# MODULO BOOTSTRAP

## DESCRIPCION DEL CASO DE ESTUDIOS

Se solicita a los desarrolladores docentes de la cátedra de Programación Visual realizar una aplicación que permita gestionar los procesos de la información que corresponden a la cursadas de materias. Se conjugará en la forma de tutoriales donde se irán desarrollando las funcionalidades a medida que se vayan incorporando más tecnologías.

## LOS REQUISITOS FUNCIONALES

Las funcionalidades de la aplicación se describen como requisitos funcionales: aquello que hace el sistema. Este aspecto se puede formalizar de muchas maneras, en este caso se usarán casos de uso. Esta técnica no corresponde a esta materia, sin embargo, se utilizará debido a que en si, no es compleja de utilizar.

Observe el siguiente requisito funcional a desarrollar.

| **RF01**: El sistema permite visualizar el listado de materias que dicta la cátedra |
|---|
| **Prioridad**: Alta |
| **Descripción**: El sistema permite visualizar un listado de las materias que dicta la cátedra. Se visualiza el Id de la materia, su nombre y la carrera a la que pertenece. |

Para este requisito funcional se describe el siguiente caso de Uso:

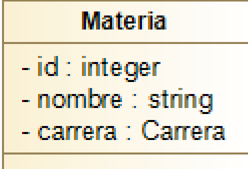
<table>
<tr><td>Caso de Uso: Visualizar Listado de materias</td><td>Codificación: UC01</td></tr>
<tr><td colspan="2">Actor Principal: Usuario</td></tr>
<tr><td colspan="2">Personal involucrado e intereses:<br><u>Usuario</u>: son los docentes de la cátedra que desean poder visualizar las materias que dicta la cátedra.</td></tr>
<tr><td colspan="2">Precondiciones: Conectividad al servidor garantizada</td></tr>
<tr><td colspan="2">Postcondicionales: En la pantalla se observarán las materias de la cátedra</td></tr>
<tr><td colspan="2">Flujo normal:</td></tr>
<tr><td>Actor</td><td>Sistema</td></tr>
<tr><td>1. Solicita la página del listado de materias.</td><td>2. Busca las materias registradas.<br>3. Genera la página del listado de materias.<br>4. Envía el resultado</td></tr>
<tr><td colspan="2">Flujos alternativos:<br>(2) No ubica las materias registradas<br>(3) No existen materias registradas</td></tr>
<tr><td colspan="2">Requisitos funcionales asociados:<br>RF01: El sistema permite visualizar el listado de materias que dicta la cátedra</td></tr>
</table>

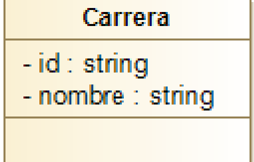
## EL DISEÑO ESTRUCTURAL DE LA APLICACION

Mínimamente necesitamos determinar las clases que serán responsables de almacenar los datos de las materias que se visualizarán. Entonces podemos representarlos de diferentes maneras. En este caso se usarán diagrama de clases, sin entrar en detalles, las clases se representan en cajas,

divididas en tres partes: en la parte superior se indica el nombre de la clase, en la parte intermedia se indican los atributos de la clase y en la tercera parte sus métodos.

Para este caso sería algo similar a lo siguiente:




## IMPLEMENTACIÓN DE LA SOLUCIÓN

### Las clases del dominio

Observe las siguientes imágenes

```java
package ar.edu.unju.fi.sistmaterias.model;

import org.springframework.stereotype.Component;

@Component
public class Carrera {
    private int id;
    private String nombre;

    public Carrera() {
        // TODO Auto-generated constructor stub
    }

    public Carrera(int id, String nombre) {
        this.id = id;
        this.nombre = nombre;
    }

    public int getId() {
        return id;
    }

    public String getNombre() {
        return nombre;
    }
}
```

```java
package ar.edu.unju.fi.sistmaterias.model;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class Materia {
    private int id;
    private String nombre;
    @Autowired
    private Carrera carrera;

    public Materia() {
        // TODO Auto-generated constructor stub
    }

    public Materia(int id, String nombre, Carrera carrera) {
        super();
        this.id = id;
        this.nombre = nombre;
        this.carrera = carrera;
    }

    public int getId() {
        return id;
    }
    public String getNombre() {
        return nombre;
    }

    public Carrera getCarrera() {
        return carrera;
    }

}
```

En primer lugar, se observa que ambas clases poseen una anotación @Component.

