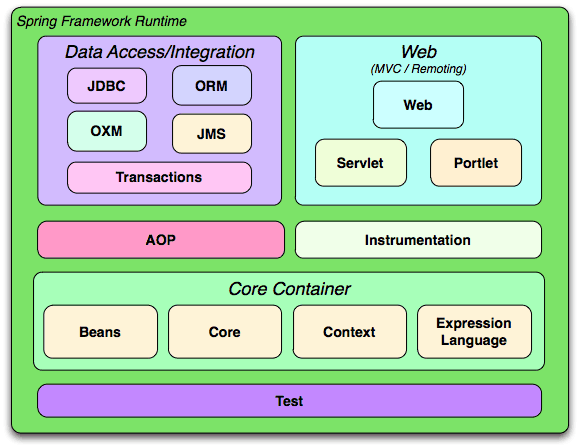
**Spring Framework** es un *framework* Open Source que facilita la creación de aplicaciones de todo tipo en **Java**, **Kotlin** y **Groovy**.

Si bien es cierto que, por lo que es más conocido es por la inyección de dependencias, Spring Framework está dividido en diversos módulos que podemos utilizar, ofreciéndonos muchas más funcionalidades:

* **Core container**: proporciona inyección de dependencias e inversión de control.
* **Web**: nos permite crear controladores Web, tanto de 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**: proporciona soporte para la instrumentación de clases.
* **Pruebas de código**: contiene un *framework* de *testing*, con soporte para JUnit y TestNG y todo lo necesario para probar los mecanismos de Spring.

**Estos módulos son opcionales**, por lo que podemos utilizar los que necesitemos sin tener que llenar nuestro classpath con clases que no vamos a usar.



**Razones para usar Spring**

Aunque no sea una característica única de Spring, el uso de inyección de dependencias facilita la programación contra interfaz, permitiendo a los distintos componentes depender únicamente de interfaces y produciendo así un **código menos acoplado**. No solo eso, también permite implementar el patrón *singleton* de una forma extremadamente sencilla (por defecto, las dependencias que inyectamos son singletons).

Hay cientos de tecnologías que Spring permite integrar. Desde bibliotecas que implementan [opentracing](https://opentracing.io/" \t "_blank) hasta las que nos generan métricas para nuestra aplicación, pasando por serialización/deserialización a JSON y XML, seguridad con OAuth2 o programación reactiva entre otras.

En general, Spring **aumenta la productividad y reduce la fricción** al ofrecernos abstracciones sobre implementaciones de tecnologías concretas. Un ejemplo claro es el de spring-data, que nos permite definir el acceso a base de datos con interfaces Java. Esto lo consigue *parseando* el nombre de los métodos y generando la consulta con la sintaxis específica para el *driver* que utilicemos. Por ejemplo, cambiar nuestra aplicación de MySQL a PostgreSQL es tan sencillo como cambiar el *driver*: Spring se encarga de la sintaxis de forma transparente.

A pesar de "la magia de Spring" 🧙🏻‍♂️, como muchos lo llaman, Spring nos permite desactivar estos "comportamientos mágicos" en caso de ser necesario, por lo que **podemos tomar el control cuando necesitemos más granularidad**. Siguiendo con el ejemplo de spring-data, este control sería necesario si tenemos que realizar consultas mucho más complejas que un SELECT \* BY name. En esos casos, entre otras opciones, podemos anotar nuestro método con @Query y escribir la consulta que deseemos. Ya no hay magia 😉

Por lo general, Spring no obliga a implementar ni extender nada, lo que nos permite escribir código que es "agnóstico" del *framework*. De esta forma, desarrolladores con cero o muy poco conocimiento de Spring pueden realizar su trabajo sin mayores complicaciones.

**Spring es de código abierto y tiene una gran comunidad detrás**. Si encuentras un *bug*, echas en falta una funcionalidad o lo que sea, siempre puedes [abrir un ticket](https://github.com/spring-projects/spring-framework/issues) o contribuir por tu cuenta.

Si bien es cierto que Spring Framework es muy potente, **la configuración inicial y la preparación de las aplicaciones para producción** son tareas bastante tediosas. **Spring Boot simplifica el proceso al máximo** gracias a sus dos principales mecanismos.

### Contenedor de aplicaciones integrado

Spring Boot permite compilar nuestras aplicaciones Web como un archivo .jar que podemos ejecutar como una aplicación Java normal (como alternativa a un archivo .war, que desplegaríamos en un servidor de aplicaciones como Tomcat).

Esto lo consigue **integrando el servidor de aplicaciones en el propio .jar** y levantándolo cuando arrancamos la aplicación. De esta forma, podemos **distribuir nuestras aplicaciones de una forma mucho más sencilla**, al poder configurar el servidor junto con la aplicación. Esto también es muy útil en **arquitecturas de microservicios**, puesto que permite distribuir nuestras aplicaciones como imágenes Docker que podemos escalar horizontalmente (algo muy complicado con un .war).

|  |
| --- |
| **Nota**: Spring boot **permite** distribuir tu aplicación como un jar, **no lo impone**. Si prefieres desplegar tu aplicación en un servidor de aplicaciones tradicional, Spring Boot te deja compilar el código como un .war que no incluya ningún servidor de aplicaciones integrado. |

### Starters

Spring Boot nos proporciona una serie de dependencias, llamadas starters, que podemos añadir a nuestro proyecto dependiendo de lo que necesitemos: crear un controlador REST, acceder a una base de datos usando JDBC, conectar con una cola de mensajes Apache ActiveMQ, etc.

Una vez añadimos un starter, éste nos proporciona todas las dependencias que necesitamos, tanto de Spring como de terceros. Además, los starters vienen configurados con valores por defecto, que pretenden **minimizar la necesidad de configuración** a la hora de desarrollar. Un buen ejemplo es el de **spring-boot-starter-actuator**: una vez que añadimos la dependencia, nuestra aplicación empezará a generar métricas tanto de la JVM como de la aplicación en sí (latencias, errores, etc).

Al igual que con Spring Framework, **cualquier configuración puede ser modificada** de ser necesario: desde el puerto en el que la aplicación escucha peticiones, hasta el banner que sale por consola al arrancar la aplicación.

**Hola Mundo con Spring Boot**

Para demostrar lo fácil que es empezar a usar Spring Boot, vamos a ver el código necesario para crear nuestro primer "Hola mundo":

