Spring

Spring framework

- Spring es un framework de código abierto que facilita la creación de aplicaciones Java, Kotlin y Groovy
- Estructura modular
- Flexibilidad para implementar distintos tipos de arquitecturas.

¿Qué es un framework?

- Es un “entorno de trabajo” compuesto por “reglas” y “herramientas” que facilitan el desarrollo de aplicaciones.

Historia

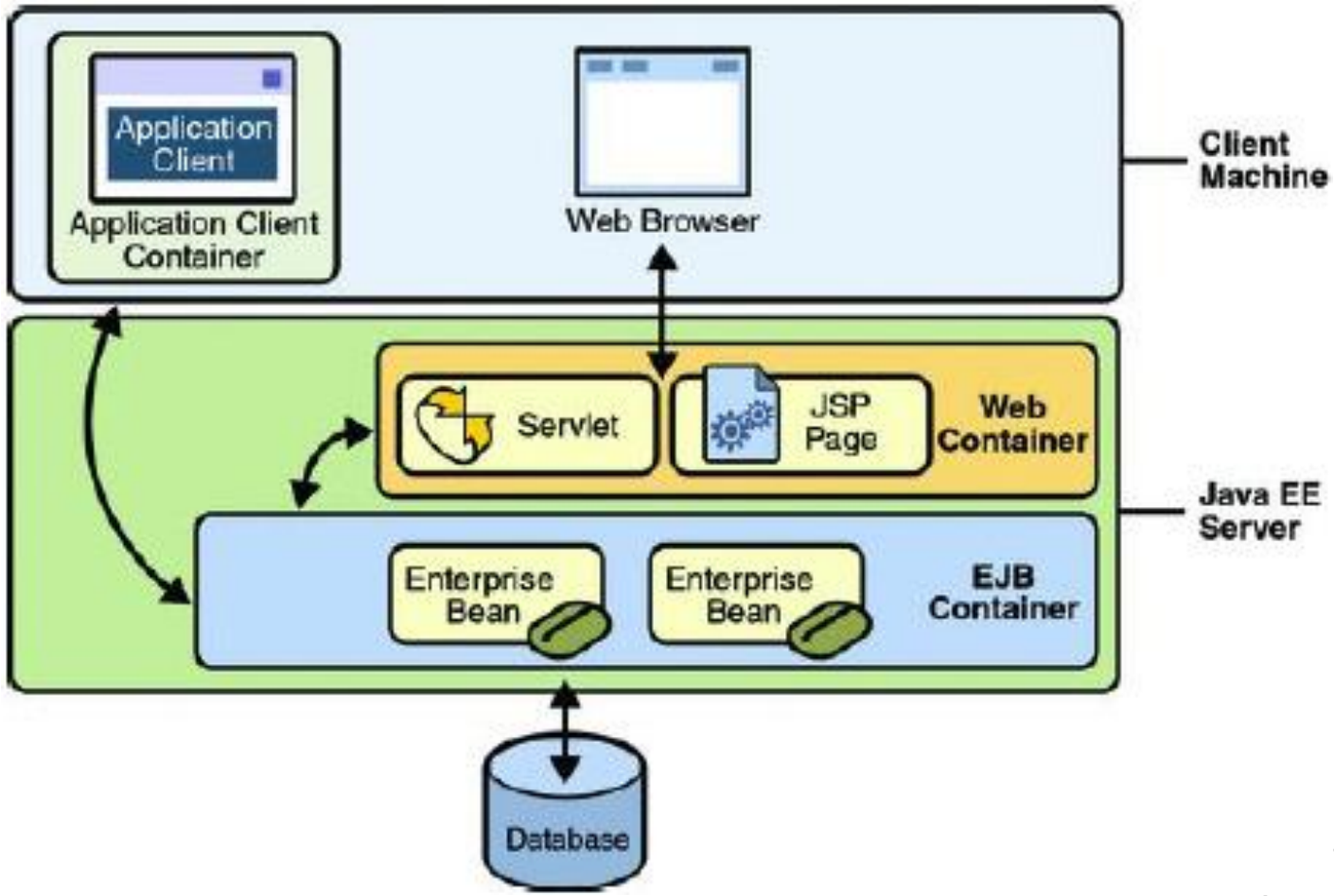
- Rod Johnson
- Expert One on One J2EE Design and Development