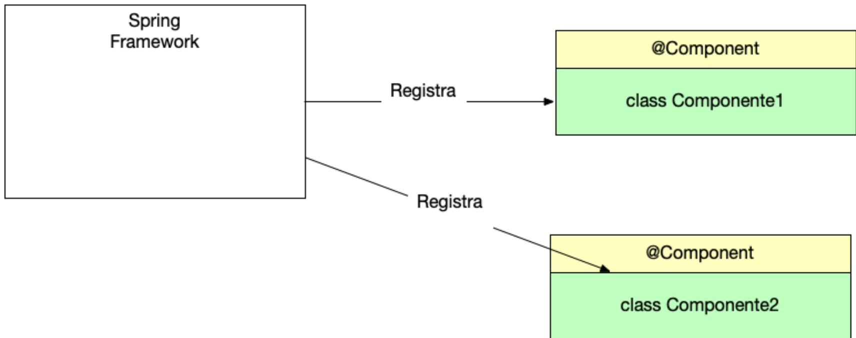
Spring @Component es una de las anotaciones fundamentales de Spring Framework a la hora de dar de alta los distintos beans con su motor de inyección de dependencia.

A nivel de Spring Framework esta anotación simplemente registra un bean dentro del framework sin mayor efecto. Es decir, si disponemos de una clase como la siguiente:

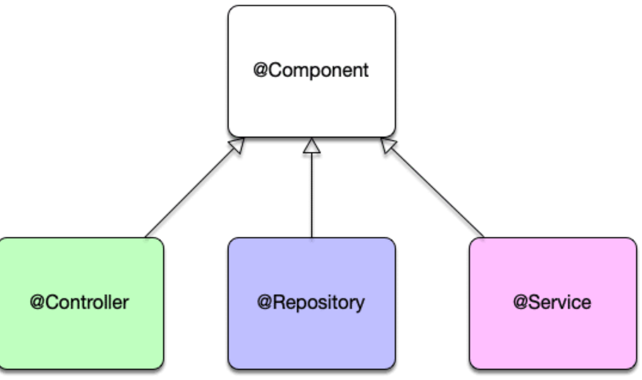
```java
@Component
public class MiComponente {
}
```

Spring la registrará dentro del framework y podremos hacer uso de ella sin problema. Esto es, Spring se encarga de su instanciación y además lo coloca en un espacio denominado Spring

Container. Cada vez que se necesite trabajar con ese objeto Spring lo extraerá del contenedor. Esto tiene varias ventajas, en primer lugar, se libera al desarrollador de la instanciación y posibles errores NullPointerExcepcion. Por otro lado, el contenedor de Spring está optimizado.




Para agregar sobre esta anotación, es la anotación padre de todas las anotaciones específicas que el framework soporta a nivel de inyección de dependencia como son @Service @Repository y @Controller




| La inyección de dependencias es un patrón de diseño que tiene como objetivo tomar la responsabilidad de crear las instancias de las clases que otro objeto necesita y suministrárselo para que esta clase los pueda utilizar |
| --- |

¿Cómo se da cuenta Java que un objeto de estas clases se halla instanciado en el Spring Container? A través, de la anotación @Autowired.

Observe la clase Materia, su atributo carrera tiene asociado el @Autowired. Esto indica que ese atributo está en el contenedor de Spring. Por lo tanto, Java debe buscarlo en ese contenedor, y no se requiere instanciarlo.

Entonces, es la anotación @Autowired la que "inyecta la dependencia", ya que indica que el objeto está en el contenedor de Spring y entonces lo asocia al atributo.

