# MANUAL DE IMPLANTACIÓN

Alejandro Llorente Corral

IES POLIGONO SUR 2ºDAW

# Índice

# Introducción

La aplicación que he desarrollado para el proyecto integrado está dividida en dos partes, por un lado la parte servidor, en el que he desarrollado un servicio REST en JAVA utilizando como principal framework, SpringFramework con Spring Boot, Hibernate y JPA; Y la parte cliente que he usado como lenguaje JavaScript utilizando como framework AngularJS.

Para la creación y el manejo de la base de datos he utilizado MySQL, utilizando la herramienta MySQL WorkBench.

# Servidor

Para el desarrollo de la parte servidor he usado eclipse, el proyecto tiene una estructura maven y utiliza JDK 1.8. Para la creación de la estructura principal del proyecto he utilizado la herramienta que nos proporciona Spring Boot en esta página https://start.spring.io/ con la que nos da la facilidad de darle un nombre a nuestro proyecto y añadirle las dependencias que vamos a necesitar.

## Configuración

Para el desarrollo de esta aplicación he añadido las siguientes dependencias utilizando la aplicación que nos proporciona la web que he descrito arriba.

### Dependencias

**- spring-boot-starter-data-jpa:** esta dependencia nos dará la posibilidad de usar JPA e Hibernate para el manejo de las entidades y relaciones de las base de datos, pudiendo anotar una clase como una entidad de una tabla de la base de datos.

**- spring-boot-starter-web:** esta dependencia lleva implícita una gran cantidad de dependencias anidadas y es la que lleva todo el potencial del framework, dándonos la posibilidad de hacer la gran mayoría de las configuraciones por anotaciones.

**- mysql-connector-java:** esta dependencia es la que se encarga de gestionar las conexiones con la base de datos que como su propio nombre indica es MySQL.

**- jackson-core:** Esta dependencia se encargara de validar y traducir los JSON que maneje toda la aplicación.

**- commons-logging:** Con esta dependencia podre pintar trazas en la consola del servidor, que me serán de mucha utilidad para depurar.

**- spring-boot-starter-mail:** Con esta dependencia añadiré la API para mandar email de JAVA

Una vez generado el paquete con dichas dependencias importare el paquete en eclipse y ejecutare un MAVEN install para que se nos cree la estructura básica del proyecto.

## Application.properties

Este archivo se nos generara solo vacío y en el iremos añadiendo las distintas configuraciones

Con esta configuración añadiremos la conexión a nuestra base de datos indicándole cuál será su url, nombre de la base de datos el usuario usado y su contraseña.

```
spring.datasource.url = jdbc:mysql://localhost:3306/ProyectoIntegradoBd?useSSL=false
spring.datasource.username = root
spring.datasource.password = root114
```

Con las siguientes configuraciones mantendremos las conexiones activas y pintaremos trazas sobre la interacción con la base de datos lo cual nos servirá bastante para saber que vamos ejecutando en cada momento, así como el dialecto de nuestra base de datos, y en último lugar en el puerto en el que queramos que si inicie nuestra aplicación.

```
spring.datasource.testWhileIdle = true
spring.datasource.validationQuery = SELECT 1
spring.jpa.show-sql = true
spring.jpa.hibernate.ddl-auto = none
spring.jpa.hibernate.naming-strategy = org.hibernate.cfg.ImprovedNamingStrategy
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL5Dialect

server.port = 8080
```

Y por último para poder mandar los emails le aplicaremos las siguientes propiedades

```
#Propiedades para envio de correos
spring.mail.protocol=smtp
spring.mail.host=smtp.gmail.com
spring.mail.port=587
spring.mail.username= TU CUENTA DE CORREO
spring.mail.password= CONTRASEÑA DE ESE CORREO
spring.mail.properties.mail.smtp.auth = true
spring.mail.properties.mail.smtp.starttls.enable = true
```

## Control de acceso HTTP (CORS)

Por razones de seguridad, los exploradores restringen las solicitudes HTTP de origen cruzado iniciadas dentro de un script. Por ejemplo, XMLHttpRequest sigue la política de mismo-origen. Por lo que, una aplicación usando XMLHttpRequest solo puede hacer solicitudes HTTP a su propio dominio, como mi aplicación está dividida en dos partes me hará falta configurar el CORS añadiéndole este @Bean a la clase principal de proyecto.

```
@Bean
public WebMvcConfigurer corsConfigurer() {
    return new WebMvcConfigurerAdapter() {
        @Override
        public void addCorsMappings(CorsRegistry registry) {
            registry.addMapping("/**");
        }
    };
}
```

Además de añadir este @Bean también tendremos que añadir algo más en los controladores que explicare más adelante

## Modelos

En el paquete com.proyectoIntegrado.model es donde voy a crear los diferentes POJOS de la aplicación, cada POJO será hará una referencia a una entidad de la base de datos, para así poder manejar correctamente los datos de las base de datos, como dispongo de bastantes modelos iré explicando lo más relevante de cada uno de ellos.

