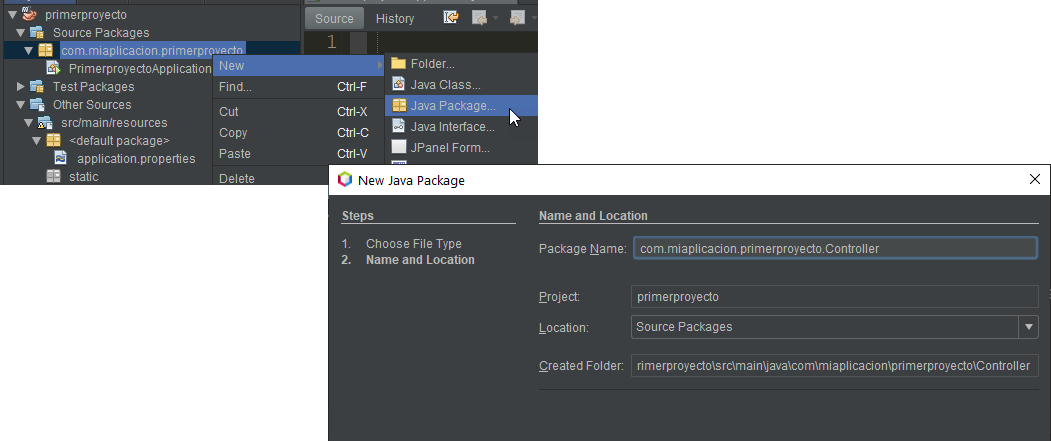
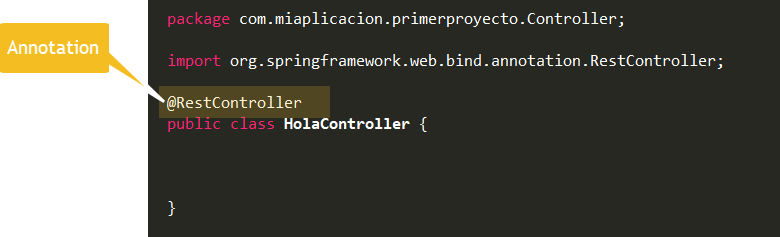
## Creando una API con Spring Boot

Crear una Api con SpringBoot es bastante sencillo. En primera instancia, crearemos un proyecto en Initializr (como se especificó antes). Una vez creado el proyecto y que se encuentre levantado en el IDE, vamos a crear un nuevo paquete llamado Controller: 

**Creando un @Controller**

El nuevo paquete creado representará la capa controladora del proyecto (siguiendo el MVC). Dentro del paquete, se creará una nueva clase que se encargará de recibir las distintas peticiones o solicitudes, es decir, crearemos un controlador. Para ello será necesario mapear la clase con la annotation @RestController.

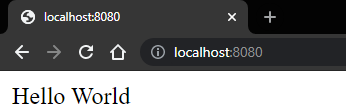


**Creando un @GetMapping**

Como se mencionó anteriormente, las APIs reciben solicitudes mediante el protocolo HTTP, es decir, pueden recibir solicitudes mediante los diferentes métodos que HTTP provee, como ser por ejemplo GET, POST, PUT, etc. Cuando se desea llevar a cabo un end-point que se ejecuta al recibir una solicitud GET, ésta debe ser etiquetada dentro del controller con la annotation @GetMapping. Te mostramos un ejemplo de un HelloWorld:



De esta manera, cuando ingresemos desde el navegador (que por defecto utiliza el método GET) a la url de la aplicación, obtendremos el resultado de la función decirHola() que se va a ejecutar, tal como puede verse a continuación:

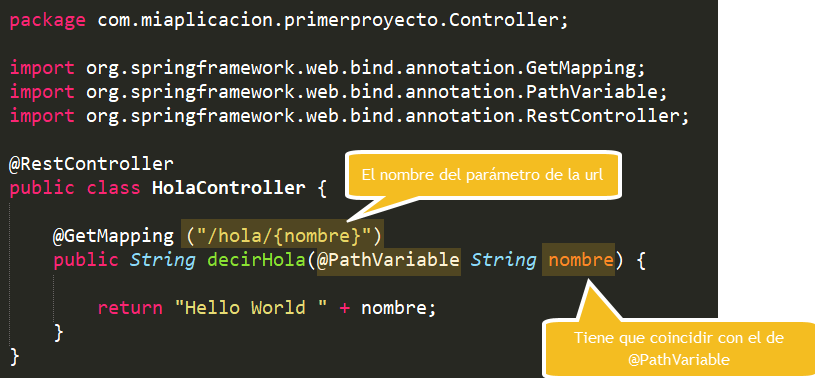


*En caso que estés utilizando una versión más reciente (de la****3.0****en adelante), deberás hacer una pequeña indicación más en la annotation****@GetMapping****que es la****URL****a donde deberá dirigirse para poder ejecutar el método* ***decirHola****. En este caso, como vamos a dejarlo en el directorio raiz, debemos agregar****(«/»)****, quedando de la siguiente manera:****@GetMapping («/»).***

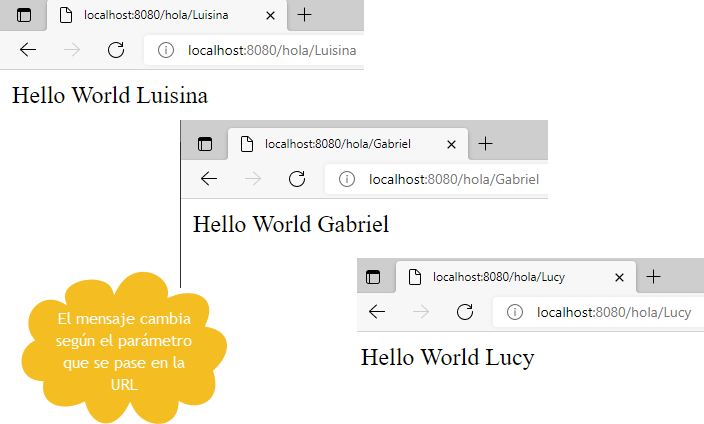
## @GetMapping + Parámetros: @PathVariable

**Parámetros con @GetMapping**

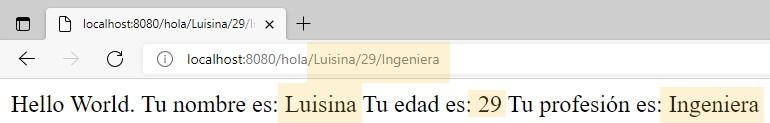
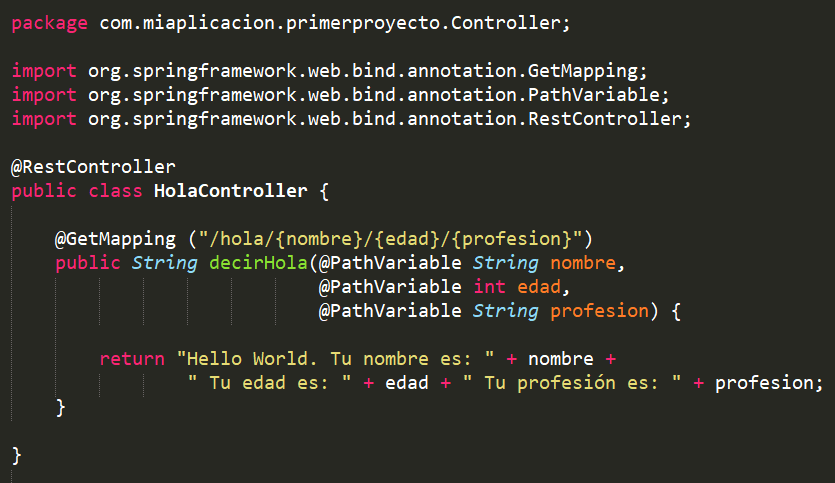
Así como en un @GetMapping puede establecerse un determinado path, también es posible recibir, tal como el método HTTP GET lo indica, valores o parámetros mediante la URL. Éstos, pueden asignarse a las diferentes funciones que tengamos en el controller a partir de la annotation @PathVariable. Un ejemplo de esto puede verse en la siguiente ilustración:



Con este ejemplo, recibiremos mediante el método GET y por la URL un nombre como parámetro. Con @PathVariable asignamos este valor a una variable (que debe tener el mismo nombre) para luego utilizarla en la función en cuestión, en este caso, para agregar el nombre al saludo “Hello World”. Ejemplo de la devolución de una solicitud con un nombre:

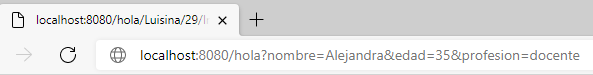


Así como lo hicimos con el parámetro nombre, podemos agregar nuevos parámetros e ir recibiéndolos por la URL de la misma manera. Te mostramos un ejemplo completo con los parámetros edad y profesión:



**Creando @RequestParam**

Otra forma de recibir múltiples parámetros en un endpoint dentro de un controller, es mediante el uso de la annotation @RequestParam. Ésta es una anotación que permite recibir parámetros mediante el método GET. La especificación de los parámetros se manifiesta mediante un signo “?” luego del path en cuestión y luego, cada uno de los parámetros se indica mediante su nombre, un símbolo igual y su valor. En caso de que haya más de un parámetro, se unen entre si mediante el símbolo “&”. Ejemplo de cómo se vería la URL:



Dentro del endpoint creado, se debe especificar la annotation @RequestParam por cada uno de los parámetros que recibiremos con la variable asociada que recibirá dicho valor. Un ejemplo de esto a nivel código puede verse en la siguiente Ilustración, mientras que en la segunda ilustración puede verse la respuesta a la solicitud:





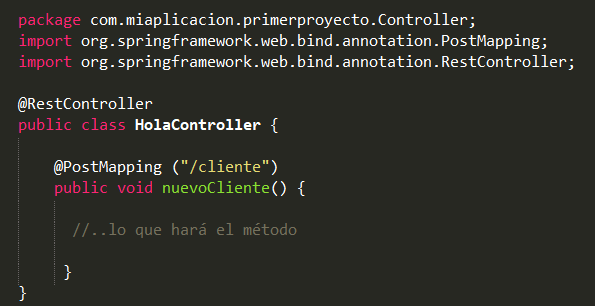
@PathVariable y @RequestParam cumplen funciones muy similares, sin embargo, sus implementaciones son diferentes. En @PathVariable, los parámetros se brindan mediante diferentes apartados path (“/”) de la dirección, mientras que en @RequestParam, los datos se especifican dentro de un mismo path mediante el símbolo “?” y separando a cada uno de ellos mediante el símbolo “&”.

Más sobre @annotation

**Creando @PostMapping**

Así como para recibir solicitudes mediante el método GET se utiliza en el controller la annotation “@GetMapping”, también existe una que permite la recepción y tratamiento de solicitudes o requests que se envíen mediante el método POST, esta annotation es el @PostMapping.

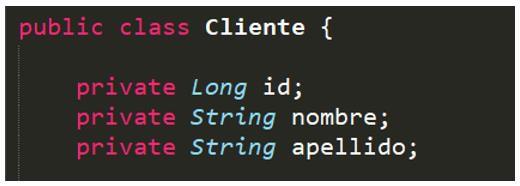
La annotation se coloca para mapear el método que se ejecutará al recibir la petición POST en la determinada URL que se especifique. Un ejemplo a nivel código puede verse a continuación:



Recepción de un objeto mediante @RequestBody

@PostMapping tiene un fiel aliado a la hora de recibir valores en una solicitud o request que es el @RequestBody. Esta annotation permite recibir objetos de dominio completos al endpoint creado mediante el cuerpo del mensaje de la solicitud para convertirlos luego en la aplicación en objetos Java.

Si suponemos una clase Cliente y el método nuevoCliente como se ve en la ilustración a continuación, mediante @RequestBody se podrían recibir los datos correspondientes a un cliente y transformarlos en un objeto Cliente Java:



Como se mencionó en otras ocasiones, el método POST no envía valores mediante la URL en cuestión, sino mediante el cuerpo o la cabecera de los mensajes HTTP.

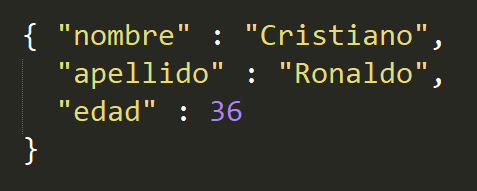
Como los navegadores utilizan por defecto el método GET, para poder probar los endpoints que utilizan el método POST, es necesario simular las solicitudes que se realizan mediante este verbo con algún software adicional. Uno de los más utilizados por excelencia es Postman.

## JSON

**JSON** o mejor conocido como “*Java Script Object Notation”*, es un formato de texto que es utilizado principalmente para el intercambio de datos mediante el**protocolo HTTP** entre diferentes sistemas o **APIS** interconectados entre sí.