- 1ª versión: Marzo 2004

- Spring alternativa al desarrollo de aplicaciones JavaEE

Historia

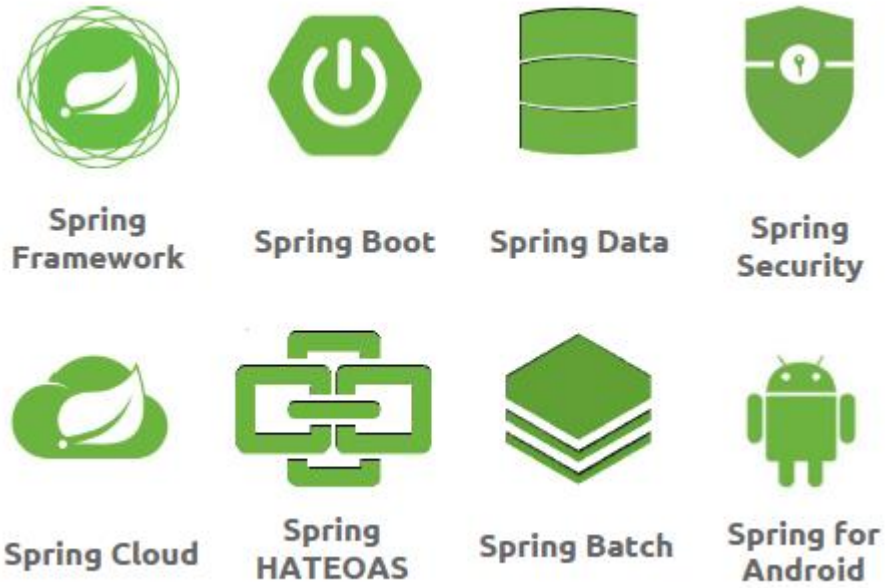
- Respuesta al modelo EJB



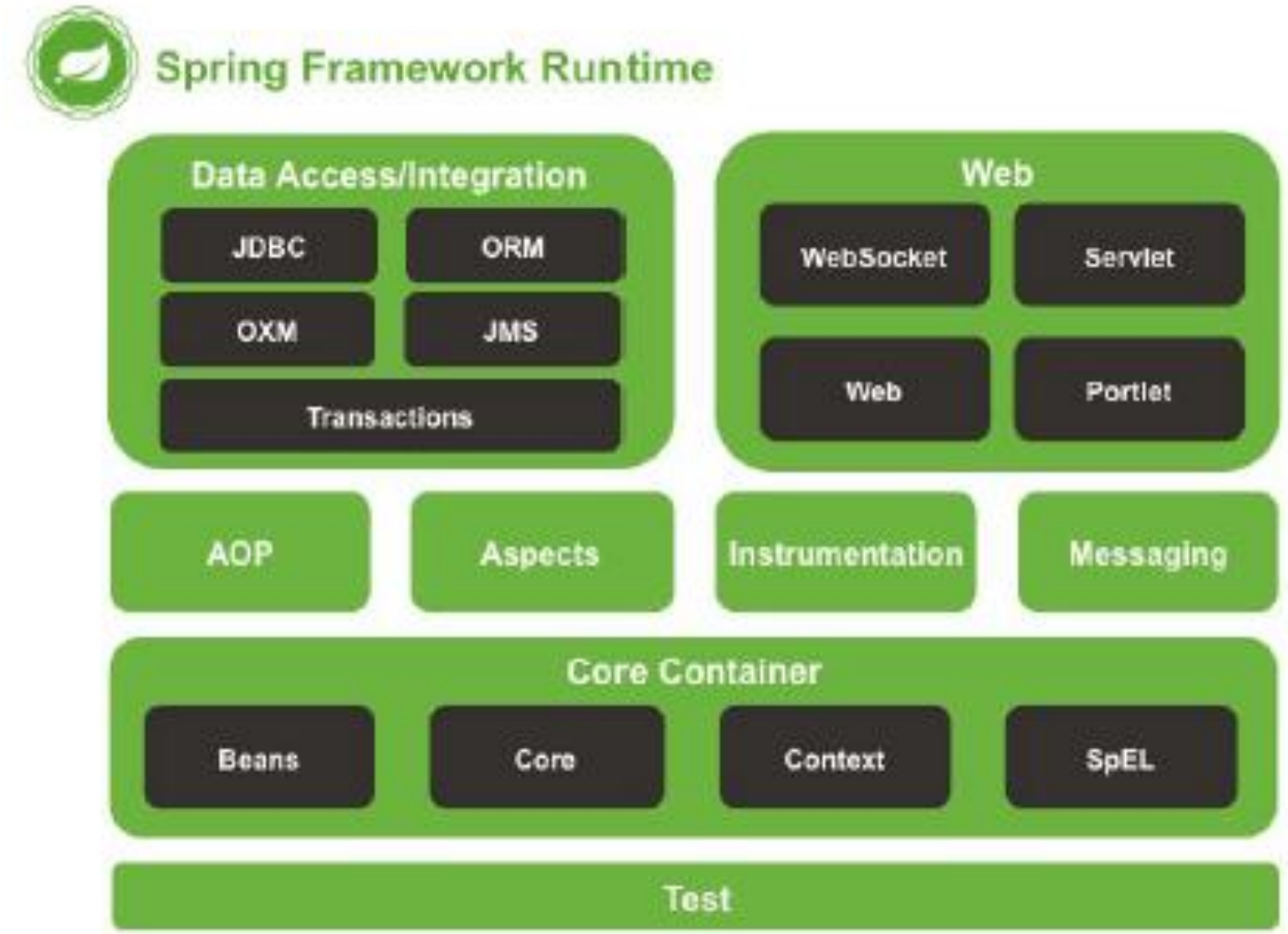
Ventajas de Spring

- Flexibilidad (Integración con otras herramientas)
- Inyección de dependencias (favorece “loose coupling”, desacoplamiento)
- Desarrollo sencillo con POJOS (Plan Old Java Objects)
- Minimiza el boilerplate code (código repetitivo)
- Simplifica el acceso a datos
- Programación orientada a aspectos (AOP)

Proyectos Spring



Módulos Spring



Módulos de Spring

- **Core container**: proporciona inyección de dependencias e inversión de control.
- **Web**: permite crear controladores Web, tanto vistas MVC como aplicaciones REST.
- **Acceso a datos**: abstracciones sobre JDBC, ORMs como Hibernate, sistemas OXM (Object XML Mappers), JSM y transacciones.
- **Programación orientada a Aspectos (AOP)**: ofrece el soporte para aspectos.
- **Instrumentación**: soporte para la instrumentación de clases.
- **Pruebas de código**: soporte para Junit y TestNG

Módulos de Spring

- Es una separación lógica que facilita la granularidad y mantiene los componentes desacoplados.

