Server-side Web Development

Unit 8. Web services.

Fidel Oltra, Ricardo Sánchez

IES Jaume II El Just
Tavernes de la Valldigna
Departament d'Informàtica
Curs 2024-25

Index

1 Web services

A web service is a method for exchanging messages between client and server applications. The web service includes a specific set of methods that can be invoked from the client application. These methods usually return data in a standard format, so the client application, whatever the language or technology used, would be able to read them. A web service is a resource commonly used for this purpose: sending requests to a web server and then receiving data in a response form.

In practice, any software, application, or cloud technology that uses standard web protocols like HTTP/HTTPS and interoperates with other apps, exchanging messages in formats like XML or JSON, for example, can be a web service.

1.1 Advantages of a web service

- Allows programs developed in different languages to connect each other and exchange data
- A web service works both in Internet and Intranet networks
- Independence from the language, operating system or technology
- Easy access
- Allows different sources to access to an unique database

1.2 Types of web services

Basically we can create a web service using two main methods: * SOAP * REST

1.2.1 SOAP

SOAP is an acronym for **Simple Object Access Protocol**. SOAP is a messaging protocol for exchanging structured information between a server and a client. It uses XML for the message format and works mainly over the HTTP protocol, although SMTP, TCP or UDP can be used too. SOAP allows developers to invoke processes running on different operating systems, so the client can receive responses from different languages, operating systems and platforms.

To work, SOAP needs a *Web Services Description Language* (**WSDL**) method definition file. WSDL contains methods, parameters, and all other information needed to send web service requests.

Nowadays SOAP is less popular than in previous years. This is because of the increasing use of REST.

1.2.2 REST

REST is an acronym for **REpresentational State Transfer**. REST is a software architectural style that was in the roots of the World Wide Web. It defines some constraints and principles to be followed when creating a web service.

REST web services can generate response information both in **JSON** and **XML** format. REST supports all the main four HTTP methods (POST, GET, PUT and DELETE) for CRUD (Create, Read, Update and Delete) operations.

METHOD	FUNCTION
GET	Retrieve information from the service
POST	Create an new resource
PUT	Update a particular resource
DELETE	Remove a particular resource

1.3 API REST

An **API** (**Application Programming Interface**) is a set of definitions and protocols for building and integrating application software. Usually it's defined as a kind of contract between the server and the client for exchanging data.

An **API REST** is an API that conforms to the requirements of REST architectural style and allows interaction between clients and web services.

An API is considered REST if it has these conditions:

- A client-server architecture
- HTTP protocol
- Independence between transactions
- A standard interface and a standard form for the transferred information
- Internal details are transparent for the client

1.4 HTTP response status codes

When an API returns a response, it adds an HTTP response status code. **HTTP response status codes** indicate whether a specific HTTP request has been successfully completed. Responses are grouped in five classes:

- Informational responses (100 - 199)
- Successful responses (200 - 299)
- Redirection messages (300 - 399)
- Client error responses (400 - 499)
- Server error responses (500 - 599)

The most common codes are:

- **200 OK**: Request successful. The result meaning of "success" depends on the HTTP method. For a GET request usually means that the resource has been fetched and transmitted in the message body.
- **201 Created**: The request succeeded, and a new resource was created as a result. This is typically the response sent after POST and PUT requests.
- **400 Bad Request**: The server cannot process the request due to a client error (e.g., malformed request syntax, invalid parameters…).
- **401 Unauthorized**: Means that the client is "unauthenticated". The client must authenticate itself to get the requested response.
- **404 Not Found**: The server cannot find the requested resource.
- **500 Internal Server Error**: The server has encountered a situation it does not know how to handle.

Complete HTTP status codes

2 Working with JSON files in PHP

As you surely know, **JSON** stands for **JavaScript Object Notation**, and is a standard for storing and exchanging data in a text-based format. Thus, the JSON files can be easily exchanged between the server and the client and they can be used by any programming language. Examples of JSON variables:

```
{"name":"Joe", "age":"25"}

[
    {"name":"Joe", "age":"25"},
    {"name":"Jane", "age":"22"}
]
```

We can convert a PHP object into an JSON object by using the `json_encode()` function:

```
$book = new Book();
$book->setTitle("...");
$book->setYear("...");
...
$jsonBook=json_encode($book);
```

The same function can be used to transform an array into a JSON object:

```
$person = array("Name"=>"John","Age"=>36,"City"=>"Tavernes");
$jsonPerson = json_encode($person);
```

Let's see an example:

```
#[Route("/jsoncheck", name: "jsoncheck")]
public function jsoncheck(): Response {
    $person = array("Name"=>"John","Age"=>36,"City"=>"Tavernes");
    $jsonPerson = json_encode($person);
    return new Response("
        <html>
        <body>
        <h2>$jsonPerson</h2>
        </body>
        </html>
    ");
}
```

When we go to `http://localhost:8080/jsoncheck` we get:

Figure 1: Showing a JSON object in PHP

By using the function `json_decode()` we can convert a JSON object into a PHP object or into an associative array. The function returns an object by default. If we add a second parameter set to true, then the result will be an associative array.

```php
$jsonString='{"Name": "Peter","Phone": "123123","City": "Tavernes"}';
$phpObject=json_decode($jsonString); // returns an object
$array=json_decode($jsonString,true); // returns an associative array
```

3 Building the API REST

In this example we will make a REST web service based on the Provider application. It follows the last guided example.

In the databases unit we made entity classes and the database access functionalities. These parts will remain the same, the only thing we have to do is to add API REST functionality.

3.1 Transforming objects to JSON

As we need the objects we send and get via the API to be encoded as JSON, we will do this task in the entity class.

First, we will encode a login object as an array to pass it later to `json_encode`. Go to the Provider class and add the next method:

```php
public function toArray(): array
{
    return array(
        'id' => $this->getId(),
        'name' => $this->getName(),
        'email' => $this->getEmail(),
        'cif' => $this->getCif()
    );
}
```

As you can see, this function transforms a login object in an array.