Sirve como un lenguaje “intermedio”, dado que independientemente del lenguaje de programación, es posible traducir los datos que se requieren transferir entre sistemas a JSON.

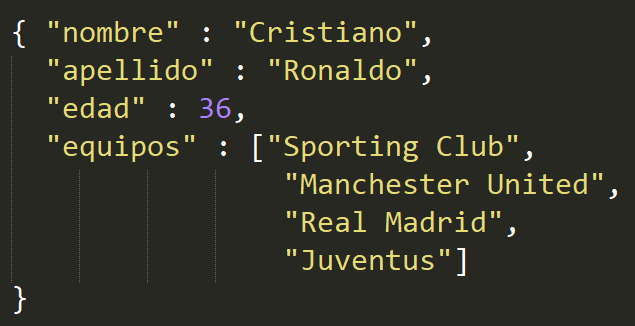
**JSON** se vale del concepto **“clave-valor”**, donde para cada clave existe un valor asociado. Un conjunto de claves y valores conforman un objeto, que en JSON se representa mediante la apertura y cierre de llaves {}. Un ejemplo sencillo de la sintaxis de un mensaje JSON podemos verlo a continuación:



Ejemplo sencillo de JSON

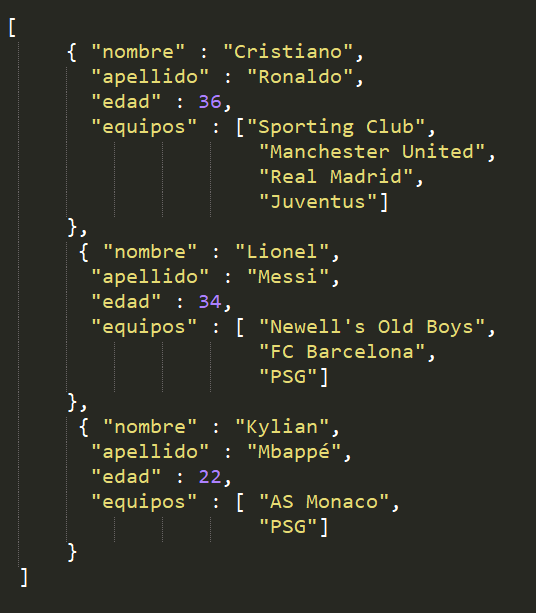
Ahora supongamos que queremos incorporar al JSON de la ilustración, los equipos en los cuales jugó Cristiano Ronaldo.

Podemos hacer esto mediante la incorporación de [] donde la clave “equipos” no contendrá solo un valor, sino un conjunto de valores.



Ejemplo de clave con valores múltiples

De igual manera, así como en el JSON de la imagen mostramos un solo objeto, es posible incorporar varios objetos, donde cada uno de ellos estará separado por “,” (coma) y un nuevo par de llaves {}. Por ejemplo:



Ejemplo de varios objetos dentro de un JSON

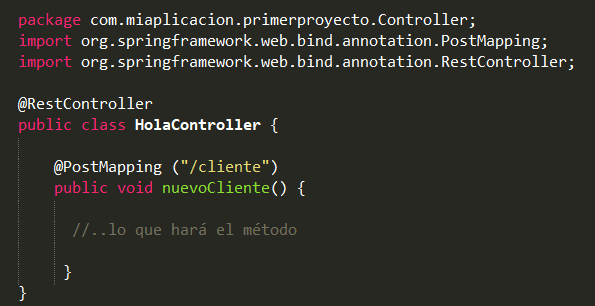
¿Por qué es importante manejar y conocer JSON? Porque la mayor parte de las API REST que se encuentran productivas en la actualidad, utilizan a JSON como formato de mensaje para comunicarse mediante el protocolo HTTP.

## @PostMapping + @RequestBody

## @PostMapping

Así como para recibir solicitudes mediante el método GET se utiliza en el controller la annotation “@GetMapping”, también existe una que permite la recepción y tratamiento de solicitudes o requests que se envíen mediante el método POST, esta annotation es el @PostMapping.

