El Diseño API REST , es una de las cosas que mas buenas practicas y opciones soporta. Al tratarse de un “Estilo” y no tanto de un “patronaje” existen muchas opciones a la hora de diseñar un API REST . Hoy voy a hablar en este artículo de uno de los temas claves de las APIS REST . ¿Cómo conseguir un mayor rendimiento? . Existen muchas casuísticas que se pueden abordar pero una de las principales es el concepto de OverFeching. Es decir cuando nuestro API REST nos devuelve más información de la que realmente necesitamos y queremos optimizarla.

[ihc-hide-content ihc_mb_type=”show” ihc_mb_who=”4″ ihc_mb_template=”1″]

Diseño API REST

Para construir este ejemplo vamos a apoyarnos en la clase Persona . Esta clase tiene la siguiente estructura:

```
package com.arquitecturajjava.web;

public class Persona {

  private String nombre;
  private String apellidos;
  private String profesion;
  private int edad;
  public String getNombre() {
    return nombre;
  }
  public void setNombre(String nombre) {
    this.nombre = nombre;
  }
  public String getApellidos() {
    return apellidos;
```

```
  }
  public void setApellidos(String apellidos) {
    this.apellidos = apellidos;
  }
  public String getProfesion() {
    return profesion;
  }
  public void setProfesion(String profesion) {
    this.profesion = profesion;
  }
  public int getEdad() {
    return edad;
  }
  public void setEdad(int edad) {
    this.edad = edad;
  }
  public Persona(String nombre, String apellidos, String profesion,
int edad) {
    super();
    this.nombre = nombre;
    this.apellidos = apellidos;
    this.profesion = profesion;
    this.edad = edad;
  }
}
```

Una vez tenemos la clase persona usaremos un proyecto de Spring Boot que contiene el starter web para arrancar :

```
<dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
```

```
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
      <scope>test</scope>
    </dependency>
  </dependencies>
```

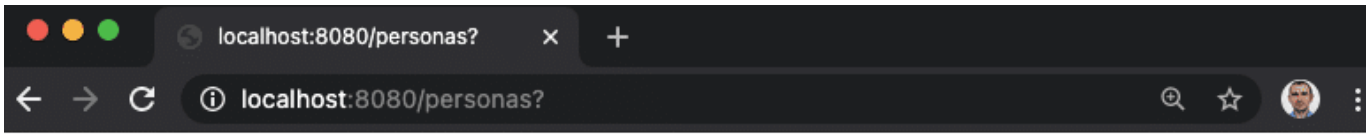
REST Controller Clásico

Es momento de construir un @RestController que nos devuelva la lista de todas las Personas . Se trata de una de las operaciones más sencillas que existen a nivel de API REST y Spring

```
package com.arquitecturajjava.web;

import java.lang.reflect.Field;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/personas")
public class PersonaREST {
```

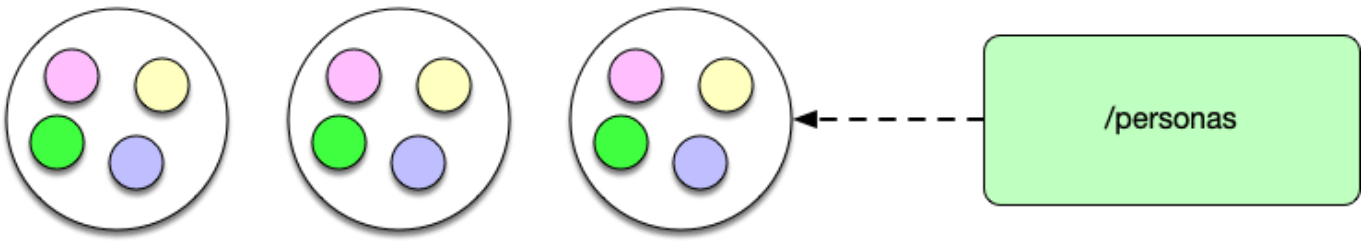
```
  static List<Persona>  lista= new ArrayList<Persona>();
  static {
    Persona p1= new Persona("pedro","perez","informatico",20);
    Persona p2= new Persona("angel","gomez","administrativo",40);
    Persona p3= new Persona("ana","blanco","ingeniera",50);
    lista.add(p1);
    lista.add(p2);
    lista.add(p3);
  }
  @RequestMapping
  public List<Persona> buscarTodas() {
    return lista;
  }
}
```

Esto nos permite que cuando lancemos la aplicación web podamos solicitar la lista de todas las Personas.




```
[{"nombre":"pedro","apellidos":"perez","profesion":"informatico","edad":20},
{"nombre":"angel","apellidos":"gomez","profesion":"administrativo","edad":40},
{"nombre":"ana","apellidos":"blanco","profesion":"ingeniera","edad":50}]
```