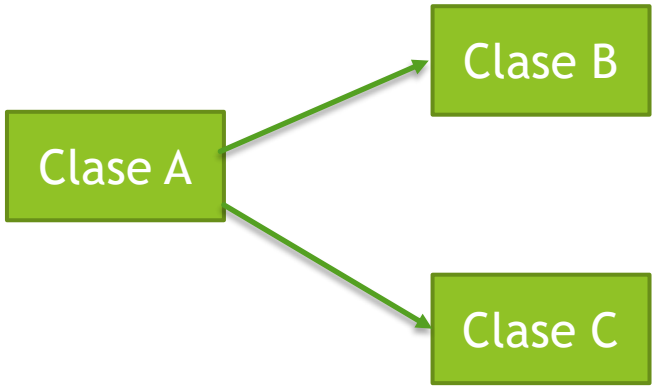
- El uso de inyección de dependencias facilita la programación contra interfaz, permitiendo a los diferentes componentes depender únicamente de interfaces y produciendo, por tanto, un código menos acoplado.

Principios clave en Spring

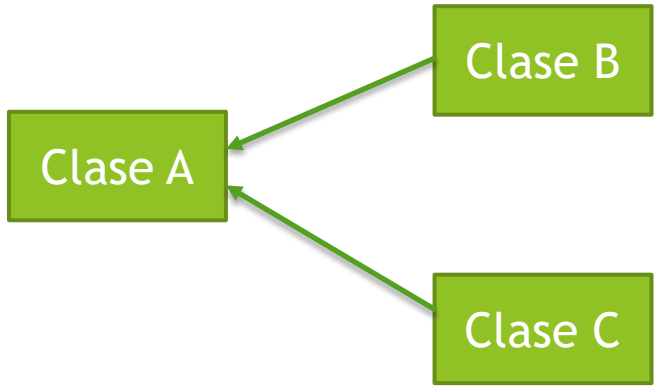
- Inyección de dependencias (DI)
- Inversión de control (IoC)

Inyección de dependencias

- Mecanismo mediante el que un objeto recibe sus dependencias ya instanciadas en vez de que sea el objeto el que tenga la responsabilidad de iniciarlas.



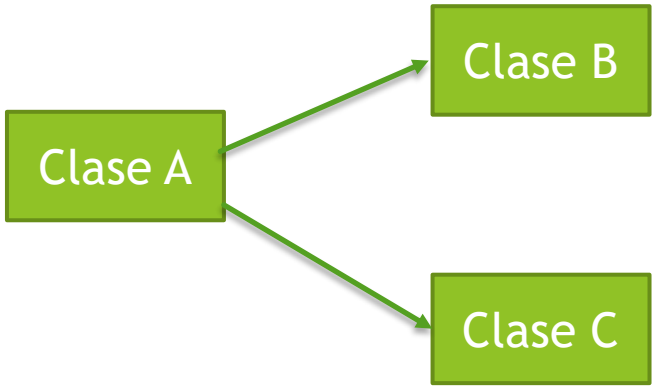
- **Modelo tradicional**



Inversión de control

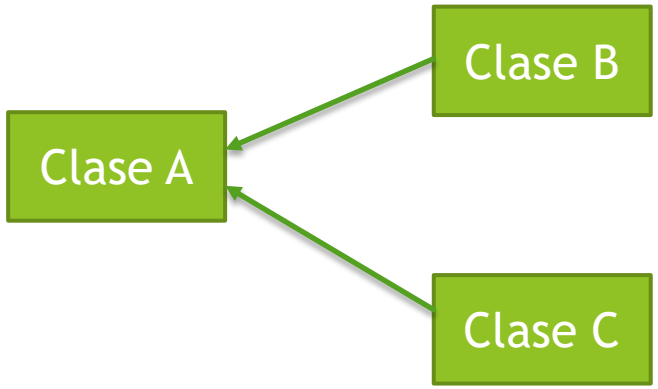
Inyección de dependencias

- **Modelo tradicional**
- Si un objeto A necesita a otros puede invocar la construcción de los mismos invocando los constructores de las otras clases



Inyección de dependencias

- Inversión de control por inyección de dependencias.
- La clase A se despreocupa de la creación de los objetos de la clase B o de la clase C, simplemente especifica que requiere un objeto de la clase B o C



Inversión de control

- Principio de diseño mediante el cual el control de un programa o una parte de él se transfiere a un framework.

- Inversión de control (IoC)