La annotation se coloca para mapear el método que se ejecutará al recibir la petición POST en la determinada URL que se especifique. Un ejemplo a nivel código puede verse a continuación:

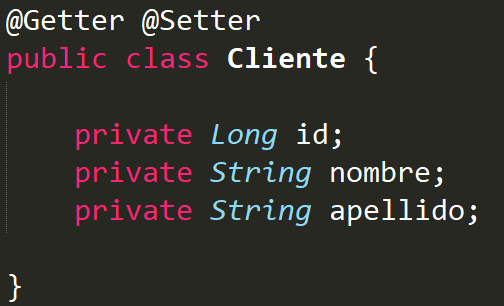


Ejemplo de uso del @PostMapping

## @RequestBody

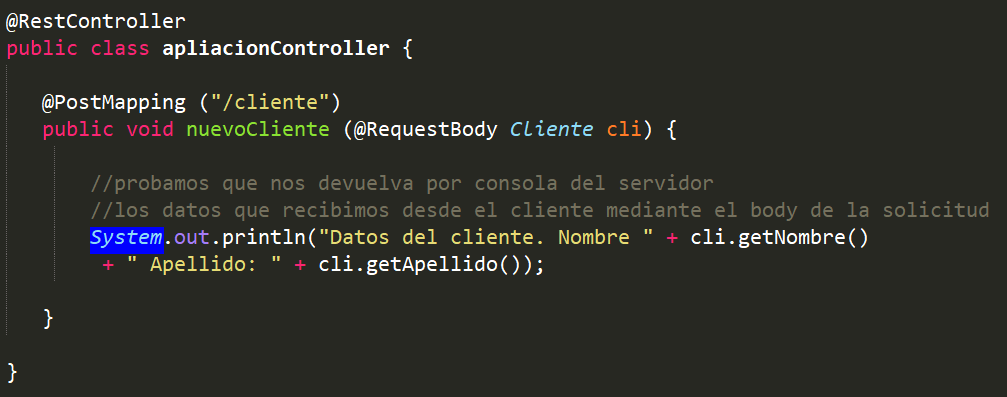
**@PostMapping** tiene un fiel aliado a la hora de recibir valores en una solicitud o request, la annotation **@RequestBody**. Ésta permite recibir objetos de dominio completos en cualquier endpoint de nuestra API dentro del cuerpo (o body) de la request. Estos datos que se reciben en el body serán transformados luego en objetos de Java dentro de nuestra aplicación.

Supongamos que tenemos una clase llamada Cliente en nuestra aplicación con los atributos id, nombre, apellido y sus respectivos getters y setters que simplificaremos mediante Lombok con dos annotations @Getter y  @Setter, tal como podemos ver en la siguiente imagen:



A partir de esta clase, supongamos que en nuestro controller tenemos el método nuevoCliente, mapeado con el @PostMapping y en el path «/cliente» (tal como vimos en la imagen anterior).

Con esto armado, mediante @RequestBody se podrían recibir los datos correspondientes a un cliente y transformarlos en un objeto de la clase Cliente en Java. Pero… ¿Cómo podríamos hacer esto? A continuación te dejamos un ejemplo de código:

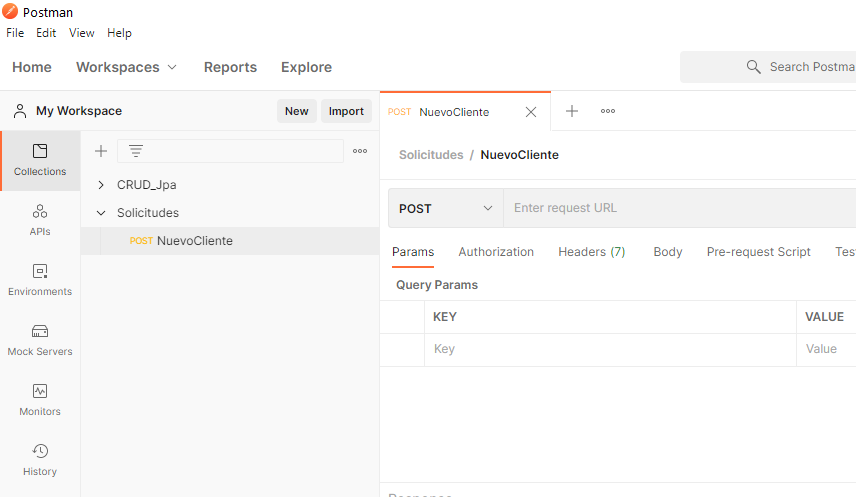
Ejemplo @PostMapping + @RequestBody

Como se mencionó en otras ocasiones, el método **POST** no envía valores mediante la URL en cuestión, sino mediante el body o la cabecera de las solicitudes o requests HTTP. A partir de esto entonces… **¿Cómo generamos una solicitud POST para poder probar si nuestra API funciona correctamente?**

