

UD07. DESARROLLO DE MÓDULOS. WEB CONTROLLERS

Sistemas de Gestión Empresarial

2 Curso // CFGS DAM // Informática y Comunicaciones

Alfredo Oltra

**Cicles
Formatius**

ÍNDIX

1 INTRODUCCIÓN	4
2 WEB CONTROLLERS	5
2.1 Paso de parámetros	8
2.1.1 Parámetros por el POST o GET	8
2.1.2 Parámetros por REST	8
2.1.3 Parámetros por JSON	8
2.2 Retorno de datos	9
2.3 Autenticación	10
2.4 Plugins de apoyo	10
2.5 Banner route	11
3 API REST	12
3.1 Comunicar una SPA Vue/React/Angular con <i>Odoo</i>	13
4 BIBLIOGRAFIA	14
5 AUTORES	14

Versión: 240118.1217


Licencia




Reconocimiento – NoComercial – CompartirIgual (by-nc-sa). No se permite un uso comercial de la obra original ni de las posibles obras derivadas, la distribución de las cuales se debe hacer con una licencia igual a la que regula la obra original.

Nomenclatura

A lo largo de este tema se utilizarán distintos símbolos para distinguir elementos importantes dentro del contenido. Estos símbolos son:

 **Atención.** Importante prestar atención a esta información.

 **Interesante.** Ofrece información sobre algún detalle a tener en cuenta.

1 INTRODUCCIÓN

La conexión con otros sistemas es una parte muy importante en la implantación de *ERPs*. Si hablamos de *Odoo* las formas de conectarse a él son muy variadas. Lo habitual es hacerlo a través del **cliente web** ya sea mediante el TPV, el frontend o, especialmente, a través del *backend*. Las tres funcionan de forma independiente y con algunas diferencias.

Aun así, es posible que en muchas ocasiones en las que únicamente se necesite obtener datos (**web services**), obviando la parte gráfica, es posible realizar la conexión mediante el uso de XML-RPC/JSON-RPC, unas librerías que permiten implementar la conexión de manera bastante sencilla (aunque en algunas cosas limitadas) desde lenguajes como Python, Java o PHP (XML-RPC) o javascript (JSON-RPC).

Por último, una tercera posibilidad es realizar la conexión a través de API REST.

En algunas ocasiones podríamos usarlos de manera indistinta, pero la principal diferencia entre ellos es que a través de *web services* se obtienen únicamente datos, mientras que a través de API REST podemos obtener no solo datos, también código *html* para ser renderizado. Esto último abre la opción a, por ejemplo, poder incorporar elementos o disposiciones que no estaban previstos en vistas.



Existe otra opción para disponer datos de una manera personalizada, como es la creación de *widgets* o componentes, aunque es un proceso más complicado.

Como se puede observar, hay muchas opciones y casi siempre se pueden hacer las cosas de muchas formas. Además, la documentación oficial de *Odoo* a este respecto es muy escueta y en ocasiones obsoleta. Elegir no siempre es fácil, aunque por dar algunos puntos de referencia:

- Añadir un cambio menor en la apariencia: añadir reglas CSS nuevas en el directorio *static*.
- Una nueva forma de visualizar o editar los datos: crear un componente o *widget*.
- Una nueva forma de visualizar o editar un registro entero: crear una vista.
- Comunicar una página web hecha con PHP con *Odoo*: *XML-RPC*.
- Hacer una web: en general se recomienda usar el módulo de web de *Odoo* y modificarla.
- Hacer una app con Java que se conecta al servidor *Odoo*: usar XML-RPC o hacer una API REST con los *Web Controllers*.
- Hacer una web estática desde cero: usar los *Web Controller* para generar HTML a partir de plantillas *QWeb*.
- Hacer una SPA desde cero con *Vue*, *React* o *Angular*: utilizar los *Web Controllers* para generar JSON a partir de los datos. Hacer una API REST con *Web Controllers*.

Las dos últimas opciones son las que van a ser estudiadas en este apartado.