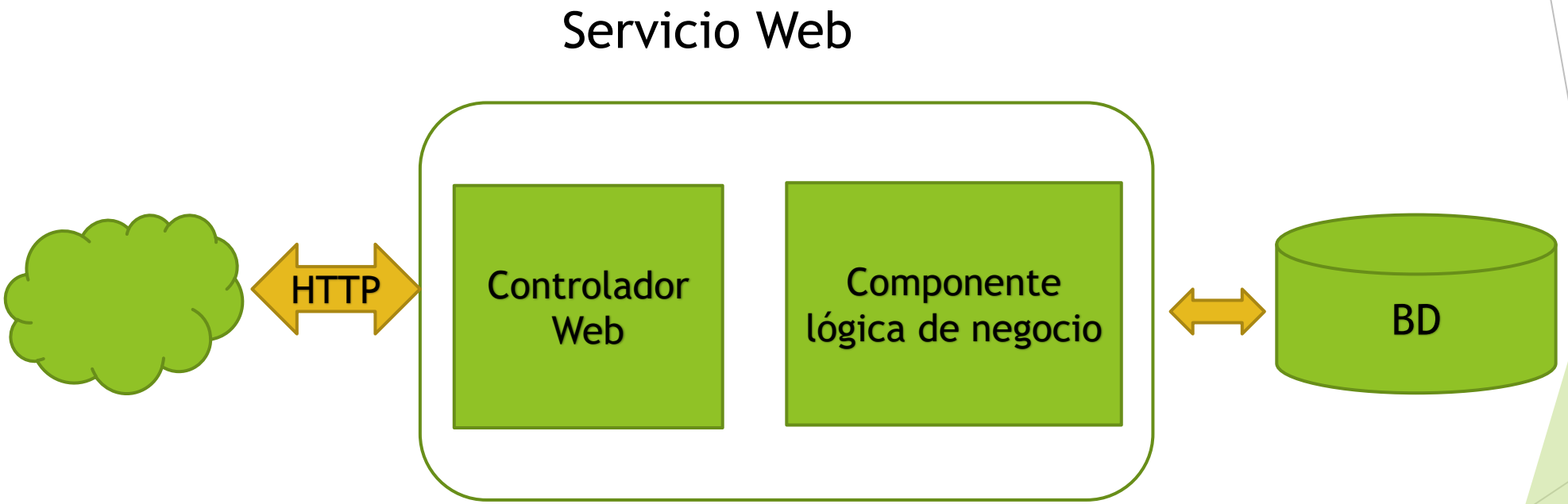
Inversión de control

- Inversión de control es un concepto junto a unas técnicas de programación:
- – en las que el flujo de ejecución de un programa se invierte respecto a los métodos de programación tradicionales,
- – en los que la interacción se expresa de forma imperativa haciendo llamadas a procedimientos o funciones.

Inversión de control

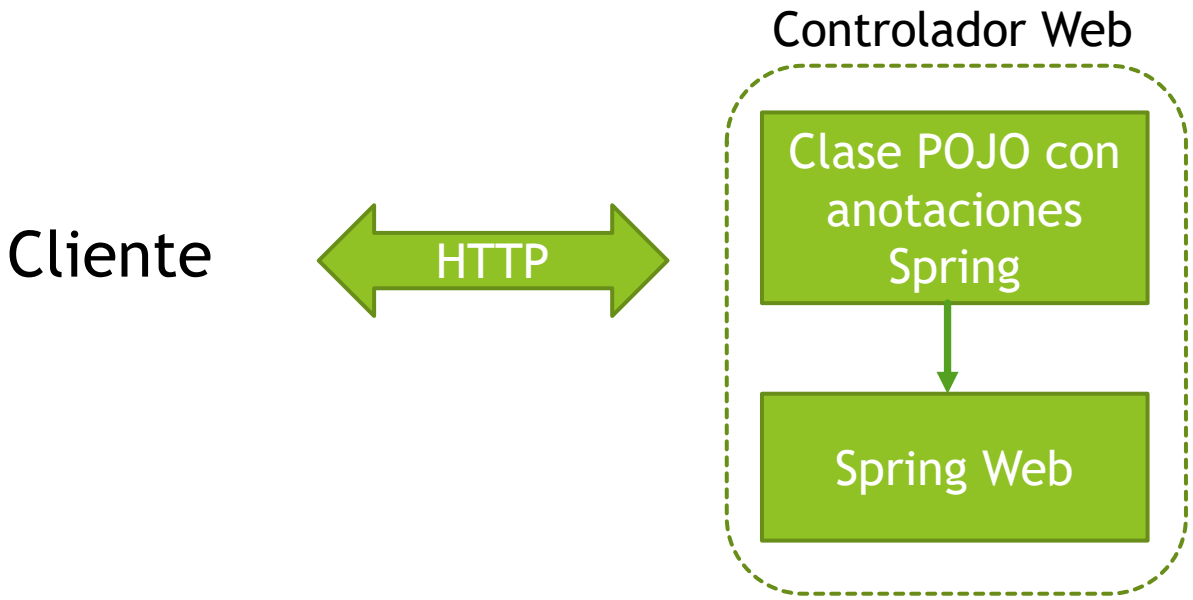
- Tradicionalmente el programador especifica la secuencia de decisiones y procedimientos que pueden darse durante el ciclo de vida de un programa mediante llamadas a funciones.
- En su lugar, en la inversión de control se especifican respuestas deseadas a sucesos o solicitudes de datos concretas, dejando que algún tipo de entidad o arquitectura externa lleve a cabo las acciones de control que se requieran en el orden necesario y para el conjunto de sucesos que tengan que ocurrir.

Estructura de un servicio Web



Servicios REST con Spring

- El módulo Web proporciona el soporte necesario para la creación de servicios REST con Spring.
- Incluye anotaciones específicas de Spring.



Spring Boot

Spring Boot

INTRODUCCIÓN A SPRING BOOT

- Módulo de la plataforma Spring cuyo objetivo es simplificar la creación de aplicaciones y servicios listos para ejecutarse.

- **Objetivos:**
- Ofrecer una forma sencilla de arrancar proyectos spring.
- Disponer de funcionalidad en función de la naturaleza del proyecto (web, jpa, nosql, etc..)

