Universidad de Alcalá
Escuela Politécnica Superior

MÁSTER UNIVERSITARIO EN INGENIERÍA DEL SOFTWARE PARA LA WEB

Trabajo Fin de Máster

Desarrollo de una aplicación Web con Spring Boot, Vaadin y Microservicios

Autor: Felipe Oliva Encabo

Tutor/es: Salvador Otón Tortosa

2021

UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

MÁSTER UNIVERSITARIO EN INGENIERÍA DEL SOFTWARE PARA LA WEB

Trabajo Fin de Máster

DESARROLLO DE UNA APLICACIÓN WEB CON SPRING BOOT, VAADIN Y MICROSERVICIOS

Felipe Oliva Encabo

Septiembre / 2021

UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

MÁSTER UNIVERSITARIO EN INGENIERÍA DEL SOFTWARE PARA LA WEB

Trabajo Fin de Máster

DESARROLLO DE UNA APLICACIÓN WEB CON SPRING BOOT, VAADIN Y MICROSERVICIOS

Autor: **Felipe Oliva Encabo**

Director: **Salvador Otón Tortosa**

Tribunal:

Presidente: _____

Vocal 1º: _____

Vocal 2º: _____

Calificación: _____

Alcalá de Henares a de de 202X

ÍNDICE RESUMIDO

INTRODUCCIÓN **1**

OBJETIVOS DEL PROYECTO **5**

ESTADO DEL ARTE **9**

SPRING FRAMEWORK **13**

VAADIN **31**

MICROSERVICIOS **47**

DESARROLLO DEL PROTOTIPO **57**

RESUMEN Y CONCLUSIÓN **74**

BIBLIOGRAFÍA **79**

ÍNDICE DETALLADO

INTRODUCCIÓN **1**

OBJETIVOS DEL PROYECTO **5**

ESTADO DEL ARTE **9**

SPRING FRAMEWORK **13**

1.1. INTRODUCCIÓN 15
1.2. INYECCIÓN DE DEPENDENCIA (DI): 16
1.3. PROYECTOS QUE COMPONEN SPRING 17
1.4. SPRING BOOT 18
1.5. SPRING INITIALIZR 20
1.6. PROGRAMANDO EN SPRING 21
1.6.1. El contenedor de Spring *21*
1.6.1.1. Bean 21
1.6.1.2. ApplicationContext 21
1.7. SERVICIOS REST EN SPRING 24
1.7.1. Implementación de API Rest en Spring *24*
1.7.2. Consumo de servicios REST en Spring *25*
1.7.2.1. La clase ResTemplate 25
1.7.2.2. Cliente Feign 25
1.7.3. Spring Cloud *27*
1.7.3.1. Spring Cloud Netflix Eureka 28
1.7.3.2. Spring Cloud Discovery 28

VAADIN **31**

1.1. ¿QUÉ ES VAADIN? 33
1.2. ARQUITECTURA DE LA APLICACIÓN 35
1.3. HERRAMIENTAS REQUERIDAS 36
1.4. DEPENDENCIAS DE LA PLATAFORMA VAADIN 37
1.4.1. Dependencias de componentes individuales *37*
1.5. COMPONENTES VAADIN 39
1.5.1. Listado de componentes *39*
1.5.2. Características básicas de los componentes *40*
1.5.3. Creación de un componente *41*
1.5.4. Vincular datos a formularios *42*
1.6. RUTAS Y NAVEGACIÓN 43
1.6.1. Ciclo de vida de la navegación *43*
1.6.1.1. BeforLeaveEvent 43
1.6.1.2. BeforeEnterEvent 44
1.6.1.3. AfterNavigationEvent 44

MICROSERVICIOS **47**

1.1. ¿QUÉ ES UN MICROSERVICIO? 49

1.2. APLICACIONES BASADAS EN MICROSERVICIOS 51
1.3. DEFINICIÓN Y PRINCIPIOS DE LOS MICROSERVICIOS 52
1.4. CUANDO NO USAR MICROSERVICIOS 55

DESARROLLO DEL PROTOTIPO 57

1.1. OBJETIVO DEL PROTOTIPO 59
1.2. DISEÑO TECNOLÓGICO DEL PROTOTIPO 60
1.2.1. Cliente Front End 60
1.2.1.1. Descripción del proyecto 60
1.2.1.2. Vistas de la aplicación. 61
1.2.2. Eureka 65
1.2.3. API Gateway 66
1.2.4. Microservicios 68
1.2.4.1. Microservicio Alumnos 70
1.2.4.2. Microservicio Academia 71
1.2.4.3. Empresas 72
1.2.4.4. Fcts 72

RESUMEN Y CONCLUSIÓN 74

1.3. PRESUPUESTO 76
1.4. CONCLUSIONES Y FUTURAS LÍNEAS DE TRABAJO 77

BIBLIOGRAFÍA 79

ÍNDICE DE FIGURAS

1. INTRODUCCIÓN

2. OBJETIVOS DEL PROYECTO

3. ESTADO DEL ARTE

FIGURA 1: ARQUITECTURA WEB MONOLÍTICA 11
FIGURA 2: PROYECTOS QUE COMPONEN SPRING 17
FIGURA 3: SPRING INITIALIZR 20
FIGURA 4: CREACIÓN DE CUN CLIENTE FEIGN EN SPRING 26
FIGURA 5: DECLARACIÓN DE LA DEPENDENCIA VAADIN.PLATFORM 37
FIGURA 6: INFORMACIÓN DE UN COMPONENTES VAADIN 40
FIGURA 7: DEFINICIÓN DEL COMPONENTES SOMEPATHCOMPONENT COMO EL OBTIVO PARA LA RUTA ESPECÍFICA SOME/PATH 43
FIGURA 8: DEFINICIÓN DEL COMPONENTE HELLOWORLD COMO EL DESTINO DE RUTA PREDETERMINADO (RUTA VACÍA) PARA SU APLICACIÓN 43
FIGURA 9: DISEÑO TECNOLÓGICO DEL PROTOTIPO 60
FIGURA 10: FICHERO APPLICATION.PROPERTIES DEL FRONT-END 61
FIGURA 11: VISTA WEB - ALUMNOS 62
FIGURA 12: VISTA WEB - EMPRESAS 62
FIGURA 13: VISTA WEB - ALUMNOS 63
FIGURA 14: INYECCIÓN DE LAS ENTIDADES @SERVICE EN LAS VISTAS DE VAADIN 63
FIGURA 15: CONFIGURACIÓN DEL PROYECTO PARA SERVIDOR DE REGISTRO EUREKA 65
FIGURA 16: INTERFACE DE ADMINISTRACIÓN EUREKA SERVER 66
FIGURA 17: CONFIGURACIÓN DEL PROYECTO PARA SERVIDOR DE REGISTRO EUREKA 67
FIGURA 18: CONFIGURACIÓN DEL ARCHIVO APPLICATION.PROPERTIES DEL SERVICIO API GATEWAY 68
FIGURA 19: CONFIGURACIÓN DEL PROYECTO PARA MICROSERVICIO ALUMNOS 68
FIGURA 20: APPLICATION.PROPERTIES DEL SERVICIO ALUMNOS 69
FIGURA 21: DIAGRAMA DE LA BASE DE DATOS DEL MICROSERVICIO ALUMNOS 70
FIGURA 22: DIAGRAMA DE LA BASE DE DATOS DEL MICROSERVICIO ACADEMIA 71
FIGURA 24: DIAGRAMA DE LA BASE DE DATO DEL MICROSERVICIO EMPRESAS 72
FIGURA 25: DIAGRAMA DE LA BASE DE DATOS DEL MICROSERVICIO FCTS 73

INTRODUCCIÓN

Actualmente todos los sectores productivos se encuentra un proceso de digitalización impulsado por las Administraciones públicas y con un objetivo claro de aumentar la productividad de las empresas. Se entiende por digitalización cambiar a un modelo de negocio digital en que todos los procesos están principalmente orquestados por tecnologías digitales.

Hablar de digitalización en el año 2021 puede parecer sorprendente sobre todo porque ya es una realidad desde los inicios de los 1990s, principalmente con la difusión de Internet. Sin embargo, es ahora cuando el contexto hace sin duda más ventajoso llevar a cabo ese proceso. Tecnologías como el Big Data, la robotización, la inteligencia artificial o las nuevas redes de telecomunicaciones móviles plantean un escenario en que la disponibilidad de datos y procesos digitales en todo tipo de organizaciones hace que estas puedan ser más notablemente competitivas en sus desempeños y tengas abiertas nuevas formas de negocio y actuación.

En este proceso de digitalización, el desarrollo de aplicaciones web toma una especial relevancia en tanto permite que los datos y procesos sean accesibles a todos los participantes de las organizaciones, incluidos los clientes. Dicha relevancia es especialmente notable en el ámbito del mercado de trabajo que demanda perfiles profesionales con las capacidades técnicas de desarrollar estas plataformas.

En el presente trabajo tiene por objeto el estudio de dos frameworks de desarrollo de aplicaciones web: Spring y Vaadin. Ambos tienen por objeto ofrecer a los programadores facilidades eliminado tareas rutinarias y costosas en tiempo y proveyendo un entorno de desarrollo flexible adaptable a las necesidades actuales de los sistemas.

También se realizará un análisis sobre las arquitecturas de aplicaciones basadas en microservicios que se basan en la separación de los componentes de la aplicación en pequeños servicios web independientes.

Para la evaluación de estas tecnologías se implementará un prototipo para la digitalización del proceso de planificación, seguimiento y evaluación de las prácticas en empresas que los alumnos de Ciclos Formativos de Formación Profesional realizan como parte de sus estudios.

OBJETIVOS DEL PROYECTO

El presente trabajo tiene los siguientes objetivos:

1. Estudio del framework Spring para el desarrollo de microservicios.

2. Estudio del framework Vaadin para el desarrollo de interfaces web.

3. Investigar sobre la aplicación de la arquitectura de microservicios.

4. Estudio de los módulos de Spring relacionados el desarrollo de microservicios.

5. Desarrollar un prototipo de una aplicación web que ayude en la gestión de la información necesaria para la supervisión de las prácticas en empresas de los alumnos que cursan estudio de Formación Profesional en España.

ESTADO DEL ARTE

El desarrollo tradicional de aplicaciones ha estado centrado tradicionalmente en el concepto monolítico de las mismas. Bajo dicho paradigma, desde un solo servidor de aplicaciones o un solo proceso, se implementa toda la solución.

Dicho concepto fue extrapolado a las aplicaciones web desarrolladas a partir de la web 2.0 y sobre todo web 3.0. En dichas aplicaciones un servidor implementaba toda la aplicación haciendo uso de tecnologías como Java Enterprise Edition y siguiendo fundamentalmente el patrón Modelo, Vista, Controlador.

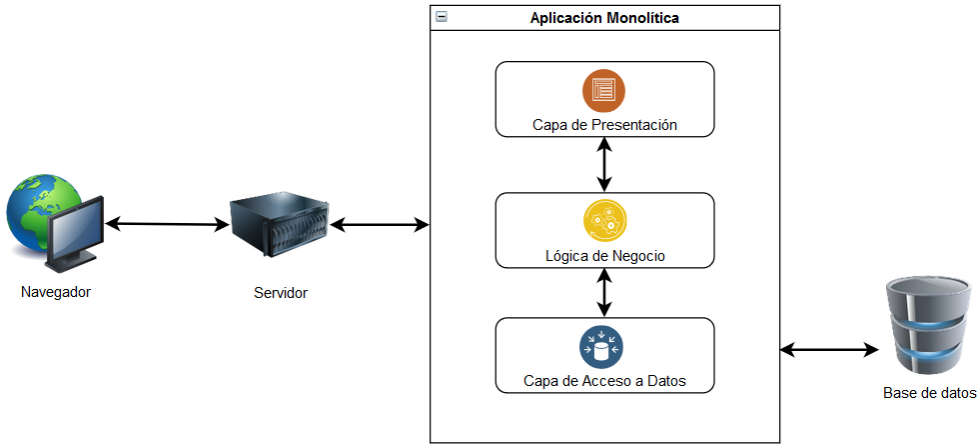


Figura 1: Arquitectura Web monolítica

La irrupción de las tecnologías de virtualización en los años 2000 abre la posibilidad de disponer de máquinas virtuales de forma rápida, ágil y barata. Además, con el desarrollo de dichas tecnologías se ha incrementado notablemente la disponibilidad de los sistemas y se ha conseguido un abaratamiento en los gastos de esta infraestructura.

De forma paralela surgen nuevas propuestas de arquitectura de software como la arquitectura orienta a servicios (SOA por sus siglas en ingles). La Arquitectura Orientada a Servicios (SOA) surgió de una combinación de los conceptos de la programación orientada a objeto y la consonantización de aplicaciones. En SOA, una aplicación se divide en varias partes conocidas como servicios. Un servicio proporciona una funcionalidad accesible para otros servicios a través de varios protocolos, como el Protocolo simple de acceso a objetos (SOAP). (Dragoni, et al., 2016)

Esta arquitectura nace como solución a los problemas que presentan la arquitectura monolítica, basando su enfoque en la reutilización del software a través de servicios bien definidos, en los cuales, varias aplicaciones puedan utilizar de forma transparente e independiente dichos servicios sin la necesidad de realizar un desarrollo dentro del sistema monolítico.

Por último, es destacable la tecnología REST introducida y definida en 2000 por Roy Fielding en su disertación doctoral. REST es una alternativa ligera a mecanismos como RPC (Llamada a procedimiento remoto) y Servicios web (SOAP, WSDL, etc.). (Cosmina, 2020).

Con este escenario tecnológico, se hace posible la implementación de las aplicaciones web basadas en microservicios. Un microservicio se puede definir como un pequeño servicio que se ejecuta sobre un proceso totalmente independiente del resto de los componentes de una aplicación. Éste puede comunicarse con otros bajo mecanismos muy livianos (API / Interfaces de red). El término microservicio, está asociado generalmente a un nuevo estilo de arquitectura que busca desarrollar aplicaciones bajo estos y aumentar la resiliencia. El hecho de descomponer aplicaciones muy complejas genera mayor tolerancia a fallas, mantenibilidad y escalabilidad de todas sus funcionalidades principales.

Un microservicio puede correr en un servidor, una máquina virtual, un contenedor o cualquier tipo de nodo de la arquitectura, lo importante es que pueda ser identificado por una dirección IP y un puerto lógico.

Spring Framework surge en 2004 como una evolución de Java Enterprise Edition. Durante su desarrollo en los últimos años ha evolucionado hacia un modelo basado en multitud de proyectos, muchos de ellos enfocados al desarrollo de aplicaciones basadas en microservicios. También ha evolucionado hacia la integración de otros proyectos que facilitan el desarrollo web como por ejemplo Vaadin.

SPRING FRAMEWORK

1.1. Introducción

Spring es el marco de desarrollo de aplicaciones más popular para Java empresarial. Millones de desarrolladores de todo el mundo utilizan Spring Framework para crear código de alto rendimiento, fácilmente comprobable y reutilizable.

Spring fue inicialmente escrito por Rod Johnson y se lanzó en 2003 bajo licencia Apache 2.0. La versión actual es la 5.1.6

Las características principales de Spring Framework se pueden utilizar para desarrollar cualquier aplicación Java, pero existen extensiones para crear aplicaciones web sobre la plataforma Java EE. Spring Framework tiene como objetivo facilitar el uso del desarrollo J2EE y promueve las buenas prácticas de programación al permitir un modelo de programación basado en POJO.

