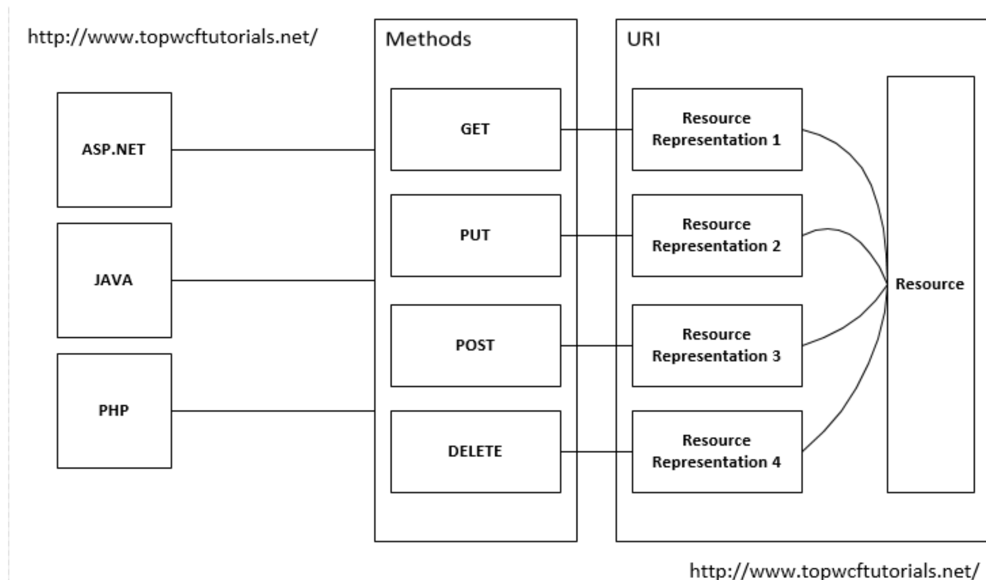
Université Pierre Mendès-France
Sciences sociales & humaines

Service Oriented Programming: **Developing a RESTful Service**

In Service Oriented Programming a RESTful service is a service conforming to the REST (**Re**presentational **S**tate **T**ransfer) architecture style. In this kind of architecture (i) you think in terms of *resources* and their *representations* (instead of thinking about operations, inputs and outputs), and (ii) you address and transfer *resource states* using the URI and HTTP standards.

When a service follows the REST architecture, every client knowing how to use HTTP can access the states and representations of the resources of a RESful service (cf. figure below). In this exercise you will develop a RESTful service using Javascript and NodeJS for exposing a music catalogue as a set of resources.



Requirements

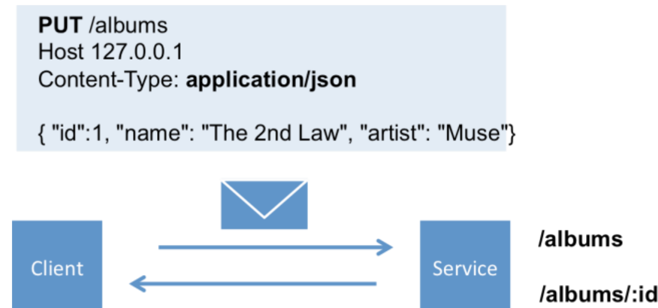
- NodeJS
- Express (NodeJS module)
- cURL or Postman (Chrome extension)
- CouchDB
- MusicCatalog REST Service and Client (distributed on course)

Objectives

- Implement a RESTful service and understand the use of HTTP methods for interacting with it.
- Develop an application that uses your RESTful service.
- Use a NoSQL database for storing the RESTful service data.

Music catalog overview

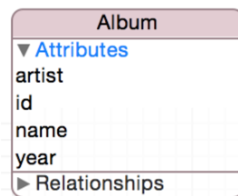
For this exercise you will use the **Music Catalog** application. The following figure illustrates its architecture (REST service and client parts).



As shown in the figure, the Music Catalog service exposes two resources:

- **/albums** — represents a music catalog as a collection of albums.
- **/albums/:id** — represents an album identified by **:id**.

The figure also shows that clients and service exchange information about albums using the **JSON** data representation. The following diagram the structure representing an album instance.



Service behaviour

In order to implement a RESTful service you have to specify *what to do* when a client executes an HTTP operation *over* one of your resources. In web programming this is called **routing** (i.e. the association of a HTTP operation with a resource).

The table below summarizes the semantic of HTTP operations when executed over the Music Catalog resources.