INTRODUCCIÓN A SPRING BOOT

- Simplifica la configuración de maven/gradle (Starter)
- Configurar automáticamente Spring siempre que sea posible (**Convención sobre configuración**)
- Simplifica el proceso de configuración de aplicaciones y gestión de dependencias
- Permite integrar entorno de ejecución en la aplicación.
- Aplicaciones java estándar (.jar)
- Ideal para creación de microservicios

Starters

- La inclusión de dependencias en una aplicación Spring boot se simplifica mediante los **starters**.
- Un starter incluye un conjunto de dependencias básicas para desarrollar un tipo de aplicación.
- Ejemplo:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Configuración de aplicaciones

- Se eliminan los archivos de configuración .xml
- Se asumen una serie de configuraciones por defecto.
- Para indicar parámetros de configuración específicos
  - **application.properties**
  - **application.yml**

La clase main

```
@SpringBootApplication(scanBasePackages=" ")
public class Application {

    public static void main(String[] args) {
    SpringApplication.run(Application.class, args);
}

}
```

Estructura controlador REST

- Clase POJO con anotaciones spring MVC

```
@RestController
public class ClaseServicio{
    @GetMapping(produces=…)
    public tipo método1(…) {….}

      @PostMapping(consumes=…)
    public tipo método2(…) {….}
        …
}
```

Controlador Rest

- Clase pojo que define los métodos

```
@RestController
public class SaludoController {

    @GetMapping(value="saludo", produces=MediaType.TEXT_PLAIN_VALUE)
    public String saludar() {
    return "Hola mundo";
    }
}
```

Parámetros y variables Variables en URL

- http://servidor:8080/app/path/var1/var2
- http://servidor:8080/saludo/pepito/44
- Datos que se envían como parte de la URL
- **Recogida de variables**
- Las variables de la URL se deben mapear a los parámetros del método que procesa la petición mediante @PathVariable

```
@RestController
public class TestService{
   @GetMapping(value="saludo/{x}/{y}", produces=MediaType.TEXT_PLAIN_VALUE)
   public String saludar(@PathVariable ("x")  String a, @PathVariable("y")  int b){
            :
   }
}
```

Parámetros en QueryString

- http://servidor:8080/app/path?x=var1&y=var2
- Los parámetros se envían en parejas nombre=valor, separados de la dirección por ?
- **Recogida de variables**
- Se mapean a parámetros del método mediante **@RequestParam**

```
@RestController
public class TestService{
   @GetMapping(value="saludo", produces(MediaType.TEXT_PLAIN_VALUE)
   public String saludar(@RequestParam("x")  String a,@ RequestParam("y")  int b){
            :
   }
}
```

Principales anotaciones REST

- @RestController: indica que la clase es un controlador
- @GetMapping, @PostMapping,@PutMapping y @DeleteMapping: asocian a los métodos del servicio un determinado método HTTP. A través de su atributo **value**, se indica también la url a la que se asociará el método.
- @PathVariable: asocia una variable de la URL a un parámetro de método.
- @RequestBody: asocia el contenido del cuerpo de la petición a un parámetro objeto, dentro del método de respuesta.

Generación de respuestas

- Transformación a JSON a través de la librería Jackson
- Spring se encargará de hacer el mapeo de Java a Json y viceversa

```
public class Libro{
        private String titulo;
        private String autor;
        public Libro() {}
        public String getTitulo() {
                return titulo;
        }
        public void setTitulo(String titulo) {
                this.titulo = titulo;
        }
}
```

{“titulo”:”Spring”,
“autor”:”J.Gil”}

Tipo de devolución del recurso

- Mediante el atributo **produces** indicamos el formato al que tiene que transformar el objeto en la respuesta.

```
public class TestService{
   @GetMapping( value="path" ,  produces=MediaType.APPLICATION_JSON_VALUE)
   public Libro datosLibro(){
                :
   }
}
```

DELETE

```
@DeleteMapping(value=..  produces=..)
public tipo metodoDelete(){
        //operación de eliminación
}
```

POST

```
@PostMapping( value=..  consumes=MediaType.APPLICATION_JSON_VALUE)
public void metodoPost (@RequestBody Persona p){
        //operación de inserción del objeto Persona
}
```

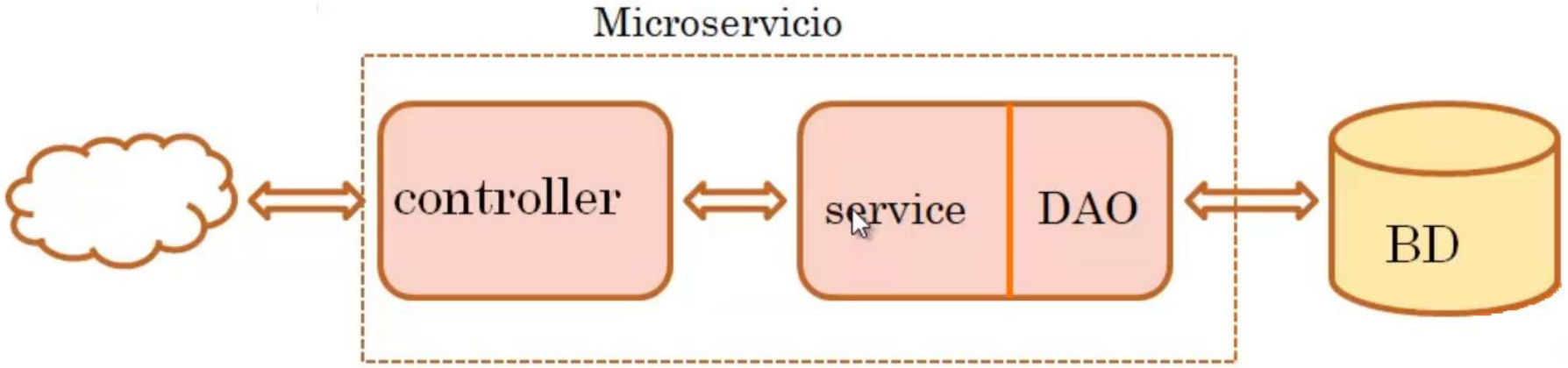
El tipo de devolución puede ser void, o de un tipo específico, en cuyo caso se indicará en ***produces*** el MediaType correspondiente

