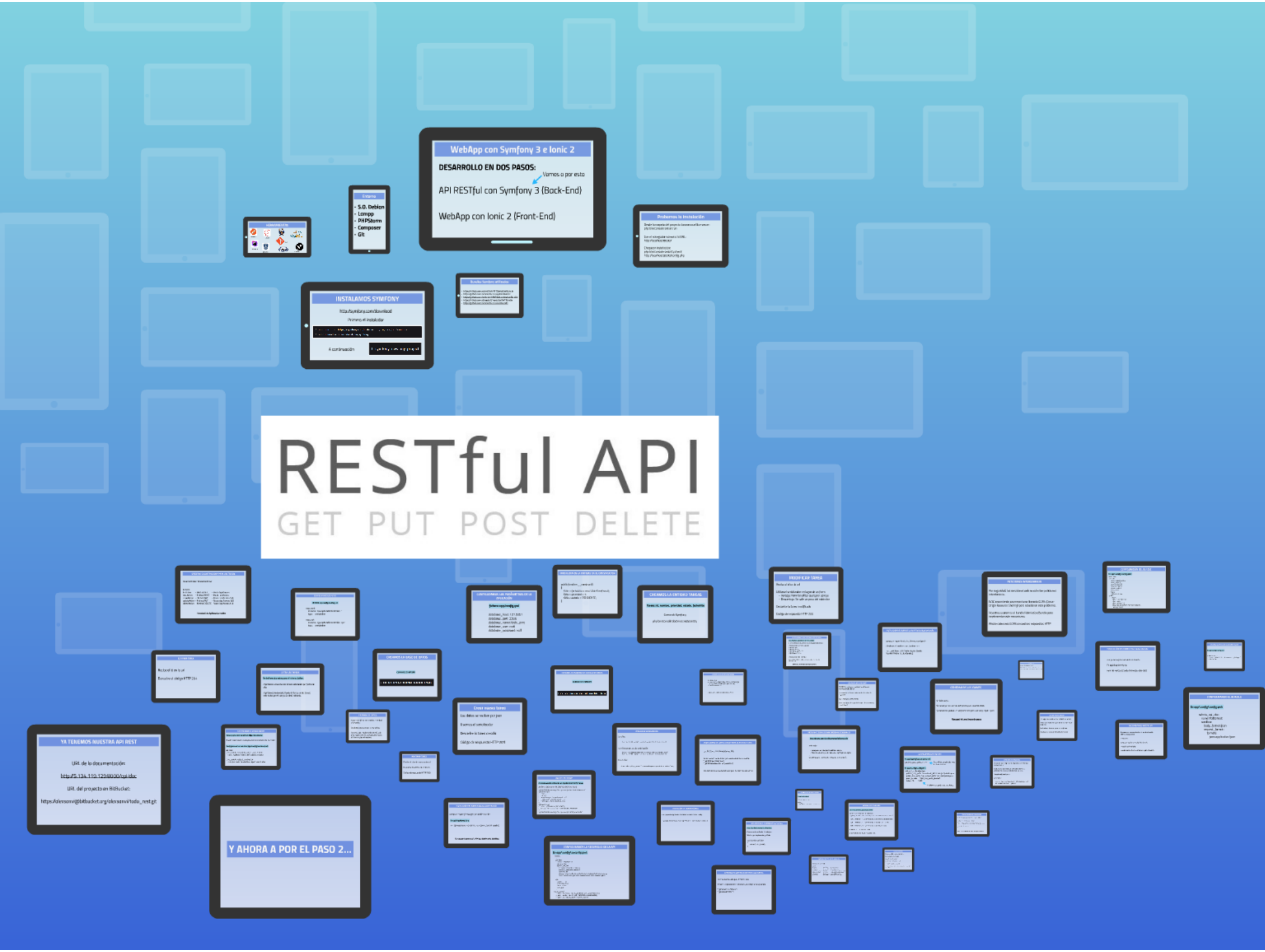
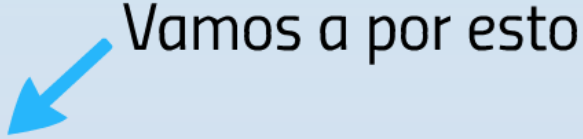

