ZF - Session 9

1. REST API

1.1. ¿Qué es REST?

REST son las siglas de **Representational State Transfer**. Fue definido hace una década por Roy Fielding en su tesis doctoral, y proporciona una forma sencilla de interacción entre sistemas, la mayor parte de las veces a través de un navegador web y HTTP. Esta cohesión con HTTP viene también de que Roy es uno de los principales autores de HTTP.

REST es un estilo arquitectónico, un conjunto de convenciones para aplicaciones web y servicios web, que se centra principalmente en la manipulación de recursos a través de especificaciones HTTP. Podemos decir que REST es una interfaz web estándar y simple que nos permite interactuar con servicios web de una manera muy cómoda.

Gracias a REST la web ha disfrutado de escalabilidad como resultado de una serie de diseños fundamentales clave:

- Un **protocolo cliente/servidor sin estado**: cada mensaje HTTP contiene toda la información necesaria para comprender la petición. Como resultado, ni el cliente ni el servidor necesitan recordar ningún estado de las comunicaciones entre mensajes. Sin embargo, en la práctica, muchas aplicaciones basadas en HTTP utilizan cookies y otros mecanismos para mantener el estado de la sesión (algunas de estas prácticas, como la reescritura de URLs, no son permitidas por REST).
- Un **conjunto de operaciones bien definidas** que se aplican a todos los recursos de información: HTTP en sí define un conjunto pequeño de operaciones, las más importantes son **POST**, **GET**, **PUT** y **DELETE**. Con frecuencia estas operaciones se equiparan a las operaciones CRUD que se requieren para la persistencia de datos, aunque POST no encaja exactamente en este esquema.

- Una **sintaxis universal para identificar los recursos**. En un sistema REST, cada recurso es direccionable únicamente a través de su URI.

- El **uso de hipermedios**, tanto para la información de la aplicación como para las transiciones de estado de la aplicación: la representación de este estado en un sistema REST son típicamente HTML o XML. Como resultado de esto, es posible navegar de un recurso REST a muchos otros, simplemente siguiendo enlaces sin requerir el uso de registros u otra infraestructura adicional.

Vamos a ver primero lo que son las URIs. Una URI es esencialmente un identificador de un recurso. Veamos el siguiente ejemplo:

```
GET /amigos
```

Podríamos llamar a esto un recurso. Cuando esta ruta es llamada, siguiendo los patrones REST, se obtendrán todos los amigos (generalmente de una base de datos), y se mostrarán en pantalla o se devolverán en un formato determinado a quien lo solicite.

Pero, cómo haremos si queremos especificar un amigo en particular?.

```
GET /amigos/marta
```

Como se puede ver es fácilmente comprensible. Esa es una de las claves de la arquitectura RESTful. Permite el uso de URIs que son fácilmente comprensibles por los humanos y las máquinas.

Piensa en un recurso como un nombre en plural. Contactos, estados, usuarios, fotos --- todos éstos serían elecciones perfectas.

Hasta ahora, hemos visto como identificar a una colección y a elementos individuales en esa colección:

```
GET /amigos GET /amigos/marta
```

De hecho, encontrarás que estos dos segmentos son todo lo que tendrías que haber necesitado siempre. Pero podemos profundizar un poco más en la potencia de HTTP para indicar cómo queremos que el servidor responda a esas peticiones. Veamos:

Cada petición HTTP especifica un método, o un verbo, en sus encabezados. Generalmente te sonarán un par de ellos como GET y POST.

Por defecto el verbo utilizado cuando accedemos o vemos una página web es **GET**.

Para cualquier URI dada, podemos referenciar hasta 4 tipos diferentes de métodos: **GET, POST, PUT, PATCH y DELETE**.

```
GET /amigos POST /amigos PUT /amigos DELETE /amigos
```

Esencialmente, estos verbos HTTP indican al servidor que hacer con los datos especificados en la URI. Una forma fácil de asociar estos verbos a las acciones realizadas, es comparándolo con **CRUD** (Create-Read-Update-Delete).

```
GET => READ POST => CREATE PUT => UPDATE DELETE => DELETE
```

Anteriormente hemos dicho que GET es el método utilizado por defecto, pero también te debería sonar POST. Cuando enviamos datos desde un formulario al servidor, solemos utilizar el método POST. Por ejemplo si quisiéramos añadir nuevos Tweets a nuestra base de datos, el formulario debería hacer un POST de los tweets POST /tweets, en lugar de hacer /tweets/añadirNuevoTweet.php.