Empezare con la clase Alumno que es la que más anotaciones y configuraciones tiene, como todo POJO, en alumno tendremos todas las propiedades privadas del alumno con lo setter y getter correspondientes.

### Anotaciones de JPA:

Con la anotacion @Entity le estaremos diciendo que esta clase se comporte como una entidad de la base de datos y con @Table estaremos diciendo como se va a llamar esta entidad en la base de datos.

```
@Entity
@Table(name = "alumno")
public class Alumno implements Serializable {
```

Con la anotación @Id estaremos declarando esta propiedad como el id de esta tabla en la base de datos y con @GeneratedValue estaremos diciendo que este id se ira generando al persistir en la base de datos y será autoincrementado

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Integer id;
```

Con @Column estaremos diciendo que esa propiedad de mi clase será una columna de esa tabla de la base de datos y con name le estaremos dando un nombre a esa columna

```
@Column(name = "nombre")
private String nombre;
```

### Relaciones:

Para poder relacional varias clases o entidades de nuestro modelo como la tenemos relacionadas en la base de datos también se usan anotaciones.

Relación uno a uno: cuando existe una relación uno a uno en nuestra base de datos tendremos que crear una propiedad de en nuestra clase que sea una instancia de la entidad con la que se relaciona, con @OnetoOne estaremos diciendo de que tipo es nuestra relación, y con la anotación @JoinColumn estaremos marcando que esta tabla es la dueña de la relación y que por tanto esta tabla tendrá una columna que contendrá en este caso el id del usuario. Con CascadeType.ALL conseguiremos que si se actualiza algún valor de esa propiedad también se actualizara en la base en el otro lado de la relación

```
@OneToOne(cascade = CascadeType.ALL)
@JoinColumn(name = "usuario_id")
private Usuario usuario;
```

Relación muchos a muchos: para este tipo de relaciones JPA nos asiste con una serie de anotaciones pero en los casos que tengo nos valen puesto que las tablas intermedias que se crean tienen atributos propios por lo que tendremos que partir la relación en uno a muchos y muchos a uno. Y tendremos que crearnos también esa clase que haga de intermedia en la relación.

Para ello tendremos que crearnos una propiedad que contendrá una lista de la clase intermedia, y al no ser la tabla alumno la dueña de la relación ya que esta será Alumno_ciclo, solo tendremos que mapearla para decirle que será un alumno.

```
@OneToMany(mappedBy = "alumno")
private List<Alumno_ciclo> alumno_ciclo;
```

Mientras que en la clase que tenemos que crear de manera intermedia tendremos que hacer la relación inversa y a su vez asociarla con la tabla ciclos para que exista la relación.

```
@ManyToOne
@JoinColumn(name = "alumno_id")
@JsonIgnore
private Alumno alumno;

@ManyToOne
@JoinColumn(name = "ciclo_id")
private Ciclo ciclo;
```

Tambien podemos observar la anotación @JsonIgnore la cual cuando hagamos una petición GET de ese alumno no nos traerá el id del alumno por lo que nos crearía una recursividad infinita.

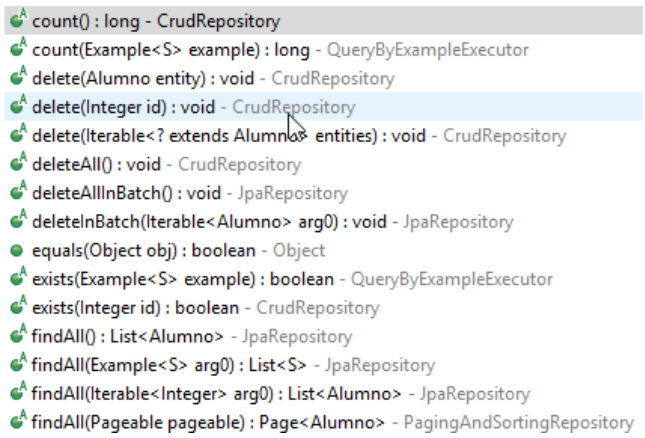
## Repositorios

Esta es una de las grandes ventajas de SpringFramework, cada una de las entidades que hemos creado, y que tendrán por lo tanto que interactuar con la base de datos deberemos de crearle una interface repositorio que extienda de JpaRepository, indicándole a la clase que hace referencia y su id.

```
@Repository
public interface AlumnoRepository extends JpaRepository<Alumno, Integer>{

}
```

Al hacer esto, cuando creemos una instancia de esta interface, tendremos una gran serie de métodos implementados que nos serán de gran ayuda. Como pueden ser el ejemplo del de listar todos los elementos, buscar uno por id, salvar un elemento, etc.




Aunque también podremos crearnos nosotros mismos nuestras propias implementaciones como en el siguiente caso.

En el que vamos a implementar un query propia de MySQL, ya que nos va a hacer falta.