## El controlador

Vamos a generar un controlador que se encargue de cargar la página. Además, dado que no tenemos un motor de persistencia real desde el cual obtener las materias, vamos a simularla en ese lugar.

Nuestro MateriasController posee la siguiente forma:

```java
1  package ar.edu.unju.fi.sistmaterias.controller;
2
3⊕ import java.util.List;
12
13 @Controller
14 public class MateriasController {
15⊖     @Autowired
16     private List<Materia> materias;
17
18⊖     @GetMapping("/materias")
19     public String getMateriasPage(Model model) {
20         System.out.println(materias.get(0));
21         if(materias.size()==1 && materias.get(0).getNombre()==null) {
22             Materia unaMateria = new Materia(1, "Programación Visual", new Carrera(1, "APU 2008"));
23             Materia otraMateria = new Materia(2, "Programación Orientada a Objetos", new Carrera(1, "APU 2008"));
24             materias.add(unaMateria);
25             materias.add(otraMateria);
26         }
27
28         model.addAttribute("materias", materias);
29         return "materias";
30     }
31 }
```

Observe que se captura la petición "/materias". Por otro lado se crean dos objetos de tipo Materia y se loa agrega al listado de materias. Observe que ese listado también ha sido generado mediante inyección de dependencias mediante la anotación @Autowired. El listado de materias es agregado a model, el cual de manera similar a ModelAndView es gestionado por el SpringFramework. Esto posibilita que el listado esté disponible por thymeleaf en la plantilla de la página html. Finalmente se invoca la página materias.html.

## La vista

A continuación, se presenta la forma de la página web materias.html

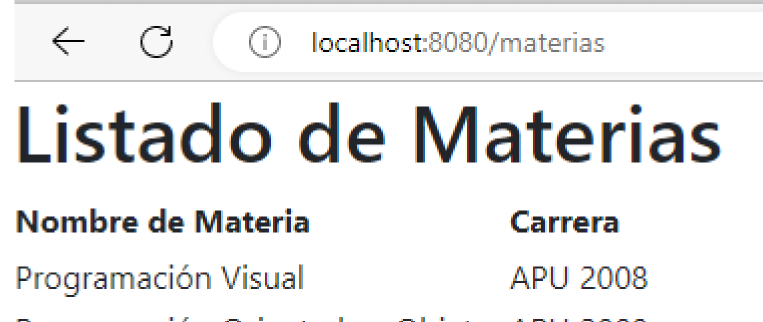
```html
1  <!doctype html>
2⊖ <html lang="es"
3        xmlns:th="http://www.thymeleaf.org">
4⊖   <head>
5      <meta charset="utf-8">
6      <meta name="viewport" content="width=device-width, initial-scale=1,maximum-scale=1, user-scalable=no">
7      <title>Gestion de Materias</title>
8      <link th:rel="stylesheet" type="text/css" th:href="@{/webjars/bootstrap/css/bootstrap.min.css}"/>
9    </head>
10⊖  <body>
11     <h1>Listado de Materias</h1>
12⊖    <table>
13⊖        <thead>
14⊖            <tr>
15                <th>Nombre de Materia</th>
16                <th>Carrera</th>
17            </tr>
18        </thead>
19⊖        <tbody>
20⊖            <div th:if="!${#lists.isEmpty(materias)}">
21⊖                <tr th:each = "materia : ${materias}">
22                    <td th:text = "${materia.nombre}"></td>
23                    <td th:text="${materia.carrera.nombre}"></td>
24                </td>
25                </tr>
26            </div>
27⊖            <div th:if="${#lists.isEmpty(materias)}">
28⊖                <tr>
29                    <td>No existen materias</td>
30                </tr>
31            </div>
32        </tbody>
33    </table>
34
35    <script type="text/javascript" th:src="@{/webjars/bootstrap/js/bootstrap.min.js}"></script>
36  </body>
37 </html>
```

Esta página esta preparada para que se utilice Bootstrap.