WebApp con Symfony 3 e Ionic 2
DESARROLLO EN DOS PASOS:
Vamos a por esto
API RESTful con Symfony 3 (Back-End)
WebApp con Ionic 2 (Front-End)
Entorno
- S.O. Debian
- Lompp
- PHPStorm
- Composer
- Git
Probemos la instalación
INSTALAMOS SYMFONY
A continuación
RESTful API
GET PUT POST DELETE
YA TENEMOS NUESTRA API REST
URL de la documentación:
http://5.134.119.129:8000/api/doc
URL del proyecto en BitBucket:
https://alexsanv@bitbucket.org/alexsanv/todo_rest.git
Y AHORA A POR EL PASO 2...

WebApp con Symfony 3 e Ionic 2

DESARROLLO EN DOS PASOS:



API RESTful con Symfony 3 (Back-End)

WebApp con Ionic 2 (Front-End)

HERRAMIENTAS

Postman

Debian

PHPStorm

Bitbucket

GIT

Symfony


Entorno
• S.O. Debian
• Lampp
• PHPStorm
• Composer
• Git

INSTALAMOS SYMFONY

http://symfony.com/download

Primero el instalador

```
$ sudo curl -LsS https://symfony.com/installer -o /usr/local/bin/symfony
$ sudo chmod a+x /usr/local/bin/symfony
```

A continuación

```
$ symfony new my_project
```

Bundles Symfony utilizados

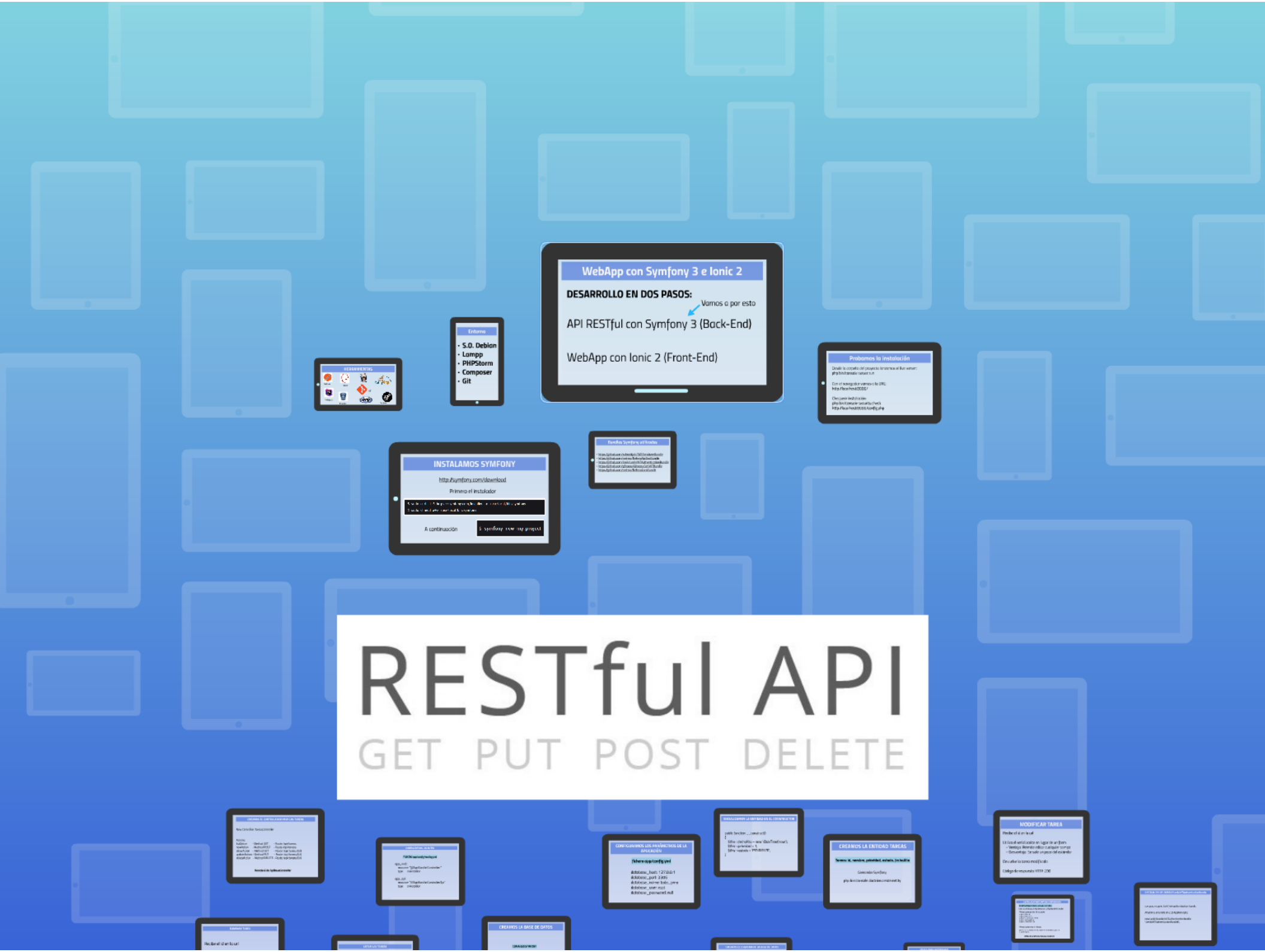
- https://github.com/schmittjoh/JMSSerializerBundle
- https://github.com/nelmio/NelmioApiDocBundle
- https://github.com/lexik/LexikJWTAuthenticationBundle
- https://github.com/gfreeau/GfreeauGetJWTBundle
- https://github.com/nelmio/NelmioCorsBundle

