pour `DELETE` sur un URL inconnu (différent à chaque fois).

4.2.2. Réponses aux requêtes sur `/livres`

1. Dans le dossier `tests`, copiez `TestsDefaut404.robot` en `TestsLivres.robot` et remplacez toute la partie `*** Test Cases ***` par ceci :

```
*** Variables ***

# pour ajouter le header Accept nécessaire
&{AcceptXML}=       Accept=application/xml,text/plain

*** Test Cases ***

GET /livres doit retourner 200
    ${response}=  GET on Session  SERVEUR  /livres  headers=&{AcceptXML}
    Status Should Be  200  ${response}
```

2. Ajoutez trois nouveaux tests qui vérifient que `POST`, `PUT` et `DELETE` sur `/livres` échouent. Copiez-les de `TestsDefaut404.robot`.

On sait maintenant que `GET /livres` répond, mais on va vérifier que la réponse possède la structure XML attendue. Il faut importer deux librairies supplémentaires et avoir un schéma XSD de validation XML.

3. Téléchargez le schéma livres.xsd (enregistrer la cible du lien sous...+renommer) dans le dossier `tests/files`. Corrigez-le pour qu'il valide le document que vous aviez mis dans le *stub* et qu'il soit le plus strict possible (attribut obligatoire, identifiant strictement positif).
4. Ensuite ajoutez ceci dans le script `TestsLivres.robot` :

```
*** Settings ***
... settings précédents à conserver...

# doc: https://robotframework.org/robotframework/latest/libraries/XML.html
Library     XML  use_lxml=True

# bibliothèque perso pour la validation d'un XML par un XSD
Library     XMLvalidation
```

```
*** Test Cases ***
...tests précédents à conserver...

GET /livres doit retourner une liste de livres XML valide
    ${response}=  GET on Session  SERVEUR  /livres  headers=&{AcceptXML}
    String Must Validate Schema  ${response.text}  tests/files/livres.xsd
```

Comme précédemment, remarquez la séparation des mots-clés et des paramètres, avec les deux espaces. Le mot-clé `String Must Validate Schema` est défini dans la librarie `XMLvalidation` dont vous ne trouverez la documentation nulle part, car c'est l'auteur de cet énoncé qui l'a programmée. Si ça vous intéresse, son code source est XMLvalidation.py.

5. Lancez le test avec `run_robot.sh`. Il doit réussir, sinon consultez `reports/report.html` : chemin incorrect, données XML mal mises dans le *stub*, invalides, etc.
6. Pour faire volontairement échouer le test et vérifier la validation, modifiez la réponse du serveur dans le *stub* `GET /livres`, par exemple ajoutez ou renommez un attribut, ou remplacez le numéro d'un livre par un mot au lieu d'un entier. N'oubliez pas de sauver chaque modification du *stub*. Revenez à la situation correcte à la fin.

Il reste une dernière sorte de vérification à faire, qu'il y a les valeurs attendues dans le document retourné par le serveur. La validation ne vérifie que la structure globale, mais pas les valeurs de la réponse. On fait cela avec XPath.

7. Ajoutez ce test dans `TestsLivres.robot` :

```
GET /livres doit retourner un livre appelé "Comprendre XSLT", id et genre
    ${response}=  GET on Session  SERVEUR  /livres  headers=&{AcceptXML}
    ${xml}=  Parse XML  ${response.text}
    Element Should Exist  ${xml}  //livre[titre="Comprendre XSLT"]
    Element Attribute Should Be  ${xml}  id  2714  //livre[titre="Comprendre XSLT"]
    Element Text Should Be  ${xml}  Informatique
    ...  //livre[titre="Comprendre XSLT"]/genre
```

Consultez la documentation de ces mots-clés dans la documentation de la librairie XML.

8. Idem : altérez la réponse du serveur dans la définition du *stub* pour que uniquement ce test échoue puis remettez dans l'état.

5. Nouvelle méthode et tests

On va maintenant rajouter une méthode `POST` sur la terminaison `/livres`. Dans toutes les API REST, cette méthode fait créer une nouvelle ressource dans la collection. La charge utile (*body*) de la requête HTTP contient la ressource. Dans ce TP, elle sera encodée en XML[1].

[1] C'est très souvent du JSON, mais la librairie JSON de RobotFramework n'est pas encore à la hauteur de ce qui existe pour XML, en validation et extraction d'informations.