Por otro lado, el único aspecto interesante por recalcar se halla en la línea 23, donde mediante Expression Language se accede al nombre de la carrera a la cual pertenece la materia.

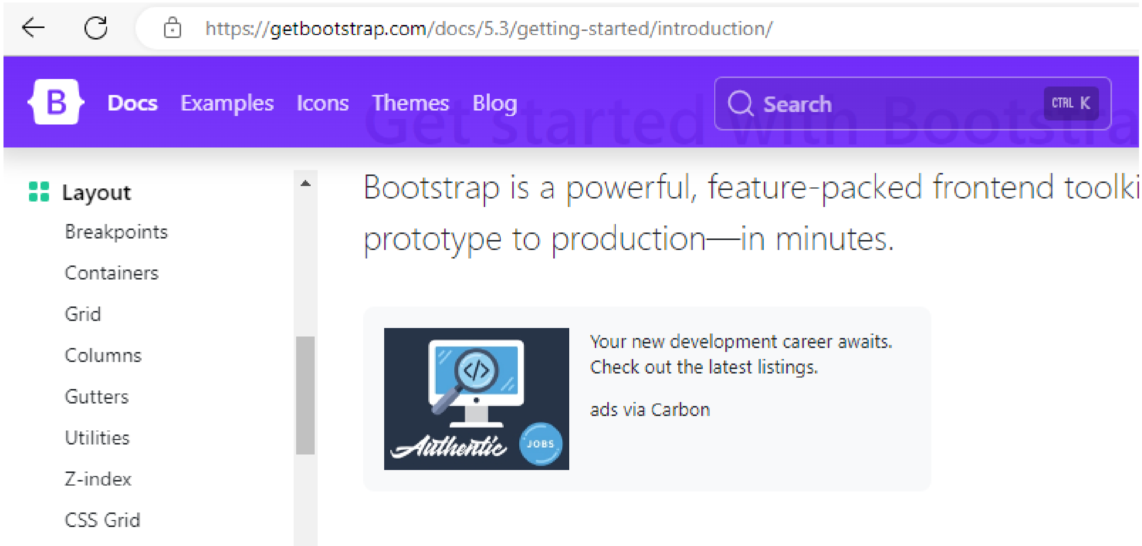
## APLICANDO ESTILOS CON BOOTSTRAP

El aspecto visual de nuestra página es el siguiente




Mediate, Bootstrap, empezaremos a cambiar esto.

En primer lugar, la idea central será centrar el contenido. En la ventana del sitio web de Bootstrap podemos visitar la sección de Layouts




Vamos a seleccionar los containers. Los contenedores son un bloque de construcción fundamental de Bootstrap que contienen, rellenan y alinean su contenido dentro de un dispositivo o ventana gráfica determinada.

Los contenedores son el elemento de diseño más básico en Bootstrap y son necesarios cuando se utiliza el sistema de cuadrícula predeterminado que ofrece sistema para el diseño responsive. Los contenedores se utilizan para contener, rellenar y (a veces) centrar el contenido dentro de ellos. Si bien los contenedores se pueden anidar, la mayoría de los diseños no requieren un contenedor anidado.

Según la documentación de Bootstrap existen diferentes contenedores responsivos:

## Responsive containers

Responsive containers allow you to specify a class that is 100% wide until the specified breakpoint is reached, after which we apply `max-width`s for each of the higher breakpoints. For example, `.container-sm` is 100% wide to start until the `sm` breakpoint is reached, where it will scale up with `md`, `lg`, `xl`, and `xxl`.

```
<div class="container-sm">100% wide until small breakpoint</div>
<div class="container-md">100% wide until medium breakpoint</div>
<div class="container-lg">100% wide until large breakpoint</div>
<div class="container-xl">100% wide until extra large breakpoint</div>
<div class="container-xxl">100% wide until extra extra large breakpoint</div>
```

Los puntos de ruptura indican el punto o tamaño tanto en alto o en ancho a partir de la cual la página empieza a cambiar la estructura de la ubicación de los componentes, e incluso la forma de los componentes. Existen plantillas de estos valores.