PUT

```
@PutMapping( value=..  consumes=MediaType.APPLICATION_JSON_VALUE)
public void metodoPut(@RequestBody Persona p){
        //operación de actualización del objeto Persona
}
```

Acceso a datos

- El acceso a los datos se encapsula en una capa independiente (capa DAO)



Acceso a datos

- El acceso a los datos se encapsula en una capa independiente (capa DAO)

- Tecnologías:
- JPA/Hibernate
- Spring Data JPA

- **Starters**:
- Spring Data JPA
- MySql Driver
- Spring Web

Configuración de acceso a datos

- En **application.properties** se definen las propiedades de conexión a la base de datos

```
spring.datasource.driver-class-name =com.mysql.cj.dbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/libreria
spring.datasource.username=root
spring.datasource.password=root
```

Ejemplo application.properties

```
#propiedades para que hibernate cree adecuadamente las instrucciones SQL
spring.jpa.hibernate.naming.implicit-
strategy=org.hibernate.boot.model.naming.ImplicitNamingStrategyLegacyJpaImpl
spring.jpa.hibernate.naming.physical-
strategy=org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl


spring.jpa.show-sql=true



#spring.datasource.url=jdbc:mysql://localhost:3306/test?serverTimezone=UTC
```

Acceso a datos

- En **application.properties** se definen las propiedades de conexión

- spring.jpa.hibernate.ddl-auto=update
- spring.datasource.url=jdbc:mysql:3306/db_example
- spring.datasource.username=usuario
- spring.datasource.password=password
- spring.datasource.driver-class-name =com.mysql.jdbc.Driver
- spring.jpa.show-sql= true

Spring Data

- Spring Data es un proyecto de SpringSource
- Propósito es unificar y facilitar el acceso a distintos tipos de tecnologías de persistencia, tanto a bases de datos relacionales como a las del tipo NoSQL
- Integra las tecnologías de acceso a datos tradicionales, simplificando el trabajo a la hora de crear las implementaciones concretas.
- Con cada tipo de tecnología de persistencia los DAOs (Data Access Objects) ofrecen las funcionalidades típicas de un CRUD (Create-Read-Update-Delete) para objetos de dominio propios, métodos de búsqueda, ordenación y paginación

Spring Data

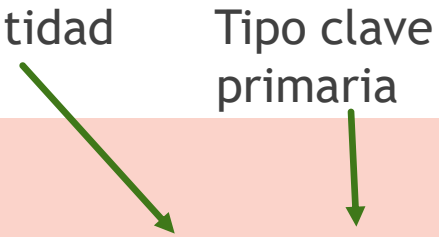
- Proporciona soporte para las siguientes tecnologías de persistencia:
- JPA y JDBC
- Redis
- MongoDB
- Apache Hadoop
- GemFire
- Neo4j
- HBase

Spring Data

- Spring Data JPA
- Spring Data MongoDB
- Spring Data Redis
- Spring Data REST
- …

Spring Data JPA

- Elimina el código de acceso a los datos
- Definición de una subinterfaz de JpaRepository que proporciona métodos CRUD




```
public interface AgendaSpringDao extends JpaRepository<Contacto, Integer>{

}
```

Métodos de JpaRepository

- T save(T entidad) Salva la entidad en la unidad de persistencia, si la entidad ya existe la actualiza. Devuelve la propia entidad.
- Optional<T> findById(ID id) Devuelve la entidad a partir de su primary key, envuelta en un Optional.
- List<T> findAll() Recupera todas las entidades
- void deleteById(ID, id) Elimina la entidad a partir de su clave primaria

Spring Data JPA

- Spring implementa automáticamente métodos a partir del nombre del mismo o instrucciones JPQL

Implementación a partir del nombre de método

Implementación a partir de la query

```
public interface AgendaSpringDao extends JpaRepository<Contacto, Integer>{

    Contacto findByEmail(String email);

    @Query("Select c from Contacto c Where c.edad<=?1")
    List<Contacto> buscarPorEdadMaxima(int edad);

    @Transactional
    @Modifying
    @Query("Delete from Contacto c Where c.email=?1")
    void eliminarPorEmail(String email);
}
```

Ejemplos

```
@Repository
public interface UserRepository extends JpaRepository<UserEntity, Integer> {

    public List<UserEntity> findByEdadLessThan(int edad);
    public List<UserEntity> findByEdadGreaterThanEqual(int edad);
    public List<UserEntity> findByNameLike(String name);
    public List<UserEntity> findByNameContaining(String name);

    @Query(value="select * from ms_users where name = ?1 and edad >= ?2 and edad <= ?3" ,nativeQuery = true)
    public List<UserEntity> findAllUsersBetweenAgeAndName(String name,int ageBegin,int ageEnd);
```