```
@Transactional
@Modifying
@Query(value = "insert into alumno_ciclo (annio_fin,alumno_id,ciclo_id) VALUES (:annio_fin, :alumno_id, :ciclo_id)", nativeQuery = true)
void save(@Param("annio_fin")Date annio_fin, @Param("alumno_id") int alumno_id, @Param("ciclo_id") int ciclo_id);
```

Donde estamos anotando que va a ser transaccional (persistente en la base de datos) y que va a ser una Query nativa a la que le voy a pasar varios parámetros.

## Servicios

Esta es la capa que va a llevar la lógica de negocio de nuestra API REST, es decir es la encargada de procesar y validar los datos para interactuar con la base de datos que hayamos configurado

En mi caso para tener más estructurado y mantenerle el código vamos a separar los servicios creando interfaces y luego implementando dichas interfaces

En las interfaces simplemente creamos los métodos vacíos para darle nombres y ajustar los parámetros de entrada que van a recibir y los que nos van a devolver aquí muestro varios ejemplos.

```
public interface AlumnoService {
    Alumno save(Alumno a);
    Alumno getAlumno(int i);
    List<Alumno> getAlumnos();
}
```

```
public interface Alumno_cicloService {

    void save(int alumno_id, List<Alumno_ciclo> alumno_ciclos);
}
```

Ahora una vez creadas las interfaces continuaremos creando las implementaciones de dichas interfaces. Para crear dichas implementaciones lo más importante será anotar a la clase con @Service la cual le dirá a Spring que esta implementación es un servicio y que será la encargada de interactuar con la base de datos. Como corresponde a la siguiente imagen que explicare más detalladamente.

```
@Service
public class AlumnoServiceImple implements AlumnoService {

    private AlumnoRepository alumnoRepository;

    @Autowired
    public AlumnoServiceImple(AlumnoRepository alumnoRepository) {
        this.alumnoRepository=alumnoRepository;
    }

    @Override
    public Alumno save(Alumno a) {
        return alumnoRepository.save(a);
    }

    @Override
    public Alumno getAlumno(int i) {
        return alumnoRepository.findOne(i);
    }

    @Override
    public List<Alumno> getAlumnos() {
        return alumnoRepository.findAll();
    }

}
```

Como podemos observar la clase está anotada con @Service, esta es la parte fundamental como he explicado antes, pero también podemos ver que tenemos la anotacion @Autowired, esta también es una de las grandes herramientas que nos proporciona SpringFramework ya que con esta anotacion estaremos inyectando la interface AlumnoRepository que he explicado más arriba por lo que ya tendremos accesos a todos los métodos que comente antes, por lo que ya podré listar o salvar un alumno fácilmente.

Otro ejemplo de implementación de un Service sería el de Alumno_aptidudesService. Ya que en este usaremos la Query nativa que hemos creado antes, y salvaremos uno a uno los objetos de Alumno_aptitudes ya que como he comentado antes en la relación contiene una lista de los Alumno_aptitudes

```
@Service
public class Alumno_aptitudServiceImpl implements Alumno_aptitudService {

    private Alumno_aptitudRepository alumno_aptitudRepository;

    @Autowired
    public Alumno_aptitudServiceImpl(Alumno_aptitudRepository alumno_aptitudRepository) {
        this.alumno_aptitudRepository = alumno_aptitudRepository;
    }

    @Override
    public void save(int alumno_id, List<Alumno_aptitud> alumnoAptitudes) {
        for (Alumno_aptitud alumnoAptitud : alumnoAptitudes) {
            alumno_aptitudRepository.save(alumnoAptitud.getNivel(), alumno_id, alumnoAptitud.getAptitud().getId());
        }

    }
```

Otro de los servicios que tengo implementados es el de envíos de email, que hace uso de la API JavaMailSender. Donde le pasas a quien o quienes va dirigido el mensaje, el asunto y el propio mensaje, y el encapsula el correo para poder mandarlo.

```
@Service
public class EmailService {

    private JavaMailSender javaMailSender;

    @Autowired
    public EmailService(JavaMailSender javaMailSender) {
        this.javaMailSender = javaMailSender;
    }

    public void sendMail(String toEmail, String subject, String message) {
        SimpleMailMessage mailMessage = new SimpleMailMessage();
        mailMessage.setTo(toEmail);
        mailMessage.setSubject(subject);
        mailMessage.setText(message);
        javaMailSender.send(mailMessage);
    }
}
```

## RestController

Como lo que he usado para el servidor es una API REST, lo que harán dichos controladores serán recibir peticiones HTTP y a partir de ellas manejar la base de datos y para ello tendremos que hacer uso de la anotacion @RestController, pero como comente antes para no tener problemas con el cros origin antes de nada me voy a crear una clase abstracta de la que extenderán todos los controladores de la aplicación para ello lo único que deberemos hacer es crearnos dicha clase con la anotación @CrossOrigin como vemos en la siguiente imagen.