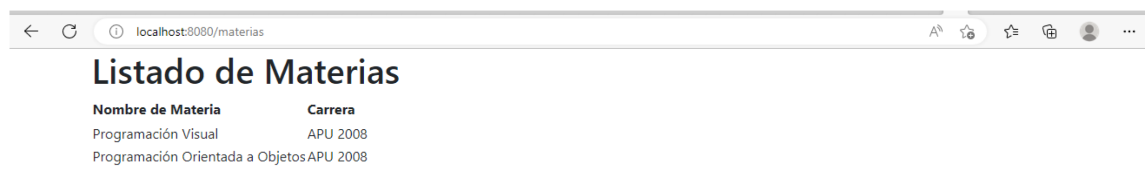
La siguiente tabla ilustra cómo se compara el ancho máximo de cada contenedor con el .container y el .container-fluid originales en cada punto de interrupción.

| | **Extra small** <576px | **Small** ≥576px | **Medium** ≥768px | **Large** ≥992px | **X-Large** ≥1200px | **XX-Large** ≥1400px |
|---|---|---|---|---|---|---|
| `.container` | 100% | 540px | 720px | 960px | 1140px | 1320px |
| `.container-sm` | 100% | 540px | 720px | 960px | 1140px | 1320px |
| `.container-md` | 100% | 100% | 720px | 960px | 1140px | 1320px |
| `.container-lg` | 100% | 100% | 100% | 960px | 1140px | 1320px |
| `.container-xl` | 100% | 100% | 100% | 100% | 1140px | 1320px |
| `.container-xxl` | 100% | 100% | 100% | 100% | 100% | 1320px |
| `.container-fluid` | 100% | 100% | 100% | 100% | 100% | 100% |

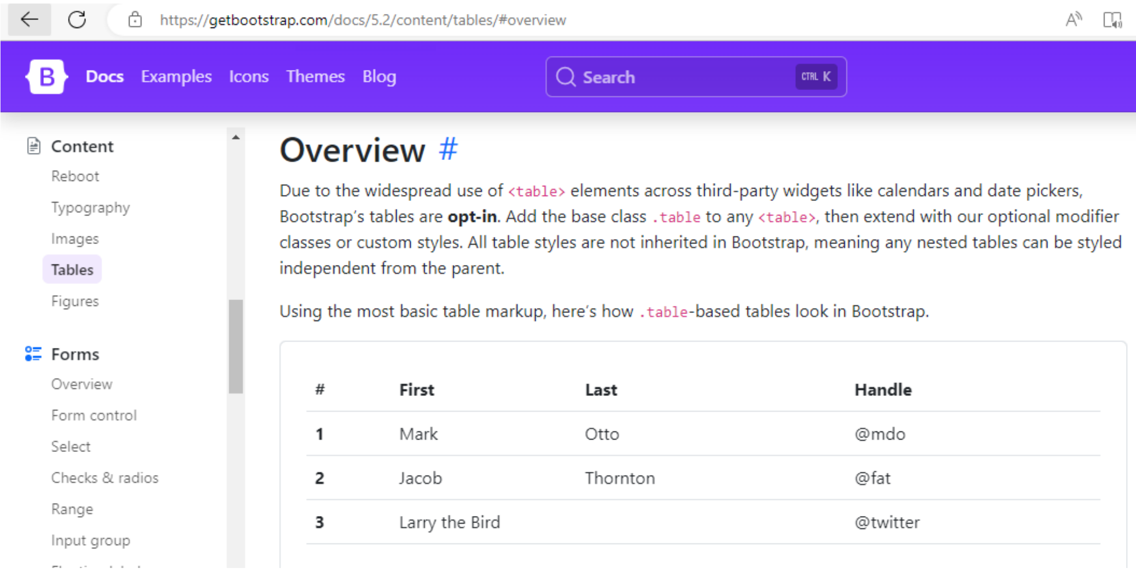
Para aplicar estas propiedades simplemente agregamos un atributo de clase, por ejemplo, a un div:

```
<div class="container-sm">
    <h1>Listado de Materias</h1>
      <table>
          <thead>
              <tr>
                  <th>Nombre de Materia</th>
                  <th>Carrera</th>
              </tr>
          </thead>
          <tbody>
              <div th:if="!${#lists.isEmpty(materias)}">
                  <tr th:each = "materia : ${materias}">
                      <td th:text = "${materia.nombre}"></td>
                      <td th:text="${materia.carrera.nombre}"></td>
                  </td>
                  </tr>
              </div>
              <div th:if="${#lists.isEmpty(materias)}">
                  <tr>
                      <td>No existen materias</td>
                  </tr>
              </div>
          </tbody>
      </table>
</div>
```