2 WEB CONTROLLERS

Se trata de funciones que responden a *URIs* (Identificador de Recursos Uniformes) concretas.

El servidor *Odoo* tiene un sistema de rutas para atender las peticiones. Por ejemplo, cuando accedemos a */web* lo que proporciona es el *backend*. Si accedemos a */pos/web* lo que nos proporciona es el *Point of Sale*. Nosotros podemos crear nuestras propias rutas para obtener páginas web personalizadas o otros datos como XML o JSON.

Para hacer estas rutas se usa los *Web Controllers*. De hecho, cuando se crea un nuevo módulo con *scaffold*, se crea un fichero *controllers/controllers.py* que, por defecto, está comentado.

Un ejemplo mínimo de controlador para analizarlo:

```
class MyController(http.Controller):

    @http.route('/school/course/', auth='user', type='json')
    def course(self):
    ...
```

Lo primero que podemos ver es que la clase hereda de `http.controller`. Recordemos, por ejemplo, que los modelos heredan de `models.Model`.

Esta clase le da las propiedades necesarias para atender peticiones HTTP y queda registrada como un controlador web. Esta clase tendrá como atributos unas funciones con un decorador específico. Estas funciones se ejecutarán cada vez que el usuario acceda a la ruta determinada en el decorador.

El decorador `@http.route` permite indicar:

- La ruta que atenderá.
- El tipo de autenticación con `auth=`, que puede ser: *users* si requiere que esté autenticado, *public* si puede estar autenticado (y si no lo está, lo trata como un usuario público) o *none* para no tener en cuenta para nada la autenticación o el usuario actual.
- El tipo de datos que espera recibir y enviar con `type="http"` o `type="json"`. Esto quiere decir que puede aceptar datos con la sintaxis HTTP de GET o POST o que espera a interpretar un body con información en formato JSON.
- Los métodos HTTP que acepta con `methods=`.
- La manera en la que trata las peticiones *Cross-origin* con `cors=`.



CORS, o *Cross-Origin Resource Sharing*, es un mecanismo que permite a los navegadores web acceder a recursos que se encuentra en un dominio diferente al dominio de la página web que lo está solicitando (un recurso de origen cruzado).

Por defecto, los navegadores web no permiten que las páginas web accedan a recursos de origen cruzado por razones de seguridad (de esta manera se evita que pueda ser inyectado código malicioso en una página web). Sin embargo, CORS permite a los desarrolladores, mediante el uso de cabeceras *http*, especificar qué recursos de origen cruzado pueden ser accedidos por una página web.

- Si necesita una autenticación con `csrf=`.



CSRF, o Cross-Site Request Forgery, es una vulnerabilidad de seguridad que permite a un atacante engañar a un usuario autenticado para que realice una acción que no desea realizar. Por ejemplo, el atacante consigue que el usuario envíe un formulario o ejecute un script malicioso con sus datos pero en beneficio del atacante.

Para evitarlo las aplicaciones web utilizan *tokens* (cadenas alfanuméricas aleatorias) que se generan para cada sesión de usuario. Cuando el usuario se autentica el servidor le envía un *token* que tendrá que ser reenviado por cada operación que se realice.

Las dos últimas opciones se deben poner en caso de hacer peticiones por una aplicación externa. En ese caso se pondrán los valores por defecto: `cors='*'`, `csrf=False`, ya que queremos que acepte cualquier petición externa y que no autentique con csrf, ya que tendremos que implementar nuestra propia autenticación.

Como veremos en el siguiente apartado, esta función recibirá parámetros, y retornará una respuesta. En caso de ser `type='json'` retornará un JSON y en caso de ser `type='http'`, retornará preferiblemente un HTML. Dejemos el JSON para más adelante y centrémonos en crear HTML.

Odoo tiene su motor de plantillas HTML que es *QWeb*, a partir del cual se puede construir el HTML con datos que queramos. También es posible no usar el motor de plantillas y generar algorítmicamente el HTML desde Python. Esta segunda opción solo es recomendada si va a ser muy simple o estático o si queremos tener un gran control sobre el HTML generado.