A continuación, se muestra la lista de algunas de las principales características de Spring Framework:

- Spring permite a los desarrolladores desarrollar aplicaciones de ámbito empresarial utilizando POJO. El beneficio de usar solo POJO es que no necesita un producto contenedor EJB como un servidor de aplicaciones, con un contenedor de servlets robusto como Tomcat o algún producto comercial es suficiente para desplegar una aplicación en Spring.
- Spring está organizada de forma modular. Aunque la cantidad de paquetes y clases es considerable, debe preocuparse solo por los que necesita e ignorar el resto.
- Spring no reinventa la rueda, sino que realmente hace uso de algunas de las tecnologías existentes, como varios framework ORM, framework de looging, JEE, y otras tecnologías de visualización.
- Probar una aplicación escrita con Spring es simple porque el código dependiente del entorno está incluido en el framework. Además, al utilizar POJO de estilo JavaBean, resulta más fácil utilizar la inyección de dependencia para inyectar datos de prueba.
- El framework web de Spring es un framework MVC bien diseñado, que proporciona una gran alternativa a los marcos web como Struts u otros marcos web menos populares.
- Spring proporciona una API para traducir excepciones específicas de tecnología (lanzadas por JDBC, Hibernate o JDO, por ejemplo) en excepciones consistentes y no verificadas.
- Los contenedores de IoC livianos tienden a ser livianos, especialmente en comparación con los contenedores EJB, por ejemplo. Esto es beneficioso para desarrollar e implementar aplicaciones en equipos con memoria y recursos de CPU limitados.
- Spring proporciona una interfaz de administración de transacciones consistente que puede escalar a una transacción local (usando una sola base de datos, por ejemplo) y escalar a transacciones globales (usando JTA, por ejemplo).

1.2. Inyección de dependencia (DI):

La Inversión de Control (IoC) es un concepto general, y se puede expresar de muchas formas diferentes y la Inyección de Dependencia es simplemente un ejemplo concreto de Inversión de Control.

Al escribir una aplicación Java compleja, las clases de aplicación deben ser lo más independientes posible de otras clases de Java para aumentar la posibilidad de reutilizar estas clases y probarlas independientemente de otras clases mientras se realizan pruebas unitarias. Dependency Injection ayuda a unir estas clases y al mismo tiempo a mantenerlas independientes.

¿Qué es exactamente la inyección de dependencia? Miremos estas dos palabras por separado. Aquí, la parte de dependencia se traduce en una asociación entre dos clases. Por ejemplo, la clase A depende de la clase B. Ahora, veamos la segunda parte, la inyección. Todo esto significa que la clase B será inyectada en la clase A por el IoC.

La inyección de dependencia puede ocurrir en la forma de pasar parámetros al constructor o después de la construcción usando métodos setter. Como la inyección de dependencia es el corazón de Spring Framework, explicaré este concepto en un capítulo posterior con un buen ejemplo.

1.3. Proyectos que componen Spring

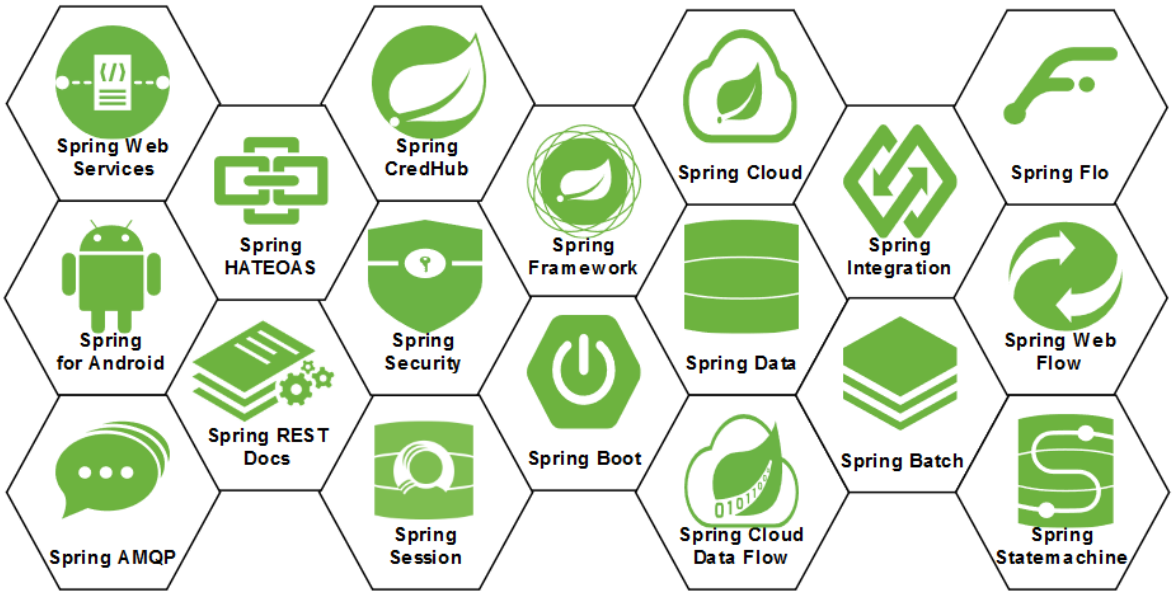


Figura 2: Proyectos que componen Spring

Actualmente Spring incluye un total 23 proyectos algunos de los cuales se muestran en la figura anterior. De todos ellos, en este trabajo se utilizan los siguientes:

- **Spring Framework**: Proporciona un modelo integral de programación y configuración para aplicaciones empresariales modernas basadas en Java, en cualquier tipo de plataforma de implementación.

- **Spring Boot**: Facilita la creación de aplicaciones independientes basadas en aplicación de producción de Spring que se pueden "ejecutar".

- **Spring Data**: Su misión es proporcionar un modelo de programación familiar y consistente basado en Spring para el acceso a los datos, al mismo tiempo que conserva los rasgos especiales del almacén de datos subyacente.

- **Spring Cloud**: Proporciona herramientas para que los desarrolladores construyan rápidamente algunos de los patrones comunes en sistemas distribuidos, por ejemplo, gestión de configuración, descubrimiento de servicios, interruptores automáticos, enrutamiento inteligente, micro-proxy, bus de control, tokens únicos, bloqueos globales, elección de liderazgo, distribuido sesiones, estado del clúster.

1.4. Spring Boot

Spring Boot es una extensión de Spring que permite la creación de proyecto de Spring de forma ágil y fácil. Spring Boot sigue el enfoque de Convención sobre la Configuración que es un paradigma de programación que minimiza las decisiones que tienes que tomar los desarrolladores, simplificando las tareas que éstos tienes que desarrollar sin ello supongo una pérdida de flexibilidad y adaptabilidad del proyecto.

Spring Boot facilita la creación de aplicaciones sin que los desarrolladores tengas que escribir la misma configuración de Spring de forma repetitiva para la creación de cada uno de sus proyectos.

Adicionalmente, Spring Boot también se puede caracterizar por los siguientes aspectos:

- Implementa el principio de diseño de Abierto-Cerrado, donde esta es cerrada a modificación, pero abierto a extensión cuando se desee tener un mayor control de sus componentes.
- Posee servidores de aplicaciones y contenedores de Servlet embebido.
- Eliminar la necesidad de configurar la aplicación por medio de código o XML. Para ello utiliza ficheros .properties o .yml.
- Es una gran opción para crear aplicaciones basadas en microservicios.
- Posee configuraciones automáticas para diferentes Frameworks como son Spring Security, Spring Batch y otros.
- Posee todas las características de Spring Framework.
- Ofrece un amplio soporte a diferentes tecnologías como son bases de datos relaciones y no relacionales, almacenamiento en caché, mensajería, procesamiento por lotes y más.
- Ofrece métricas de la aplicación por medio de una librería llamada Actuator, utilizada para exponer información importante sobre la aplicación que se encuentra en ejecución a través del monitoreo.

Spring Boot incluye los starters, un conjunto de biblioteca que contiene una colección de todas las dependencias relevantes preconfiguradas que se necesitan para iniciar un proyecto con una funcionalidad particular. Con los starters se aplican una serie de configuración automáticas por defecto que hacen que la aplicación pueda desarrollarse y desplegarse rápidamente, de forma sencilla y también segura, reduciendo considerablemente la cantidad que pasos a realizar con respecto a un proyecto Spring.

Los starters están compuestos por una agrupación de las dependencias relacionadas a las tecnologías y módulos más comunes que se usan en el día a día en el desarrollo de aplicaciones web con Spring Framework. Estos solucionan la tediosa tarea de tener que añadir cada una de las dependencias requeridas para implementar una tecnología dentro del proyecto y simplifican el fichero que contiene las configuraciones para la compilación de la aplicación.

Todos los starters inician con el prefijo de spring-boot-starter en el nombre y vienen con una configuración por defecto.

Actualmente Spring Boot contiene más de 50 starters divididos en 3 grupos:

- **Spring Boot application starters:** Son los más comúnmente utilizados. A este grupo pertenecen los siguientes: spring-boot-starter-web, spring-boot-starter-test, spring-boot-starter-security y spring-boot-starter-data-jpa.

- **Spring Boot production starters**: Estos starters proveen un conjunto de dependencias relacionadas con la aplicación cuando se encuentra en un ambiente de producción. Actualmente solo existe spring-boot-starter-actuator que provee una librería de monitoreo y administración de aplicación.

- **Spring Boot technical starters**: Estos proveen un conjunto de dependencias a tecnologías alternativas a las utilizadas por defecto en Spring Boot.

1.5. Spring Initializr

Spring Initializr es una herramienta web que proporciona Spring y que está disponible en Spring.io para generar los proyectos de una forma rápida. De dicha página se obtendrá un archivo rar compatible con los principales entornos de desarrollo en Java: Netbeans, Eclipse o IntelliJ.

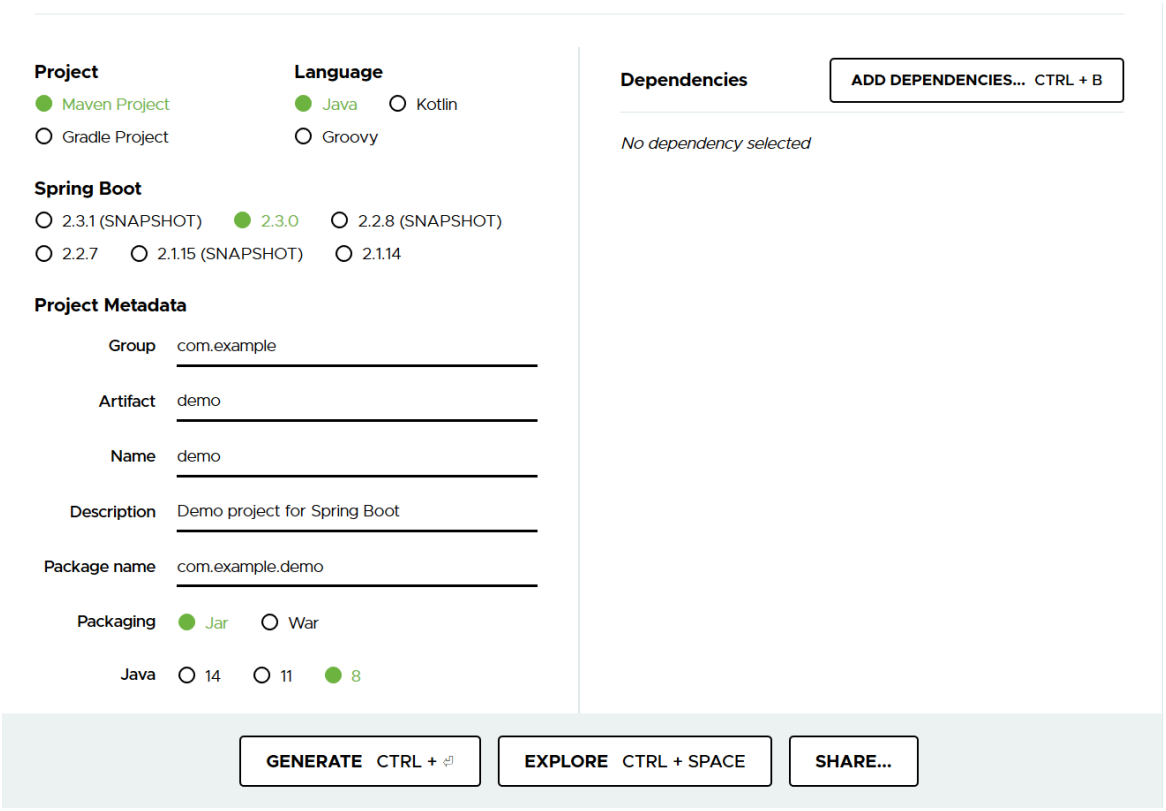


Figura 3: Spring Initializr

En la herramienta hay que definir los siguientes parámetros del proyecto:

- El lenguaje del proyecto
- El tipo de gestor de proyecto (maven o gradle),
- La versión del Spring Boot
- Los datos identificativos del proyecto
- El tipo empaqueta final del mismo
- O la versión de Java

Además, desde esta herramienta, también se pueden añadir los módulos de Spring que estarán disponible en la aplicación. La forma de realizar esto es mediante dependencias que se incluirán en el fichero pon.xml.

1.6. Programando en Spring

Para programar en Spring es fundamental entender que todo el framework se baja es en concepto de Inyección de Dependencia que es una forma de implementar el patrón Inversión de Control (Ioc). Tradicionalmente antes de utilizar un objeto era necesario instanciarlo para poder usarlo, ahora esa instanciación la realiza de forma automática Spring.

1.6.1. El contenedor de Spring

Ambos conceptos, Inyección de Dependencia e Inversión de Control, se implementan en Spring sobre sus Bean y el Application.Context.

1.6.1.1. Bean

Un bean es un objeto que Spring se encarga de crear y almacenar en su contenedor IoC. Además, Spring se encarga de manejar el ciclo de vida de estos objetos para que puedan ser inyectados cuando y donde sean solicitados. Estos beans se configuran mediante código Java añadiendo al objeto que implementan una sería de metadatos.

La forma más fácil de crear un bean es añadiendo la notación @Bean. Además, los módulos de Spring implementan otros tipos de Beans entre los que cabe destacar los siguientes:

Algunos de los componentes disponible en Spring son los siguientes:

- @Repository
- @Service
- @Controller
- @RestController
- @Component
- @Bean

1.6.1.2. ApplicationContext

ApplicationContext es un contenedor avanzado de Spring. Puede cargar definiciones de beans, unir (wire) beans y administrarlos siempre que se solicite. Este contenedor está definido en la interfaz *org.springframework.context.ApplicationContext*. Aparte de tener todas las funcionalidades de BeanFactory, añade otras como la capacidad de obtener mensajes de texto de un fichero .properties y la capacidad de publicar eventos a los elementos interesados.

En esta aplicación, se utiliza SpringApplication de Spring Boot, que genera un ApplicationContext de tipo ConfigurableApplicationContext. Para configurar beans, en vez de utilizar XML, también se puede utilizar anotaciones de Java. Dada la siguiente clase:

```
@Configuration
@ComponentScan
@EnableAutoConfiguration
public class Application {
   public static void main(String[] args) {
       SpringApplication.run(Application.class, args);
   }
}
```

- @EnableAutoConfiguration de Spring Boot inyecta automáticamente los beans que considera adecuados para que la aplicación pueda ejecutarse, mediante clases de auto-configuración. Estos beans irán siendo reemplazados por los que introduzca el desarrollador si lo desea. Las clases de auto-configuración son componentes de tipo @Configuration. Estas tres anotaciones suelen ir juntas en la clase principal de la aplicación, por lo tanto, se puede reemplazar por @SpringBootApplication.
- @ComponentScan realizar la búsqueda de beans para inyectarlos en el ApplicationContext,
- @Configuration indica que es un vean de tipo configuración.

Una vez que estos componentes han sido instanciados por Spring y están disponibles en su contenedor es posible inyectarlos en cualquier lugar de la aplicación usando la notación @AutoWired.

El siguiente ejemplo muestra una clase anotada con @Component, Spring se encarga de instanciar un objeto de ExampleComponent y dejarlo en su contendor a la espera de una petición

```
package com.welbits.spring.example;

import org.springframework.stereotype.Component;

@Component
public class ExampleComponent {

      public void method1() {
            ...
      }

      public boolean method2() {
            ...
      }

}
```

ExampleComponent se crea como un bean en el contendor de ApplicationContext y queda listo para ser inyectado mediante anotaciones. Como, por ejemplo:

```
package com.welbits.spring.example;

import org.springframework.beans.factory.annotation.Autowired;
import com.welbits.spring.example.ExampleComponent;

@Controller
public class ExampleController {

   private ExampleComponent exampleConponent;

   @Autowired
   public RootController(ExampleComponent exampleConponent) {
      this.exampleConponent = exampleConponent;
   }

   ...
}
```

La notación @Qualifier permite anexar un nombre a los componentes para hacer referencia a él posteriormente en el momento de su inyección. El añadir un identificador a los componentes evita errores cuando varios componentes implementan en mismo tipo, ya que ese identificador es univoco.

```
@Autowired
public UserServiceImpl(@Qualifier("userRepository1") UserRepository userRepository)
{
       this.userRepository = userRepository;
}
```

1.7. Servicios REST en Spring

1.7.1. Implementación de API Rest en Spring

Spring facilita notablemente la implementación de REST API gracias a su anotación @RestController que convierte a la clase en cuestión en controlador cuya respuesta no es una vista sino una respuesta de tipo JSON.