Ejemplos de URIs que son no RESTful y que no se recomienda utilizar:

```
 /tweets/añadirNuevoTweet.php  /amigos/borrarAmigoPorNombre.php
/contactos/actualizarContacto.php
```

Ejemplos de URIs que son RESTful y que serían un buen ejemplo:

```
 GET /tickets/12/messages - Devuelve una lista de mensajes para el ticket
#12  GET /tickets/12/messages/5 - Devuelve el mensaje #5 para el ticket
#12  POST /tickets/12/messages - Crea un nuevo mensaje en el ticket #12
PUT /tickets/12/messages/5 – Actualiza el mensaje #5 para el ticket #12
PATCH /tickets/12/messages/5 - Actualiza parcialmente el mensaje #5 para
el ticket #12  DELETE /tickets/12/messages/5 - Borra el mensaje #5 para el
ticket #12
```

¿Pero entonces, cuáles serían las URIs correctas para presentar un formulario al usuario, con el objetivo de añadir o editar un recurso?

En situaciones como esta, tiene más sentido añadir URIs como:

```
GET /amigos/nuevo GET /amigos/marta/editar
```

La primera parte de la trayectoria /amigos/nuevo, debería presentar un formulario al usuario para añadir un amigo nuevo. Inmediatamente después de enviar el formulario, debería usarse una solicitud POST, ya que estamos añadiendo un nuevo amigo.

Para el segundo caso /amigos/marta/editar, este formulario debería editar un usuario existente en la base de datos. Cuando actualizamos los datos de un recurso, se debería utilizar una solicitud PUT.

Más información de cómo nombrar las URI en una API REST:

http://www.restapitutorial.com/lessons/restfulresourcenaming.html

Otro libro recomendable sobre RESTful: http://restcookbook.com/

1.2. Verbos disponibles en REST

Antes de seguir adelante con ejemplos concretos, vamos a revisar un poco más los verbos utilizados en las peticiones a una API REST.

GET (Recuperar)

GET es el método HTTP utilizado por defecto en las peticiones web. Una advertencia a tener en cuenta es que deberíamos utilizar GET, para hacer peticiones sólo de lectura, y deberíamos obtener siempre el mismo tipo de resultado, independientemente de las veces que sea llamado ese método.

Como programador puedes hacer lo que quieras cuando se hace una llamada a las rutas en la URI, pero una buena práctica es seguir las reglas generales para diseñar una API REST correctamente.

POST (Crear)

El segundo método que te resultará familiar es POST. Se utilizará para indicar que vamos a crear un subconjunto del recurso especificado, o también si estamos actualizando uno o más subconjuntos del recurso especificado.

Por ejemplo vamos a crear un recurso nuevo por ejemplo enviando un nuevo usuario para darlo de alta, entonces lo haremos a la URL /amigos

```
POST /amigos
```

PUT (Actualizar)

Utiliza PUT cuando quieras actualizar un recurso específico a través de su localizador en la URL.

Por ejemplo si un artículo está en la URL http://miweb.local/api/articulos/1333, podemos actualizar ese recurso enviando todos los campos a actualizar desde un formulario (método POST):

```
PUT /api/articulos/1333
```

Si no conocemos la dirección del recurso actual, for ejemplo para añadir un nuevo artículo, entonces utilizaríamos la acción POST. Por ejemplo.

```
POST /api/articulos
```

DELETE (Borrado)

Por último DELETE debería se usado cuando queremos borrar el recurso especificado en la URI. Por ejemplo si ya no somos más amigos de macarena, siguiendo los principios de REST, podríamos borrarla usando una petición delete a la URI:

```
DELETE /amigos/macarena
```

PATCH (Actualizaciones parciales)

Una solicitud de tipo PATCH se utiliza para realizar una actualización parcial de un recurso es decir para actualizar ciertos campos del recurso y no el recurso al completo. Los campos a actualizar se enviarían desde un formulario por POST y el tipo de petición es PATCH.

```
PATCH /api/articulos/1333
```

HEAD (Solicitud de cabeceras)

- Una solicitud de tipo HEAD es como una solicitud GET, con la salvedad que solamente se devuelven las cabeceras HTTP y el código de respuesta, no el documento en sí mismo.
- Con este método el navegador puede comprobar si un documento ha sido modificado, por temas de caché por ejemplo. También puede comprobar si un documento existe o no.
- Por ejemplo, si tienes un montón de enlaces en tu web, periódicamente podrías comprobar mediante peticiones HEAD si los hiperenlaces son correctos o están rotos. Éste tipo de comprobación es muchísimo más rápido que usar GET.
- **Más información sobre cabeceras HTTP en: http://code.tutsplus.com/tutorials/http-headers-for-dummies--net-8039**