Para responder a esto, tenemos que recordar que los navegadores utilizan por defecto el método GET y es por ello que pudimos probar todos nuestros endpoints directamente desde ellos para ver su correcto funcionamiento; sin embargo, para poder probar los endpoints que utilizan el **método POST**, es necesario simular las solicitudes como si se estuvieran realizando desde otra aplicación o un frontend mediante este verbo, es por ello que necesitamos, para simular solicitudes POST un **software adicional**. Uno de los más utilizados para realizar este tipo de pruebas, es **Postman.**

## Postman

Postman es un software que permite simular el envío de solicitudes HTTP REST mediante diferentes métodos sin la necesidad de desarrollar una aplicación cliente o de contar con un front-end en particular. Para utilizarlo únicamente es necesario descargarlo de su página oficial <https://www.postman.com/downloads/>, instalarlo y empezar con sus configuraciones iniciales. En la siguiente imagen tenemos un ejemplo de cómo se ve Postman.



## Pruebas con Postman

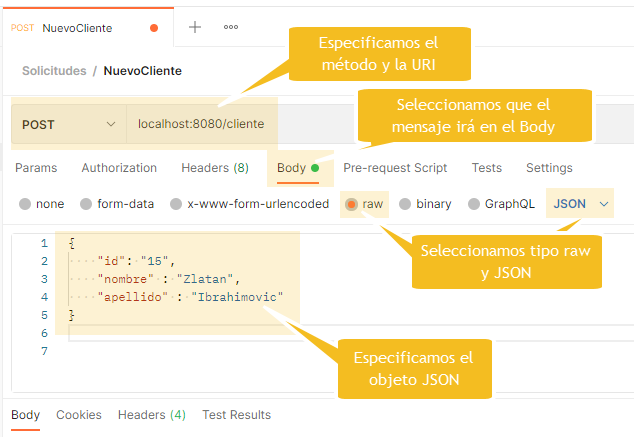
En la clase anterior, desarrollamos una API en donde se esperaba la recepción de los datos de un cliente mediante el método POST dentro del body de una solicitud. El problema que teníamos era que NO PODÍAMOS PROBAR si funcionaba correctamente dado que no sabíamos cómo simular una solicitud POST.

Como ahora sabemos qué es Postman, vamos a intentar probar si nuestra API funciona correctamente y recibe los datos necesarios.

Para ello crearemos una nueva solicitud y en primer lugar indicaremos el método que vamos a utilizar (en este caso POST) y la URL de nuestra API (en este caso localhost:8080/cliente); luego en las pestañas que se encuentran debajo, tenemos todas las partes de un paquete, como vamos a recibir los datos mediante @RequestBody, vamos a seleccionar Body, haciendo referencia que allí viajarán los datos. Además de esto, debemos seleccionar formato raw en el radio button que allí figura y JSON como formato en el combo box que se encuentra a la derecha.

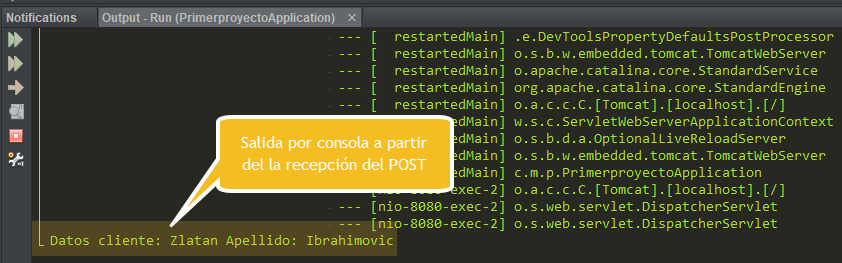
Una vez hecho esto, escribiremos un archivo JSON manualmente simulando un objeto, para ello utilizaremos las claves id, nombre y apellido (que coinciden con los nombres de los atributos de nuestra app en Java) y le asignaremos sus correspondientes valores.

Un ejemplo de cómo hacer todas estas configuraciones se muestra en la imagen a continuación:



Una vez que tengamos todo configurado, podemos hacer click en Send y esto enviará la solicitud POST a nuestra aplicación (obviamente esta debe estar siendo ejecutada).