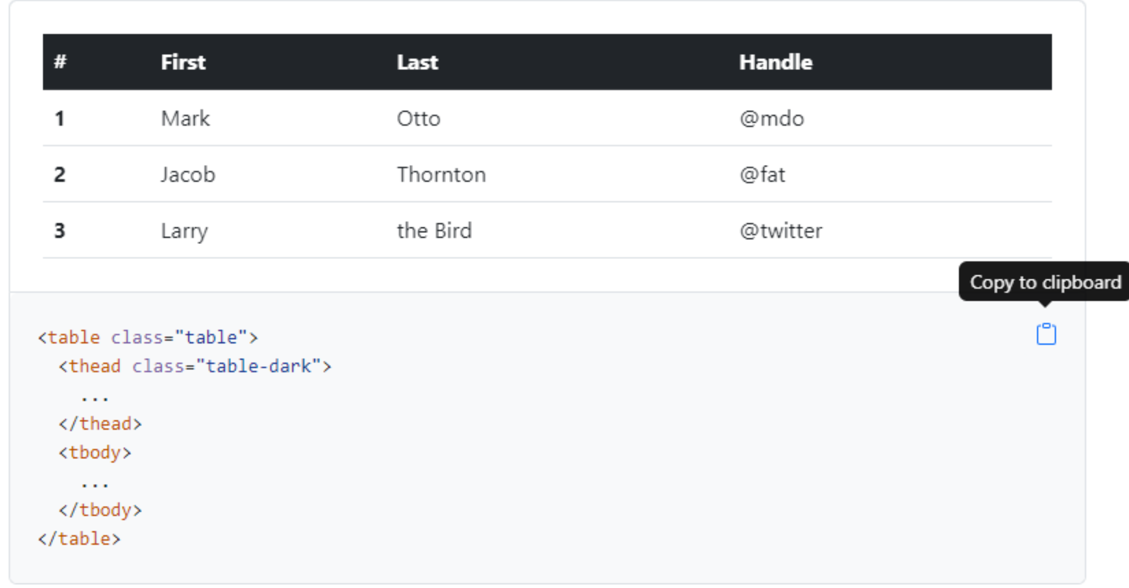
Lo cual genera el siguiente resultado:




Ahora agregaremos un estilo a la tabla. En la página de Bootstrap podemos buscar estilos para la tabla en su buscador



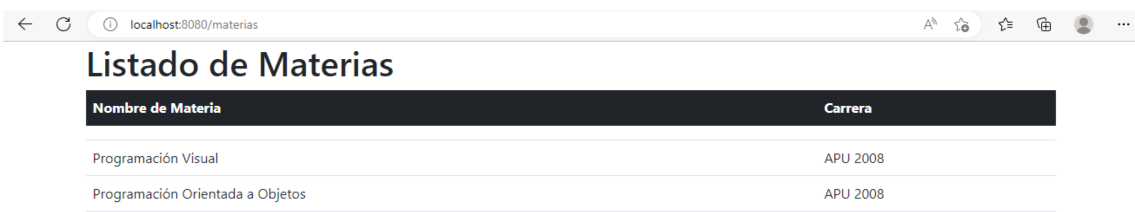

Ahora buscamos alguna de las plantillas que nos guste, por ejemplo, el siguiente permite brindar un estilo con colores. Como puede observar simplemente debemos indicar los estilos mediante los atributos adecuados. Como nosotros ya hemos creado la tabla podemos directamente aplicar los estilos indicados en la plantilla:



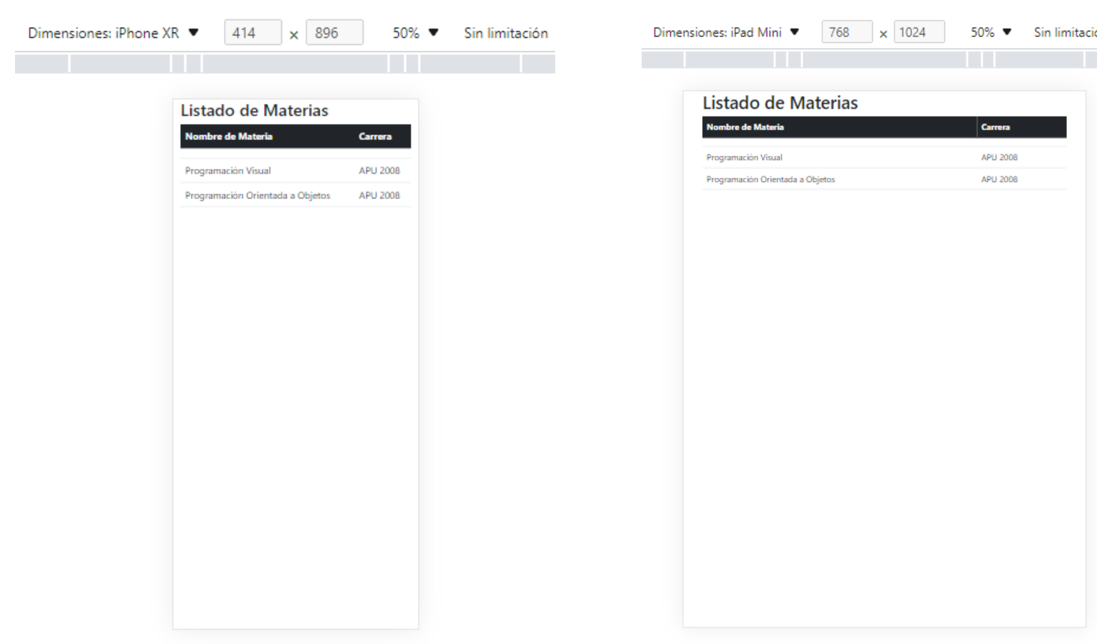
Con lo cual para nuestro ejemplo queda así

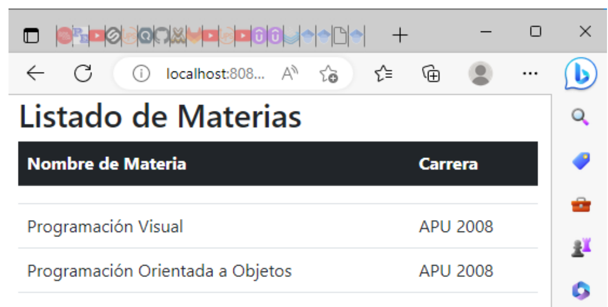
```html
<table class="table">
    <thead class="table-dark">
        <tr>
            <th>Nombre de Materia</th>
            <th>Carrera</th>
        </tr>
    </thead>
    <tbody>
        <div th:if="!${#lists.isEmpty(materias)}">
            <tr th:each = "materia : ${materias}">
                <td th:text = "${materia.nombre}"></td>
                <td th:text="${materia.carrera.nombre}"></td>
            </td>
            </tr>
        </div>
        <div th:if="${#lists.isEmpty(materias)}">
            <tr>
                <td>No existen materias</td>
            </tr>
        </div>
    </tbody>
</table>
```

Y visualmente se observa así:




Con lo cual, el aspecto visual se ha modificado enormemente. Además, esto es responsive:






De esta manera hemos introducido al uso de componentes o plantillas de Bootstrap