5.1. Spécifications

La requête HTTP est émise par le client. Elle se fait sur l'URL `/livres`, avec la méthode `POST` et l'entête : `Content-Type: application/xml` Le *body* est donc encodé en XML et contient un seul élément `<livre>` avec ses sous-éléments, tel qu'un de ceux de l'exemple précédent.

Le serveur HTTP de l'API REST répond le code 201 si le livre a pu être inséré : *body* valide et nouveau livre (identifiant différent de ceux déjà présents dans sa BDD), ainsi qu'une réponse texte `Success: insert OK`. Dans le cas d'un envoi XML non valide, le serveur répond un code 400 avec le message `Error: bad data`. Si le livre existe déjà (identifiant déjà présent), c'est un code 409 avec le message `Error: data with same ID exists`.

5.2. *Stub* pour intercepter la requête

Quels nouveaux *stubs* ajouter pour simuler toutes les réponses à ces requêtes ? Le client va essayer de créer un vrai nouveau livre et aussi tenter de faire une collision d'identifiant, et aussi de fournir des données erronées. Ça fait trois situations à gérer.

Un point important, c'est qu'on va avoir trois *stubs* qui couvrent le même triplet (URL, méthode et autre), mais sur des conditions différentes. Donc il faut jouer sur les priorités pour que certains *stubs* répondent avant d'autres, ceux qui ont les conditions les plus restrictives en premier :

- priorité 4 : `POST /livres` avec un livre correct mais identifiant en double ⇒ code 409,
- priorité 5 : `POST /livres` avec un livre correct et sans doublon ⇒ code 201,
- priorité 6 : `POST /livres` sans condition sur le *body* ⇒ code 400.

Cet ordre vient des limitations sur les conditions qu'on peut tester.

Voici maintenant comment créer ces trois *stubs*. On commence par le premier.

1. Cliquez sur le bouton `+ New`
   a. Name: "`POST /livres (échec doublon)`",
   b. Sélectionnez la méthode et l'URL,
   c. Dans `> Advanced`, sous `URL match type`, choisissez `Path` et mettez 4 pour `Priority`,
   d. Cliquez sur `+ Header`, et mettez `Content-Type` contains `application/xml`,
   e. Juste en dessous, dans la partie `Body`, choisissez `matchesXPath` et dans la zone de texte mettez `/livre[@id=2714 or @id=4265]`. C'est une expression XPath qui compare l'identifiant fourni à ceux des deux livres retournés par `/livres`.
   f. Remplissez la partie `Response` conformément aux spécifications,
   g. Cliquez sur `Save` tout en bas.
2. Cliquez sur le bouton `Clone` qui est juste à côté de `Save`
   a. Name: "`POST /livres (succès)`",
   b. Mettez `Priority` à 5,
   c. Juste en dessous, dans la partie `Body`, choisissez `equalToXml`, cochez `Enable XMLUnit placeholders` et dans la zone de texte, mettez ce qu'il y a dans ce fichier. C'est la structure de ce que doit envoyer le client, avec des sortes de jokers, voir ci-dessous.
   d. Remplissez la partie `Response`,
   e. Cliquez sur `Save` tout en bas.
3. Cliquez sur le bouton `Clone` qui est juste à côté de `Save`.
   a. Name: "`POST /livres (échec données)`",
   b. Mettez 6 comme `Priority`,

c. Enlevez le header `Content-Type` avec le bouton poubelle,
d. Enlevez le `equalToXml` avec le bouton poubelle,
e. Remplissez la partie `Response`,
f. Cliquez sur `Save` tout en bas.

À propos des jokers XMLUnit, ils sont documentés sur cette page, mais ça semble encore expérimental. Il est très dommage que `WireMock Studio` ne propose carrément pas de valider par rapport à un schéma. Voici les possibilités :

- `${xmlunit.ignore}` correspond à toute chaîne vide ou non-vide,
- `${xmlunit.isDateTime}` valide une date/heure correspondant à différents formats, dont `yyyy-MM-dd` et `yyyy-MM-dd HH:mm:ss`,
- `${xmlunit.isNumber}` correspond à un nombre quelconque entier, décimal ou en notation scientifique,
- `${xmlunit.matchesRegex(expression)}` correspond à l'expression régulière indiquée. Ce sont les expressions régulières Java. Par exemple, pour reconnaître un mot : `${xmlunit.matchesRegex(^\w+$)}`, un nombre entier : `${xmlunit.matchesRegex(^\d+$)}`, une chaîne non vide : `${xmlunit.matchesRegex(^.+$)}`. Pour en savoir plus, voir ce tutoriel. NB: ne pas copier-coller à cause du ^ transformé en `^`.