3.2 Making the Response

To return a response we will create a **Response** class with a static method called `result`:

```php
<?php
declare(strict_types=1);

class Response
{
    public static function result(int $code, array $response): bool|string {
        header("Content-type:application/json; charset=utf-8");
        http_response_code($code);
```

```
        return json_encode($response);
    }
}
```

In the `header` function we are telling that the response will have content of type `application/json` and `utf8` codification.

In the function `http_response_code` we pass the HTTP status code (200, 400, etc).

Finally, the `json_encode` returns a JSON object from the `$response` array or `false` if `$response` has errors.

3.3 Making the services

To serve the API, we create a new script called `api.php`. In it we import the needed classes:

```
<?php
require_once __DIR__ . '/models/Provider.php';
require_once __DIR__ . '/models/ProviderRepository.php';
require_once __DIR__ . '/models/Response.php';
```

All the logic will be managed inside a `switch` statement that gets the request method via the `$_SERVER` superglobal:

```
switch($_SERVER['REQUEST_METHOD']) {
```

The `'REQUEST_METHOD'` key stores the method used by the client to make the request.

3.3.1 Getting the logins (GET)

In the **case** of a GET request we have 2 cases: a request of a single login by its `id`, or a request of all the logins.

First we get the query string params with the `parse_string` function. It parses the query variables into the array passed as the second argument.

```
parse_str($_SERVER['QUERY_STRING'], $params);
```

If the client is requesting a single login, then it will set a parameter with the name 'id', so we can know if the request is for a single element. Then, we retrieve the requested login by its id, convert it to an array and create a response array with the data and the result value 'OK', that indicates the request has been successful:

```
if(isset($params['id'])) {
    //Get by id
    $id=trim(strip_tags($params['id']));
    $provider = ProviderRepository::select($id);
    if($provider) {
        $response = array('result'=>'OK', 'data'=>$provider->toArray());
    } else {
        $response = array('result'=>'Error', 'data'=>'Provider not found');
    }
}
```

If we don't have an `id` param, the we get all the elements. Here, we are using the `toArray` method inside a foreach loop in order to transform all the objects in arrays. Another method is to use the `PDO::FETCH_ASSOC` constant in the `ProviderRepository` class select methods, but for this example we won't do it.

```
if(isset($params['id'])) {
    //Get by id
    ...
} else {
    //Get all
    $result = ProviderRepository::getAll();
    $resultArray = [];
    foreach ($result as $object){
        $resultArray[] = $object->toArray();
    }
    $response = array('result'=>'OK', 'data'=>$resultArray);
}
```

Finally, call the `Response::result` static method and echo the response.

```
echo Response::result(200,$response);
```

The full code block:

```php
case 'GET':
  //parses the query variables into the params array
  parse_str($_SERVER['QUERY_STRING'], $params);

  if(isset($params['id'])) {
    //Get by id
    $id=trim(strip_tags($params['id']));
    $provider = ProviderRepository::select($id);
    if($provider) {
        $response = array('result'=>'OK', 'data'=>$provider->toArray());
    } else {
        $response = array('result'=>'Error', 'data'=>'Provider not found');
    }
  } else {
      //Get all
      $result = ProviderRepository::getAll();
      $resultArray = [];
      foreach ($result as $object){
          $resultArray[] = $object->toArray();
      }
      $response = array('result'=>'OK', 'data'=>$resultArray);
  }
  echo Response::result(200,$response);
  break;
```

We can check the request simply writing the URL in a browser. For the url `../api.php`:

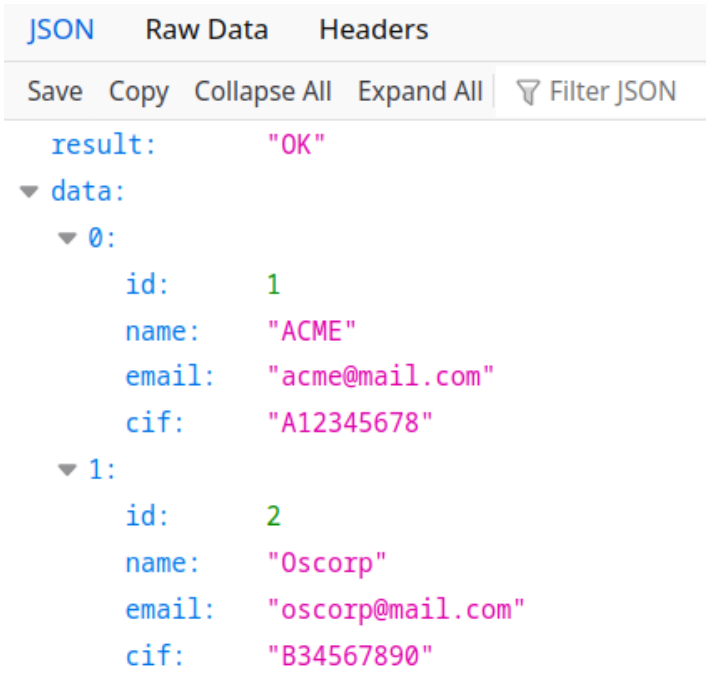

Figure 2: All the logins in Firefox

And for the url `../api.php?id=2`:

You can also use Postman to do that. The use of Postman will be explained later.

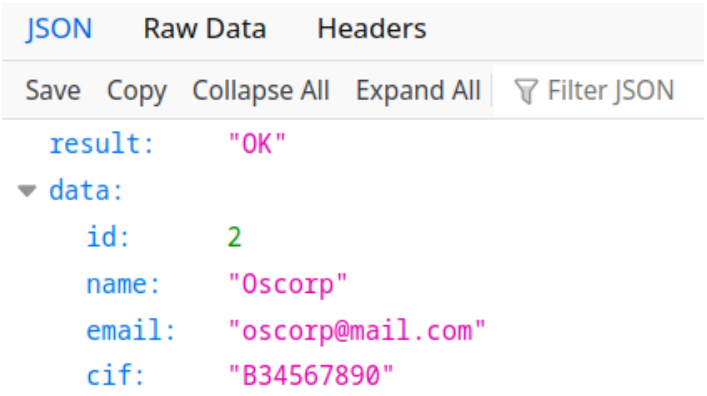

Figure 3: A single login request

3.3.2 Using Postman

We have seen that testing GET services is simple with a browser. Even testing a POST service could be done via an HTML form, but the modification (PUT) or deletion (DELETE) services require other tools if we want to test them. One of the most useful tools for this purpose is **Postman**.