```
@CrossOrigin
public abstract class AbstractResourceController {

}
```

En la aplicación tendré 3 controladores, uno para manejar el login, uno para manejar el envío de email, y otro que es el que más peso llevara que llevara todo lo relacionado con el alumno, es decir alta, listado, etc. Los tres controladores compartirán tres cosas la anotacion @RestController, que es la encargada de decirle a spring que esta clase hará de controlador de la API REST, extenderán de la clase abstracta antes creada AbstractResourceController, y todas llevaran inyectados los servicios de los que dependan para hacer las peticiones en el constructor de dicha clase como podemos ver en la siguiente imagen.

```
@RestController
public class AlumnoController extends AbstractResourceController {

    private final AlumnoService alumnoService;
    private final CicloService cicloService;
    private final Alumno_cicloService alumno_cicloService;
    private final RedesService redesService;
    private final AptitudService aptitudService;
    private final Alumno_aptitudService alumno_aptitudService;
    private final Alumno_otService alumno_otService;
    private final CentrosService centrosService;

    @Autowired
    public AlumnoController(AlumnoService alumnoService, CicloService cicloService,
            Alumno_cicloService alumno_cicloService, RedesService redesService, AptitudService aptitudService,
            Alumno_aptitudService alumno_aptitudService, Alumno_otService alumno_otService,
            CentrosService centrosService) {
        this.alumnoService = alumnoService;
        this.cicloService = cicloService;
        this.alumno_cicloService = alumno_cicloService;
        this.redesService = redesService;
        this.aptitudService = aptitudService;
        this.alumno_aptitudService = alumno_aptitudService;
        this.alumno_otService = alumno_otService;
        this.centrosService = centrosService;
    }
```

Ahora para mapear las peticiones Spring también nos proveen de varias anotaciones, con las que le decimos de que petición se trata, cuál va a ser la url a la que se mantendrá a la escucha y el método que ejecutara en dicho caso.

```
@RequestMapping(value = "/alumnos", method = RequestMethod.GET)
public List<Alumno> alumnos() {
    return alumnoService.getAlumnos();
}
```

Como podemos ver en esta imagen, la petición será del tipo GET, la url en la que estará a la escucha será /alumnos, y lo que hará dicho métodos es listar todos los alumnos que tengamos en la base de datos haciendo uso del servicio antes implementado.

Para poder encontrar a tan solo un alumno buscándolo por su id de la base de datos tendremos el siguiente método.

```
@RequestMapping(value = "/alumno/{id}", method = RequestMethod.GET)
public Alumno alumno(@PathVariable("id") int id) {
    return alumnoService.getAlumno(id);
}
```

Como podemos ver en el value lleva entre llaves un parámetro que en este caso será la id del alumno que queremos buscar, con la anotacion @PathVariable cogeremos el parámetro que nos venga en la url y la convertiremos a entero ya que será su id y con ella buscamos a dicho alumno haciendo uso del servicio antes implementado.

Para el alta de nuevos alumnos utilizaremos una petición POST como la que veremos en la siguiente imagen que iré explicando paso a paso.

```
@RequestMapping(value = "/alumno", method = RequestMethod.POST)
public void alumno(@RequestBody Alumno alumno) {
    Alumno a = alumnoService.save(alumno);
    int alumno_id = a.getId();
    redesService.save(a.getId(), alumno.getRedes());
    alumno_cicloService.save(alumno_id, alumno.getalumno_ciclo());
    alumno_aptitudService.save(alumno_id, alumno.getAlumno_aptitudes());
    alumno_otService.save(alumno_id, alumno.getAlumno_ot());
}
```

Como podemos observar lo nuevo aquí es la anotacion @RequestBody con ella lo que le estaremos diciéndole a spring será que va a tener un parámetro de entrada que en este caso será en formato Json y que tendrá la misma forma que la que presenta la clase Alumno, es decir que contendrá todas sus propiedades, incluidas las relaciones que este tenga.

Como podemos ver lo primero que haremos será salvar al alumno que nos llegue en dicha petición y a partir de guardar a dicho alumno y a partir del ir actualizando las tablas intermedias para poder ir guardando los id correspondientes en sus tablas, para que todas queden perfectamente confeccionadas.

# Cliente

Para el desarrollo de la parte cliente he usado AngularJS, html y bootstrap, y como IDE he usado Brackets que es un editor de texto que encapsula un navegador local para ejecutar en ese servidor la aplicación sin tener que estar montando en un servidor.

## Librerías

Para cargar las librerías solo se necesitará cargar el script en las páginas en las que las necesites, pero como veremos más adelante gracias al enrutador las cargaremos en el Index.html, y lo que debemos saber es que la primera que debernos cargar será la propia de Angular y acto seguido la de app.js, que será un script propio donde especificaremos las configuraciones de nuestra aplicación.

**Angular:** He usado la librería de angular 1.6.4, para la creación completa de la aplicación en la parte cliente.

**app.js:** aquí cargaremos las demás librerías que nos hagan falta en nuestra aplicación además de las configuraciones necesarias.

**ui.router:** Angular es conocido por crear aplicaciones SPA (Aplicaciones de una sola página), pues con esta librería podremos simular la navegación entre páginas, los que nos será de gran utilidad como veremos más adelante.

**ngSanitize:** con esta librería podremos sanitizar el html de nuestra aplicación, además nos hará falta como dependencia de otras librerías que usaremos.

**ngAnimate :** con esta librería podremos crear animaciones con angular.

**ui.bootstrap:** con esta librería podremos hacer uso de algunos de los componentes que nos proporciona bootstrap.

**ui.select:** esta librería nos proporcionara la forma de manejar los select para poder filtrarlos.

**ngDialog:** con esta librería podremos crear modales con Angular.

**ngCookies:** nos permitirá crear modificar y borrar cookies fácilmente.

**ngCsv:** no proporcionara facilidades para exportar datos que estén en formato Json a CSV

## Index.html

Al utilizar la librería enrutador, todos los Script que nos hagan falta los cargaremos en el índex, es decir las librerías y los script propio que vayamos necesitando como los controladores, factorías, filtros, componentes, etc. Tambien en este archivo deberemos de especificar el nombre que tendrá el app.js.

Como podemos ver este será el body de nuestro archivo