Probamos la instalación

Desde la carpeta del proyecto lanzamos el live server:
php bin/console server:run

Con el navegador vamos a la URL:
http://localhost:8000/

Chequear instalación:
php bin/console security:check
http://localhost:8000/config.php


WebApp con Symfony 3 e Ionic 2
DESARROLLO EN DOS PASOS:
Vamos a por esto
API RESTful con Symfony 3 (Back-End)
WebApp con Ionic 2 (Front-End)
Entorno
• S.O. Debian
• Lampp
• PHPStorm
• Composer
• Git
INSTALAMOS SYMFONY
http://symfony.com/download
Primero el instalador
A continuación
Probamos la instalación
RESTful API
GET PUT POST DELETE
CREAMOS LA ENTIDAD TAREAS
MODIFICAR TAREA

CONFIGURAMOS LOS PARÁMETROS DE LA APLICACIÓN

fichero app/config.yml

```
database_host: 127.0.0.1
database_port: 3306
database_name: todo_pmo
database_user: root
database_password: null
```

CREAMOS LA BASE DE DATOS

COMANDO SYMFONY

```
$ php bin/console doctrine:database:create
```

CREAMOS LA ENTIDAD TAREAS

Tareas: id, nombre, prioridad, estado, fechaAlta

Comando Symfony

php bin/console doctrine:create:entity

CREAMOS EL ESQUEMA DE LA BASE DE DATOS

COMANDO SYMFONY

```
$ php bin/console doctrine:schema:update --force
```

INICIALIZAMOS LA ENTIDAD EN EL CONSTRUCTOR

```
public function __construct()
{
    $this->fechaAlta = new \DateTime('now');
    $this->prioridad = 1;
    $this->estado = 'PENDIENTE';
}
```

CONTROLADOR PADRE (MÉTODOS REUTILIZABLES)

CLASE ApiBaseController extends Controller

Creamos el directorio Api dentro de src/AppBundle/Controller

Métodos para generar la respuesta

respuestaCorrecta
respuestaConErrores
respuestaUsuarioNoAutorizado
respuestaNoEncontrado
respuestaDatosNoValidos

Métodos para tratar entidades:

procesaDatos: Valida la entidad, la guarda y devuelve la respuesta
eliminaEntidad

Utiliza el serializador, hay que instalarlo

INSTALAR EL SERIALIZADOR (JMSSERIALIZERBUNDLE)

http://jmsyst.com/bundles/JMSSerializerBundle

Instalación:

- composer require jms/serializer-bundle
- Añadirlo al array de bundles de app/AppKernel.php:

...
new JMS\SerializerBundle\JMSSerializerBundle(),
...