Anotaciones en clase main

- @EntityScan En el atributo basePackages se indican los paquetes en donde se encuentran definidas las entidades.
- @Enable.JpaRepositories En el atributo basePackages se indican los paquetes en donde se encuentran definidas las subinterfaces de JpaRepository

```
@EnableJpaRepositories(basePackages={"com.dao"})
@EntityScan(basePackages={"com.model"})
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

Spring MVC

- Spring Web MVC es un subproyecto Spring que esta dirigido a facilitar y optimizar el proceso creación de aplicaciones web utilizando el patrón MVC (Modelo-Vista-Controlador)
- El **Modelo** representa los datos o información que manejará la aplicación web
- La **Vista** son todos los elementos de la UI (Interfaz de Usuario), con ellos el usuario interactúa con la aplicación, ejemplo: botones, campos de texto, etc.
- El **Controlador** será el encargado manipular los datos en base a la interacción del usuario

Spring MVC

- El Controlador Frontal (FrontController) es el encargado de soportar todas las peticiones Web y redirigirlas a los componentes que sean necesarios.
- En este caso el controlador de Spring se denomina **ServletDispatcher** y viene configurado por defecto por Spring Boot.

Spring MVC

- Dependencias necesarias:
- Spring Web
- Thymeleaf
- Thymeleaf es un motor de plantillas

Spring MVC

- **@RequestMapping**
- Permite asignar solicitudes web a clases de controlador y/o métodos

```
@Controller
@RequestMapping("/sitio")
public class SaludoController {
@GetMapping(value="/saludo")
public String mostrarHome() {
return "saludo";
}
```

Spring MVC

- **@RequestMapping**
- Permite asignar solicitudes web a clases de controlador y/o métodos

```
@Controller
@RequestMapping("/sitio")
public class SaludoController {
@GetMapping(value="/saludo")
public String mostrarHome() {
return "saludo";
}
```

Spring MVC- Thymeleaf

```
<html xmlns:th="http://www.thymeleaf.org">
<body>
<table>
<tr th:each="persona: ${personas}">
<td th:text="${persona.nombre}" />
<td th:text="${persona.apellidos}" />
<td th:text="${persona.edad}" />
</tr>
</table>
</body>
</html>
```

ResponseEntity

- Es una extension de **HttpEntity** que añade un código de status en HttpStatusCode
- **Ejemplo:**

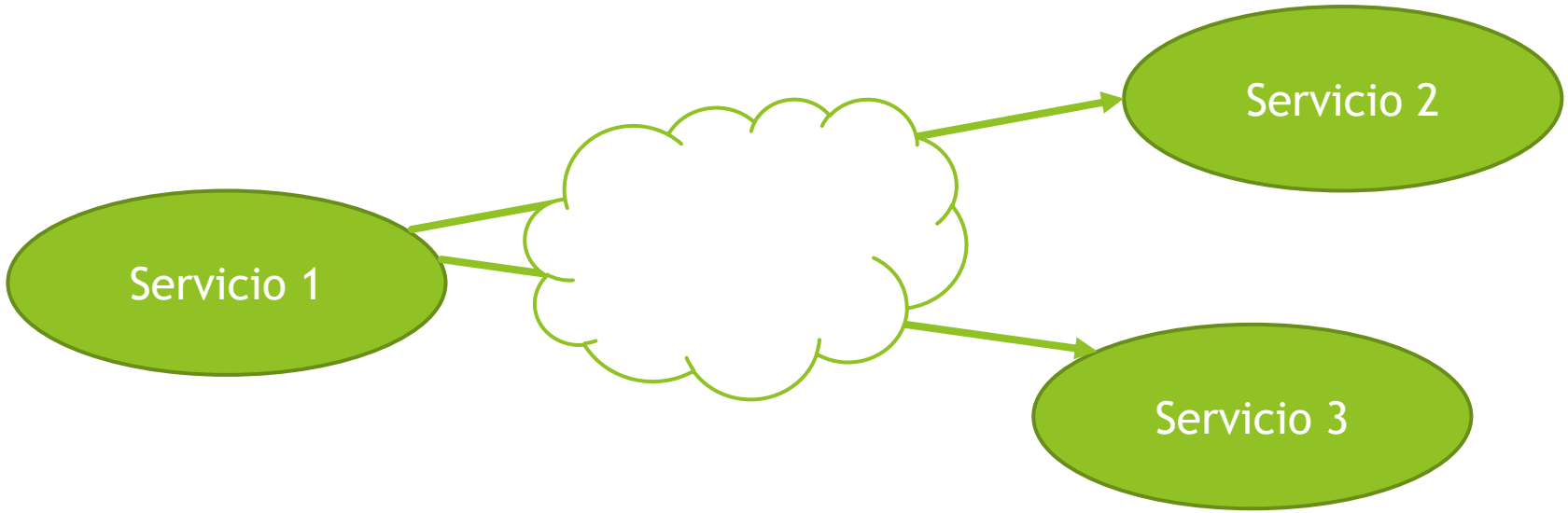
```
public ResponseEntity<?> getById(@PathVariable("codCurso") int codCurso) {
try {
    Curso curso = service.getById(codCurso);
    return ResponseEntity.ok(curso);

} catch (IllegalArgumentException e) {
return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(Collections.singleton(new
Curso("Curso no valido", 0, 0)));

}
```

ResponseEntity

- En RestTemplate, se devuelve al invocar a exchange():