```
<body ng-app="proyectoIntegrado" ng-controller="LoginController as vm" class="bg-success">
    <div class="wrap">
        <div class="container-fluid" ui-view>
        </div>
    </div>
    <footer id="footer" class="bg-success">
       <div class="row">
            <div class="col-xs-6 col-md-offset-1 col-md-3 margen">
                © 2017<span style="color:#0a93a6">  IES Poligono Sur</span>
            </div>
            <div class="col-md-offset-6 col-md-2 col-xs-6">
                <a href="https://www.facebook.com/iespsur" target="_blank"><i id="social-fb" class="fa fa-facebook-square fa-3x social">
                </i></a>
                <a href="https://twitter.com/iespsur" target="_blank"><i id="social-tw" class="fa fa-twitter-square fa-3x social"></i>
                </a>
                <a href="http://iespoligonosur.org" target="_blank"><i id="social-gp" class="fa fa-globe fa-3x social"></i></a>
                <a href="mailto:info@iespoligonosur.org"><i id="social-em" class="fa fa-envelope-square fa-3x social"></i></a>
            </div>
        </div>
    </footer>
```

En el que incluimos la directiva ng-app donde daremos el nombre de nuestro app.js y lo que permanecerá en toda la aplicación, que en este caso será el footer.

En las siguientes imágenes mostrare la carga de los diversos Script que he creado o importado de diversas librerías, algunos de estos script los recojo por CDN por lo que para desplegar a aplicación nos hará falta una conexión a internet.

```
<!--Cargamos Angular-->
<!-- <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.4/angular.min.js"></script>-->
<script src="resources/lib/angular.min.js"></script>

<!-- Cargamos ui.router y ngCookies-->
<!--<script src="http://unpkg.com/angular-ui-router/release/angular-ui-router.min.js"></script>-->
<script src="resources/lib/ui-router.min.js"></script>
<script src="resources/lib/cookies.js"></script>


<!-- Cargamos ng-animate y ui.bootstrap-->
<script src="resources/lib/angular-animate.js"></script>
<script src="resources/lib/ui-bootstrap-tpls-2.5.0.min.js"></script>

<!-- Cargamos ng-csv , ng-sanitize y select CDN-->
<script src="https://cdnjs.cloudflare.com/ajax/libs/angular-sanitize/1.6.4/angular-sanitize.min.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/ng-csv/0.3.6/ng-csv.min.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/angular-ui-select/0.20.0/select.js"></script>

<!-- Cargamos Dialogs-->
<script src="https://cdnjs.cloudflare.com/ajax/libs/ng-dialog/1.3.0/js/ngDialog.js"></script>
```

```
<!-- Cargamos script de la aplicacion -->
<script src="resources/lib/app.js"></script>
<script src="controllers/loginController.js"></script>
<script src="controllers/profesorController.js"></script>
<script src="controllers/alumnoDetalleController.js"></script>
<script src="controllers/registroController.js"></script>
<script src="factories/loginFactory.js"></script>
<script src="factories/profesorFactory.js"></script>
<script src="factories/alumnoDetalleFactory.js"></script>
<script src="filter/filtros.js"></script>
<script src="factories/cicloFactory.js"></script>
<script src="factories/alumnoFactory.js"></script>
<script src="factories/csvService.js"></script>
<script src="factories/authFactory.js"></script>
<!-- Formulario por pasos -->
<script src="form/form.component.js"></script>
<script src="form/form.controller.js"></script>
<script src="form/formData.value.js"></script>
<script src="form/usuario.component.js"></script>
<script src="form/usuario.controller.js"></script>
<script src="form/alumno.component.js"></script>
<script src="form/alumno.controller.js"></script>
<script src="form/direccion.component.js"></script>
<script src="form/direccion.controller.js"></script>
<script src="form/ciclo.component.js"></script>
<script src="form/ciclo.controller.js"></script>
<script src="form/ot.component.js"></script>
<script src="form/ot.controller.js"></script>
<script src="form/redes.component.js"></script>
<script src="form/redes.controller.js"></script>
<script src="form/result.component.js"></script>
<script src="form/result.controller.js"></script>
```

## app.js

En este archivos cargaremos todas las librerías antes declaradas, y añadiremos sus configuraciones. De esta manera