CONFIGURAR EL SERIALIZADOR

Clase AppBundle\Serializer\ObjectConstructor

El serializador creará los objetos con el constructor de la entidad

Configuramos los servicios (app\config\services.yml)

```
services:
  jms_serializer.unserialize_object_constructor:
    class:  AppBundle\Serializer\ObjectConstructor

  jms_serializer.object_constructor:
    alias: jms_serializer.doctrine_object_constructor
```

UTILIZAR EL SERIALIZADOR

Serialize:

```
$json_result = $this->get('jms_serializer')->serialize($result, "json", $context);
```

Serialize con grupos de serialización:

```
$context = SerializationContext::create()->setGroups(array($serialization_group));
$json_result = $this->get('jms_serializer')->serialize($result, "json", $context);
```

Deserialize:

```
$tarea = $this->get('jms_serializer')->deserialize($request->getContent(), Tarea::class, "json");
```

CONFIGURAMOS LAS RUTAS

FICHERO app/config/routing.yml

```
app_web:
    resource: "@AppBundle/Controller/"
    type:     annotation

app_api:
    resource: "@AppBundle/Controller/Api"
    type:     annotation
```

CREAMOS EL CONTROLADOR PARA LAS TAREAS

New Controller: TareasController

Actions:

listAction - Method GET - Route /api/tareas
newAction - Method POST - Route /api/tareas
showAction - Method GET - Route /api/tareas/{id}
updateAction - Method PUT - Route /api/tareas/{id}
deleteAction - Method DELETE - Route /api/tareas/{id}

Heredará de ApiBaseController

LISTAR LAS TAREAS

Se definen dos rutas para el mismo Action:

/api/tareas: devuelve las tareas ordenadas por fecha de alta

/api/tareas/ordenadas/{orden}: Devuelve las tareas ordenados por el campo {orden} indicado.

ANOTACIONES DEL SERIALIZADOR PARA LA ENTIDAD TAREA

use JMS\Serializer\Annotation as JMS;

Indicamos el tipo de datos y el nombre de la fecha de alta:
* @JMS\Type("DateTime")
* @JMS\SerializedName("fechaAlta")

Por defecto no se usa Camel Case para los nombres de campo

FILTRADO DE DATOS

Se permite filtrar por nombre, prioridad y/o estado.

Los datos se pasan por query string.

Tenemos que implementar un método en el repositorio (TareaRepository) para el filtrado y la ordenación.

MOSTRAR TAREA

Recibe el id de la tarea en la url

Devuelve los datos de la tarea

Código de respuesta HTTP 200

Crear nueva tarea

Los datos se reciben por json

Usamos el serializador

Devuelve la tarea creada

Código de respuesta HTTP 201

USO DEL VALIDADOR

Anotaciones de validación en AppBundle\Entity\Tarea

```
use Symfony\Component\Validator\Constraints as Assert;

* @Assert\NotBlank(message="No se puede dejar el nombre de la tarea vacío")
* @Assert\Range(
*      min = 1,
*      max = 3,
*      invalidMessage = "La prioridad va de 1 a 3",
*      minMessage = "La prioridad va de 1 a 3",
*      maxMessage = "La prioridad va de 1 a 3"
* )
* @Assert\Choice(
*     choices = {"TERMINADA", "PENDIENTE"},
*     message = "El estado debe de ser TERMINADA o PENDIENTE"
* )
* @Assert\NotBlank(message="No se puede dejar vacía la fecha de alta")
```

VALIDACIÓN DE NOMBRE ÚNICO

```
use Symfony\Bridge\Doctrine\Validator\Constraints\UniqueEntity;


 * @UniqueEntity("nombre", message="Ya existe una tarea con ese nombre")
```

MODIFICAR TAREA

Recibe el id en la url

Utiliza el serializador en lugar de un form

- Ventaja: Permite editar cualquier campo
- Desventaja: Se sale un poco del estándar

Devuelve la tarea modificada

Código de respuesta HTTP 200

ELIMINAR TAREA

Recibe el id en la url

Devuelve el código HTTP 204

SEGURIDAD DE LA API (JWT)