```
ResponseEntity<String> entity = template.getForEntity("https://ejemplo.com", String.class);
String body = entity.getBody();
MediaType contentType = entity.getHeaders().getContentType();
HttpStatus statusCode = entity.getStatusCode();
```

Interacción con microservicios



Interacción entre microservicios

- La clase RestTemplate
- Forma parte del módulo Web de Spring, por lo que hay que añadir el starter Web.
- Su creación se define en una clase de configuración

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

Interacción entre microservicios

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

- Para inyectarlo en una variable:

```
@Autowired
RestTemplate template;
}
```

Petición GET con RestTemplate

- Se realiza mediante el método getForObject
- getForObject(String url,Class<T> responseType, Object…uriVariables)

```
String url="http://localhost:8080/cursos";
Curso[] cursos=template.getForObject(url,Curso[].class);
Curso cur = template.getForObject(url+"/{name},Curso.class,"PHP");
```

Petición POST con RestTemplate

- Se realiza mediante el método **postForObject** si se devuelve resultado

postForObject(String url, Object request,Class<T> responseType, Object…uriVariables)

- Se realiza mediante el método **postForLocation** si no se devuelve resultado:

postForLocation(String url, Object request, Object…uriVariables)

```
String url="http://localhost:8080/curso";
Curso curso=new Curso("PHP", 30, "tardes");
template.postForLocation(url, curso);
```

Petición PUT con RestTemplate

- Se asume que no devuelve resultado
- void put(String url, Object request, Object…uriVariables)

```
String url="http://localhost:8080/curso";
Curso curso = template.getForObject(url+"/{name}",Curso.class,"PHP");
Curso.setDuration(70);
template.put(url,curso);
```

Petición DELETE con RestTemplate

- Se asume que no devuelve resultado
- void delete(String url, Object request, Object…uriVariables)

Método exchange

- Permite realizar cualquier tipo de petición
- Adecuado cuando los métodos anteriores no permiten hacer la petición deseada (por ejemplo petición PUT con devolución de resultados):

ResponseEntity<T> exchange(String url, HttpMethod método,

HttpEntity<?> entity, Class<T> responseType, Object…uriVariables)

```
Curso curso = template.getForObject(url+"/{name}",Curso.class,"PHP");
Curso.setDuration(70);
ResponseEntity<Curso[]> rp =
    tmp.exchange(url, HttpMethod.PUT, new HttpEntity<Curso),Curso[].class;
Curso[] cursos= rp.getBody();
```

Petición POST con RestTemplate

- Se realiza mediante el método **postForObject** si se devuelve resultado

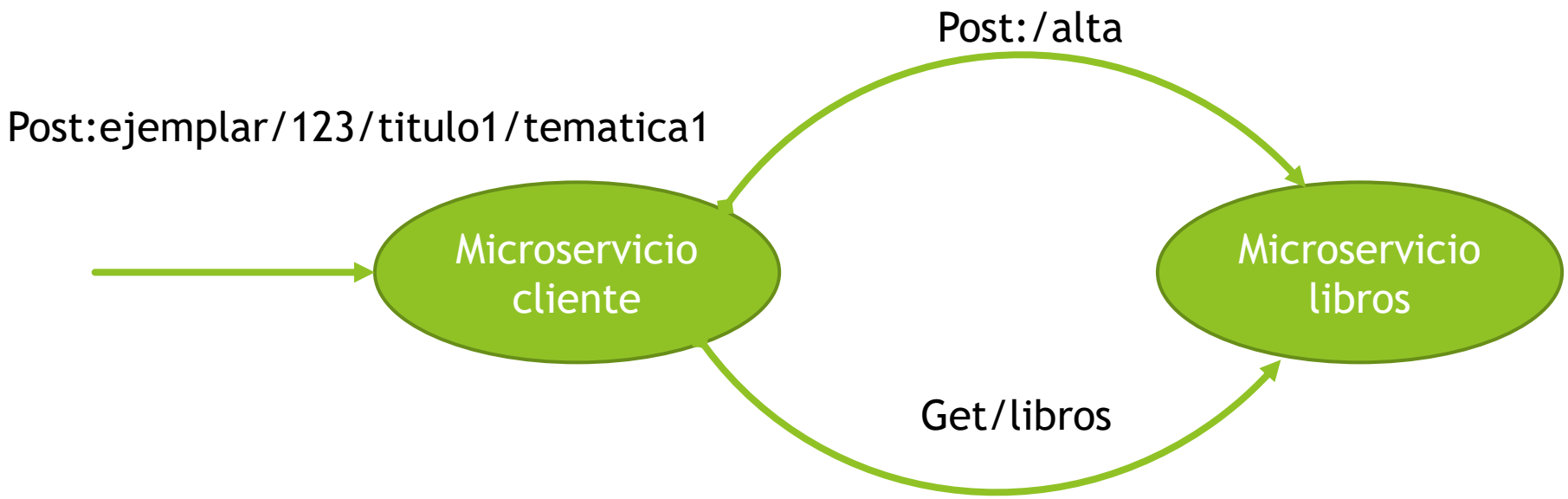
postForObject(String url, Object request,Class<T> responseType, Object…uriVariables)

- Se realiza mediante el método **postForLocation** si no se devuelve resultado:

postForLocation(String url, Object request, Object…uriVariables)

```
String url="http://localhost:8080/curso";
Curso curso=new Curso("PHP", 30, "tardes";
template.postForLocation(url, curso);
```

Ejemplo



ejemplar/123/Delphi/programación