Veamos el ejemplo que proporciona *Odoo* cuando hacemos *scaffold* para crear un módulo:

```
@http.route('/school/school/objects/', auth='public')
def list(self, **kw):
    return http.request.render('school.listing', {
        'root': '/school/school',
        'objects': http.request.env['school.school'].search([]),
    })
```

Aquí se utiliza una nueva herramienta que es `http.request.render()`. Esta función utiliza una plantilla (*school.listing*) para crear un HTML a partir de unos datos.

Veamos ahora la plantilla *school.listing*:

```
<template id="school.listing">
    <ul>
        <li t-foreach="objects" t-as="object">
            <a t-attf-href="#{root}/objects/#{object.id}">
                <t t-esc="object.display_name"/>
            </a>
        </li>
    </ul>
</template>
```

En esta plantilla, *objects* es el *recordset* que recorrerá y mostrará una lista de los registros enviados por el *request.render*.

Todo lo anterior funciona perfectamente si accedemos a esta ruta desde el mismo navegador en el que hemos hecho *login* anteriormente. De esta manera se cumplirá que no estamos haciendo una petición *Cross-origin* (desde otro dominio) y que estamos autenticados.

Es posible que queramos mostrar cosas a usuarios que no tenemos autenticados, como por ejemplo un catálogo en una web. Ya tenemos `auth='public'`, no obstante si no está autenticado, la función `search` va a fallar. Para que no falle se puede utilizar la función `sudo()` antes de la función. De esta manera ignora si el usuario tiene permisos o si está autenticado.

```
'objects': http.request.env['school.school'].sudo().search([],
```



De manera similar a lo que ocurre con el uso de *sudo* en sistemas Linux, hay que tener mucho cuidado con el uso de *sudo()*. En general es mejor confiar en la autenticación de *Odoo* para todo. Si no es posible, hay que establecer un sistema de autenticación adecuado. En caso de ser información totalmente pública hay que limitar el acceso de los usuarios a lo mínimo imprescindible.

2.1 Paso de parámetros

Los métodos decorados con `@http.route` aceptan parámetros enviados por el *body* mediante POST, parámetros enviados mediante `?` de GET o en la misma URL como en el caso de un servicio REST

2.1.1 Parámetros por el POST o GET

En caso de ser `type='http'`, espera un *body* de POST tradicional o un GET con `?` y `&`, similar a:

```
parametro=valor&otroparametro=otrovalor
```

Si sabemos el nombre de los parámetros, los podemos poner en la función decorada. En caso de desconocerlos, podemos poner `**args` o `**kw` de manera que `args` será una lista de parámetros.

```
@http.route('/school', auth='public', cors='*', type='http')
def get_course(self, model, obj, **kw):
    model = http.request.env[model].sudo().search([('id', '=', obj)])
    .mapped(lambda p: p.read()[0])
    return model
```

Este ejemplo se puede llamar con un POST en el que enviemos los parámetros `model=course&obj=23` o con un GET en el que la URI, suponiendo estamos haciendo pruebas en un servidor en `localhost`, sería `http://localhost:8069/school?model=course&obj=23`

2.1.2 Parámetros por REST

Modifiquemos este ejemplo para aceptar peticiones al modo de los servicios REST:

```
@http.route('/school/<model>/<obj>', auth='public', cors='*', type='http')
def get_course(self, model, obj, **kw):
    model = http.request.env[model].sudo().search([('id', '=', obj)])
    .mapped(lambda p: p.read()[0])
    return model
```

No hay más que hacer una petición POST o GET sin enviar nada pero en este caso con la URL (suponiendo `localhost`): `http://localhost:8069/school/course/23`

2.1.3 Parámetros por JSON

Tan solo tenemos que cambiar `type='http'` por `type='json'` y enviar una petición POST. Odoo necesita que el POST tenga como cabecera `Content-Type: application/json` y el *body* tendrá (pensando en el ejemplo anterior) una sintaxis similar a esta:

```
'{"jsonrpc": "2.0", "method": "call", "params": {"model": "course", "obj": "23"}}'
```