2. Creación de un controller REST

2.1. Configuración

2.1.1. Nueva ruta para nuestro controller REST

```
<?php
return array(
    'router' => array(
        'routes' => array(
            'user\usersREST' => array(
                'type'    => 'Literal',
                'options' => array(
                    'route'    => '/users-rest/',
                    'defaults' => array(
                        'controller' => 'User\Controller\UsersREST',
                    ),
                ),
                'may_terminate' => true, // parent route can be alone
                'child_routes' => array(
                    'withID' => array(
                        'type' => 'Segment',
                        'options' => array(
                            'route' => 'id/[:id]/',
                            'constraints' => array(
                                'id' => '[0-9]+',
                            ),
                            'defaults' => array(
                                // Same as parent. We can also avoid this 'defaults' key
                            ),
                        ),
                    ),
                ),
            ),
        ),
```

Esta ruta será la que utilizará en todas las peticiones, pudiendo utilizar la id del recurso:

```
http://localhost:8080/users-rest/
```

```
http://localhost:8080/users-rest/id/4/
```

o bien utilizando curl desde el terminal:

```
curl -i -H "Accept: application/json" http://localhost:8080/users-rest/id/?/
```

2.1.2. Factory para el controller REST

```
'controllers' => array(
    'factories' => array(
```

```
            'User\Controller\UsersREST'     => 'User\Controller\Factory\UsersRESTControllerFactory',
    ),
),
```

2.1.3. Añadir ViewJasonStrategy al View Manager

```
'view_manager' => array(
    'template_map'              => array(
    ),
    'template_path_stack'       => array(
        __DIR__ . '/../view',
    ),
    'strategies'                => array(
        'ViewJsonStrategy',
    ),
),
```

2.2. UserRESTControllerFactory

```
<?php
/**
 * xenFramework (http://xenframework.com/)
 *
 * This file is part of the xenframework package.
 *
 * (c) Ismael Trascastro <itrascastro@xenframework.com>
 *
 * @link        http://github.com/xenframework for the canonical source repository
 * @copyright   Copyright (c) xenFramework. (http://xenframework.com)
 * @license     MIT License - http://en.wikipedia.org/wiki/MIT_License
 *
 * For the full copyright and license information, please view the LICENSE
 * file that was distributed with this source code.
 */

namespace User\Controller\Factory;


use User\Controller\UsersRESTController;
use Zend\ServiceManager\FactoryInterface;
use Zend\ServiceManager\ServiceLocatorInterface;

class UsersRESTControllerFactory implements FactoryInterface
{

    /**
     * Create service
     *
     * @param ServiceLocatorInterface $serviceLocator
     *
     * @return mixed
     */
    public function createService(ServiceLocatorInterface $serviceLocator)
    {
        $sm = $serviceLocator->getServiceLocator();
        $model = $sm->get('User\Model\UsersModel');
        $form = $sm->get('User\Form\User');
```

```
        return new UsersRESTController($model, $form);
    }
}
```

De igual manera que hacíamos con nuestro controller User normal, inyectamos las dependencias (el módelo y el formulario) a nuestro controller REST.