URL	Method	Parameter	Description
http://localhost:3000/albums	GET	-	List all the albums
	PUT	Album	Add a new album
	POST	-	Unused
	DELETE	-	Unused
http://localhost:3000/albums/:id	GET	-	Get the album
	PUT	-	Update the album
	POST	-	Unused
	DELETE	-	Delete the album

In the Music Catalog REST service, these semantics are implemented by the routings specified in the **CatalogService.js** file. The content of this file is shown below.

Université Pierre Mendès-France Grenoble
Sciences sociales & humaines

```
var express    = require('express');
var bodyParser = require('body-parser');

var app = express();
app.use(bodyParser.json());

var albums = {};
albums['1'] = { "id":1, "name": 'The 2nd Law', "artist": 'Muse', "year": 2012 };

app.get('/albums', function (request, response) {
   response.json(albums);
});

app.get('/albums/:id', function (request, response) {
   response.json(albums[request.params.id]);
});

app.put('/albums/:id', function (request, response) {
   var album = request.body;
   albums[request.params.id] = album;
   response.json('OK');
});

app.delete('/albums/:id', function (request, response) {
   var deleted = delete albums[request.params.id];
   response.json(deleted);
});

var server = app.listen(3000, function () {
   console.log('Listening at http://localhost:%s', server.address().port);
});
```

As you can see, the catalog service defines routes by calling the GET/PUT/POST/DELETE methods of an *express object* (see Express for more information). For instance, the following code snippet illustrates *how to retrieve a specific album* using its ID by **routing** the **GET** operation to the resource **/albums/:id**.

```
app.get('/albums/:id', function (request, response) {
   response.json(albums[request.params.id]);
});
```

Note that the ID of an album is *resolved dynamically* by matching the request' URI with the string **/albums/:id**. Also note that you can access this value at runtime by using the object **request.params**. Finally note that, before sending information back to the client (i.e., the set of albums), the service transforms the *JavaScript object* containing the album information (**albums[request.params.id]**) into its *JSON representation* (**response.json(OBJECT)**).

The following code snippet illustrates how to *add a new album to the catalog* by **routing** a **PUT** operation to the resource **/albums/:id**.

```
var app = express();
app.use(bodyParser.json());

app.put('/albums/:id', function (request, response) {
   var album = request.body;
   albums[request.params.id] = album;
```

```
    response.json('OK');
});
```

In this example the service retrieves the information about the new album by accessing the HTTP request body (**request.body**). Then, as in the previous example, it uses the *id* specified in the *album resource* (**request.params.id**) in order to store the album into the music catalog (**albums[request.params.id] = album**).

Recall that client and service send albums' information using the JSON data representation. This implies that you have to **ALWAYS** transform a JavaScript object into JSON (and vice versa) in order to communicate. Note that in the previous example Express converts automatically the **request.body** to Javascript because it assumes that all body requests are represented in JSON (**app.use(bodyParser.json())**).

TESTING THE SERVICE CATALOG

Run the Music Catalog service by executing the following command inside the Music Catalog application folder:

```
node CatalogService
```

Once running, test the service by issuing the following HTTP requests with cURL (or Postman).

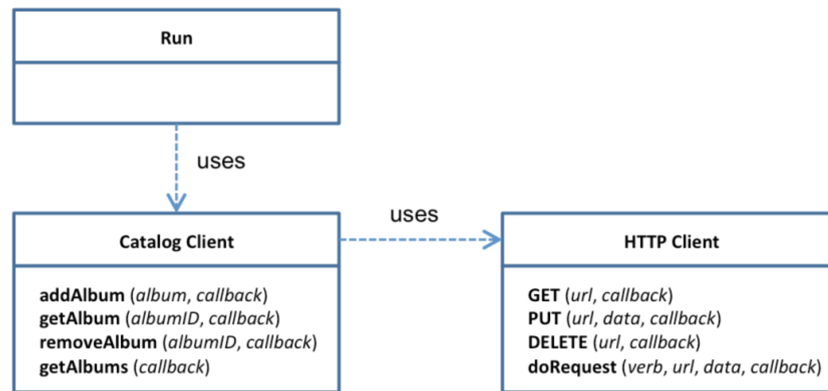
```
# GET all albums
curl localhost:3000/albums

# Add new album
curl -X PUT localhost:3000/albums/2 -H "Content-Type: application/json" -d @-
{
  "id": 2,
  "name": "The Resistance",
  "artist": "Muse",
  "year": 2009
}
Press CTRL + D

# GET all albums
curl localhost:3000/albums
```

SERVICE CLIENT

Until now you have used a third-party tool for interacting with the Music Catalog service (e.g., Web Browser, cURL, Postman). In this section you will use the Catalog Client (a JavaScript application) for sending HTTP request to the Catalog Service.