Para probar todas estas peticiones se recomienda usar programas como *PostMan* o la extensión de *Visual Studio Code Thunder Client*, que permiten hacer muchas pruebas rápidamente. En caso de hacer alguna prueba puntual, se puede usar *curl* con comandos como este:

```
curl -i -X POST -H "Content-Type: application/json" \
-d '{"jsonrpc":"2.0","method":"call","params":{"model":"course","obj":"23"}}' \
localhost:8069/school
```

2.2 Retorno de datos

El hecho de poner `type='json'` o `type='http'` cambia el comportamiento de la función decorada, tanto para la recepción como para la devolución de los datos.

Cuando es de tipo `http`, la función no únicamente devolverá un HTML, también modifica las cabeceras para indicar que es de tipo `http`. El HTML a devolver lo puede enviar directamente (montándolo mediante Python en el cuerpo de la función):

```
return "<p>Hola</p>"
```

o montándolo a través de una plantilla (lo más habitual):

```
return http.request.render('school.listing', {
    'root': '/school/school',
    'objects': http.request.env['school.school'].search([]),
})
```

El caso de devolver un JSON la solución es similar:

```
return "{ user: { id: 0, name: \"Leo\", surname: \"Nadal López\"} }"
```

Sin embargo es posible que queramos recibir los datos vía GET (`type='http'`) y devolverlos mediante un JSON (`type='json'`). En esos casos el retorno de datos se debe realizar de manera manual manipulando la respuesta con `http.response`:

```
return http.Response(
    json.dumps(resp),
    status=200,
    mimetype='application/json')
```



El uso de `http.response` no es necesario en caso de retornar un JSON por un decorador de tipo `json` o un HTML por un decorador `http`, ya que el propio decorador ya añade las cabeceras necesarias y serializa los datos.

En ocasiones puede haber un problema con el objeto que retornamos, ya que la función `json.dumps` no siempre podrá serializar todo tipo de datos. Es posible indicarle que función aplicar para que intente serializar datos no serializables con el parámetro `default`. Por ejemplo:

```
return http.Response(
    json.dumps(resp), default=str
    status=200,
    mimetype='application/json')
```

Con este ejemplo, a los datos no serializables, les intenta aplicar la función `str`. Esto es útil por ejemplo con los datos en formato `datetime`. Sin embargo, habrá casos en los que no será suficiente y deberemos realizarlos manualmente.

2.3 Autenticación

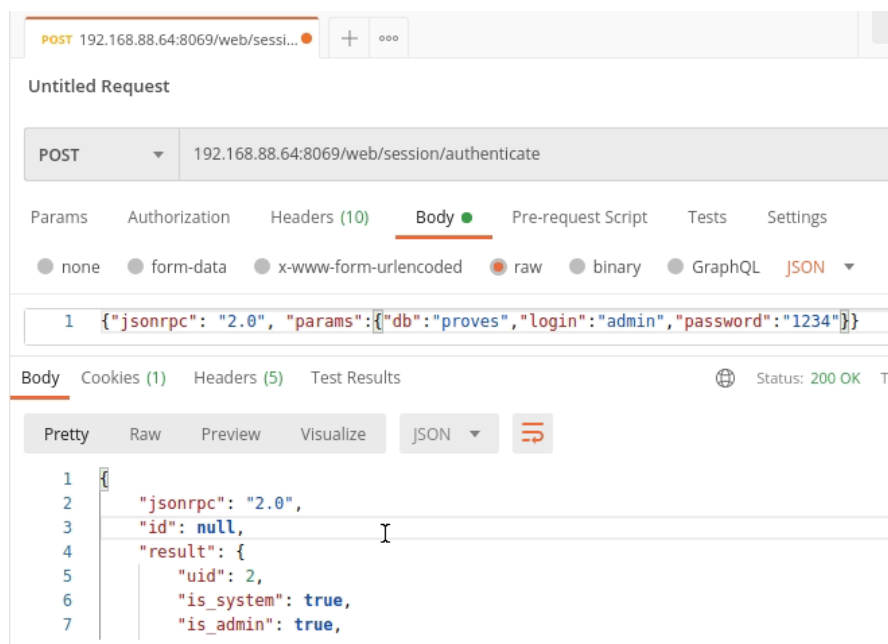
Este es el código del controlador de autenticación en *Odoo*:

```
@http.route('/web/session/authenticate', type='json', auth="none")
def authenticate(self, db, login, password, base_location=None):
    request.session.authenticate(db, login, password)
    return request.env['ir.http'].session_info()
```