```
var proyectoIntegradoApp = angular.module("proyectoIntegrado",['ui.router',"ngSanitize", "ngCsv", "ngAnimate",
"ui.bootstrap",'ui.select','ngDialog','ngCookies']);
```

Después en este mismo Script cargaremos las siguientes templates que manejara la librería ui.router, aquí pongo algunas de las configuraciones.

```
proyectoIntegradoApp.config(function($stateProvider,$urlRouterProvider){

    $urlRouterProvider.otherwise("login");

    $stateProvider
        .state('login',{
            url: "/login"    ,
            templateUrl: "templates/login.html",
            controller: 'LoginController',
            controllerAs: 'vm'
        })
        .state('alumnos',{
            url: "/alumnos"    ,
            templateUrl: "templates/alumnos.html",
            controller: 'AlumnoController',
            controllerAs: 'vm'
        })
        .state('profesor',{
            url: "/profesor"    ,
            templateUrl: "templates/profesor.html",
            controller: 'ProfesorController',
            controllerAs: 'vm'
        })
```

Donde con el $urlRouterProvider le decimos que si intenta acceder a alguna url que no está especificada más abajo esta la redigirará a la template de login.

Con el $stateProvider formaremos la navegación que simularemos entre las pagina, donde el state es la url donde estará a la escucha, url será lo que mostrara en la barra de navegación, templateUrl, especificaremos donde tendremos el archivo html que contendrá dicha página, con controller, asignaremos un controlador a dicha página y con controllerAs le diremos como podremos llamar al controlador desde la vista.

## Enrutador

Como he explicado arriba gracias a la librería de ui.router podremos simular la simulación entre página, lo único que deberemos hacer será especificar en el índex esta directiva.

```
<div class="container-fluid" ui-view></div>
```

Con la directiva ui-view estaremos encapsulando cualquier template que hayamos creado dentro de ese div

## Factorías

Estas serán las encargadas de hacer las peticiones al servidor y proporcionarles los resultados de dicha petición a controlador que será el encargado de renderizar la página. Aquí pongo un ejemplo de una factoría que se encarga de hacer una petición POST al servidor pasándole un objeto en formato Json que el servidor está esperando recibirla, lo único que deberemos hacer será inyectarle el $http.

```
(function() {
    angular.module('proyectoIntegrado')
        .factory('AlumnoFactory', alumnoFactory);
    function alumnoFactory($http){

        //var currentUsuario;

        var interfaz = {
            //Metodos del servicio
            createAlumno: function(alumno){
                return $http.post("http://localhost:8080/alumno",alumno);
            }
        }
        return interfaz;
    }
}())
```

Una vez creada esta factoría podremos inyectársela a cualquier controlador y podremos llamar a cualquieras de los métodos aquí creados.

## Controladores

Cada página que tengamos tendrá asignado un controlador, que será el encargado de realizar las distintas operaciones que nos sean necesarias dentro de nuestra aplicación. Además a los controladores podremos inyectarles las diversas factorías o servicios creados, así como los diferentes filtros.

Para crear un controlador deberemos empaquetarlos en una función anónima y cargar el módulo de nuestra aplicación es decir el app.js, después llamar a la función del controlador inyectando las dependencias que nos hagan falta en dicho controlador. Después lo siguiente que deberemos hacer será declarar una variable que por convección se llamará vm donde agregaremos todo el controlador con la palabra reservada this.

```
(function() {

    angular.module('proyectoIntegrado')
        .controller('LoginController', loginController);

    function loginController(loginFactory,$state,$stateParams) {
        var vm = this;

}());
```

Dentro de dicho controlador podremos crear funciones que podremos llamar desde la vista haciendo por ejemplo un ng-click como por ejemplo esta.