API REST = stateless (el servidor no almacena información del cliente)

La seguridad se basa en intercambio de tokens de seguridad

Json Web Tokens (RFC 7519)

Genera un token de seguridad a partir de un usuario y un password

INSTALACIÓN DEL BUNDLE LexikJWTAuthenticationBundle

composer require lexik/jwt-authentication-bundle

Añadimos el bundle en app\AppKenel.php

new Lexik\Bundle\JWTAuthenticationBundle
\LexikJWTAuthenticationBundle(),

GENERAMOS LAS CLAVES

```
$ mkdir var/jwt

$ openssl genrsa -out var/jwt/private.pem -aes256 4096

$ openssl rsa -pubout -in var/jwt/private.pem -out var/jwt/public.pem
```

Nos pedirá una frase de paso

CONFIGURAMOS EL BUNDLE

En app\config\parameters.yml:

```
jwt_key_pass_phrase: alex
```

frase de paso que pusimos al crear la clave

En app/config/config.yml:

```
lexik_jwt_authentication:
    private_key_path: %kernel.root_dir%/../var/jwt/private.pem
    public_key_path:  %kernel.root_dir%/../var/jwt/public.pem
    pass_phrase:      %jwt_key_pass_phrase%
    token_ttl:        7200
```

El token caducará a las dos horas

CREAMOS LA ENTIDAD USER

User: id, username, password, avatar

Creamos las anotaciones de validación

* @UniqueEntity("username", message="Ya existe un usuario con ese nombre")

* @Assert\NotBlank(message="No se puede dejar el nombre de usuario vacío")

* @Assert\NotBlank(message="No se puede dejar el password vacío")

* @Assert\NotBlank(message="No se puede dejar el avatar vacío")

Actualizamos el esquema de la BBDD

php bin/console doctrine:schema:update --force

IMPLEMENTAMOS EL INTERFACE UserInterface

class User implements UserInterface

Generamos los métodos del interface

Sólo hay que implementar getRoles

```
public function getRoles()
{
        return ['ROLE_USER'];
}
```

CREAMOS USUARIO ADMINISTRADOR

username: alex
password: wFuohBMiGiNRQwkCUt5stJXaK9XAIwPgDn
+7E76fJmwe/hH9v5fxEBhpEJruq91FhfOa5Gj
+ZdkM7DuimxdoKw==

El password es alex encriptado con SHA512

CONFIGURAMOS LA SEGURIDAD DE USUARIO

En app/config/security.yml

```
encoders:
    AppBundle\Entity\User: sha512

providers:
    our_users:
        entity: { class: AppBundle\Entity\User, property: username }
```

INSTALACIÓN DEL BUNDLE GfreeauGetJWTBundle

composer require gfreeau/get-jwt-bundle:2.0.x-dev

En app\AppKernel.php:

new Gfreeau\Bundle\GetJWTBundle\GfreeauGetJWTBundle(),

Es necesario para que la API sea totalmente stateless

CONFIGURAMOS LA SEGURIDAD DE LA API

En app\config\security.yml:

```
firewalls:
    ...
    gettoken:
        pattern:  ^/api/tokens$
        stateless: true
        gfreeau_get_jwt:
            username_parameter: username
            password_parameter: password
            post_only: true
            success_handler: lexik_jwt_authentication.handler.authentication_success
            failure_handler: lexik_jwt_authentication.handler.authentication_failure

    api:
        pattern:   ^/api
        anonymous: true
        stateless: true
        lexik_jwt: ~
    ...
access_control:
    - { path: ^/api/tokens, roles: IS_AUTHENTICATED_ANONYMOUSLY }
    - { path: ^/api/doc, roles: IS_AUTHENTICATED_ANONYMOUSLY }
    - { path: ^/api, roles: IS_AUTHENTICATED_FULLY }
```

CREAMOS EL CONTROLADOR PARA LOS TOKENS

Lo necesitamos sólo para definir la ruta

El bundle capturará las peticiones y el código no se ejecutará

* @Route("/api/tokens")
* @Method("POST")