En caso de hacer una aplicación web en la misma URL y necesitar autenticación con un usuario de *Odoo*, tendremos que hacer una petición con JSON a esta ruta.

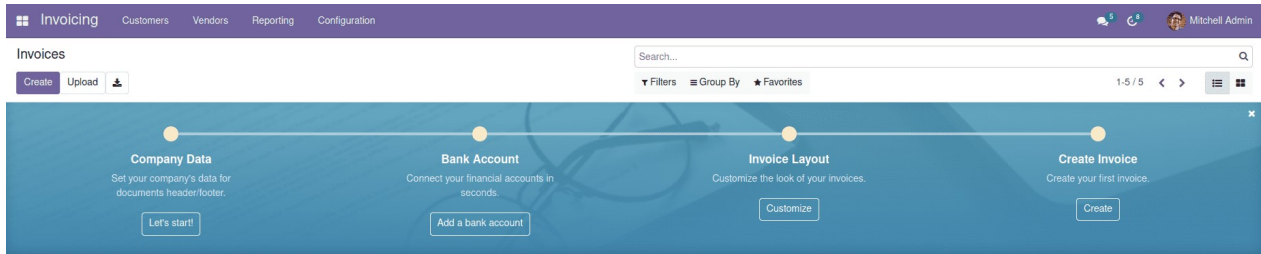
2.4 Plugins de apoyo

Uno de los sistemas más habituales para realizar pruebas es Postman, aunque en caso de estar utilizando Visual Studio Code como entorno de desarrollo es posible utilizar plugin como *Thunder Client*.



2.5 Banner route

Uno de los web controller más utilizados en Odoo es el *banner_route*. Su misión es la de mostrar en las vistas un banner en la parte superior con información extra.



El proceso es similar a cualquier otro *web controller*, salvo que en este caso la autenticación es de tipo user (solo los usuarios autenticados pueden acceder a ella).

```
@http.route('/school/school_banner/', auth='user', type='json')
def get_banner_data(self, **kw):
    num_student = request.env['school.student'].search_count([])
    return {
        'html': request.env.ref('school.school_template')._render({
            'num_student': num_student,
        })
    }
```

En este caso, por poner otro ejemplo, la renderización se realiza a través de json (type=' json') indicando el código html en el atributo *html*.

La diferencia radica en la llamada al web controller, que se tiene que realizar desde la vista.

```
<record model="ir.ui.view" id="school.student_tree">
    <field name="name">Lista de estudiante</field>
    <field name="model">school.student</field>
    <field name="arch" type="xml">
        <tree banner_route="/student/student_banner" limit="20">
            <field name="name"/>
            <field name="surname"/>
            <field name="nia"/>
        </tree>
    </field>
</record>
```

3 API REST

Odoo está más orientado a crear webs con su framework o en su URL que para hacer de API. Pero si queremos hacer una aplicación móvil o una aplicación web externa que consulte sus datos, podemos optar por crear una API REST. Para ello debemos tener en cuenta los siguientes factores:

- Hay que poner `cors="*"` para poder acceder.
- Debemos desactivar `csrf`, ya que no lo podemos usar.
- No podemos utilizar directamente la autenticación con *Odoo*. Necesitamos implementar algún protocolo para mantener la sesión, algo como *Tokens JWT*.
- En las API REST, el método de la petición es el verbo, así que hay que obtener el método para hacer cosas distintas.
- Si en el decorador ponemos `type='json'`, no puede aceptar peticiones GET, ya que no tienen un *body*. No obstante si ponemos `type='http'` hay que retornar un JSON igualmente. Depende de lo que pidamos por GET y cómo lo implementemos, podemos tener un problema al convertir de recordset a JSON con `json.dumps()`, pero lo podemos solventar con `default=str` (como hemos comentado antes) o con una herramienta interna de *Odoo* sacada de la biblioteca `odoo.tools.date_utils`.