4. Améliorez la validation des charges utiles à l'aide d'expressions régulières, c'est à dire remplacez les `${xmlunit.ignore}` et autres par mieux que ça.

5.3. Tests de `POST /livres`

Vous devez ajouter de nouveaux tests pour essayer les trois situations décrites ci-dessus. Là aussi, ce n'est pas simple. Pour chaque test, il faut fabriquer une charge utile spécifique allant du bon livre aux mauvais possibles.

1. Voici les ajouts à faire dans `TestsLivres.robot` :

```
*** Variables ***

# pour ajouter les headers nécessaires
&{AcceptXML}=       Accept=application/xml,text/plain
&{ContentTypeXML}=  Content-Type=application/xml

# livres à ajouter, bons et mauvais
${livre9876}=        <livre id="9876">
                ...    <auteur>inconnu</auteur>
                ...    <titre>OuiOui et NonNon</titre>
                ...    <genre>Jeunesse</genre>
                ...    <publication>11/12/2013</publication>
                ...    <description>OuiOui fait une rencontre.</description>
                ...  </livre>
${livre4265}=        <livre id="4265">
                ...    <auteur>Milou</auteur>
                ...    <titre>Ouaf</titre>
                ...    <genre>Animalier</genre>
                ...    <publication>18/09/2000</publication>
```

```
                ...    <description>Grr grr</description>
                ...  </livre>
${livrebad}=        <livre id="9876">
                ...    <titre>Ouaf</titre>
                ...    <genre>Animalier</genre>
                ...  </livre>

*** Test Cases ***

... conserver les tests précédents sauf l'échec sur POST /livres ...

POST /livres valide doit retourner 201
    ${response}=  POST on Session  SERVEUR  /livres
    ...  headers=&{ContentTypeXML}  data=${livre9876}
    Status Should Be  201  ${response}
    Should Be Equal  ${response.text}
    ...  Success: insert OK
```

La charge utile d'un `POST` est spécifiée par l'option `data`.

2. Rajoutez les deux autres tests qui vont essayer l'un `data=${livre4265}`, et l'autre `data=${livrebad}`. N'oubliez pas le paramètre `expected_status` spécifique.

6. Nouvelles terminaisons et tests

On va maintenant rajouter deux nouvelles terminaisons pour apprendre un dernier aspect de `WireMock`. On veut que le serveur REST réponde à `/livres/`*id*, c'est à dire des chemins ayant une partie variable, l'identifiant du livre voulu.

6.1. *Stub* `GET /livres/{id}`

1. Dans la partie centre gauche, cliquez sur le *stub* `Get /livres`. Ça affiche toutes ses informations. Allez tout en bas et cliquez sur le bouton `Clone` et ensuite, modifiez la copie comme ceci.
   a. Name: `GET /livres/{id}`
   b. À côté de `GET`, mettez `/livres/[0-9]+` C'est la syntaxe qui permet d'intercepter les chemins portant un numéro valide défini par une expression régulière.
   c. Dans `Advanced`, choisissez `Path regex` dans `URL match type`
   d. Modifiez le document retourné dans `Response`, `Body`, au lieu de la liste de deux livres, vous allez faire retourner un livre aléatoire. Mettez ce qu'il y a dans ce fichier. Il emploie des jokers `{{...}}` qui sont documentés ici et là.
   e. Cliquez sur `Save` tout en bas.
2. Clonez ce *stub* en `GET /livres/{id} (mauvais id)` pour qu'il réponde une erreur 406 à un identifiant incorrect, non entier. L'expression régulière devient `/livres/[^/]+` (au moins un caractère quelconque sauf un `/`).
3. Pour tester rapidement, essayez successivement :

```
curl -H 'Accept: application/xml' http://localhost:8000/livres/12
curl -H 'Accept: application/xml' http://localhost:8000/livres/123456
curl -H 'Accept: application/xml' http://localhost:8000/livres/-2
curl -H 'Accept: application/xml' http://localhost:8000/livres/abc
```

Les deux premières réussissent en retournant des valeurs aléatoires, les deux dernières échouent. Il faut maintenant écrire les tests qui vont vérifier ça.