CONTROLADOR DEL API DE USUARIOS

New Controller: UserController

Actions:

listAction	- Method GET	- Route /api/usuarios
newAction	- Method POST	- Route /api/usuarios
showAction	- Method GET	- Route /api/usuarios/{id}
updatePasswordAction	- Method PUT	- Route /api/usuarios/{id}/password
updateAvatarAction	- Method PUT	- Route /api/usuarios/{id}/avatar
deleteAction	- Method DELETE	- Route /api/usuarios/{id}
profileAction	- Method GET	- Route /api/usuarios/profile

LISTAR LOS USUARIOS

No se debe serializar el password del usuario

Creamos un grupo de serialización

Anotación en la entidad User:

```
use JMS\Serializer\Annotation as JMS;

    * @JMS\Groups({"sinpassword"})
```

Se añadirá la anotación a todos los campos que queremos que se serialicen

ENCRIPTACIÓN DEL PASSWORD

Se debe encriptar el password con el encoder indicado

```
private function encriptaPassword(User $usuario)
{
    $encoder = $this->container->get('security.password_encoder');
    $encoded = $encoder->encodePassword($usuario, $usuario->getPassword());

    $usuario->setPassword($encoded);
}
```

Este método se utilizará en los actions new y updatePassword

DIRECTORIO DE IMÁGENES

Tendremos una imagen que se asignará por defecto a los nuevos usuarios

Crearemos un parámetro con el directorio donde se guardarán las imágenes subidas y otro con la url.

En app\config\config.yml:

parameters:

```
...
avatars_directory: '%kernel.root_dir%/../web/uploads/'
url_avatars_directory: '/uploads/'
```

ACTION UPDATE AVATAR

La imagen se enviará por Json codificada en base64

Habrá que decodificarla y guardarla en el directorio de imágenes

Se devolverá el usuario que se ha modificado

El campo avatar contendrá la url de la imagen

ACTION PROFILE

Se devolverán los datos del usuario actual

Para ello, habrá que obtener el usuario del token de seguridad

```
$usuario = $this->get('security.token_storage')->getToken()->getUser();
```

PETICIONES INTERDOMINIO

Por seguridad, los servidores web no admiten peticiones interdominio.

W3C recomienda un mecanismo llamado CORS (Cross-Origin Resource Sharing) para solucionar este problema.

Nosotros usaremos el bundle NelmioCorsBundle para implementar este mecanismo.

Añade cabeceras CORS a nuestras respuestas HTTP

INSTALACIÓN DEL BUNDLE NelmioCorsBundle

composer require nelmio/cors-bundle

En app\AppKernel.php:

new Nelmio\CorsBundle\NelmioCorsBundle(),

CONFIGURACIÓN DEL BUNDLE

En app\config\config.yml:

```
nelmio_cors:
    defaults:
        allow_credentials: false
        allow_origin: []
        allow_headers: []
        allow_methods: []
        expose_headers: []
        max_age: 0
        hosts: []
        origin_regex: false
    paths:
        '^/api/':
            allow_credentials: true
            allow_origin: ['*']
            allow_headers: ['*']
            allow_methods: ['POST', 'PUT', 'GET', 'DELETE']
            max_age: 7200
```

DOCUMENTANDO NUESTRA API

Crearemos la documentación utilizando el bundle NelmioApiDocBundle

Instalación:

composer require nelmio/api-doc-bundle

En app\AppKernel.php:

new Nelmio\ApiDocBundle\NelmioApiDocBundle(),

CREAMOS LAS RUTAS DE LA DOCUMENTACIÓN

En app\config\routing.yml:

```
NelmioApiDocBundle:
    resource: "@NelmioApiDocBundle/Resources/config/routing.yml"
    prefix:   /api/doc
```

CONFIGURANDO EL BUNDLE

En app\config\config.yml:

```
nelmio_api_doc:
    name: TODO Rest
    sandbox:
        body_format: json
        request_format:
            formats:
                json: application/json
```

YA TENEMOS NUESTRA API REST

URL de la documentación:

http://5.134.119.129:8000/api/doc

URL del proyecto en BitBucket:

https://alexsanvi@bitbucket.org/alexsanvi/todo_rest.git

Y AHORA A POR EL PASO 2...