```
vm.login = function () {
    if(!vm.usuario){
        vm.error = true;
        return vm.msgerror = "Introduzca usuario y contraseña";
    }
    loginFactory.login(vm.usuario)
        .then(function (response) {
            vm.error = false;
            vm.msgerror = '';
            vm.currentUsuario = response.data;
            if(typeof(vm.currentUsuario.id) == "undefined"){
                vm.error = true;
                vm.msgerror = "Email o contraseña incorrecto";
            }else{
                loginFactory.setCurrentUsuario(vm.currentUsuario);
                if(vm.currentUsuario.rol_profesor==0){
                    //$state.go('alumnos');
                }else{
                    $state.go('profesor');
                }

            }
        }, function (response) {
            vm.error = true;
            vm.msgerror = response.data;
        });

};
```

Donde decimos que cuando se llame a vm.login ejecutaremos el siguiente código, donde haremos uso de la factoría factoryLogin, que es la encargada de realizar las peticiones al servidor para logarse. Y conde comprobaremos si ese usuario existe y que rol dentro de la aplicación tiene, ya que con la directiva $state.go("nombrePlantilla"), iremos a la template antes definida en el app.js con su correspondiente controlador.

Dentro de los controladores podemos definir tantas funciones y variables como queramos a los que podremos acceder desde la vista gracias al $scope, también dentro de nuestro controlador podemos estar observando el cambio de alguna de las variables que se cambien en la vista, gracias a la directiva $scope.$watch por lo que estará a la escucha de cualquier cambio que se haga en dicha variable como podemos ver en este ejemplo.

```
$scope.$watch('vm.text', function() {
    if(vm.text == null || vm.text == ""){
        vm.alumnos = vm.alumnos2;
    }else{
        vm.alumnos = $filter('filterSearch')(vm.text.toLowerCase(), vm.selectFilter.toLowerCase(), vm.alumnos2);
    }
});

$scope.$watch('vm.selectFilter', function() {
    if(vm.text == null || vm.text == ""){
        vm.alumnos = vm.alumnos2;
    }else{
        vm.alumnos = $filter('filterSearch')(vm.text.toLowerCase(), vm.selectFilter.toLowerCase(), vm.alumnos2);
    }
});
```

Como podemos ver así confeccionare el filtro de las búsquedas, ya que mantendré a la escucha el texto que introduzca el usuario en el filtro y también estará a la escucha el campo de la selección del filtro, lo que hará esta función será aplicarle el filtro que ahora explicare, al array de alumnos del que dispones devolviéndote ese array pero ya filtrado

## Filtros

Para el manejo del filtro que puede hacer el usuario con el rol de profesor, he tenido que crearme un filtro personalizado al igual que uno para manejar la paginación de la vista de los alumnos.

Así declaramos que va a ser un script de filtro, donde le pasamos el modulo del proyectoIntegrado que sería nuestro app.js y añadiéndole la función .filter

```
(function (){
    angular.module('proyectoIntegrado')
    .filter('filterSearch', function(){
        return function(text, cat, array){
            var arr = new Array();
```

Como podemos ver este filtro recibirá un text que será el que escriba el usuario en el área de filtrar, un cat que será la categoría seleccionada por el que el usuario querrá filtrar y un array que será el array con todos los alumnos que tengas almacenados en la base de datos. Y nos devolverá un array nuevo con los datos ya filtrados, vamos a verlo por partes.

```
for(item in array){
    if(cat == "disponibilidad"){
        if(text == "s" || text == "si"){
            text=1;
        }else if(text == "n" || text == "no"){
            text=0;
        }
        if(array[item][cat]==text){
            arr.push(array[item]);
        }
    }else if(cat == "ciclo"){
        var ciclo = array[item]["alumno_ciclo"];
        for(i in ciclo){
            if(ciclo[i]["ciclo"]["siglas"].indexOf(text.toUpperCase())!=-1){
                arr.push(array[item]);
            }
        }
```

Como podemos observar lo primero que haremos será recorrer el array que le pasamos, y comprobaremos que categoría está seleccionada, en el primer caso comprobaremos si es disponibilidad la que esta seleccionada y como la disponibilidad es realmente es un booleano es decir un  o un  1, lo que tendremos que hacer será transformar el texto de entrada en caso de que sea “s” o “si” a 1 y en caso de “n” o “no” a 0 y después lo que haremos será ir metiendo en el array aquellos elementos que tengan como disponibilidad la que hayamos descrito.

En el caso del ciclo al ser un objeto dentro de otro objeto es decir que el Json está anidado tendremos que recorrer ese objeto también y nos centraremos en las siglas para hacer el filtrado, donde convertiremos lo que escriba el usuario a mayúsculas ya que en la base de datos las siglas están en mayúsculas, y de igual manera que en la disponibilidad iremos añadiendo elementos al array que vamos a devolver si el texto coincide con las siglas del ciclo que tenga dicho alumno.

Ahora vamos a ver como conseguimos el filtrado por edad y aptitudes

Con las aptitudes al tratarse también de un Json anidado hacemos prácticamente lo mismos que con la búsqueda del ciclo.