El controlador implementado servirá de interfaz para definir las distintas operaciones que el servicio REST permitirá realizar. A continuación, se muestra un ejemplo simple que permite recibir entidades del tipo User y entidades del tipo Recipe.

```
package com.welbits.spring.core;
…
@RestController
@RequestMapping("/api")
public class RestApiController {
    @Autowired
    private RecipeService recipeService;
    @Autowired
    private UserService userService;

    @RequestMapping(value = "/recipe/{recipeId}",
                    method = RequestMethod.GET)
    public Recipe viewRecipe(@PathVariable long recipeId) {
      Recipe recipe = recipeService.findById(recipeId);
      return recipe;
    }
    @RequestMapping(value = "/user/{userId}", method = RequestMethod.GET)
    public User viewUser(@PathVariable long userId) {
      User user = userService.findById(userId);
      return user;
    }
}
```

En el código anterior, desde el controlador se hace referencia a los repositorios de cada entidad para obtener un objeto de cierto tipo dado según el identificador de cada petición de tipo GET. La anotación @RequestMapping ("/api") reserva un espacio específico de URIs para el servicio.

Es común programar de forma explícita el estado de la respuesta que se incluirá. El siguiente código hace uso de la clase HttpServletResponse para generar esa respuesta 404 (Not Found).

```
@RequestMapping(value = "/user/{userId}", method = RequestMethod.GET)
public User viewUser(@PathVariable long userId, HttpServletResponse
response) {
      User user = userService.findById(userId);
      if (user == null) {
            response.setStatus(HttpServletResponse.SC_NOT_FOUND);
      }
      return user;

}
```

1.7.2. Consumo de servicios REST en Spring

La programación de aplicaciones con microservicios supone que éstos puedan comunicar entre sí para poder así colaborar en el objetivo final de la aplicación. Spring ofrece distintas alternativas para el consumo de API Rest: Rest Template y el cliente Feign

1.7.2.1. La clase ResTemplate

RESTTemplate es la clase que ofrece Spring para el acceso desde la parte cliente a Servicios REST. Esta clase forma parte del módulo web de Spring y basta con añadir el starter web para poder utilizarla.

Para su creación se define de la siguiente forma en una clase de configuración de la aplicación o en la clase principal. Al estar anotada con @Bean estará disponible en el contenedor de Spring para su inyección donde sea necesario.

```
@Bean
public RestTemplate getTemplate() {
        return new RestTemplate();
}
```

Para inyectarla en una variable se emplea la anotación @Autowired de Spring.

```
@Autowired
RestTemplate template;
```

La clase proporciona los siguientes métodos:

- método *getForObject* para la realización de peticiones GET de HTTP
- método *postForObject* para la realización de peticiones POST
- método *put* para la realización de peticiones PUT
- método *delete* para la realización de peticiones DELETE.

Para peticiones más avanzadas incluye el método *exchange()* que hace uso de la clase *RequestEntity* para crear la petición y *ResponseEntity* para devolver el resultado obtenido.

.

1.7.2.2. Cliente Feign

Feign es una librería que forma parte del stack de Spring Cloud, desarrollada por Netflix, para generar clientes de servicios REST de forma declarativa.

Al estilo de los repositorios de Spring Data, lo único que debemos hacer es anotar una interfaz con las operaciones de mapeo de los servicios que queremos consumir, parametrizando apropiadamente la entrada y salida de los mismos, para que se correspondan con los verbos y los datos de las operaciones de los servicios que queremos consumir.

Desde el punto de vista del soporte que tenemos a día de hoy con Spring, Feign nos facilitaría el trabajo, así como lo hace Spring Data, sin necesidad de «bajar» al nivel de *RestTemplate*, como Spring Data nos evita trabajar directamente con *EntityManager* o *JdbcTemplate*. Y, siguiendo con la comparación, igualmente la implementación se genera al vuelo en tiempo de arranque del contexto de Spring.

De entre sus características podemos encontrar las siguientes:

- Es altamente configurable, pudiendo usarse diversos encoders y decoders para formatear la información que viaja en cada petición y respuesta.
- Soporta las anotaciones de JAX-RS y Spring MVC para la declaración de los endPoints de los servicios REST.
- Se integra perfectamente con el resto de componentes del stack de Spring Cloud:
    - balanceo de carga con Ribbon,
    - circuit breaker con Hystrix,
    - registro de servicios en Eureka.

A continuación, se muestra la creación de un cliente Feign. La anotación @Feign recibe como parámetro el nombre del servicio al que deseamos lazar la petición.

AlumnoFeignClient.java

Project JDK is not defined

```
1   package com.formacionbdi.microservicios.app.cursos.clients;
2
3   import ...
8
9   @FeignClient(name = "microservicio-usuarios")
10  public interface AlumnoFeignClient {
11
12      @GetMapping("/alumnos-por-curso")
13      public Iterable<Alumno> obtenerAlumnosPorCurso(@RequestParam Iterable<Long> ids);
14  }
15
```

Figura 4: Creación de cun cliente Feign en Spring

A nivel de método, las anotaciones @GetMapping, @PostMapping, @PutMapping, etc. utilizadas para la creación de los servicios definen el tipo de petición que se realizará así mismo las anotaciones @PathVariable y @RequestParam definen los parámetros que se recibirán tras la petición, así como su tipo.

1.7.3. Spring Cloud

Spring Cloud es un módulo de Spring que proporciona herramientas para que los desarrolladores creen rápidamente algunos de los patrones comunes en los sistemas distribuidos, como lo son las aplicaciones implementadas con microservicios (por ejemplo, gestión de la configuración, descubrimiento de servicios, disyuntores enrutamiento inteligente, etc.).

Entre las ventajas que ofrece Spring Cloud están:

- Permite a los desarrolladores enfocarse en el desarrollo de la lógica de negocio y la creación de servicios.
- Modularidad y descomposición de todo el entorno para que al realizar cambios a un componente especifico, los demás no se vean afectados.
- Implementación y mantenimiento de forma sencilla y flexible a través de ficheros de configuración.
- Escalabilidad de las aplicaciones.

En la web principal de Spring Cloud se mencionan las características que posee el Framework, como se visualiza a continuación:

- Configuración distribuida / versionada.
- Servicio de registro y descubrimiento.
- Enrutamiento.
- Llamadas de servicio a servicio.
- Balanceo de carga.
- Disyuntores.
- Cerraduras globales.
- Gestión de clúster.
- Mensajería distribuida.

Es importante mencionar que el primer conjunto de herramientas lanzadas para desarrollar las aplicaciones distribuidas basadas en una arquitectura de microservicio en Java fue desarrollado por el equipo de Netflix con el nombre de Netflix OSS, las cuales buscaban solucionar los problemas que presentaban los sistemas distribuidos a gran escala.

Poco tiempo después, Spring Framework desarrolló su conjunto de herramientas Cloud basándose en la integración de los componentes de Netflix OSS; donde trabajando en conjunto con el equipo de la compañía de Streaming se formó Spring-Cloud-Netflix como herramienta principal para el desarrollo en un entorno en la nube.

Recientemente Spring ha comunicado que no seguirá desarrollando funcionalidades para las herramientas ese primer conjunto de Netflix. Por ahora si que seguirá ofreciendo soporte para la resolución de errores y bug. Por ello, es recomendable que los nuevos proyectos ya no usen esas primeras herramientas.

Spring Cloud está formado por 23 proyectos, de ellos, en este trabajo se utilizarán solamente dos:

- **Spring Cloud Netflix Eureka**: permite gestionar el registro y estado de los microservicios que forman la aplicación.
- **Spring Cloud Gateway**: implementa un único punto de acceso único a los componentes y microservicios que contiene una infraestructura.

A la hora de implementar un proyecto con Spring Cloud, es necesario tener en cuenta la compatibilidad de versiones entre este módulo y Spring Boot. En la página de Spring se especifica la siguiente tabla de compatibilidades.

Versión de Spring Cloud	Versión de Spring Boot
2020.0.x aka Ilford	2.4.x, 2.5.x (Starting with 2020.0.3)
Hoxton	2.2.x, 2.3.x (Starting with SR5)
Greenwich	2.1.x
Finchley	2.0.x
Edgware	1.5.x
Dalston	1.5.x

1.7.3.1. Spring Cloud Netflix Eureka

En de un sistema distribuido los componentes deben intercambiar información continuamente para que la aplicación pueda ser operativa, además unos componentes deben de conocer la existencia de otros y ser capaces de balancear la carga entrega las distintas instancias de un mismo componente.

Spring Cloud Netflix Eureka es un servidor de nombres basado en REST que viene cubrir esa necesidad en el entorno de desarrollo de microservicios de Spring.

Por defecto, todos los clientes del servicio, los microservicios, actualizarán su estado y sus datos de conexión (dirección IP y puerto) cada 30 segundos, a esto se le denomina "hearbeats". Dicha información de estado estará disponible para su consulta por parte de los microservicios que deseen conectar con otro de ellos.

Cuando un servicio no actualiza su estado en 90 segundo se marcará como no disponible para evitar que otros microservicios le sigan haciendo llegar peticiones.

El servidor Eureka (Listado 13.7) es una aplicación Spring Boot simple, que se convierte en un servidor Eureka a través de la anotación @EnableEurekaServer. Además, el servidor necesita una dependencia de spring-cloud-starter-eureka-server.

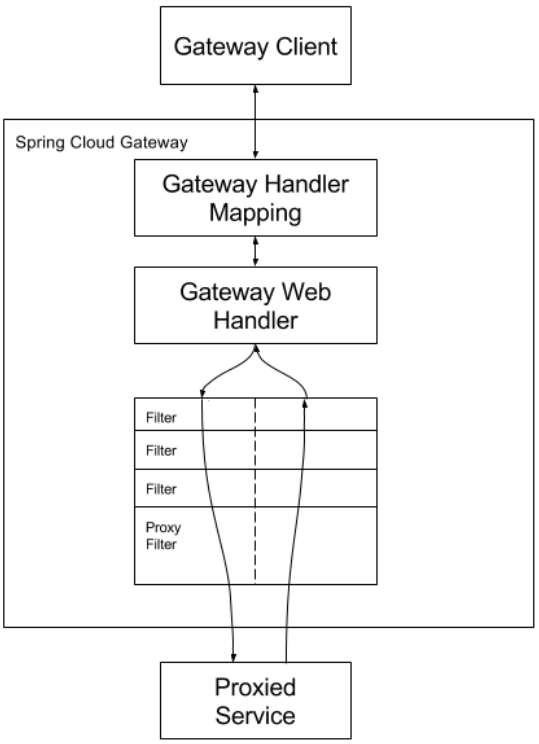
1.7.3.2. Spring Cloud Discovery

Este proyecto proporciona una biblioteca para construir una puerta de enlace API sobre Spring WebFlux. A través de dicha puerta de enlace Spring Cloud Disvery ofrecerá acceso a los microservicios de una infraestructura sin que el usuario tenga el detalle de donde se están ejecutando ese microservicio.

Con ello, Spring Cloud Gateway proporciona una forma simple de acceso toda la infraestructura. Además de realizar este enrutamiento, también proporciona funcionalidades como seguridad, monitoreo/métricas y resiliencia.

1.7.3.2.1. ¿Cómo funciona Spring Cloud Discovery?

El siguiente diagrama proporciona una descripción una descripción general de alto nivel de cómo funciona Spring Cloud Gateway



Los clientes realizan solicitudes a Spring Cloud Gateway. Si el Gateway Handler Mapping (controlado de mapeo) determina que una solicitud coincide con una ruta definida, se envía al Gateway Web Handler (controlador web de la puerta de enlace). Este controlador ejecuta la solicitud a través de una cadena de filtros que es específica de la solicitud. La razón por la que los filtros están divididos por la línea de puntos es que los filtros pueden ejecutar la lógica antes y después de que se envíe la solicitud del proxy. Se ejecuta toda la lógica del "pre" filtro. Luego se realiza la solicitud de proxy. Una vez realizada la solicitud de proxy, se ejecuta la lógica del filtro "posterior".

Esa cadena de filtros, así como el resto de la funcionalidad de Gateway puedes ser establecida y configuradas de forma programática haciendo uso de clases específicas de este módulo de Spring. Con dichas clases será posible definir tanto los predicados como los filtros de las rutas.

No obstante, el proyecto inicial de Spring Gateway ya viene configurado con una funcionalidad básica totalmente operativa siendo solamente necesaria establecer algunos parámetros básicos en el fichero application.properties con conseguir su operatividad.

1.1. ¿Qué es Vaadin?

Vaadin es un marco de interfaz de usuario de Java junto con componentes web de JavaScript que permite la creación de aplicaciones web en Java controlado por servidor, o en JavaScript, utilizando los componentes web de Vaadin con cualquier backend.

La parte relacionada con TypeScript se puede considerar un subproyecto de Vaadin denominado Vaadin Fusión y no será tratado ni utilizado en este proyecto.

Se utilizará Vaadin Flow. Vaadin Flow es un framework para el desarrollo web full-stack que se programa en Java y cuyas principales características son:

- Tiene una API de componentes basada en Java con más de 40 componentes de interfaz de usuario. Dichos componentes han sido cuidadosamente diseñados y se enfocan tanto en la experiencia de usuario final como en el desarrollador.
- Tiene también una API de enlace a datos para conectar los componentes de la interfaz de usuario o cualquier otro backend Java son seguridad de tipos.
- Ofrece comunicación automática entre el cliente y el servidor a través de XHR o WebSockets. Su arquitectura permite centrarse en la interfaz de usuario sin tener que pensar en la comunicación cliente-servidor.
- Ofrece enrutamiento entre vistas, formularios, internacionalización. Su API de enrutamiento permite crear estructuras de página jerárquica para que el usuario navegue.
- Implementa el patrón de inyección de dependencia lo cual lo hace compatible con Spring y CDI.

Por ejemplo, puede crear una interfaz de usuario simple en Java de la siguiente manera:

```java
// Create a layout for other components
VerticalLayout layout = new VerticalLayout();

// Use TextField for standard text input
TextField textField = new TextField("Your name");

// Button click listeners can be defined as lambda expressions
Button button = new Button("Say hello",
          e -> Notification.show("Hello!"));

// Add the web components to the HTML element
layout.add(textField, button);
```

Cuando se usa la API de Java, los componentes controlan sus contrapartes de JavaScript en el navegador, y no necesita saber nada sobre el HTML o JavaScript que se ejecuta bajo el capó.

Con el código anterior, Vaadin crea el DOM de HTML de la siguiente manera (se omiten los atributos y el DOM de sombra):

```HTML
<div>
  <vaadin-text-field></vaadin-text-field>
  <vaadin-button>Say hello</vaadin-button>
</div>
```

De hecho, también es posible escribir plantillas HTML como el fragmento anterior para crear interfaces de usuario en lugar de usar Java, y luego agregar la lógica de la interfaz de usuario con Java.

Vaadin viene con un gran conjunto de componentes de interfaz de usuario prediseñados, también llamados widgets o controles. Es posible utilizar los componentes de JavaScript tanto a través de la API de Java como en JavaScript. También permite combinarlos para crear interfaces de usuario complejas y ampliarlas para agregar funciones.

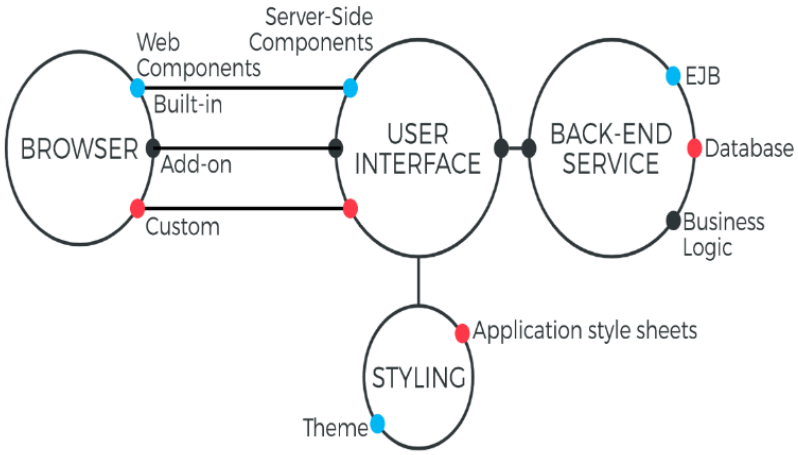
Vaadin también proporciona acceso completo al DOM, incluso desde Java del lado del servidor. La flexibilidad se extiende a la pila de programación; puede optar por escribir la interfaz de usuario en Java, TypeScript, JavaScript o cualquier combinación de ellos. Las mismas funciones están disponibles en su mayoría independientemente del idioma que utilice.