Veamos un ejemplo:

```
@http.route('/school/api/<model>', auth="none", cors='*', csrf=False,
methods=["POST", "PUT", "PATCH"], type='json')
def apiPost(self, **args):
    print('***** API POST PUT *****')
    print(args, http.request.httprequest.method)
    model = args['model']

    if (http.request.httprequest.method == 'POST'):
        record = http.request.env['school.' + model].sudo().create(args['data'])
        return record.read()

    if (http.request.httprequest.method == 'PUT' or http.request.httprequest.method ==
'PATCH'):
        record = http.request.env['school.' + model].sudo().search([('id', '=',
args['id'])])[0]
        record.write(args['data'])
        return record.read()

    return http.request.env['ir.http'].session_info()

@http.route('/school/api/<model>', auth="none", cors='*', csrf=False, methods=["GET",
"DELETE"], type='http')
def apiGet(self, **args):
    print('***** API GET DELETE *****')
    print(args, http.request.httprequest.method)
```

```
model = args['model']
search = []

if 'id' in args:
    search = [('id', '=', args['id'])]

if (http.request.httprequest.method == 'GET'):
    record = http.request.env['school.' + model].sudo().search(search)
    return http.Response( # Retornará un array sin el formato '{"jsonrpc": "2.0"...
        json.dumps(record.read(), default=date_utils.json_default),
        status=200,
        mimetype='application/json'
    )

if (http.request.httprequest.method == 'DELETE'):
    record = http.request.env['school.' + model].sudo().search(search)[0]
    record.unlink()
    return http.Response(
        json.dumps(record.read(), default=date_utils.json_default),
        status=200,
        mimetype='application/json'
    )

return http.request.env['ir.http'].session_info()
```

En este ejemplo falta todo lo relativo a la autenticación y algunas comprobaciones para evitar errores, pero se puede ver cómo hacemos una cosa distinta en función del método HTTP. Resulta más fácil de gestionar el GET y el POST por separado por el type.

3.1 Comunicar una SPA Vue/React/Angular con Odoo

Este apartado no tiene mucho que ver con el módulo, sin embargo es interesante como enlace con el módulo *Desarrollo Web en Entorno Cliente* del CFGS DAW o como introducción a un proyecto final de ciclo. Este sería el servicio de *Angular* que hace peticiones al API REST del apartado anterior:

```
@Injectable({
  providedIn: 'root'
})

export class CourseService {
  courseURL = environment.url+'course'; // La URL está en enviroment
  postOptions = { headers: new HttpHeaders({ "Content-type": "application/json;
charset=UTF-8" })};
  constructor( private http: HttpClient) {
  }
  createCourse(course:ICourse): Observable<ICourse[]> {
    let postBody = `{"jsonrpc":"2.0","method":"call","params":{"data":"$
{JSON.parse(course)}}}`;
  }
```

```
let obs: Observable<ICourse[]> =  
  this.http.post<{result: ICourse[]}>  
(this.courseURL, this.postBody, this.postOptions)  
  .pipe(map(response => response.result))  
  return obs;  
}
```

De forma similar podría realizarse con otras bibliotecas de programación reactiva como *Vue* o *React*.

En odooinvue.org un ejemplo de como conectarse a Odoo con Vue, utilizando el framework para desarrollar código multiplataforma con [Quasar](#).

4 BIBLIOGRAFIA

1. [Documentación de Odoo](#)

5 AUTORES

A continuación ofrecemos en orden alfabético (por apellido) el listado de autores que han hecho aportaciones a este documento.

- Jose Castillo Aliaga
- Sergi García Barea
- Alfredo Oltra