```
}else if(cat == "aptitudes"){
    var aptitudes = array[item]["alumno_aptitudes"];
    for(i in aptitudes){
        if(aptitudes[i]["aptitud"]["nombre"].toUpperCase().indexOf(text.toUpperCase())!=-1){
            arr.push(array[item]);
        }
    }
```

Mientras que para la edad tendremos que calcularla nosotros ya que en la base de datos esta guardada en formato fecha, por lo que o primero que haremos será crear una fecha actual y restarle la fecha que tiene dicho alumno y si cumple con los requisitos lo añadiremos a la lista

```
}else if(cat == "edad"){
    var annioActual = (new Date).getFullYear();
    var annio = parseInt((array[item]["fecha_nac"]).substring(6, (array[item]["fecha_nac"]).length));
    var edad = annioActual - annio;
    if(text.length >= 2){
        if(edad.toString()==text){
        arr.push(array[item]);
       }
    }else{
        if((array[item][cat]).toLowerCase().indexOf(text)!=-1){
            arr.push(array[item]);
        }
    }
```

## Servicios

Los servicios son maneras de quitar trabajo al controlador para que se le puedan aplicar la lógica de negocio en el servicio en vez de en el controlador para así tener este más limpio. Un ejemplo de servicio que he creado es la manera de importar los datos a CSV, los servicios se crean igual que las factorías.

Para este servicio he usado el módulo ng-csv pero este solo puede exportar los datos de un Json simple y el Json que yo tengo de los alumnos es complejo es decir esta anidado. Por lo que tendré que tratar los datos para formar un Json que no contenga objetos ni matrices, por lo que yo le pasare al servicio un Json completo que iré trabajando. Voy a explicarlo por partes.

```
createCsv: function(alumnos){
    var myData = [];
    alumnos.forEach(function(obj) {
        var redes = [];
        var ciclo = [];
        var aptitudes = [];
        if(obj.disponibilidad == 0){
            obj.disponibilidad = 'No';
        }else{
            obj.disponibilidad = 'Si';
        }
        obj.redes.forEach(function (obj){
           redes.push(obj.link);
        });
```

Lo primero que haremos será crearnos un array que será el que devolveremos para que el ng-csv nos lo exporte, y al que le iremos dando formato.

Después tendremos que recorrer el Json y por ejemplo cambiaremos la disponibilidad para que nos grave en el csv un “Si” o un “no” en vez de 0 o 1.

Después cuando encontramos la primera matriz que serán las redes sociales, por lo que recorreremos ese objeto también y lo iremos metiendo en un array de String.

Haremos lo mismo con los ciclos y las aptitudes.

```
obj.alumno_ciclo.forEach(function (obj){
   ciclo.push(obj.ciclo.siglas);
   ciclo.push(obj.ciclo.nombre);
   ciclo.push(obj.annio_fin);
});
obj.alumno_aptitudes.forEach(function (obj){
    aptitudes.push(obj.aptitud.nombre);
    aptitudes.push(obj.nivel);
});
```

Y por último añadiremos ya al array que devolveremos los datos ya con el formato deseado

```
    myData.push(
        {
            nombre: obj.nombre,
            apellido1: obj.apellido1,
            apellido2: obj.apellido2,
            fecha_nac: obj.fecha_nac,
            direccion: obj.direccion,
            localidad: obj.localidad,
            cp: obj.cp,
            provincia: obj.provincia,
            telefono: obj.telefono,
            email: obj.email,
            email2: obj.email2,
            disponibilidad: obj.disponibilidad,
            observaciones: obj.observaciones,
            usuario: obj.usuario.usuario,
            contrasena: obj.usuario.contrasena,
            redes: redes.toString().replace(",","|"),
            ciclo: ciclo.toString().replace(",","|"),
            aptitudes: aptitudes.toString().replace(",","|")
        }
    );
});
return myData;
```

Como podemos comprobar para las claves redes, ciclo y aptitudes lo que haremos será transformar los arrays que hemos ido creando a una cadena para así poder grabarlo en el csv.

Para crear la cabecera del CSV también usare una función de este servicio donde lo que hare será crearme un array con las palabras clave que quiero que aparezcan en la cabecera

```
headerCsv: function(){
    var header = ["Nombre","Primer Apellido","Segundo Apellido","Fecha
    Nacimiento","Direccion","Localidad","Codigo postal","Provincia","Telefono","Email","Email
    opcional","Disponibilidad","Observaciones","Usuario","Contraseña","Redes Sociales","Ciclo
    Cursado","Aptitudes"];
    return header;
}
```

# Despliegue

**Requisitos:**

- JDK 1.8
- Maven
- MySQL
- Servidor Apache

Para poder desplegar la aplicación deberás copiar las dos carpetas que se encuentran en el repositorio, en la ruta ProyectoIIntegradoSpringRest\ProyectoIntegrado\src\main\resources\bd encontraras el script de la base de datos, tan solo tendrás que iniciar MySQL y ejecutar este script, una vez ejecutado, deberemos abrir una consola de comandos y dirigirnos a la carpeta ProyectoIIntegradoSpringRest\ProyectoIntegrado y ejecutar **mvnw spring-boot:run** y ya tendremos la API REST en funcionamiento. Lo único que nos quedara será copiar la carpeta ProyectoIntegrado a tu servidor apache y para iniciar el funcionamiento de la aplicación deberás de abrir tu navegador y poner en la url **localhost/ProyectoIntegrado/index.html**