Postman is a free and cross-platform application that allows you to send all kind of requests to a server, and examine the response that it produces. In this way, we can verify that the services offer the adequate information before being used by a real client application.

You can download Postman from its web site, postman.com, and click on the icon below "Download the desktop app" of your operating system:

In Ubuntu you can install from the Snap repository:

```
sudo snap install postman
```

Once installed, as soon as it starts, it will ask us if we want to register, and even associate Postman with a Google account. This has the advantages of being able to store in the account the different tests that we do, and we are able to use them in other computers, but it is not a mandatory step, and we can skip it.

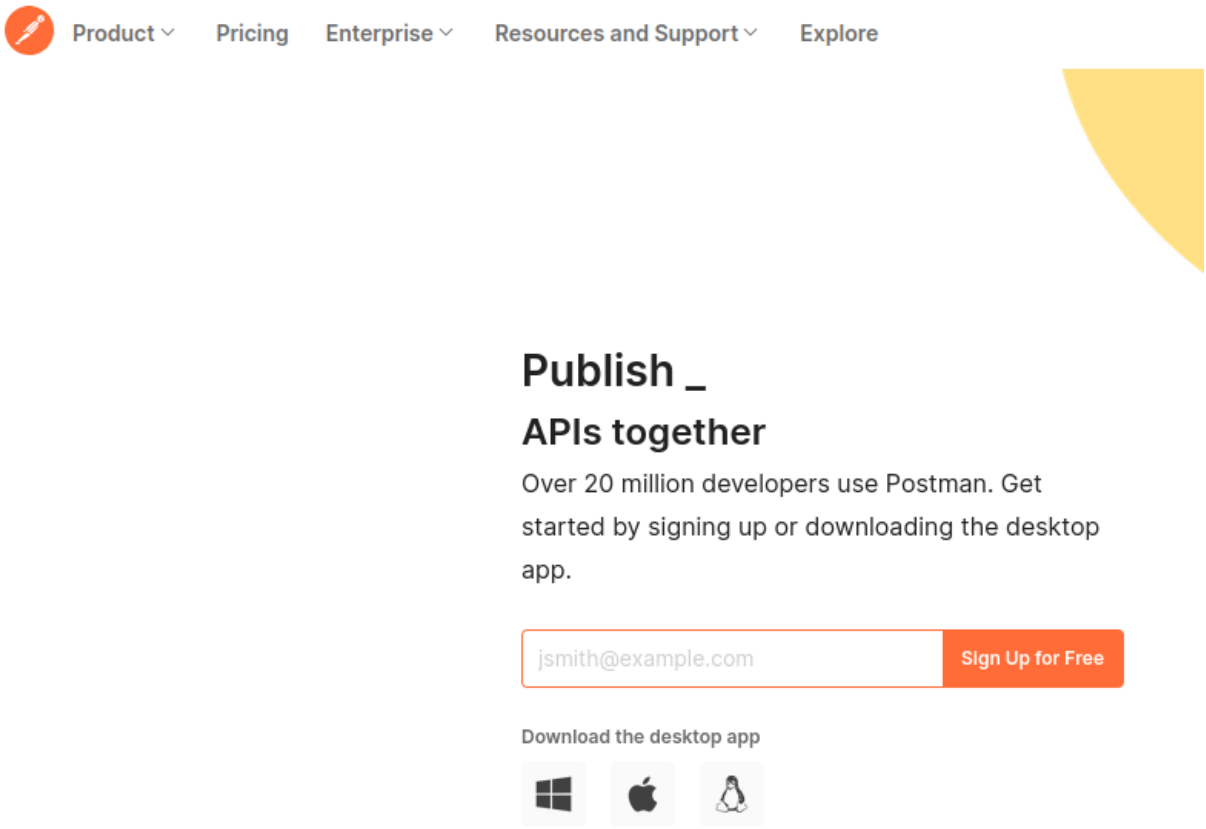

Figure 4: Download Postman