1.2. Arquitectura de la aplicación.

Trabajar con tecnologías web front-end, como HTML, CSS y JavaScript, puede ser un desafío y llevar mucho tiempo para los desarrolladores de Java. En Vaadin, todos los elementos de la interfaz de usuario se componen de componentes web. Esto hace que el desarrollo sea más fácil que nunca, porque cada elemento está desacoplado y aislado.

Vaadin incluye:

- Una API de componente de interfaz de usuario Java de tipo seguro en el lado del servidor que facilita el uso de los componentes web.
- Comunicación bidireccional automatizada entre el servidor y el navegador, que:
  - Ofrece a los desarrolladores de Java acceso completo a todas las mejoras web modernas.
  - Facilita la conexión de la interfaz de usuario a los datos a través de un robusto backend de Java, en lugar de utilizar la comunicación tradicional basada en REST.
  - Le permite elegir el protocolo de comunicación HTTP (S) básico o WebSocket para crear aplicaciones en tiempo real.
- Enlace de datos bidireccional: cuando la interfaz de usuario cambia en el cliente o en el servidor, los cambios se reflejan automáticamente en el otro lado.



Vaadin le permite acceder a las API del navegador, componentes web e incluso elementos DOM simples, directamente desde Java del lado del servidor. No es necesario comprender cómo funcionan la comunicación de cliente a servidor o los componentes web. Esto le deja libre para concentrarse en crear componentes que funcionen a un nivel de abstracción superior.

1.3. Herramientas requeridas

Para el desarrollo de una aplicación con Vaadin son necesarias las siguientes herramientas:

- Java JDK 8 (o posterior). Aunque se recomienda el uso de Java 11 (la última versión LTS) o posterior.
- Maven versión 3.5 (o posterior) o Gradle versión 6 (o posterior.
- Node.js versión 10 (o posterior)
- npm versión 5.6 (o posterior)

La administración de Vaadin y otras bibliotecas de Java puede resultar tediosa de hacer manualmente, por lo que se recomienda usar un sistema de compilación que administre las dependencias automáticamente. Vaadin se distribuye en el repositorio central de Maven y se puede utilizar con cualquier sistema de gestión de compilación o dependencia que pueda acceder a repositorios de Maven, como Ivy o Gradle, además de Maven.

El administrador de paquetes npm se usa para administrar las dependencias frontend de Vaadin.

1.4. Dependencias de la plataforma Vaadin

La plataforma Vaadin incluye un conjunto de componentes, con API de Java del lado del servidor, que puede utilizar para crear su interfaz de usuario. Los componentes, junto con Flow, se incluyen como dependencias de plataforma.

El módulo *vaadin-core* incluye todos los componentes de código abierto, como TextField, Button y Grid. El módulo *vaadin* amplía este conjunto para incluir todos los componentes admitidos oficialmente en la plataforma Vaadin, como Vaadin Charts, aunque algunos de ellos requieren licencia para su uso.

Los componentes forman parte de la plataforma Vaadin y se incluyen como dependencias, junto con Flow. Cada componente tiene una API de Java.

El uso de la dependencia de la plataforma (com.vaadin:vaadin) garantiza que todos los componentes disponibles, tanto de código abierto como comerciales, se incluyan automáticamente. Tiene la garantía de obtener versiones compatibles tanto de Flow como de los componentes.

```xml
XML
<dependencies>
    <!-- other dependencies -->
    <!-- component dependency -->
    <dependency>
        <groupId>com.vaadin</groupId>
        <artifactId>vaadin</artifactId>
        <version>${vaadin.platform.version}</version>
    </dependency>
</dependencies>
```

Figura 5: Declaración de la dependencia vaadin.platform

1.4.1. Dependencias de componentes individuales

Como alternativa al uso de la dependencia de la plataforma, puede declarar componentes individuales como dependencias. Para ello, es decir para agregar un único componente es necesario añadir el paquete *bom* de componentes Vaadin y el paquete correspondiente a cada uno de los componentes que queramos usar en el proyecto. Por ejemplo, el siguiente código agrega al proyecto sólo el componente button.

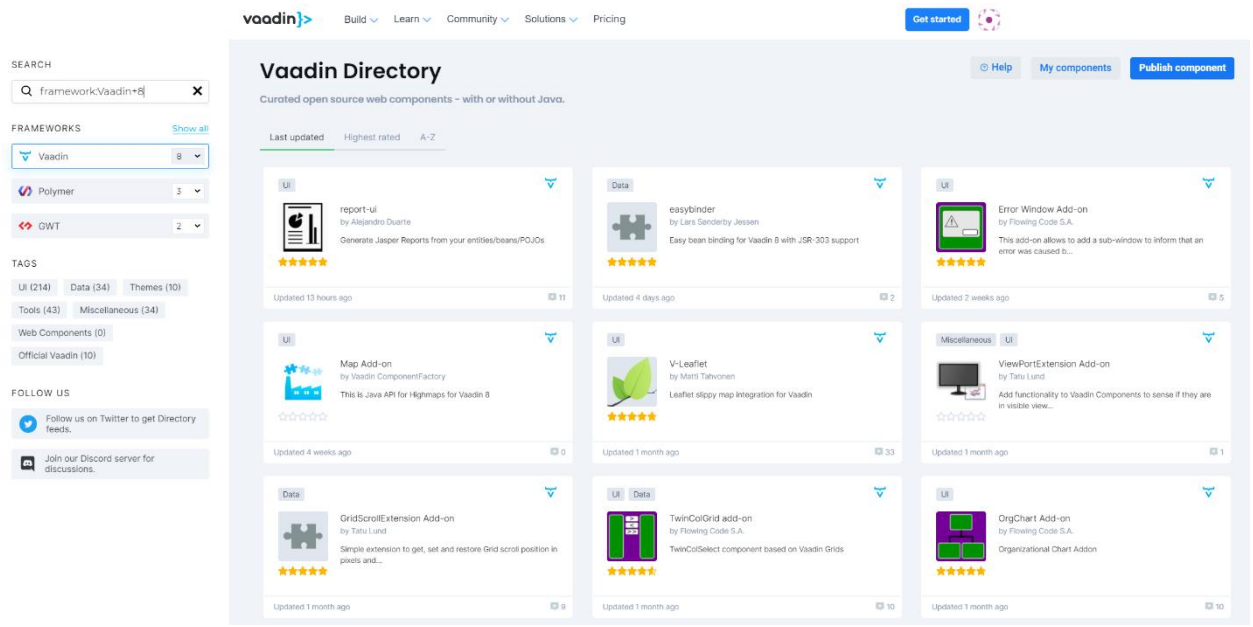
```xml
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>com.vaadin</groupId>
            <artifactId>vaadin-bom</artifactId>
            <version>
                ${vaadin.platform.version}
            </version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>

<dependencies>
    <!-- other dependencies -->

    <!-- component dependency -->
    <dependency>
        <groupId>com.vaadin</groupId>
        <artifactId>vaadin-button-flow</artifactId>
    </dependency>
</dependencies>
```

La dependencia *bom* corrige todas las dependencias relacionadas con Vaadin a una combinación probada, de modo que los componentes individuales se pueden agregar de forma segura. Sin la lista de materiales, algunas dependencias generadas por https://www.webjars.org/ pueden cambiar en el futuro, debido a nuevas versiones o debido al uso de rangos de versiones.

1.5. Componentes Vaadin

1.5.1. Listado de componentes

El sitio web de Vaadin ofrece un directorio de componentes en el que están incluidos además de los 40 que forman parte del paquete básico otros muchos componentes desarrollados por terceros a partir de las 40 iniciales.



Por cada uno de los componentes se incluye una API para su uso, ejemplos, versiones, así como su compatibilidad con la versión del framework.

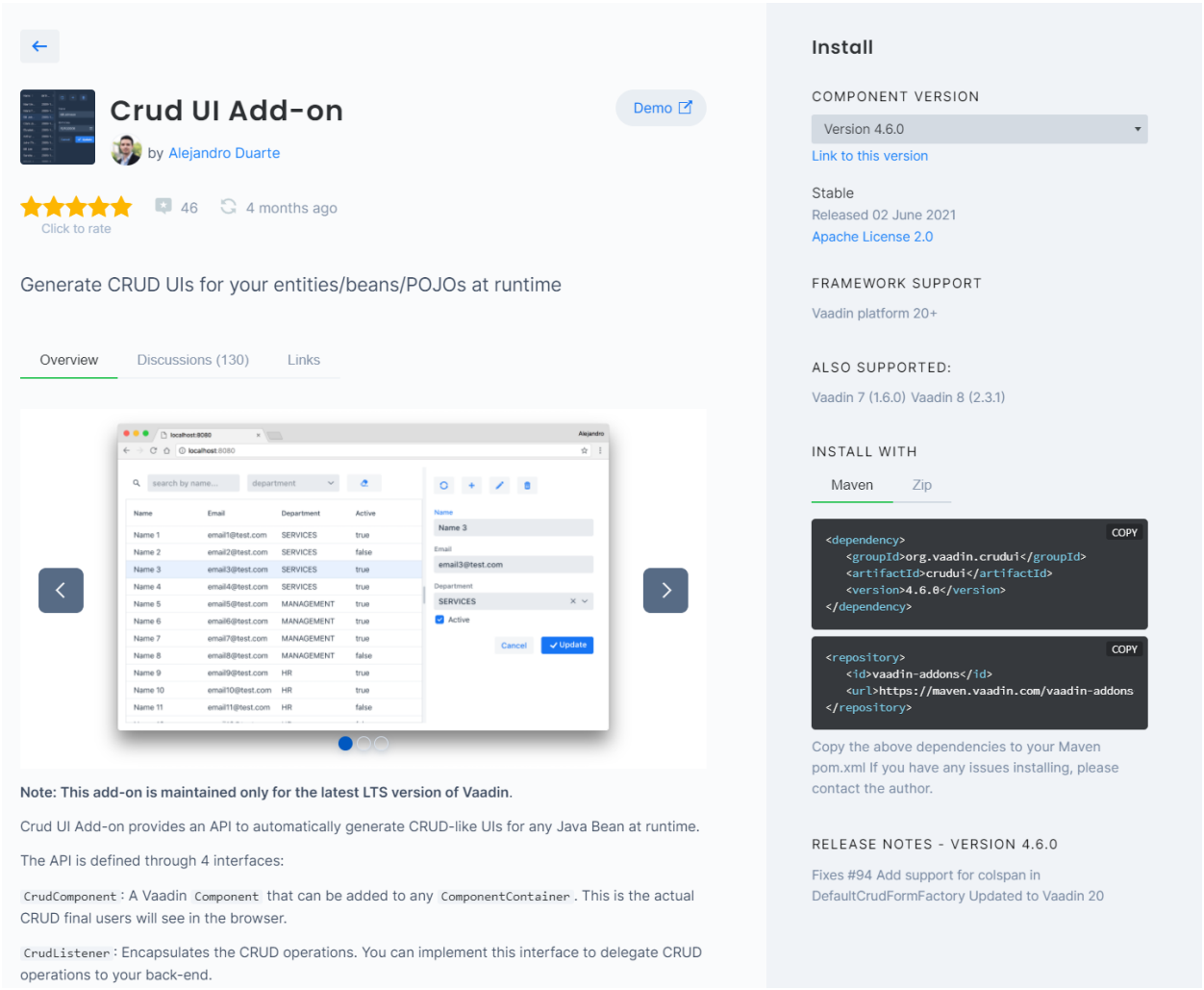


Figura 6: Información de un componentes Vaadin

1.5.2. Características básicas de los componentes

Todos los componentes tienen una serie de atributos común:

- **Id**: identificador del componente. Debe ser único en la página.
- **Element**: hace referencias al elemento raíz y permite acceder a la funcionalidad de bajo nivel utilizando el método *component.getElement()*.
- **Visible**: define si el componente es visible o no en la interfaz de usuario.
- **Enabled**: define si el componente está activado o por el contrario bloque cualquier interacción del cliente al servidor.

1.5.3. Creación de un componente

Vaadin ofrece varias formas de crear un componente, la más sencilla es instanciando un objeto de su clase tal y como se muestra en el siguiente código:

```java
@Tag("input")
public class TextField extends Component {

    public TextField(String value) {
        getElement().setProperty("value",value);
    }
}
```

En ese componente, el elemento raíz es:

- Creado automáticamente (clase *Component*)según la anotación @Tag
- Accedido usando el método *getElement()*
- Se utiliza para establecer el valor inicial del campo.

También es posible crear componentes a partir de otros existentes extendiendo la clase Composite tal y como se muestra en el siguiente código:

```java
public class TextField extends Composite<Div> {

    private Label label;
    private Input input;

    public TextField(String labelText, String value) {
        label = new Label();
        label.setText(labelText);
        input = new Input();
        input.setValue(value);

        getContent().add(label, input);
    }
}
```

En dicho código:

- el composite, crea automáticamente el componente raíz (especificado mediante genéricos con *Composite<Div>*).
- El componente raíz está disponible a través del método *getContent()*.
- En el constructor, sólo es necesario crear los componentes secundarios y agregarlos a la raíz Div
- El valor se establece mediante el método *setValue* del componente Input.

Una vez creado un componente, es común agregarle una API para modificar su comportamiento y establecer sus principales propiedades. Dicho componente y su API podrá ser reutilizado posteriormente.

También es posible crear un componente ampliando cualquier componente existente. Para la mayoría de los componentes, hay un componente del lado del cliente y un componente del lado del servidor correspondiente:

- Componente del lado del cliente: contiene HTML, CSS y JavaScript, y define un conjunto de propiedades que determinan el comportamiento del componente en el lado del cliente.
- Componente del lado del servidor: contiene código Java que permite cambiar las propiedades del lado del cliente y gestiona el comportamiento del componente en el lado del servidor.

Puede extender un componente en el lado del servidor o del cliente. Tenga en cuenta que estos son enfoques alternativos que se excluyen mutuamente.

La extensión de un componente del lado del servidor es útil cuando desea agregar una nueva funcionalidad (en lugar de aspectos visuales) a un componente existente. Los ejemplos adecuados incluyen el procesamiento automático de datos, la adición de validadores predeterminados y la combinación de varios componentes simples en un campo que administra datos complejos. Como alternativa, puede ampliar la clase Composite que tiene una API mínima. Esto oculta los métodos disponibles en la API más extensa que se expone cuando sus componentes personalizados amplían una implementación de *Component*.

1.5.4. Vincular datos a formularios

En muchas aplicaciones, los usuarios proporcionan datos estructurados al completar campos en formularios. Estos datos generalmente se representan en código como una instancia de un objeto de la lógica de negocio (JavaBean), por ejemplo, una persona en una aplicación de recursos humanos.

La clase *Binder* le permite definir cómo los valores de un objeto de la lógica de negocio se vinculan a los campos de la interfaz de usuario.

Binder lee los valores en el objeto comercial y los convierte del formato esperado por el objeto comercial al formato esperado por el campo, y viceversa. Es de destacar que *Binder* sólo puede vincular componentes que implementan la interfaz *HasValue*, por ejemplo, TextFieldy ComboBox.

Binder también permite validar la entrada del usuario y presentar el estado de validación al usuario de diferentes formas.

1.6. Rutas y navegación

Vaadin proporciona la clase *Router* para estructurar la navegación de su aplicación web en partes lógicas.

El enrutador se encarga de entregar el contenido cuando el usuario navega dentro de una aplicación Incluye soporte para rutas anidadas, acceso a parámetros de ruta y más.