6.2. Tests

1. Dans le dossier `tests` du client, copiez `TestsLivres.robot` en `TestsLivresNNN.robot`. Laissez la partie `Settings`, mais élaguez le reste pour arriver à ceci :

```
*** Variables ***

# pour ajouter les headers nécessaires
&{AcceptXML}=       Accept=application/xml,text/plain

*** Test Cases ***

GET /livres/NNN doit retourner 200
    Fail  PAS FAIT

GET /livres/NNN doit retourner un livre XML valide
    Fail  PAS FAIT

GET /livres/NNN avec un mauvais numéro doit retourner 406
    Fail  PAS FAIT

POST /livres/NNN doit retourner 404
    Fail  Faire comme dans les autres tests
```

2. Votre travail consiste à compléter les tests pour mettre à l'épreuve le serveur REST. Vous devez aussi rajouter les tests pour vérifier les échecs de `GET` quand l'identifiant du livre n'est pas bon.

7. Nouvelles méthodes et terminaisons

- Rajoutez une terminaison nommée `GET /livres/{id}/{champ}`, qui répond à un `GET` sur `/livres/NNN/CCC` avec `NNN` étant un identifiant de livre et `CCC` un nom de champ parmi `auteur`, `titre`, `genre`, `publication` et `description`. Elle retourne un élément du type `CCC` demandé, contenant un texte aléatoire, par exemple si `CCC` vaut `titre`, elle peut retourner : `<titre>pkztlt</titre>`.
  Comme pour `GET /livres/{id}` dont elle s'inspire, elle retourne 200 si tout est correct et 406 si l'identifiant est incorrect. Le code 404 est retourné pour une demande qui ne correspond pas du tout.
  La difficulté est de faire reconnaitre le chemin. Il faut écrire une expression régulière qui admet certaines valeurs seulement. Sa syntaxe est `(val1|val2|...)`.

- Rajoutez une terminaison qui répond à un `DELETE` sur `/livres/NNN` qui retourne un succès, 200, quand le numéro est compris entre 100 et 999. Il retourne 406 si le numéro est en dehors de cette plage mais quand même un entier, et 400 si le numéro n'est pas un entier. Pour cela, il faut trois *stubs*, dont le plus prioritaire a une expression régulière plus précise que `[0-9]+`. N'oubliez pas de retirer les entêtes `Accept` inutiles.
- Rajoutez une terminaison qui répond à un `PUT` sur `/livres/NNN`. Le rôle de cette terminaison est de remplacer la donnée identifiée par l'URL. Donc il doit y avoir une vérification sur l'identifiant `NNN` de l'URL ; cet identifiant doit être connu, soit 2714, soit 4265. D'autre part, les données de la charge utile doivent être valides. La réponse est 201, `Success: replace OK` si tout est correct ; 406, `Error: no data with this ID exists` si le `NNN` de l'URL n'est pas l'un des deux connus ; 400, `Error: bad data` dans les autres cas.
- Rajoutez une terminaison qui répond à un `PUT` sur `/livres/NNN/CCC` accompagné d'un élément XML donnant une nouvelle valeur au champ `CCC` comme plus haut. Code 400 si le XML n'est pas valide, 201 si valide. Vous aurez peut-être une collision avec les *stubs* précédents s'ils n'ont pas une condition de filtrage suffisamment claire.
  Il y a un souci technique pour valider la charge utile dans le *stub*. Ça doit être un élément XML simple, contenant un texte non vide, par exemple `<auteur>Untel</auteur>`. Le problème est que l'élément est défini par `CCC`, variable d'une requête à l'autre. Une manière de faire est d'utiliser XPath : `/*[name()='titre' or name()='auteur']`, à compléter pour les autres valeurs et avec un test pour que le contenu de l'élément ne soit pas vide.

Au bout d'un certain nombre de *stubs*, on se rend compte que l'interface de `WireMock Studio` n'est pas très ergonomique. On a une longue énumération qui semble difficile à maîtriser. Il est important de bien nommer les *stubs*, afin de pouvoir utiliser le bouton de filtrage. Comment faudrait-il mieux organiser cette interface ?

8. Rendu du TP

Vous devrez remettre votre travail à la fin de chaque séance. C'est ainsi que c'est noté, en contrôle continu.

Ouvrez un terminal bash : 

```
cd /Docker/$USER
tar cfvz tp6.tgz tp6
```

Puis déposez le fichier `tp6.tgz` dans la zone de dépôt du TP6 sur Moodle R4.02 Qualité de développement.

ATTENTION votre travail est personnel. Si vous copiez pour quelque raison que ce soit le travail d'un autre, vous serez tous les deux pénalisés par un zéro.

En cas de souci quelconque, envoyez un mail à `pierre.nerzic@univ-rennes1.fr` pour expliquer le problème.