Si todo llega correctamente, en la consola de salida del servidor que nos proporciona el IDE, podremos visualizar si efectivamente el System.out.println que colocamos, recibió y mostró los valores que recibimos. Un ejemplo de esto puede verse a continuación:



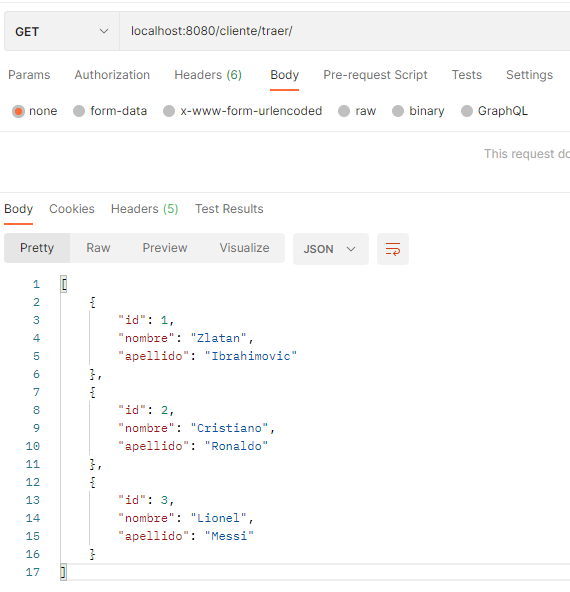
## Respuestas mediante @ResponseBody

Así como el método POST permite la utilización de una annotation @RequestBody para recibir un objeto JSON en una solicitud, el método GET también posee una annotation @ResponseBody para conformar un objeto para la devolución de un mensaje.

Supongamos que tenemos una lista de clientes, y que mediante una solicitud GET deseamos devolver estos clientes que se encuentran en ella. Para ello crearemos un método traerClientes() y lo mapearemos con la annotation @ResponseBody.



Si una vez realizado esto, ejecutamos nuestra aplicación y simulamos una solicitud GET mediante POSTMAN, veremos que vamos a recibir como resultado los 3 objetos Java que creamos en el endpoint en el body de la response que obtenemos.

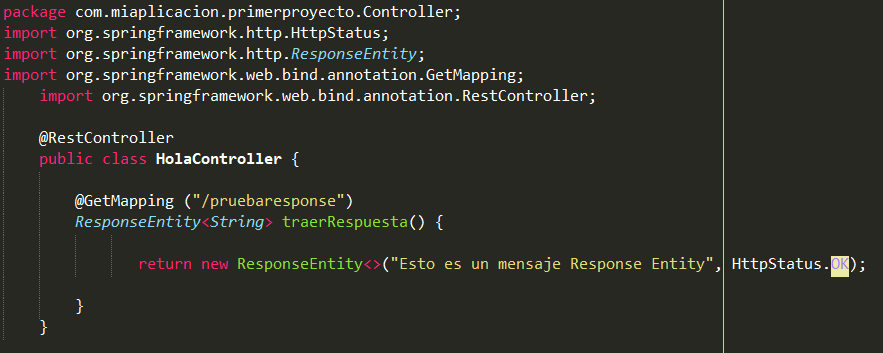


De esta manera podemos devolver los objetos o listas de objetos que creamos en nuestra lógica en el cuerpo de las responses, para que así el cliente que esté haciendo la consulta, pueda recibirlos en formato JSON y hacer uso de los mismos.

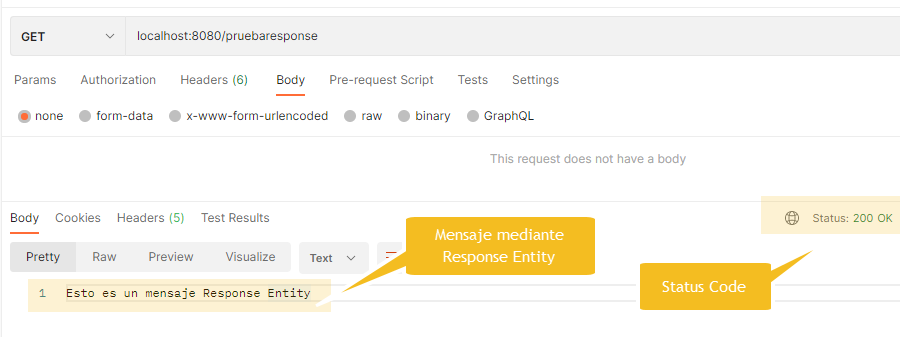
## ResponseEntity

Anteriormente vimos el funcionamiento de @ResponseBody, el cual permite enviar datos o mensajes JSON dentro del cuerpo de un mensaje HTTP, sin embargo, no es la única forma de devolver mensajes desde el controlador, como otra opción tenemos a ResponseEntity.