Para definir el contenido de una ruta de navegación se hace uso de una clase que contendrá todos los componentes de forma programática y que estará anotada con la anotación @Route. Si omite el parámetro en la anotación @Route, el destino de la ruta se deriva del nombre de la clase. Por ejemplo, se MyEditorvuelve "myeditor", se PersonViewvuelve "person"y se MainViewvuelve "". También se puede explicitar la ruta haciendo uso de parámetros. A continuación se muestran algunos ejemplos.

```java
@Route("")
public class HelloWorld extends Div {
    public HelloWorld() {
        setText("Hello world");
    }
}
```

Figura 8: Definición del componente HelloWorld como el destino de ruta predeterminado (ruta vacía) para su aplicación

```java
@Route("some/path")
public class SomePathComponent extends Div {
    public SomePathComponent() {
        setText("Hello @Route!");
    }
}
```

Figura 7: Definición del componentes SomePathComponent como el obtivo para la ruta específica some/path

1.6.1. Ciclo de vida de la navegación

El ciclo de vida de la navegación se compone de una serie de eventos que se activan cuando un usuario navega en una aplicación de un estado o vista a otro.

Los eventos se disparan a los oyentes agregados a la instancia UI y a los componentes adjuntos que implementan interfaces de observador relacionadas.

1.6.1.1. BeforLeaveEvent

BeforeLeaveEvent es el primer evento que se activa durante la navegación. Este evento permite posponer, cancelar o cambiar la navegación a un destino diferente y se entrega a cualquier instancia de componentes que implemente la interface *BeforeLeaveObserver* antes de que comience la navegación, es decir en la página

origen. También es posible registrar un listener independiente para este evento usando el método del UI *addBeforeLeaveListener(BeforeLeaveListener)*.

Un caso de uso típico de este evento es preguntar al usuario si desea guardar los cambios no guardados antes de navegar a otra parte de la aplicación.

BeforeLeaveEvent incluye el método postpone que puede ser usado para posponer la navegación en curso hasta que una cierta condición se cumpla. Este método interrumpe el proceso de notificación a los observers y listeners. Cuando se reanuda la transición, se llama a los observers restantes (los que siguen al observer que inició el aplazamiento).

1.6.1.2. BeforeEnterEvent

BeforeEnterEvent es el segundo evento que se activa durante la navegación. El evento le permite cambiar la navegación para ir a un destino diferente al original.

Este evento se utiliza normalmente para reaccionar ante situaciones especiales, por ejemplo, si no hay datos para mostrar o si el usuario no tiene los permisos adecuados.

Este evento se envía a cualquier implementación de instancia de componente *BeforeEnterObserver* que esté adjunta a la interfaz de usuario a partir de la interfaz de usuario que se mueve a través de los componentes secundarios.

El evento se dispara:

- Solo después de que se haya continuado a postpone (llamado durante a BeforeLeaveEvent).
- Antes de desconectar y adjuntar componentes para que la interfaz de usuario coincida con la ubicación a la que se navega.

También es posible registrar un oyente independiente para este evento usando el método addBeforeEnterListener (BeforeEnterListener) método en UI.

1.6.1.3. AfterNavigationEvent

AfterNavigationEvent es el tercer y último evento que se activa durante la navegación.

Este evento se usa generalmente para actualizar varias partes de la interfaz de usuario después de que se completa la navegación real. Los ejemplos incluyen ajustar el contenido de un componente de ruta de navegación y marcar visualmente el elemento de menú activo como activo.

El evento se dispara:

- Después BeforeEnterEvent, y
- Después de actualizar qué componentes se adjuntan a la interfaz de usuario.

En este punto, el estado de navegación actual se muestra al usuario y ya no es posible realizar más redireccionamientos y cambios similares.

El evento se entrega a cualquier implementación de instancia de componente AfterNavigationObserver que se adjunta después de completar la navegación.

También es posible registrar un oyente independiente para este evento utilizando el método *addAfterNavigationListener(AfterNavigationListener)* en la interfaz de usuario.

MICROSERVICIOS

1.1. ¿Qué es un microservicio?

Un microservicio es una entidad software que posee las siguientes características:

- Aislamiento de otros microservicios, unos microservicios no dependen para su ejecución de otros.
- Autonomía: los microservicios pueden ser desplegados y destruidos independientemente unos de otros.
- Tienen una granularidad fina: son responsables únicos de una determinada funcionalidad.

La idea principal detrás de los microservicios es que algunos tipos de aplicaciones son más fáciles de construir y mantener cuando se dividen en muchas piezas más pequeñas que funcionan juntas. Aunque la arquitectura ha aumentado la complejidad, los microservicios aún ofrecen muchas ventajas sobre la estructura monolítica.

La facilidad que ofrecen los microservicios de separarse recombinarse protege todo el sistema contra el deterioro y facilita los procesos ágiles, haciendo que la metodología sea atractiva para las organizaciones, especialmente para las que todavía utilizan infraestructuras monolíticas.

Los microservicios colaboran y se comunican entre sí para implementar así una aplicación completa. Esta arquitectura utiliza API para pasar información de un servicio a otros. El funcionamiento software subyacente o el hardware sobre el que se basa el servicio depende únicamente del equipo que creo el servicio. Esto hace que la aplicación final sea más resistente en su desarrollo.

La principal ventaja del uso de microservicios es que las aplicaciones creadas como un conjunto de componentes modulares independientes son más fáciles de probar, mantener y comprender ya que permiten a las organizaciones.

- Aumentar la agilidad
- Mejorar los flujos de trabajo
- Disminuir la cantidad de tiempo que se necesita para mejorar la producción.

Si bien cada componente independiente aumenta la complejidad, el componente también puede tener capacidades de monitoreo adicionales para combatirlo.

También son de destacar los siguientes aspectos a favor de los microservicios:

- Aislamiento y resiliencia: si uno de los componentes falla, debido a problemas como tecnología obsoleta o imposibilidad de desarrollar más el código, los desarrolladores pueden activar otro componente mientras el resto de la aplicación continúa funcionando de forma independiente. Esta capacidad brinda a los desarrolladores la libertad de desarrollar e implementar servicios según sea necesario, sin tener que esperar a las decisiones relativas a toda la aplicación.
- Escalabilidad: debido a que los microservicios están hechos de componentes mucho más pequeños, pueden consumir menos recursos y, por lo tanto, escalar más fácilmente para satisfacer la creciente demanda de este componente específico.

- Como resultado de su aislamiento, los microservicios pueden funcionar correctamente incluso durante grandes cambios de tamaño y volumen, lo que los hace ideales para empresas que trabajan con una amplia grama de plataformas y dispositivos.
- Desarrollo de forma autónoma: a diferencia de los monolitos, los componentes individuales son mucho más fáciles de encajar en tuberías de entrega continua y escenarios de implementación complejos. Solo el servicio identificado debe modificarse y volver a implementarse cuando se necesita un cambio. Si un servicio falla, los demás seguirán funcionando de forma independiente. Su carácter autónomo beneficia a los equipos porque:
    - Habilita el escalado y desarrollo
    - No requiere mucha coordinación con otros usuarios.
- Relación con la empresa: las arquitecturas de microservicios se dividen a lo largo de los límites del dominio empresarial, organizadas en torno a capacidades como logística, facturación, etc. Esto aumenta la independencia y la compresión en toda la organización: diferentes equipos pueden utilizar un producto específico y luego poseerlo y mantenerlo durante su vida útil.
- Evolución: cualquier arquitectura de microservicios es altamente evolutiva. Los microservicios son una excelente opción para situaciones en las que los desarrolladores no pueden predecir completamente a qué dispositivos accederá la aplicación en el futuro. También permiten cambios rápidos y controlados en el software sin ralentizar la aplicación en su conjunto, por lo que puede ser más iterativo en el desarrollo de funciones y nuevos productos.

De entre las ventajas de los microservicios cabe destacar las siguientes:

- Por supuesto, la arquitectura de microservicios viene con una curva de aprendizaje. Los usuarios nuevos pueden tener dificultades para determinar:
    - El tamaño de cada microservicio
    - Límites óptimos y puntos de conexión entre microservicios.
    - El marco adecuado para integrar servicios.
- Mayor complejidad: en primer lugar, los microservicios son un sistema muchos más complicado. Tiene una curva de aprendizaje que puede ser empinada por escalar, pero una vez que se aprende, como la mayoría de las cosas, se puede usar con facilidad. No obstante, es necesario tener en cuenta que la arquitectura de microservicios no siempre es la mejor solución para una aplicación. Para algunos casos puede ser demasiado compleja.
- Mas caro. Los microservicios también pueden ser más costosos. Por lo general, se ejecutan en sus propios entornos con sus propias CPU. Funcionan a través de llamadas API que tienen una

1.2. Aplicaciones basadas en microservicios

Desde la aparición de Internet en 1969, esta red de redes ha transformado e interconectado todos los aspectos de la sociedad. Desde un punto de vista del mercado, se benefician tanto las multinacionales como empresas locales, ya que ambas pueden distribuir su producto a nivel global, aunque también trae consigo ciertas "desventajas" ya que al aumentar tanto la base de clientes potenciales aumenta el número de competidores, ya que todas las empresas pertenecen al mismo mercado globalizado. Esta presión por sobrevivir en la lucha competitiva ha afectado a la manera en la que los desarrolladores construyen las aplicaciones. Podemos identificar varios factores que empujaron al mundo del desarrollo a crear la arquitectura basada en microservicios:

- **Complejidad**: Las empresas actualmente suelen externalizar servicios, los cuáles antes estaban incluidos en su propia estructura. Este cambio se ve reflejado en que las aplicaciones monolíticas para gestionar la empresa se hayan visto sustituidas y surjan aplicaciones que se comuniquen con varios servicios y bases de datos, que pueden pertenecer a la empresa o no.
- **Rapidez**: Hoy en día los clientes habitualmente se decantan por productos software sean actualizados en intervalos de tiempo cortos o medios, para disponer de nuevas características cuanto antes.
- **Rendimiento y escalabilidad**: Al pertenecer a un mercado global, las aplicaciones deben tener mecanismos para escalar y desescalar según la demanda y mantener el rendimiento en todas las situaciones, ya que no sabemos exactamente cuál puede ser el volumen de transacciones en un momento determinado.
- **Fiabilidad**: Nuestra aplicación debe ser altamente fiable porque un sólo problema puede hacer que nuestro cliente abandone nuestro servicio y esto empeore nuestra imagen en el mercado.

El concepto fundamental detrás de los microservicios es, aunque sea paradójico, que los desarrolladores debemos comprender que para construir servicios altamente escalables para poder dar servicio a una mayor cantidad de clientes necesitamos desgranar nuestras aplicaciones en pequeños servicios que puedan ser construidos y desplegados de manera independiente. Este concepto nos permite construir sistemas que son:

- **Flexibles**: Dado que nuestra aplicación se compone de servicios pequeños y de única función, podemos reemplazarlos o cambiarlos para ofrecer nuevas funcionalidades rápidamente.
- **Resilientes**: Esta arquitectura tiene una gran resistencia ante los fallos gracias a la modularidad del sistema. Esto nos permite que cuando un servicio falle, pueda ser sustituido rápidamente por otro. También será resistente a la degradación del código, debida al paso del tiempo y al avance en las tecnologías que rodean a una aplicación. Esta se suaviza en la arquitectura de microservicios, ya que, al igual que con los fallos, la modularidad de la aplicación nos permite ir actualizando y reemplazando los servicios obsoletos sin necesidad de reconstruir la aplicación a gran escala.
- **Escalables**: Nuestra aplicación se debe poder escalar horizontalmente a través de diferentes servidores. Esto nos permite responder a los aumentos de la demanda de cualquier servicio de la aplicación, haciendo ese coste efectivo ya que únicamente potenciamos la disponibilidad del recurso necesario.

1.3. Definición y principios de los microservicios

Los microservicios constituyen un enfoque arquitectónico y organizativo en el desarrollo software donde las aplicaciones se basan en la creación de pequeños servicios independientes que se comunican a través de APIs bien delimitadas. Nos permiten desarrollar aplicaciones con módulos físicamente separados, incluso pueden escribirse en distintos lenguajes de programación y con conexión a distintos tipos de bases de datos. La idea es que cada uno de ellos implemente una funcionalidad única y su despliegue sea de manera individual.

Amazon, Netflix y eBay son ejemplos de aplicación exitosa de esta arquitectura. Surgieron basados en la idea propuesta por Alister Cockburn en 2005 de la arquitectura hexagonal (también conocida como arquitectura de Puertos y Adaptadores). Esta arquitectura proponía encapsular las funciones de negocio del resto del mundo. Estas funciones, al estar aisladas, no conocían los canales de entrada o los formatos de mensajes que podían recibir. Esto se solucionaba convirtiendo los diferentes mensajes que recibía la aplicación desde diferentes dispositivos a uno que fuese conocido por las funciones. El proceso que llevan a cabo los puertos y adaptadores pasa desapercibido tanto por las aplicaciones externas como las funciones internas. Esto nos permite realizar cambios sin tener que preocuparnos demasiado por los diseños de interfaces.

Una analogía que nos hará entender mejor los microservicios es un panal. Las abejas construyen el panal juntando celdas hexagonales de cera. Empiezan poco a poco, construyendo este con los materiales que tienen disponibles para construirlo. Repiten el patrón de construcción de las celdas, creando una estructura muy robusta, que está compuesta por celdas individuales unidas unas con otras. Los daños a una sola celda no repercuten a las demás y se pueden reconstruir fácilmente.

No existe una definición oficial sobre qué es un microservicio, pero la concepción sobre los mismos más aceptada es que es una técnica de desarrollo de aplicaciones software según la cual podemos construir una aplicación como un conjunto de servicios modulares, pequeños e independientemente desplegables (¡Como las abejas en el panal!). Tampoco existe un consenso para definir cuál debería ser el tamaño de un microservicio ni el número de microservicios que debe haber en una infraestructura. Lo verdaderamente importante es la creación de servicios independientes y ganar autonomía en su despliegue y escalado: si un microservicio no tiene una funcionalidad propia y su existencia se basa en estar interconectado con otro para realizar cualquier acción, quizás es que esos dos microservicios deberían ser uno únicamente. Actúan como un sistema distribuido en el que cada servicio tiene deseablemente una única responsabilidad y se intercomunican a través de protocolos independientes de la implementación como HTTPS, WebSockets, AMQP, ….

En la popularización de la arquitectura de los microservicios contribuyó en gran medida el avance de tecnologías como HTML 5, Angular y la expansión de los proveedores de servicios cloud PaaS. Por último, no podemos olvidar que Docker dio el impulso final a esta arquitectura para que se lanzase de lleno a la popularidad. De estas cuestiones hablaremos más adelante, ahora vamos a repasar los principios en los que se fundamenta esta tecnología:

• Principio de única responsabilidad: Es uno de los principios de los patrones de diseño software SOLID, que están orientados al desarrollo de aplicaciones que usen la programación orientada a objetos (OOP). Describe que una unidad debe tener una única responsabilidad. Este principio abarca a funciones, clases o servicios, que actúan como unidad. Como podemos deducir, las responsabilidades no pueden ser

compartidas entre servicios, aunque existen excepciones en las que por limitaciones del negocio podemos saltarnos este principio. La unidad en una aplicación basada en microservicios sería, lógicamente, un microservicio. Este principio nos aporta una alta cohesión en cada uno de los microservicios.