Todo es correcto disponemos de la url de Personas para acceder a toda la información:

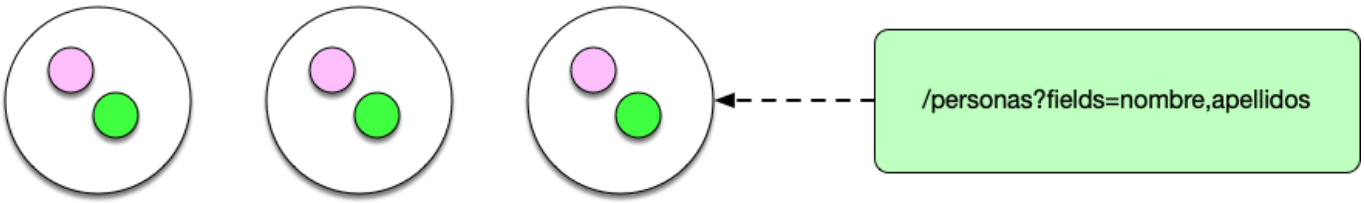


A veces aparece un problema y es que no queremos obtener todos los datos de la Persona sino simplemente un subconjunto que es lo que realmente necesita nuestra aplicación cliente. Es decir necesitamos aumentar la flexibilidad de nuestra API para eliminar un problema de OverFetching (obtener mas información de la que realmente necesitamos). ¿Cómo podemos abordar esto? . Para ello debemos hacer dos cosas una registrar una nueva url que permita filtrar los campos que nosotros realmente necesitamos . En segundo lugar usar el api de Reflection para generar esos niveles de flexibilidad en Java y publicar una Lista que contenga un Map con todas las propiedades que nosotros necesitamos.

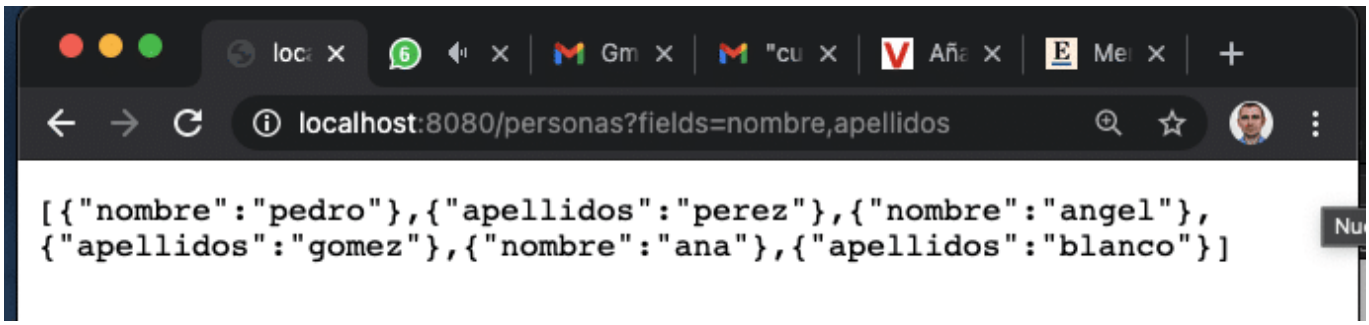
```
  @RequestMapping(params="fields")
  public List<Map<String,Object>> buscarTodasFiltradas(@RequestParam
String fields) {

    List<Map<String,Object>> listaMapa=new
ArrayList<Map<String,Object>>();
    String[] campos= fields.split(",");
    for (Persona p: lista) {
      Field[] fields2=p.getClass().getDeclaredFields();
      for (String campo :campos) {
        Map<String,Object> mapa= new HashMap<String,Object>();
        try {
          Field campoReflection=p.getClass().getDeclaredField(campo);
          campoReflection.setAccessible(true);
          mapa.put(campo,campoReflection.get(p));
        } catch (IllegalArgumentException | IllegalAccessException |
NoSuchFieldException
            | SecurityException e) {
          e.printStackTrace();

        }
        listaMapa.add(mapa);
```

```
        }
      }
      return listaMapa;
    }
}
```

En este caso lo que estamos usando es el api de Reflection para acceder a los campos que el usuario ha solicitado generar un mapa con las propiedades que realmente queremos . De esta forma el api gana en flexibilidad.



Si volvemos a lanzar la aplicación de Spring Boot y invocamos la nueva url con los campos que queremos de nuestros objetos.



Siempre es importante diseñar un API REST que sea flexible:

Otros artículos relacionados

1. REST API y su inmutabilidad
2. API REST estilos y homogeneidad

3. REST Resources Nombres vs Verbos
4. API REST

[/ihc-hide-content]