2.3. UserRESTController

```
<?php
/**
 * xenFramework (http://xenframework.com/)
 *
 * This file is part of the xenframework package.
 *
 * (c) Ismael Trascastro <itrascastro@xenframework.com>
 *
 * @link        http://github.com/xenframework for the canonical source repository
 * @copyright   Copyright (c) xenFramework. (http://xenframework.com)
 * @license     MIT License - http://en.wikipedia.org/wiki/MIT_License
 *
 * For the full copyright and license information, please view the LICENSE
 * file that was distributed with this source code.
 */

namespace User\Controller;

use User\Form\UserForm;
use User\Model\UsersModel;
use Zend\Mvc\Controller\AbstractRestfulController;
use Zend\View\Model\JsonModel;

class UsersRESTController extends AbstractRestfulController
{
    /**
     * @var UsersModel
     */
    private $model;

    /**
     * @var UserForm
     */
    private $form;

    /**
     * @param UsersModel $model
     * @param UserForm $form
     */
    function __construct(UsersModel $model, UserForm $form)
    {
        $this->model = $model;
        $this->form = $form;
    }

    // The following methods will be mapped from the HTTP request
    // http://framework.zend.com/manual/current/en/modules/zend.mvc.controllers.html#the-
abstractrestfulcontroller
```

```
    /**
     * getList
     *
     * Maps HTTP Request Method: GET
     *
     * Using from the client
     *
     * Curl
     *      curl -i -H "Accept: application/json" http://localhost:8080/users-rest/
     * Browser
     *      http://localhost:8080/users-rest/
     *
     * @return JsonModel
     */
    public function getList()
    {
        $users = $this->model->findAll(false);

        foreach ($users as $user) {
            $userArray  = $user->getArrayCopy();
            $data[]     = $userArray;
        }

        return new JsonModel([
            'data' => $data
        ]);
    }

    /**
     * get
     *
     * Maps HTTP Request Method: GET
     *
     * Curl
     *      curl -i -H "Accept: application/json" http://localhost:8080/users-rest/id/?/
     * Browser
     *      http://localhost:8080/users-rest/id/?/
     *
     * @param int $id
     * @return JsonModel
     */
    public function get($id)
    {
        $user       = $this->model->getById($id);
        $userArray  = $user->getArrayCopy();

        return new JsonModel([
            'data' => $userArray
        ]);
    }

    /**
     * create
     *
     * Maps HTTP Request Method: POST
     *
     * Curl
     *      curl -i -H "Accept: application/json" -X POST -d
"username=API&email=api@email.com&password=1234&role=user" http://localhost:8080/users-rest/
```

```
     *
     * @param mixed $data
     * @return mixed|JsonModel
     */
    public function create($data)
    {
        $this->form->setData($data);

        if ($this->form->isValid()) {
            $formData = $this->form->getData();

            $data['username']   = $formData['username'];
            $data['email']      = $formData['email'];
            $data['password']   = $formData['password'];
            $data['role']       = $formData['role'];
            $data['date']       = date('Y-m-d H:i:s');

            $id = $this->model->save($data);

            // We cannot call $this->get($id) because the request method now is POST instead of GET
            $user       = $this->model->getById($id);
            $userArray  = $user->getArrayCopy();

            return new JsonModel([
                'data' => $userArray,
            ]);
        }
    }

    /**
     * update
     *
     * Maps HTTP Request Method: PUT
     *
     * Curl
     *      curl -i -H "Accept: application/json" -X PUT -d "username=APIUPDATE"
http://localhost:8080/users-rest/id/14/
     *
     * @param mixed $id
     * @return mixed|JsonModel
     */
    public function update($id, $data)
    {
        $user = $this->model->getById($id);
        $user->exchangeArray2($data);

        if (!isset($data['password'])) {
            $user->setPassword(null);
        }

        $this->form->bind($user);
        $this->form->getInputFilter()->get('password')->setAllowEmpty(true);

        if ($this->form->isValid()) {
            $formData = $this->form->getData();

            $data['id']         = $formData->getId();
            $data['username']   = $formData->getUsername();
            $data['email']      = $formData->getEmail();
            $data['password']   = $formData->getPassword();
```

```
            $data['role']       = $formData->getRole();
            $data['date']       = $formData->getDate();

            $id = $this->model->update($data);

            // We cannot call $this->get($id) because the request method now is POST instead of GET
            $user       = $this->model->getById($id);
            $userArray  = $user->getArrayCopy();

            return new JsonModel([
                'data' => $userArray,
            ]);
        }

        return new JsonModel([
            'error' => true,
        ]);
    }

    /**
     * delete
     *
     * Maps HTTP Request Method: DELETE
     *
     * Curl
     *      curl -i -H "Accept: application/json" -X DELETE http://localhost:8080/users-rest/id/?/
     *
     * @param mixed $id
     * @return mixed|JsonModel
     */
    public function delete($id)
    {
        $this->model->delete($id);

        return new JsonModel([
            'data' => 'deleted',
        ]);
    }
}
```

Este nuevo controller extiende de AbstractRestfulController del cual tendremos que implementar las acciones REST.

A la hora de devolver una respuesta, en lugar de renderizar una vista, devolvemos un JsonModel ya que hemos añadido la estrategia a la configuración del View Manager.

En la cabecera de los métodos tenemos ejemplos de uso.

El código de la session 9 se encuentra disponible en github:

https://github.com/itrascastro/ZF-Course/tree/session-10