- Principio de Autonomía: Los microservicios son servicios autónomos e independientemente desplegables, que toman la responsabilidad completa sobre una tarea y su ejecución. Cuando un servicio es desplegado, decimos que se ha instanciado una copia de él. Pueden operar y cambiar independientemente del resto que le rodean. Recogen las dependencias como librerías, entornos de ejecución, e incluso a sí mismos en servidores, contenedores o máquinas virtuales para abstraer los recursos físicos.
- Principio de Descentralización: Las aplicaciones basadas en la arquitectura de microservicios poseen una gestión de datos descentralizada, ya que cada servicio gestiona su propia base de datos (si la necesita), dándonos mucha flexibilidad y consistencia para realizar cambios a múltiples recursos.
- Principio de bajo acoplamiento: Interactúan por interfaces claramente definidas o a través de eventos, mientras la implementación interna permanece independiente. Esto causa que los microservicios actúen como cajas negras, por lo que, desde fuera de ellos, no podemos conocer aspectos como sus estructuras de datos, tecnologías o su lógica. Esto también tiene una repercusión cultural en el mundo del desarrollo, pues permite que los equipos de desarrollo se organicen de manera independiente y la aplicación se construya rápidamente ensamblando los diferentes componentes.
- Principio de Resiliencia: Los microservicios son un mecanismo natural para aislar los fallos. Al ser independientes, si ocurre un fallo lo más probable es que ese fallo afecte sólo a una parte del sistema. También nos permite añadir cambios más pequeños a la aplicación en vez de actualizaciones muy grandes que pueden ser peligrosas.
- Principio de Antifragilidad: Recuperarse rápido de los fallos es indispensable para construir sistemas resistentes y tolerantes a ellos. Este principio asume que nuestra aplicación puede fallar (e inevitablemente ocurrirá), y cuando detecta un fallo debe recuperarse de él lo más rápido posible, eliminando los procesos que sean necesarios para no afectar al sistema completo. La auto sanación suele acompañar a este principio y se usa habitualmente en microservicios, donde el sistema puede detectar los fallos y se ajusta automáticamente para evitar seguir fallando.
- Principio de bajo peso: Al implementar una función específica, estos son muy ligeros. Debemos tener en cuenta la selección de tecnologías que lo respaldan, asegurándonos que mantendrán esta característica. Esto permite que su instanciación y despliegue sean muy rápidos y baratos.
- Principio de reusabilidad: Al estar concebidos como cajas negras, podemos reutilizar microservicios ya codificados y probados ajenos a la aplicación que encajen con una parte de esta. Debemos tener en cuenta que la interfaz de comunicación y los resultados que ofrezca el microservicio estén en consonancia con nuestra aplicación. Este principio es muy abierto ya que, además, podemos reusar las partes que nos interesen de microservicios construidos.
- Principio de Interoperabilidad: Los microservicios conviven en un ecosistema donde existen variedad de ellos, y consiguen sus objetivos mediante la comunicación.
- Principio de transparencia: La monitorización de cada microservicio es muy importante para asegurar que cuando ocurran problemas, sabremos cuál es su fuente y poder resolverlos eficazmente.

- Principio plurilingüe: Los microservicios deben comunicarse entre ellos para conseguir sus objetivos. Esto se realiza con protocolos de comunicación independientes de la implementación, lo cual nos permite que los servicios puedan estar construidos con diferentes tecnologías y lenguajes.
- Principio de automatización: El ciclo de vida de los microservicios (que explicaremos más adelante) debe estar completamente automatizado ya que gestionarlo manualmente no tendría sentido por el alto coste y esfuerzo que conllevaría, sin ofrecernos prácticamente ninguna ventaja. Implementar esta característica es una tarea compleja, pero a la larga nos ahorrará tiempo y posibles fallos en varios procesos. Los procesos de build, testeo, despliegue y escalada son los que debemos automatizar para conseguir manejar el ciclo de vida de los servicios de esta manera.
- Principios del ecosistema de la aplicación: Alrededor de los servicios también tendremos que organizar un ecosistema para mejorar la calidad de la aplicación. Entre muchos aspectos, los más importantes son procesos DevOps, gestión de un log centralizado, API Gateway y monitorización. Estos conceptos se verán en más profundidad junto con la arquitectura de los microservicios.

Tras esta revisión de principios de los microservicios podemos entender por qué los microservicios supusieron una revolución en su momento. Para implementar todas ellas, debemos conocer a fondo la arquitectura y patrones típicos de una aplicación basada en microservicios, que son las que consiguen que los microservicios nos puedan ofrecer todas estas características.

1.4. Cuando no usar microservicios

Aunque sean una arquitectura muy popular y parece estar en boca de todos los desarrolladores, los microservicios no son una solución universal y en este apartado vamos a explorar cuáles son los escenarios en los que una arquitectura de microservicios no sería deseable:

- Traen consigo una gran complejidad que las aplicaciones monolíticas no poseen a la hora de desarrollarlas. Definir los requisitos de cada servicio es una tarea que conlleva un gran esfuerzo. Si por motivos económicos o temporales no se desea invertir en todas las tareas que acompañan a los microservicios tales como la monitorización o la escalabilidad, no debemos optar por esta arquitectura.
- Una aplicación basada en microservicios puede tener una gran cantidad de servidores que usa simultáneamente, en la que se ejecutan instancias de los servicios. Aunque la nube nos facilita la tarea de gestión de los servidores y abarata costes, si no se está preparado para gestionar y monitorizar la complejidad operacional de tantos servidores, esta arquitectura podría ser un problema más que una solución.
- La arquitectura de microservicios está orientada hacia la escalabilidad y la reusabilidad. Si nuestro objetivo es diseñar una aplicación pequeña y con una base de usuarios también pequeña, sería un error optar por esta arquitectura dados los costes que conlleva adicionales al desarrollo del código fuente.

Si la aplicación que queremos desarrollar maneja multitud de datos diferentes y requieren muchas transformaciones complejas y agregaciones, los microservicios tampoco son una buena idea. Esto es por su naturaleza distribuida, sería muy complicado definir las responsabilidades de cada microservicio y se solaparían unos con otros, pudiendo violar el principio de responsabilidad única.

DESARROLLO DEL PROTOTIPO

1.1. Objetivo del prototipo

En esta sección del presente trabajo se va a desarrollar una aplicación web utilizando Spring, Vaadin y microservicios con objeto de poner en práctica los conocimientos antes detallados.

En concreto se va a desarrollar una aplicación web que ayude en la planificación, gestión y seguimiento de las prácticas en empresas que los alumnos de Ciclos Formativos de Formación Profesional cursan en su último curso. Dichas prácticas están incluidas en todos los títulos oficiales de los ciclos formativos como si fuera una asignatura más que recibe el nombre de Formación en los Centros de Trabajo (FCTs).

El prototipo a desarrollar implementará la funcionalidad básica de registrar la relación de los alumnos que hacen sus prácticas en cada una de las empresas. Para ello será necesario poder hacer las operaciones básicas CRUD de alumnos, empresas y fcts.

1.2. Diseño tecnológico del prototipo

El diseño ejecutado se corresponde con el mostrado en la siguiente figura. En los apartados posteriores se destalla cada uno de sus componentes

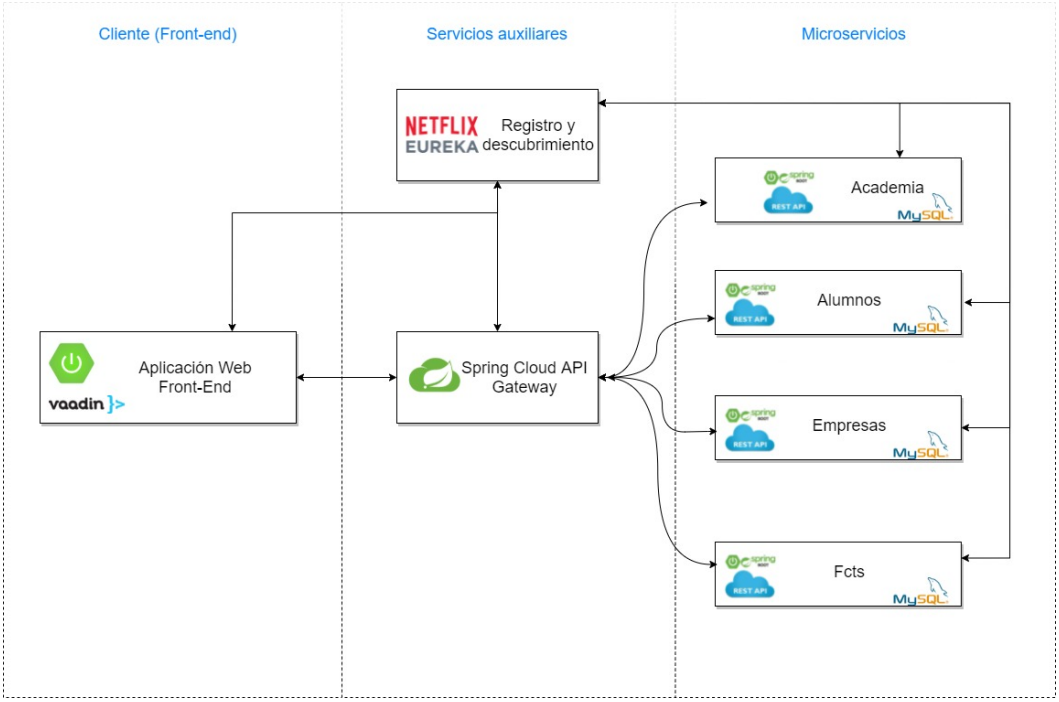


Figura 9: Diseño tecnológico del prototipo

1.2.1. Cliente Front End

1.2.1.1. Descripción del proyecto

Para la parte del cliente Front-End se utilizará Vaadin integrado en un proyecto de Spring. Se creará un proyecto en el IDE IntelliJ con las siguientes dependencias:

- Spring Boot DevTools
- Vaadin
- Spring HATEOAS
- Eureka Client Service
- y, OpenFeign.

En cuanto a las versiones se utilizarán las siguientes versiones:

- Java.version:11

- Spring-cloud.version: 2020.0.3
- Spring boot versión: 2.5.4
- Vaadin: 14.7.0

Este módulo tendrá como principal responsabilidad servir las páginas web que implementar la interfaz de usuario de la aplicación. Además, se comunicará con el resto de microservicios haciendo uso del cliente Feign disponible en el Spring.

Su archivo application.properties quedará configura tal y como se muestra a continuación:

```
spring.application.name=webfct
server.port=${PORT:8091}

eureka.instance.instance-id=${spring.application.name}:${random.value}

# Ensure application is run in Vaadin 14/npm mode
vaadin.compatibilityMode = false
logging.level.org.atmosphere = warn
```

Figura 10: Fichero application.properties del Front-End

El proyecto estará formado por un:

- Una capa de clientes Feign para la conexión con cada uno de los otros microservición a través de los cuales se lanzarán consultas. A modo de similitud esta capa a la capa de repositorios de una aplicación con persistencias en base de datos.
- Una capa de servicio que hará uso los clientes Feign e incluirá la lógica de negocio de la aplicación. Dado que estamos en un proyecto Spring, las clases de esta capa se anotan con @Service para que sean consideras como tal.
- Y, por último, una capa en la que se implementan las vista con Vaadin. Esta capa hace uso de los servicios ofrecidos por la capa de servicios.

1.2.1.2. Vistas de la aplicación.

Las vistas generadas para la aplicación son las siguientes:

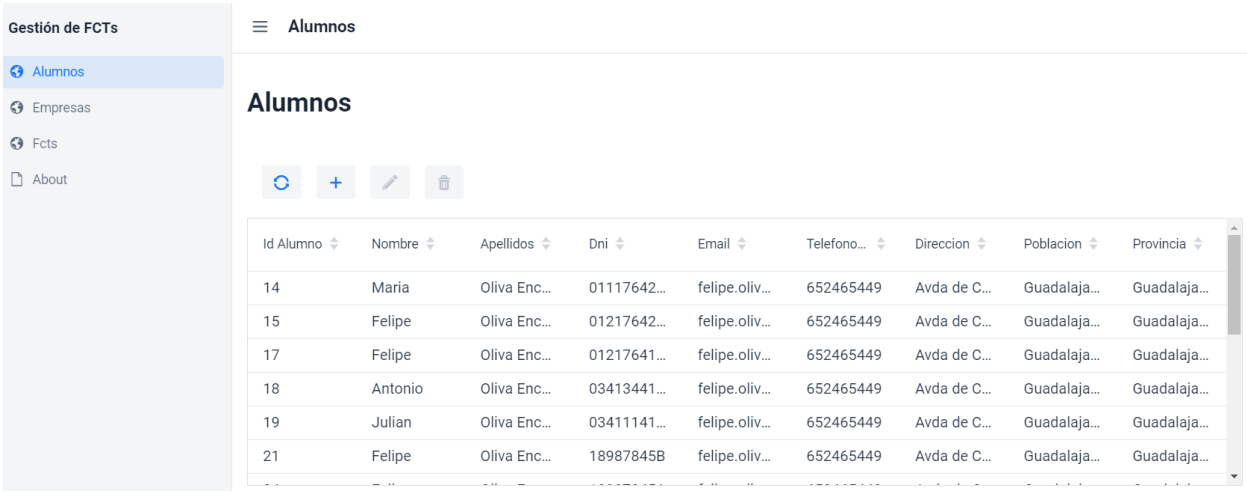


Figura 11: Vista web - Alumnos

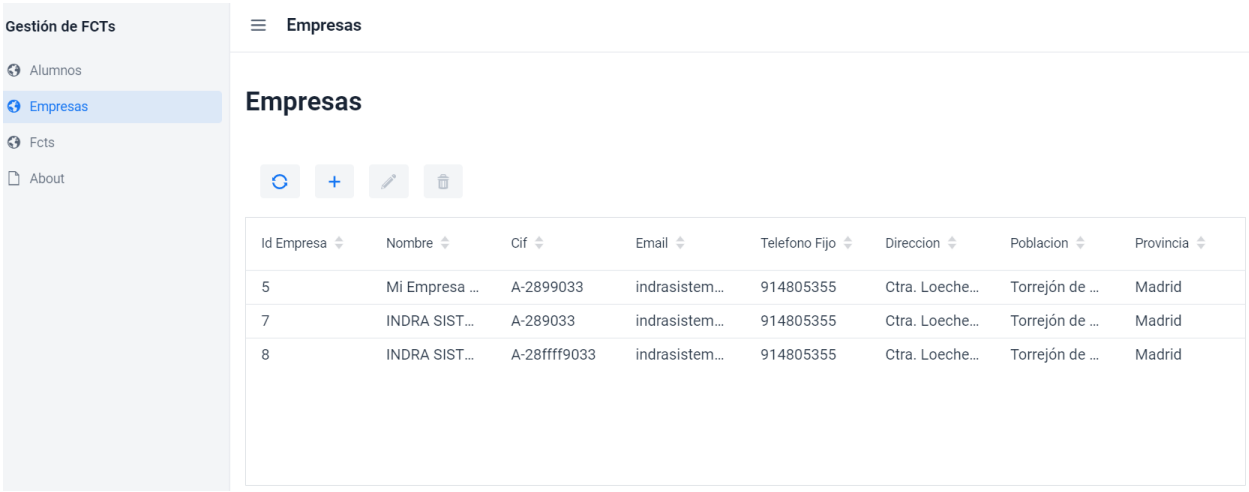


Figura 12: Vista web - Empresas

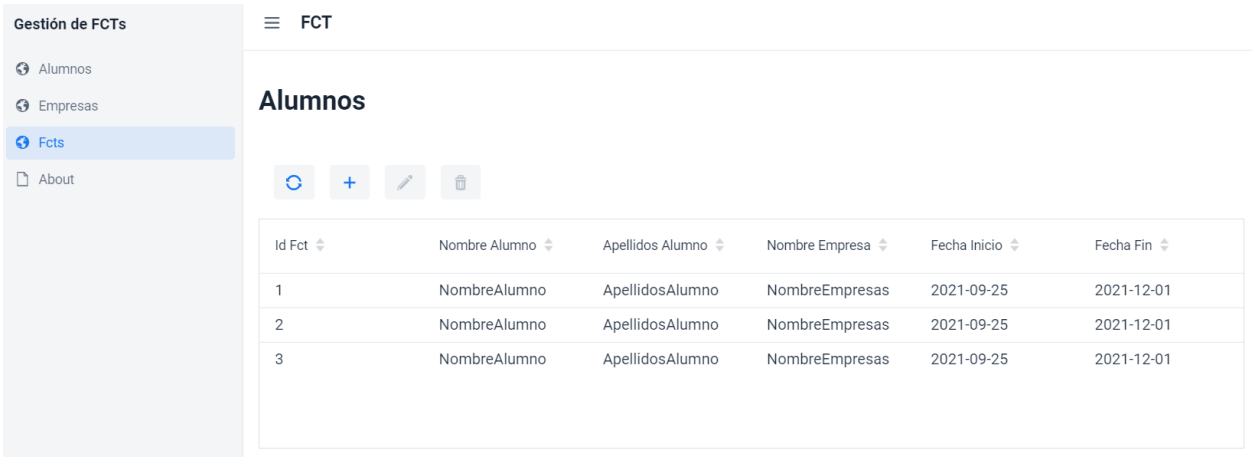


Figura 13: Vista Web - Alumnos

Todas ellas, comparten la vista padre implementada en la Clase MainView. La vista MainView implementa el menú de la aplicación, así como el layout básico del resto de las vistas.