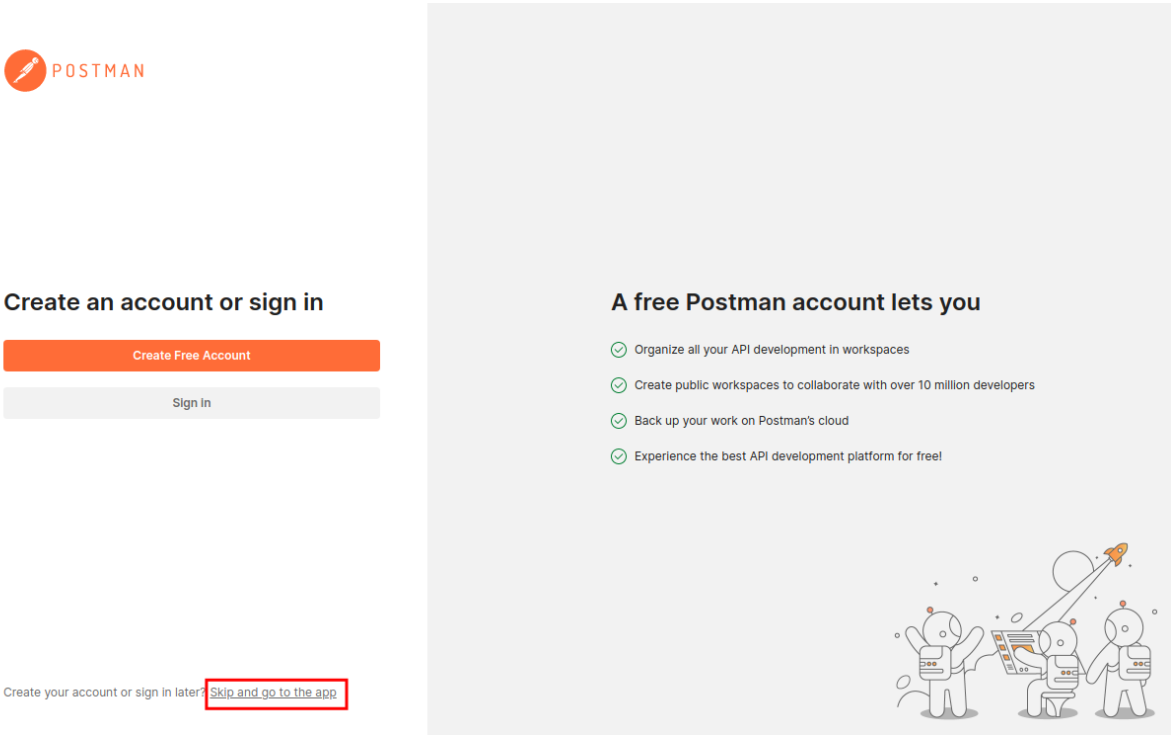

Figure 5: Postman register

After this screen, we will see a dialog to create simple **requests** or **collections** of requests (test suites for an application). Click on ***Create a collection -> From scratch*** and write the name for the new collection, *"Providers API"*.

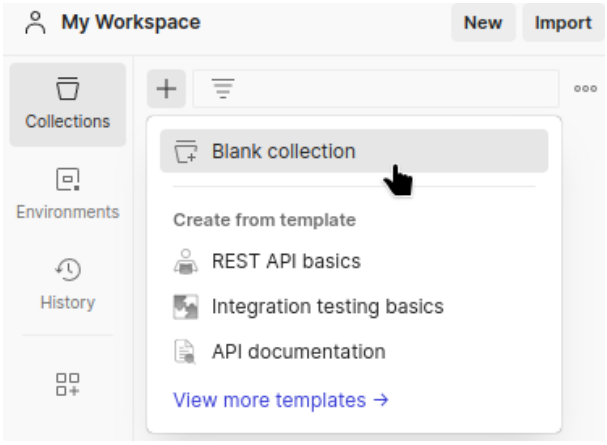

Figure 6: Create collection

We can see the collection in the left panel of Postman. To make our first request, click on **New** (above the new collection, in the left panel) and select **HTTP Request**:

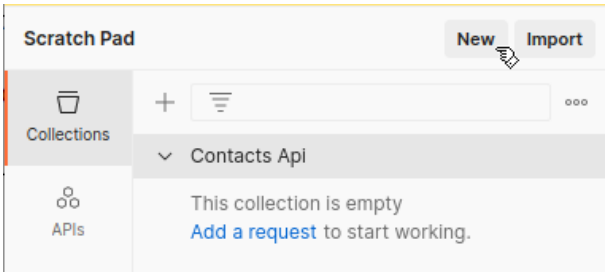

Figure 7: Postman new request

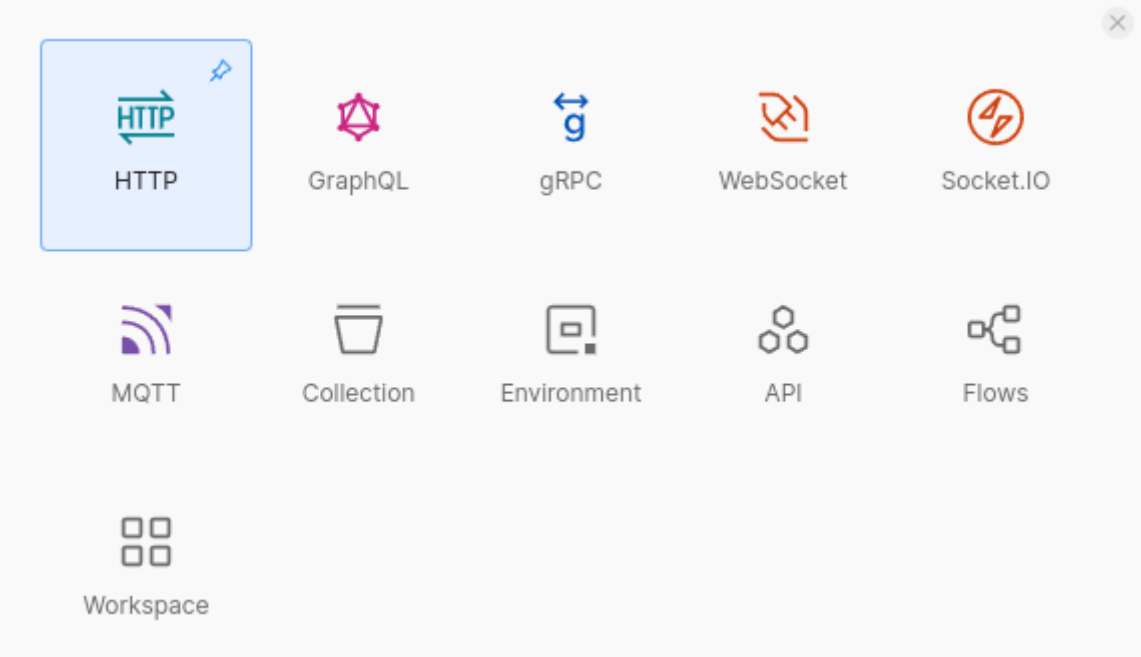

Figure 8: Postman new request

Now, write the listing URL (`localhost/yout_app/api.php`) next to the GET selection button and then click on ***Send***. In the bottom panel you should view the Json response:

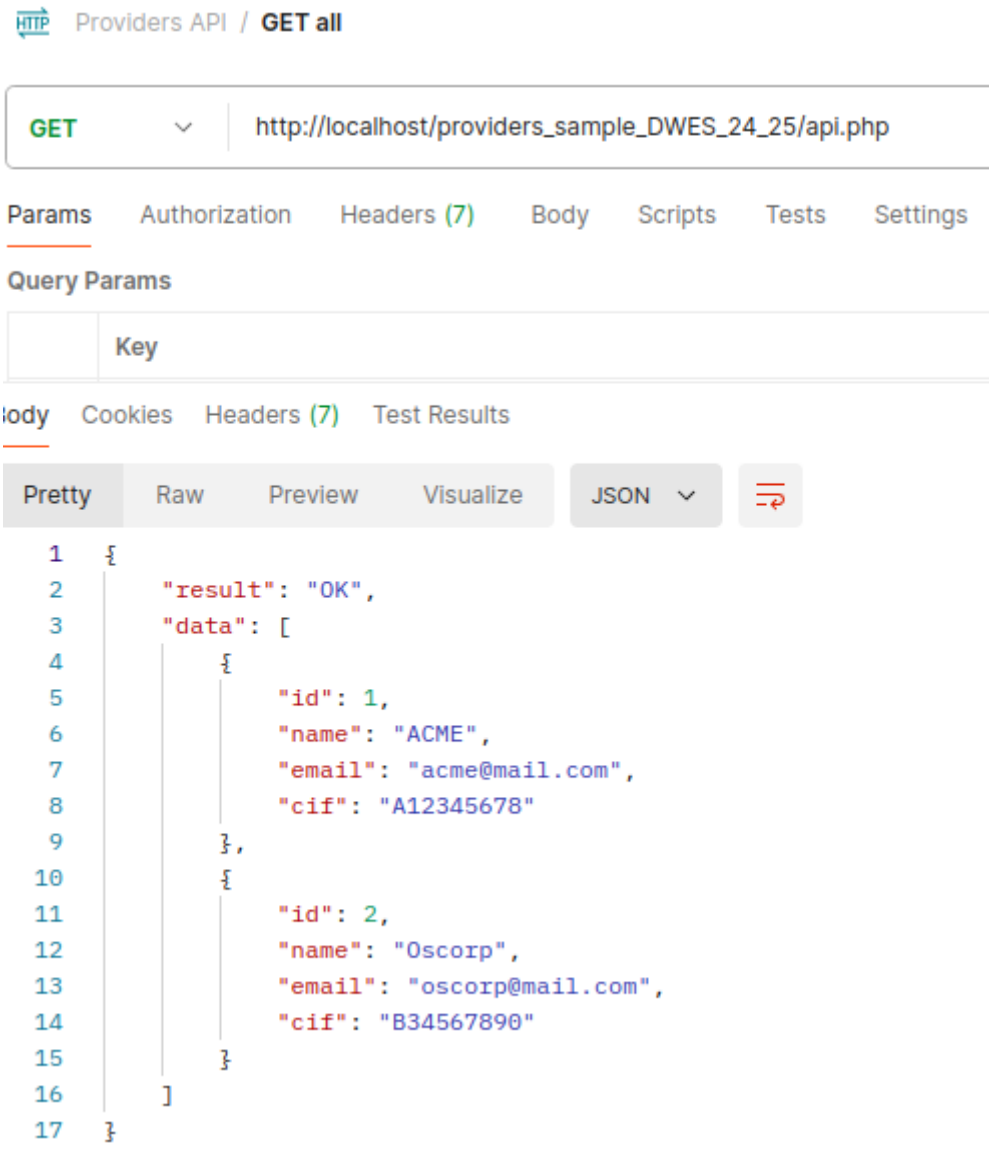

Figure 9: Postman GET request

If everything is ok, click on the ***Save*** button so you could reuse the request later.

You can test getting one login by its id in the same way.

3.3.3 Posting one element (POST)

To insert a login we use the POST method and the default route without parameters in the query string, because the new data (except the id) will be included in the request body. As the method is different from GET, we can reuse the same route with a different operation.

Here, `file_get_contents` reads the post content into a string and `json_decode` parses a json string into an array:

```
case 'POST':
    $params = json_decode(file_get_contents("php://input"),true);
    if(!isset($params)) {
        $response=array('result'=>'Error','data'=>'Empty data');
        echo Response::result(400,$response);
    } else {
        $provider = new Provider();
        $provider->setName($params['name']);
        $provider->setEmail($params['email']);
        $provider->setCif($params['cif']);
        $id = ProviderRepository::insert($provider);
        $provider->setId($id);
        $response = array(
            'result'=>'OK',
            'data'=>$provider->toArray(),
        );
        echo Response::result(201, $response);
    }
    break;
```

In the data element we are showing the new login added.

To make a POST request with Postman, change the method to POST and write in the **Body** section, in **raw** format, the data that you want to insert:

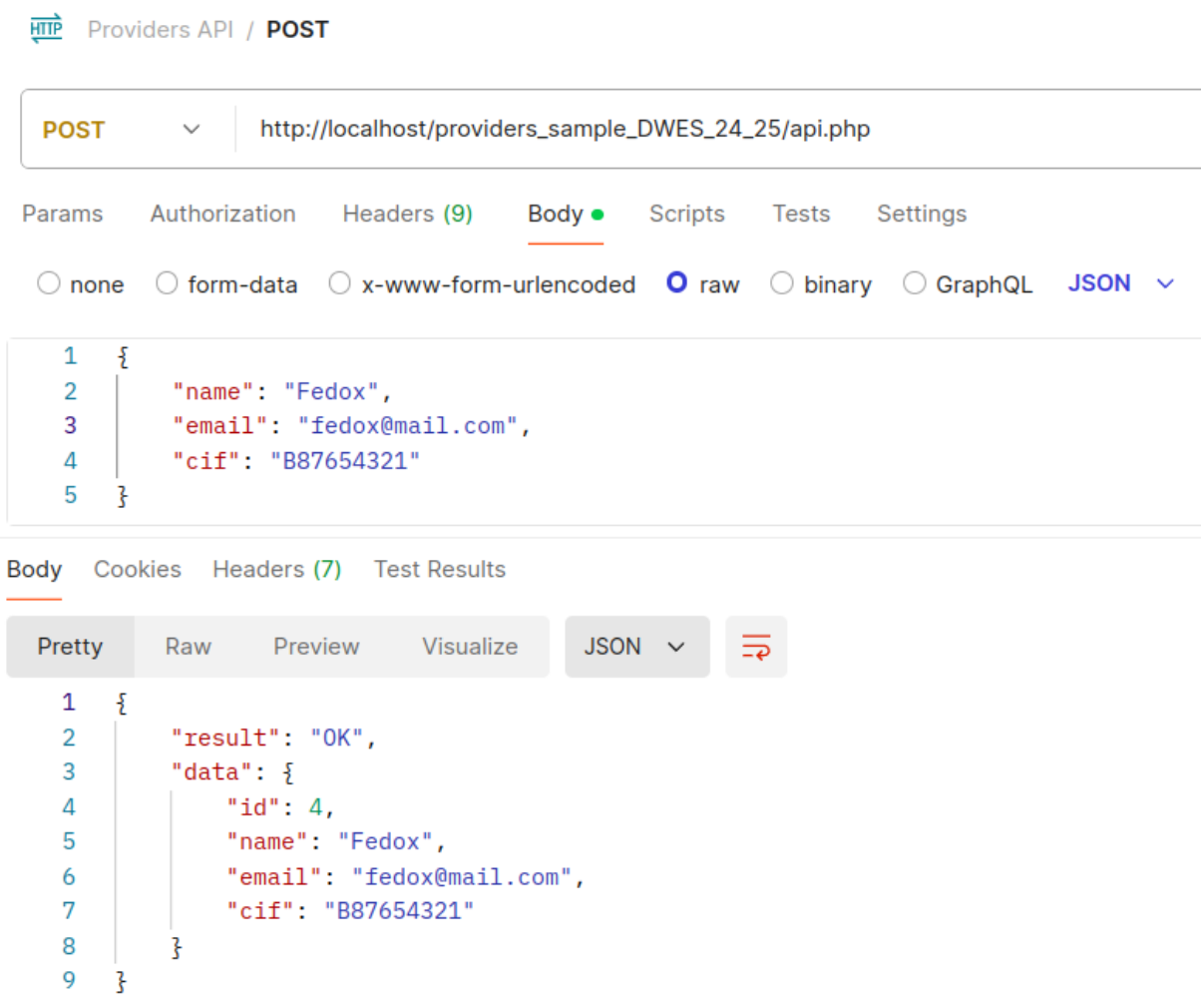

Figure 10: POST request with Postman

3.3.4 Modifying an element (PUT)

This method is similar to the previous one, but fetching first the login by its `id` and modifying it later with the request data. We are using the same url but changing the method and adding the `id` of the login. We are also checking if the provided `id` exists.

```php
case 'PUT':
    parse_str($_SERVER['QUERY_STRING'], $query);
    $params = json_decode(file_get_contents("php://input"),true);
    if(!isset($params) || empty($query['id'])) {
        $response=array('result'=>'Error','data'=>'Empty data');
        echo Response::result(400,$response);
    } else {
        $provider = new Provider();
        $provider->setId(trim(strip_tags($query['id'])));
        $provider->setName(trim(strip_tags($params['name'] ?? '')));
        $provider->setEmail(trim(strip_tags($params['email'] ?? '')));
        $provider->setCif(trim(strip_tags($params['cif'] ?? '')));
        if(ProviderRepository::update($provider)){
            $response = array(
                'result'=>'OK',
                'data'=>$provider->toArray(),
            );
            echo Response::result(201, $response);
        } else {
            $response = array(
                'result'=>'Error',
                'data'=>'Provider not found',
            );
            echo Response::result(400, $response);
        }
    }
    break;
```

The PUT request with Postman can be done changing to the PUT method and writing the data with the desired changes:

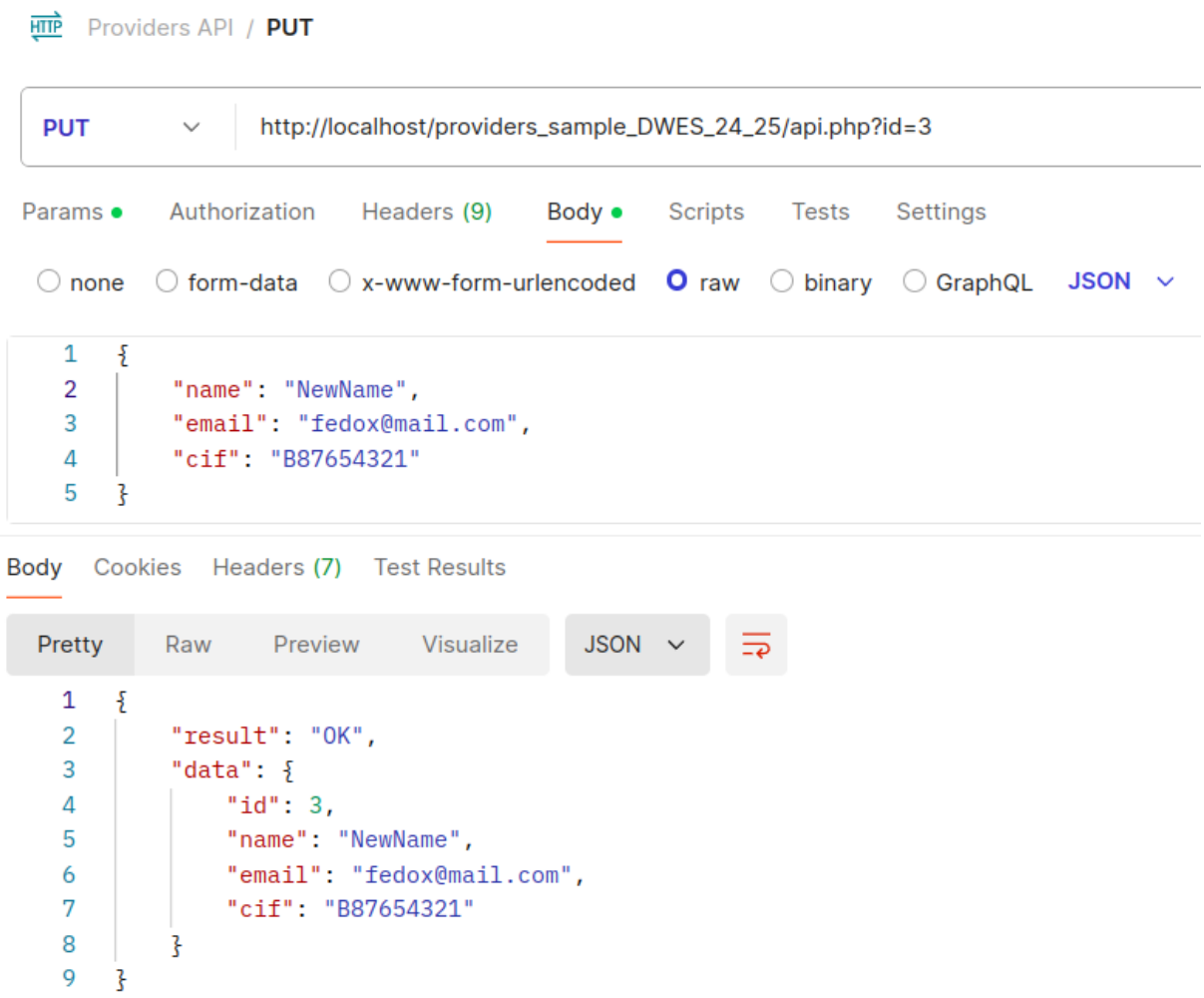

Figure 11: PUT request with Postman

It's a good practice to test with wrong data, like an non existing `id`:

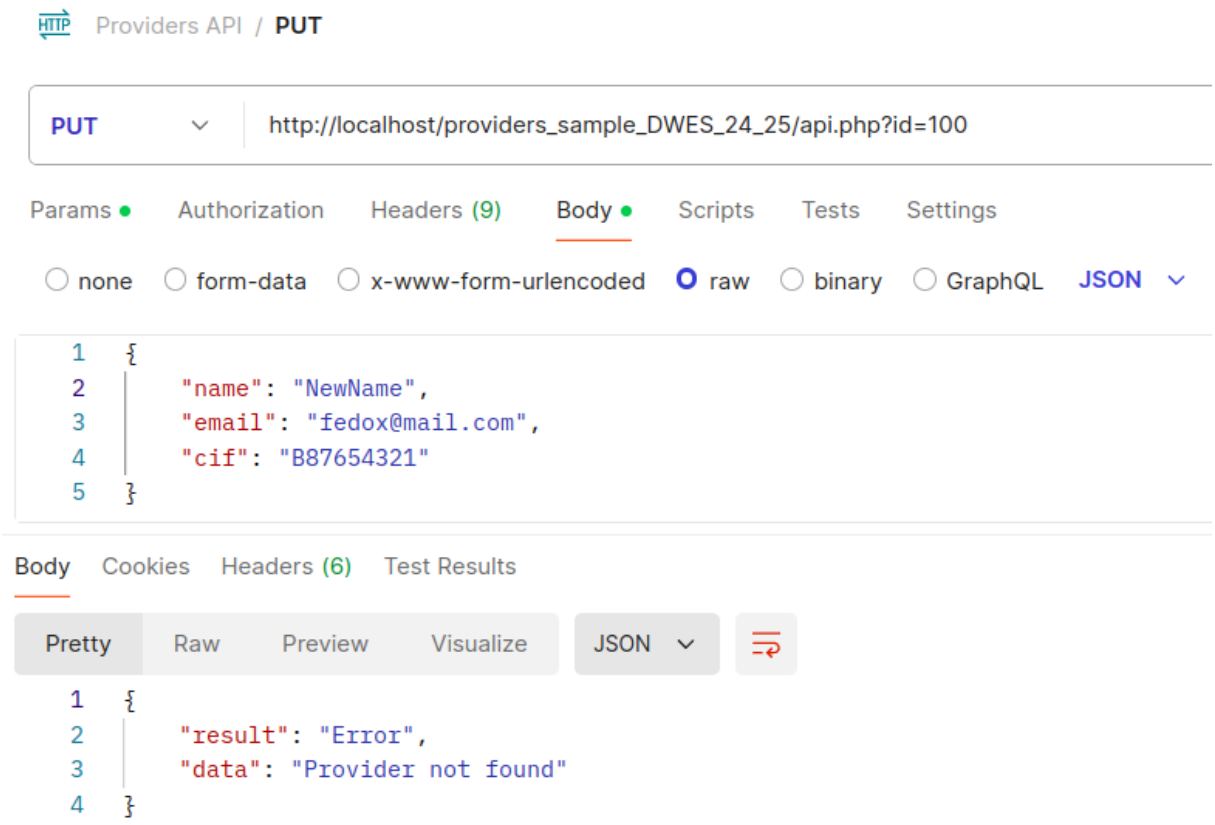

Figure 12: PUT request with wrong id

3.3.5 Removing an element (DELETE)

The same as before, but using the DELETE method. Here we check if the `id` exists and the login is deleted.

```php
case 'DELETE':
    parse_str($_SERVER['QUERY_STRING'], $query);
    if(!isset($query['id'])) {
        $response=array('result'=>'Error','data'=>'Empty data');
        echo Response::result(400, $response);
    } else {
        $id = trim(strip_tags($query['id']));
        $provider = ProviderRepository::select($id);
        if(!is_null($provider) && ProviderRepository::delete($provider)){
            $response = array(
                'result'=>'OK',
                'data'=>$provider->toArray(),
            );
            echo Response::result(200, $response);
        } else {
            $response = array(
                'result'=>'Error',
                'data'=>'No providers deleted',
            );
            echo Response::result(200, $response);
        }
    }
    break;
```

To do the DELETE request with Postman, we only need to specify the DELETE method and to pass the id with the URL:

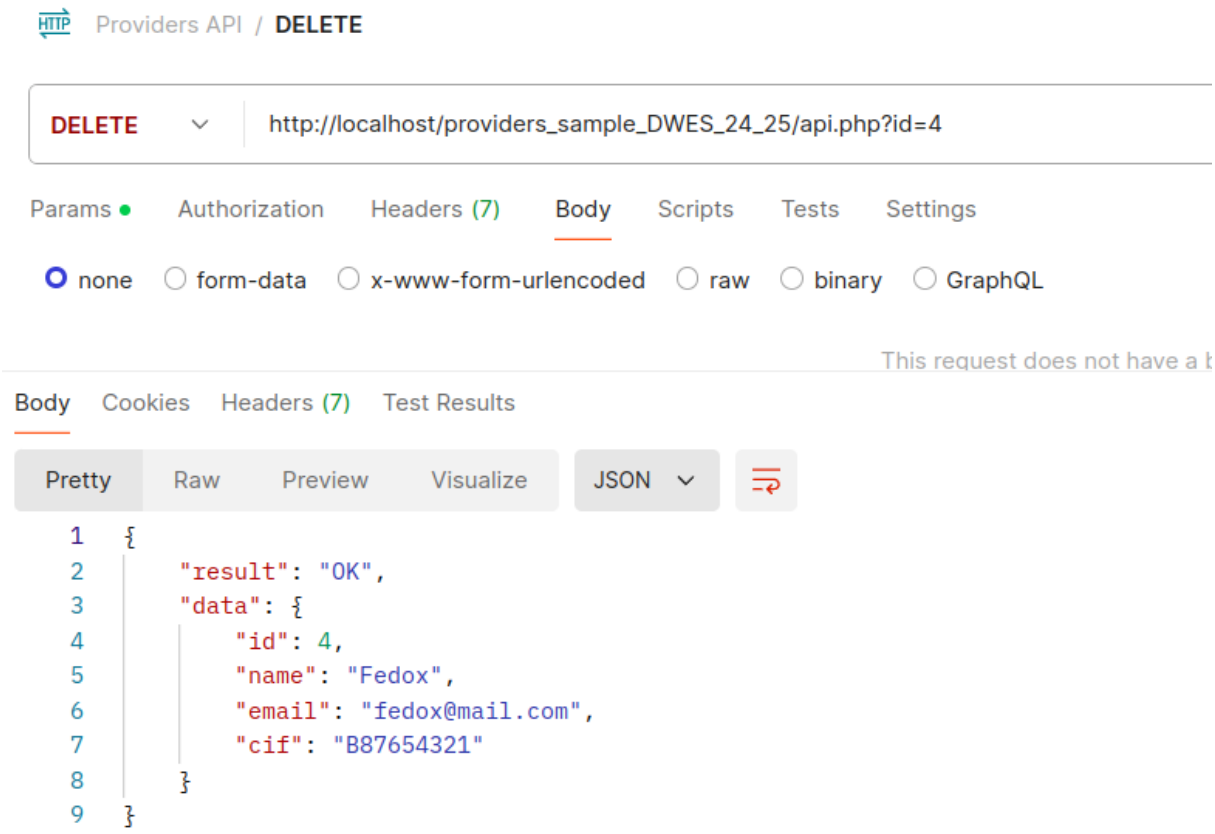

Figure 13: DELETE request and result with Postman

Remember to save your requests in Postman in order to reuse them later.

4 Validation

To do the validation we simply need to use the `validate()` method of the `Provider` class.

The code for the PUT method in the `api.php` script:

```php
//PUT
$provider = new Provider();
$provider->setId(trim(strip_tags($query['id'])));
$provider->setName(trim(strip_tags($params['name'] ?? '')));
$provider->setEmail(trim(strip_tags($params['email'] ?? '')));
$provider->setCif(trim(strip_tags($params['cif'] ?? '')));

//Validation
$errors = $provider->validate();
if(empty($errors)){
    if(ProviderRepository::update($provider)){
        $response = array(
            'result'=>'OK',
            'data'=>$provider->toArray(),
        );
        echo Response::result(201, $response);
    } else {
        $response = array(
            'result'=>'Error',
            'data'=>'Provider not found',
        );
        echo Response::result(400, $response);
    }
} else {
    //Validation errors
    $response = array(
        'result'=>'Error',
        'data'=>'Validation erros',
        'errors'=>$errors,
    );
    echo Response::result(400, $response);
}
```

Note that we are adding the errors array to the response.

Then, check some invalid PUT requests:

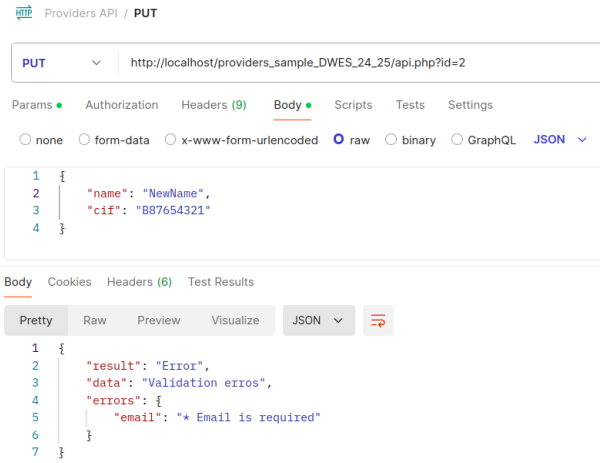

Figure 14: PUT request with empty fields

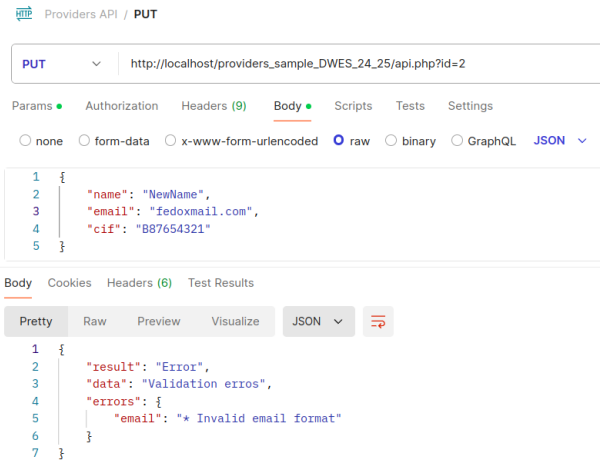

Figure 15: PUT request with invalid data

The POST method code:

```php
//POST
$provider = new Provider();
$provider->setName($params['name']);
$provider->setEmail($params['email']);
$provider->setCif($params['cif']);

$errors = $provider->validate();
if(empty($errors)){
    //Validation successful
    $id = ProviderRepository::insert($provider);
    $provider->setId($id);
    $response = array(
        'result'=>'OK',
        'data'=>$provider->toArray(),
    );
    echo Response::result(201, $response);
} else {
    //Validation errors
    $response = array(
        'result'=>'Error',
        'data'=>'Validation erros',
        'errors'=>$errors,
    );
    echo Response::result(400, $response);
}
```

You can get the full code at GitHub repository