As shown in the figure below, the Catalog Client is composed of the following classes:

- **HttpClient** — Offers basic operations for executing HTTP requests. The HTTP client assumes that all data is sent using JSON as the data format representation.
- **CatalogClient** — Offers operations for adding, getting, removing the albums in the Music Catalog. These operations use the HTTP Client.
- **Run** — Script illustrating the use of the Catalog Client.



HTTP client

The following snippet shows the code of the HTTP Client class (cf. **HttpClient.js** file).

```
this.GET = function(url, callback) {
   this.doRequest('GET', url, null, callback);
};

this.PUT = function (url, data, callback) {
   this.doRequest('PUT', url, data, callback);
};

this.DELETE = function (url, callback) {
   this.doRequest('DELETE', url, null, callback);
};

this.doRequest = function(verb, url, data, callback) {

   var options = {
      hostname: URL.parse(url).hostname,
      port:    URL.parse(url).port,
      path:    URL.parse(url).path,
      method:  verb,
      headers: {
         'Content-Type': 'application/json',
         'Content-Length': (data)? data.length: 0
      }
   };

   var req = http.request(options, function(res) {…});

   req.write( (data)? data: '');
   req.end();

};
```

Note that all HTTPClient operations (GET/PUT/DELETE) depend on the **doRequest** function, which is in charge of preparing the HTTP message: *header* (**options**) and *body* (**req.write(data)**). Also note that **doRequest** assumes that all content (**data**) sent by the HTTPClient is represented in JSON (**‘Content-Type: application/json’**).

Catalog client

The following snippet illustrates the use of the **HttpClient** for *adding* and *getting* an album (cf. **CatalogClient.js**).

```
var ALBUMS_URI = 'http://localhost:3000/albums';
var ALBUM_URI  = 'http://localhost:3000/albums/{_}';
var http = new HttpClient();

this.addAlbum = function (album, callback) {
   var url  = ALBUM_URI.replace('{_}', album.id);
   var data = JSON.stringify(album);
   http.PUT(url, data, callback)
};

this.getAlbum = function (albumID, callback) {
   var url = ALBUM_URI.replace('{_}', albumID);
   http.GET(url, function (resp) {
      var album = JSON.parse(resp.body);
      callback(album);
   });
};
```

Note that before adding a *new album* into the catalog (using **http.PUT**), the **addAlbum** function converts the album JavaScript object into JSON using the **JSON.stringify()**. In a similar way, when the **getAlbum** function receives the response from the service, it uses the **JSON.parse** function for converting the **response body** (sent in JSON format) into a JavaScript object.

Run script

Finally the following code illustrates the use of the CatalogClient class for *adding, retrieving and removing* albums programmatically (cf. **Run.js** file).

```
/// Step 0: Show the albums in the catalog
.then(function (next) {
   client.getAlbums(function (albums) {
      console.log('\n'+'ALBUMS (before insertion)');
      console.log(albums);
      next();
   });
})
/// Step 1: Insert new album in Catalog
.then(function (next) {
   var album = { "id": 2, "name": 'The Resistance', "artist": 'Muse', "year": 2009 };
   client.addAlbum(album, function (resp) {
      next();
   });
})
/// Step 2: Insert another album (with errors)
.then(function (next) {
   var album = { "id": 3, "name": 'XXXX', "artist": 'YYYY', "year": 0000 };
   client.addAlbum(album, function (resp) {
      next();
   });
})
/// Step 3: Show the albums in the catalog
.then(function (next) {
   client.getAlbums(function (albums) {
      console.log('\n'+'ALBUMS (after insertions)');
      console.log(albums);
      next();
   });
})
```

```
/// Step 4: Remove the album with errors
.then(function (next) {
   client.removeAlbum('3', function () {
     next();
   });
})
/// Step 5: Show (again) the albums in the catalog
.then(function (next) {
   client.getAlbums(function (albums) {
      console.log('\n'+'ALBUMS (after deletion)');
      console.log(albums);
      next();
   });
});
```

To Do

For this exercise you have to:

- Describe the characteristics of the Catalog REST service. For instance, it is stateful/stateless, persistent/non-persistent?
- Implement a Deezer Client using the HTTPClient class for *searching* and *retrieves* information about albums using the Deezer RESTful Service.
- Modify the Catalog Service in order to store the Catalog in CouchDB.
- Define a CouchDB view for retrieving the URLs of the albums' covers (see CouchDB: The definitive guide).