El acceso a la capa de servicio desde las vistas se ha realiza inyectando una entidad de servicio en el constructor de la vista tal y como se muestra en el siguiente código.

```
@PageTitle("Empresas")
@Route(value="/empresas", layout = MainLayout.class)
public class EmpresasView extends VerticalLayout implements CrudListener<EmpresaEntity> {

    EmpresaService empresaService;

    // Instancia de CRUID UI
    private GridCrud<EmpresaEntity> crud = new GridCrud<>(EmpresaEntity.class, new VerticalCrudLayout());

    public EmpresasView(@Autowired EmpresaService empresaService){
```

Figura 14: Inyección de las entidades @Service en las vistas de Vaadin

Todas las vistas anteriores utilizan el mismo componente básico denominado CRUD UI desarrollado por Alejandro Duarte y disponible en el directorio de componentes de Vaadin.

1.2.2. Eureka

Eureka implementa un servidor de registro en el que los microservicios pueden registrarse para que otros puedan localizar su dirección IP y puerto a partir de un nombre descriptivo

Para la implementación de este microservicio se ha construido un proyecto de IntelliJ con la siguiente configuración.

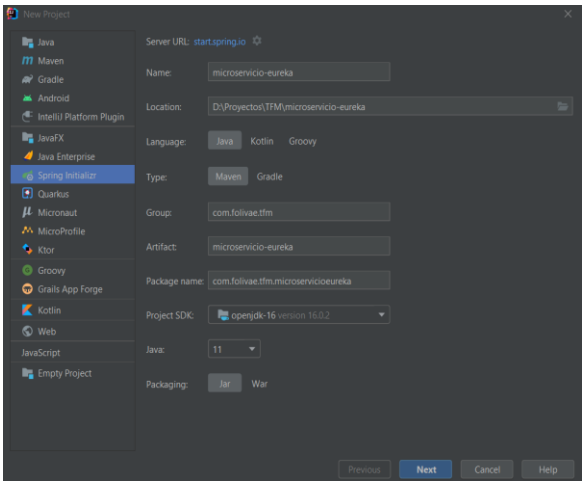


Figura 15: Configuración del proyecto para servidor de registro Eureka

A dicho proyecto, se le ha añadido la dependencia Eureka Server.

Eureka proporciona una interface de usuario para la administración de los microservicios. La siguiente imagen muestra el aspecto de dicha interface.

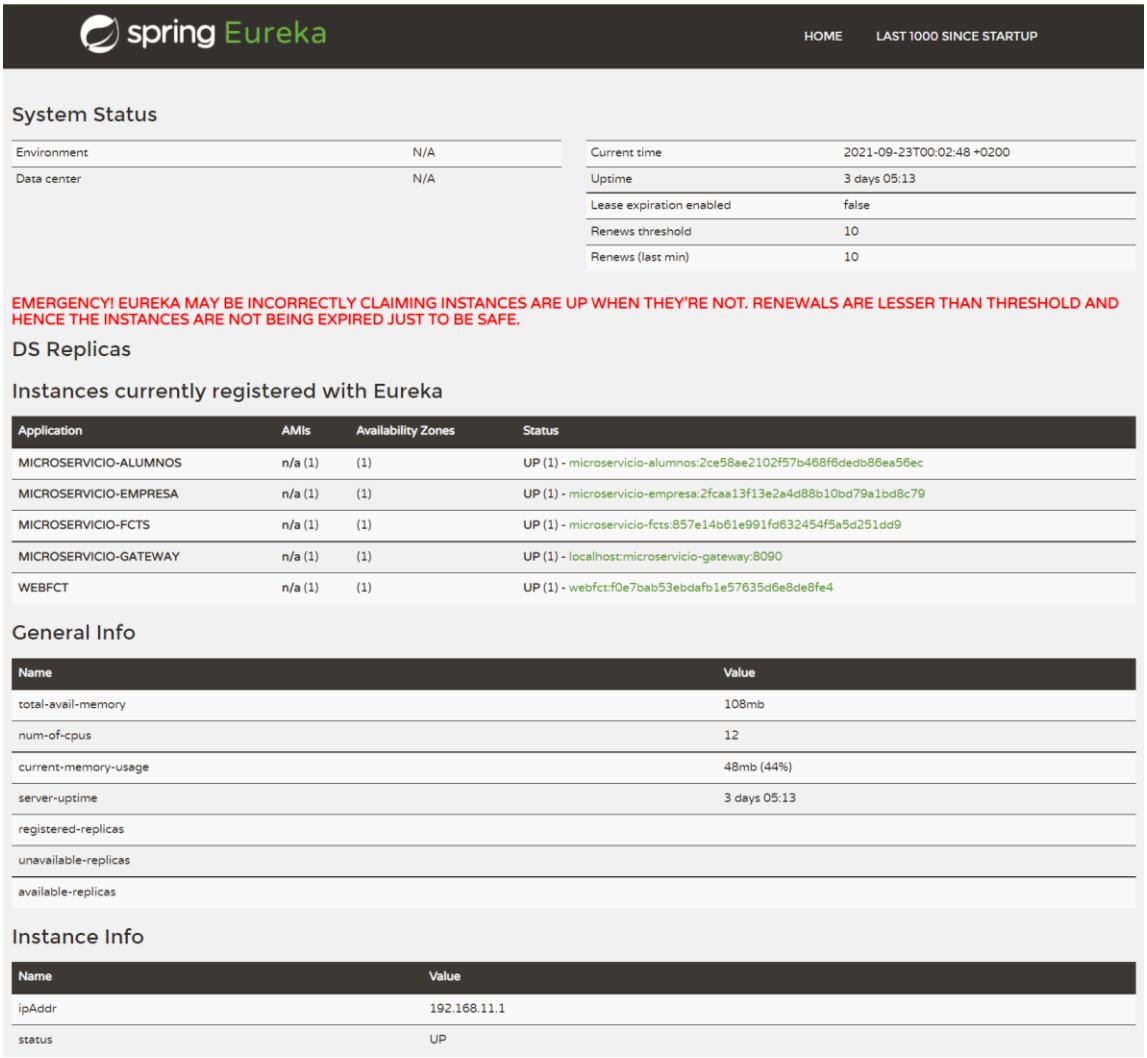


Figura 16: Interface de administración Eureka Server

1.2.3. API Gateway

El API Gateway tiene como misión servir al front-end de punto de entrada a todo el ecosistema de microservicios disponible por detrás. Además, este API Gateway implementa balanceo de carga de tal forma que si un microservicio tiene varias instancias aplicará un algoritmo round robin para distribuir la carga de procesamiento entre ellas.

Para la localización de los microservicios precisa de un servidor de registro como Eureka.

Para su implementación se construye un proyecto de Spring con las siguientes características

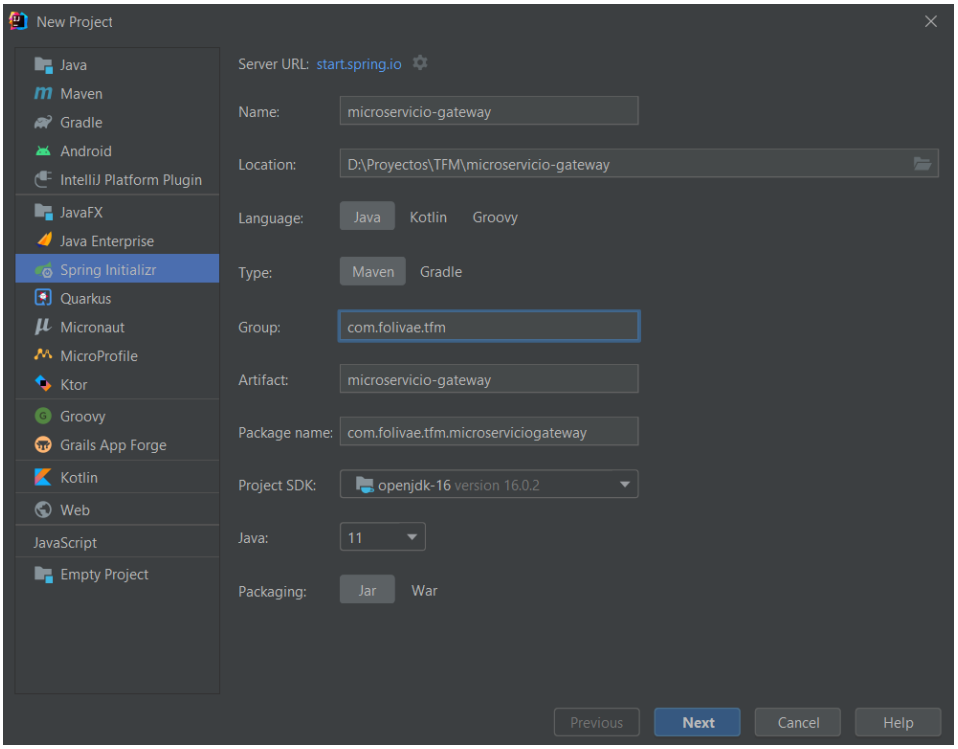


Figura 17: Configuración del proyecto para servidor de registro Eureka

Agregamos las siguientes dependencias:

- Spring Boot DevTools
- Eureka Discover Client
- Cloud LoadBalancer
- Gateway

Esta es la configuración del fichero *application.properties* con el que configuramos las rutas a cada uno de los microservicio desplegados. Por cada ruta es necesario definir cuadro parámetros:

- **Id**, que es un identificador único de la ruta
- **URI**, que hace referencia al nombre del microservicio con el que esa ruta conecta. La dirección IP y el puerto del microservicio provisto por Eureka, este servicio es a su vez cliente de Eureka. Las siglas del *lb* hace referencia a Load Balancer de cada uno de los microservicios.
- **Predicates**, en que definimos el Path y que incluye la url a partir de la cual accedemos a esta ruta en gateway. Los símbolos ** al final, definen que, a partir de ahí, se debe construir la consulta al microservicio.
- **Filters**, en el que simplemente especificamos cuándo nombres de la URL debe eliminar el Gateway para acceder al microservicio.

Por último, en la última línea del fichero deshabilitamos ribbon para que sea Load Balancer de Spring el que se encargue de balancear la carga entre distintos microservicios cuando haya varias instancias.

MicroserviciosGatewayApplication.java | *application.properties

```
spring.application.name=microservicio-gateway
server.port=8090

eureka.client.service-url.defaultZone=http://localhost:8761/eureka

spring.cloud.gateway.routes[0].id=microservicio-usuarios
spring.cloud.gateway.routes[0].uri=lb://microservicio-usuarios
spring.cloud.gateway.routes[0].predicates=Path=/api/alumnos/**
spring.cloud.gateway.routes[0].filters=StripPrefix=2

spring.cloud.gateway.routes[1].id=microservicio-cursos
spring.cloud.gateway.routes[1].uri=lb://microservicio-cursos
spring.cloud.gateway.routes[1].predicates=Path=/api/cursos/**
spring.cloud.gateway.routes[1].filters=StripPrefix=2

spring.cloud.gateway.routes[2].id=microservicio-examenes
spring.cloud.gateway.routes[2].uri=lb://microservicio-examenes
spring.cloud.gateway.routes[2].predicates=Path=/api/examenes/**
spring.cloud.gateway.routes[2].filters=StripPrefix=2

spring.cloud.gateway.routes[3].id=microservicio-respuestas
spring.cloud.gateway.routes[3].uri=lb://microservicio-respuestas
spring.cloud.gateway.routes[3].predicates=Path=/api/respuestas/**
spring.cloud.gateway.routes[3].filters=StripPrefix=2

spring.cloud.loadbalancer.ribbon.enabled=false
```

Figura 18: Configuración del archivo application.properties del servicio API Gateway

1.2.4. Microservicios

Todos los microservicios desarrollados comparten el mismo proyecto base, que se ha configurado en Spring Initializr como se muestra en la siguiente ilustración.

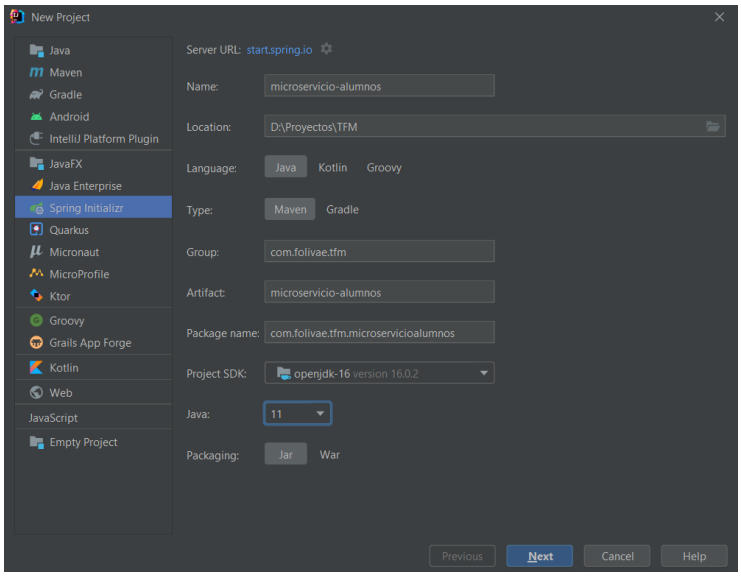


Figura 19: Configuración del proyecto para microservicio Alumnos

Adicionalmente se le han añadido las siguientes dependencias:

- Spring Boot Services
- Spring Web

- OpenFeign
- Cloud LoadBalancer
- Eureka Discovery Center
- Spring Data JPA
- MySQL Driver.

Para que cada uno de los microservicios pueda registrarse en el servidor Eureka y encontrar otros microservicios, todos, deben de tener la anotación @EnableDiscoveryClient o @EnableEurekaClient en su clase principal. Además, se debe incluir una dependencia de spring-cloud-starter-eureka en el archivo pom.xml .

En cuanto a la configuración del proyecto se ha implementado el archivo application.properties de la siguiente forma:

```
# Puerto y nombre
spring.application.name=microservicio-empresa
#server.port=8001
#Asigno un puerto aleatorio para que si creo varias instancias no haya conflicto
server.port=${PORT:0}
eureka.instance.instance-id=${spring.application.name}:${random.value}


#DATASOURCE
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/empresas?useSSL=false&serverTimezone=Europe/Madrid&allowPublicKeyRetrieval=true
spring.datasource.username=root
spring.datasource.password=admin

#JPA
spring.jpa.generate-ddl=false
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect
spring.jpa.show-sql=true
logging.level.org.hibernate.SQL=debug


# Table names physically
spring.jpa.hibernate.naming.physical-strategy=org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl

# Deshabiito el ribbon como balanceador de carga para forzar el uso de Spring Cloud Load Balancer
spring.cloud.loadbalancer.ribbon.enabled=false

config.balanceador.test=${BALANCEADOR: por defecto}
```

Figura 20: Application.properties del servicio Alumnos

De esta configuración, es destacable el escenario en el que varias instancias del mismo microservicio estén en ejecución al mismo tipo. En tal caso, el nombre de la aplicación será común a ambas (spring.application.name), pero su puerto y su identificación en Eureka deben de ser distintos entre sí. Para cumplir esto se han aleatorizado los parámetros: *server.port* y *eureka.instanace.instence-id*

En la última línea del fichero, se accede a una variable de entorno del proyecto que se configurará con un valor distinto para cada instancia. Dicha variable será accesible desde el código Java de la aplicación haciendo uso de anotación @Value de la siguiente forma.

```
@Value("${BALANCEADOR: por defecto}")
private String balanceadorTest;
```

Con ello podremos diferenciar qué instancia está atendiendo una determinada petición ya que podremos incluir en las respuestas de los API REST la variable balanceadorTest.

1.2.4.1. Microservicio Alumnos

Este microservicio tiene como principal responsabilidad la gestión de toda la información relacionada con los datos personales de los alumnos.

En su capa de persistencia accederá a una base de datos MySQL con la siguiente estructura.

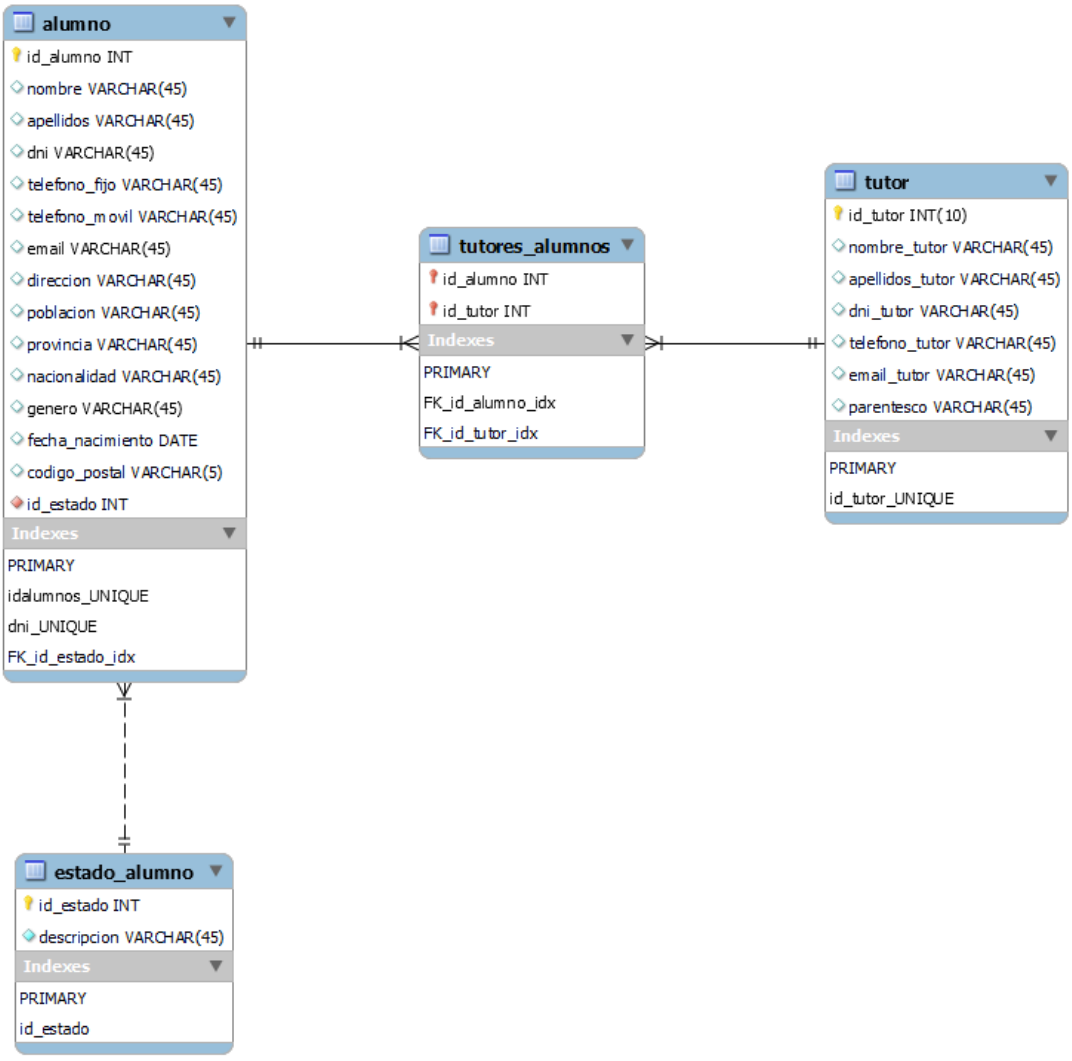


Figura 21: Diagrama de la base de datos del microservicio Alumnos

1.2.4.2. Microservicio Academia

Este microservicio tiene como principal responsabilidad la gestión de toda la información relacionada con la gestión académica de la organización.

En su capa de persistencia accederá a una base de datos MySQL con la siguiente estructura.

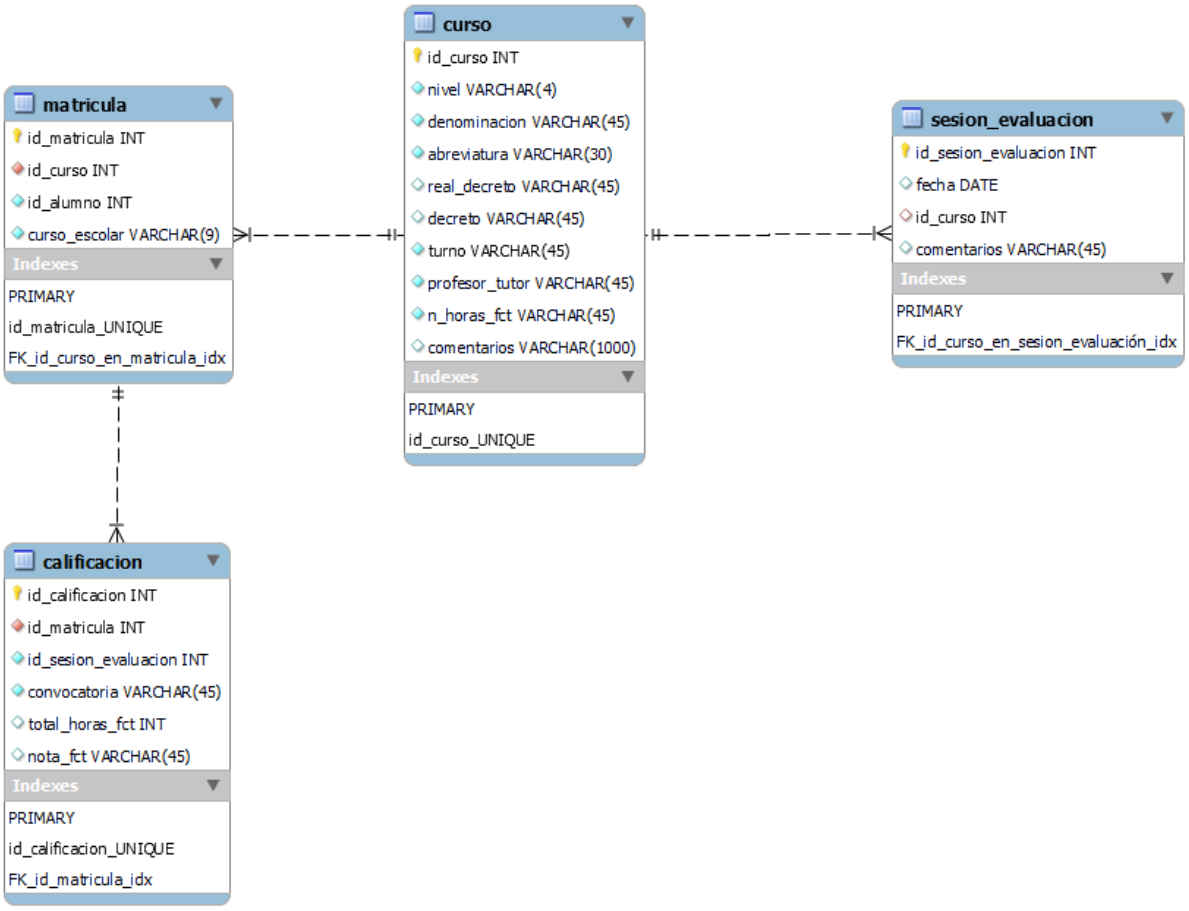


Figura 22: Diagrama de la base de datos del microservicio Academia

1.2.4.3. Empresas

Este microservicio tiene como principal responsabilidad la gestión de toda la información relacionada con datos de las empresas que colaboran en las prácticas de los alumnos.

En su capa de persistencia accederá a una base de datos MySQL con la siguiente estructura.

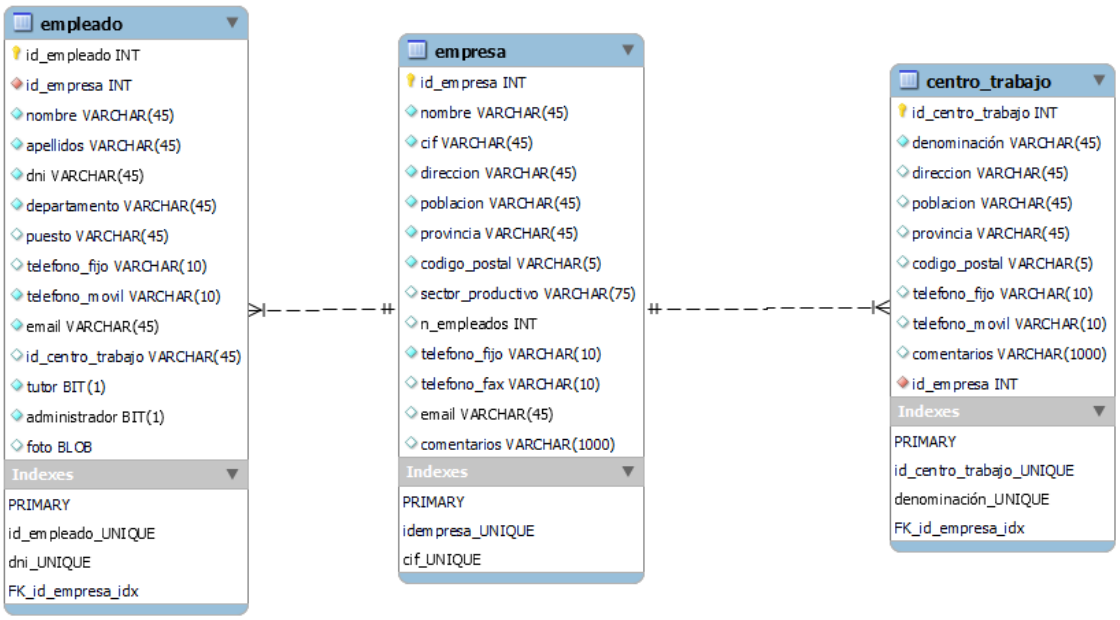


Figura 23: Diagrama de la base de dato del microservicio Empresas

1.2.4.4. Fcts

Este microservicio tiene como principal responsabilidad la gestión de la información relacionada con las prácticas de los alumnos en las empresas.

En su capa de persistencia accederá a una base de datos MySQL con la siguiente estructura.

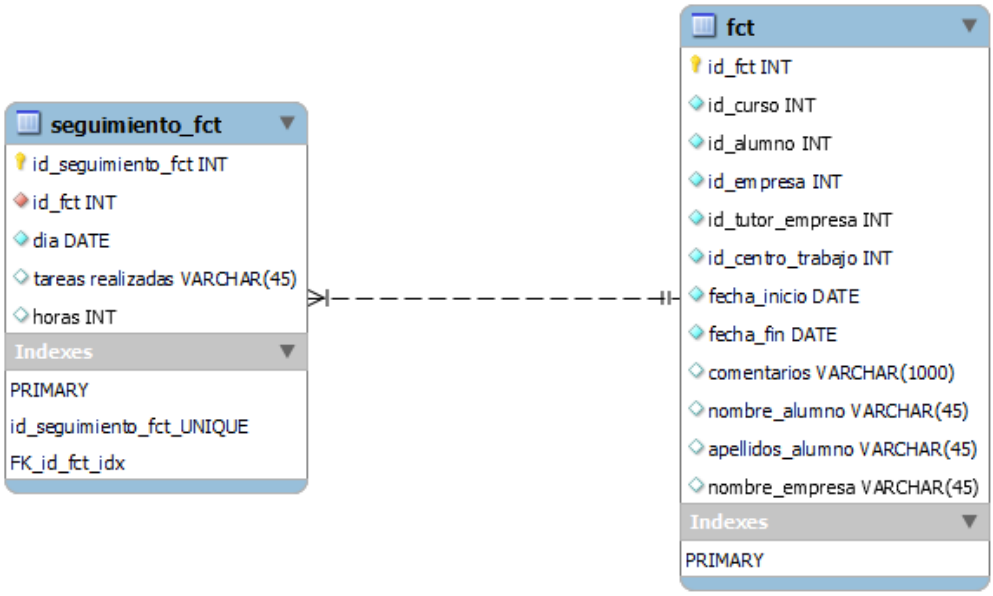


Figura 24: Diagrama de la base de datos del microservicio Fcts

RESUMEN Y CONCLUSIÓN

En comparación con las tecnologías Java equivalentes en funcionalidad estudiadas en las asignaturas del Máster, tanto Spring como Vaadin ofrecen multitud de ventajas que sin duda justifican su uso mayoritario en el ámbito productivo actualmente. De todas esas ventajas, considero especialmente reseñable la rapidez con la que es posible desarrollar una aplicación web manteniendo todos los grados de libertad que la programación en Java ofrece.

En cuanto a los microservicios, a la hora de lograr una primera aproximación, su desarrollo ha resultado ser fácil y metódico. Sin embargo, como en el caso de la aplicación implementada cada microservicio hacía uso de una base de datos distinta, no ha sido posible implementar el aspecto relacional de los datos a ese nivel. Dicha responsabilidad recae ahora en la lógica de cada uno de los microservicios. Este es un aspecto crítico de la arquitectura de microservicios que sin duda debe ser tenido en cuenta en la planificación de futuras aplicaciones.

1.3. Presupuesto

En esta sección se incluye un presupuesto estimado para el desarrollo e implementación de la aplicación prototipo definida en este trabajo.

Tarea	Horas	PVP Hora	Total
Desarrollo de microservicio REST Alumnos	15	50,00 €	750,00 €
Desarrollo de microservicio REST Academia	15	50,00 €	750,00 €
Desarrollo de microservicio REST Fcts	15	50,00 €	750,00 €
Desarrollo de microservicio REST Empresas	15	50,00 €	750,00 €
Desarrollo del front-end	25	50,00 €	1.250,00 €
Despliegue de la aplicación en la nube	10	50,00 €	500,00 €
Pruebas y validación final	10	50,00 €	500,00 €
		Subtotal	5.250,00 €
		IVA (21%)	1.102,50 €
		TOTAL	**6.352,50 €**

1.4. Conclusiones y Futuras Líneas de Trabajo

El aprendizaje de un nuevo lenguaje o Framework siempre es costoso tanto en tiempo como es esfuerzo, pero también es cierto que es gratificante y productivo cuando se alcanza la cima de la curva. Tanto en el caso de Spring como en el caso de Vaadin, considero que la inversión ha sido productiva y confío me sea útil en mi desarrollo profesional futuro.

Sin duda para el futuro será necesario continuar aprendiendo sobre los otros muchos módulos de Spring que en este trabajo no se han abordado.

BIBLIOGRAFÍA

- Prasad Reddy S. (2017), Beginning Spring Boot 2: Applications and Microservices with the Spring Framework, Apress, California.
- Cosmina I. (2020), Pivotal Certified Professional Core Spring 5 Developer Exam: A Study Guide Using Spring Framework 5
- Gutierrez F. (2019), Pro Spring Boot 2: An Authoritative Guide to Building Microservices, Web and Enterprise Applications, and Best Practices.
- Marten Deinum, Daniel Rubio, Josh Long (2017), Spring 5 Recipes: A Problem-Solution Approach, Apress, California.
- Sam Newman (2015), Building Microservices: Designing Fine-Grained Systems, O'Reilly Media, Sebastopol, California.
- Alexandru Jecan (2017), Java 9 Modularity Revealed: Project Jigsaw and Scalable Java Applications, Apress, California.
- Relan Kunal (2019), Building REST APIs with Flask: Create Python Web Services with MySQL, Apress, California.
- Laddad R. (2003), AspectJ in Action: Practical Aspect-oriented Programming, Manning, New York.
- Evans E. (2003), Domain-Driven Design: Tackling Complexity in the Heart of Software, Addison Wesley, Boston.
- Deepak Alur et. Al, (2001), Core J2EE Patterns: Best Practices and Design Strategies, Prentice Hall, Nueva Jersey.
- Wolff E. (2006), Microservices: Flexible Software Architecture, Addison Wesley, Boston.
- Meyer B. (1998), Object-Oriented Software Construction, Prentice-Hall, New Jersey.
- Spring (2021), Spring, (https://spring.io/) (Consultado continuamente durante el desarrollo de este proyecto).
- Vaadin (2021), Vaadin (https://vaadin.com//) (Consultado continuamente durante el desarrollo de este proyecto).
- Santiago Ramírez Pérez (2020), Estudio del Framework Spring, Spring Boot y Microservicios (TFM)

Universidad de Alcalá
Escuela Politécnica Superior

ESCUELA POLITECNICA
SUPERIOR