La diferencia principal es que Response Entity administra todo el paquete completo de una respuesta HTTP, es decir, puede manipular el cuerpo, la cabecera o incluso los códigos de estado, haciendo que la respuesta brindada sea totalmente personalizada. Un ejemplo de una respuesta con ResponseEntity con un Status Code puede verse en la imagen a continuación.



Si probamos con Postman el endpoint que acabamos de hacer con ResponseEntity, vamos a obtener el siguiente resultado, en donde además de ver el status code 200 (que establecimos mediante HttpStatus.OK) veremos el mensaje personalizado que hayamos seleccionado.



Como otro detalle a tener en cuenta, es que Response Entity no es una annotation como el caso de @ResponseBody, sino que se trata de una clase especial que podemos utilizar.

## Patrón DTO

Una de las problemáticas más comunes a la hora de desarrollar aplicaciones (sobre todo web) es la necesidad de **interconexión e intercambio de mensajes** entre capas u otras aplicaciones. Esto hace que sea realmente importante, el hecho de encontrar una forma de diseñar cómo o mediante qué formato debe transmitirse la información.

Es común que en estos casos se utilicen las mismas clases/entidades que tenemos creadas en el modelo de nuestra aplicación; sin embargo, existe una forma más óptima para llevar a cabo esta tarea: implementando el **patrón DTO**.

**DTO** es un **patrón de diseño** que tiene como finalidad crear un **objeto plano (POJO)** con una serie de atributos que puedan ser enviados o recuperados del servidor en una sola invocación. Un **DTO** puede contener datos de múltiples clases, fuentes o tablas de una base de datos y agruparlos en una única clase simple.

**En base a esto podemos decir que DTO nos permite:**

* Crear estructuras de datos totalmente independientes al modelo de datos (o clases entidades).
* Incorporar en una misma clase elementos o datos de clases distintas según la necesidad que tengamos.

Como ventaja principal de implementar **DTO**, podemos mencionar un caso común, que suele darse día a día en un ambiente de desarrollo, donde el modelo de datos deba cambiar por algún motivo en particular. Si esto sucede, no afectará a la forma en la que DTO devuelve los datos, ya que su estructura seguiría siendo la misma.

**Spring** permite manejar **objetos DTO** dentro del **controller** para luego transformarlos en formato **JSON** y retornarlos al cliente que haya realizado una determinada solicitud.

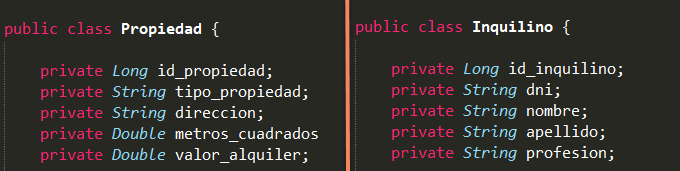
## ¿Cómo implementar DTO?

Supongamos un caso de **una inmobiliaria**, que contiene una **clase Propiedad** (que muestra los datos de una determinada propiedad en alquiler) y una **clase Inquilino** (que muestra los datos de un potencial inquilino).

En el modelo de datos, ambas clases estarán por separado, pero si aplicamos el **patrón DTO**, podemos crear una clase que incorpore los datos que necesitamos de cada una de ellas para devolverlo en una response. Veamos un ejemplo paso a paso.

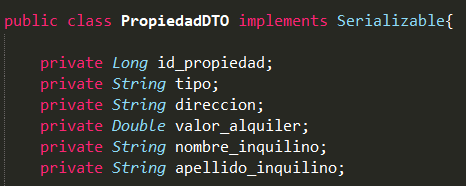
#### PASO 1

Creamos un proyecto SpringBoot con Initializr. Luego dentro de él, creamos las clases Propiedad e Inquilino.



#### PASO 2

Crearemos una clase llamada PropiedadDTO, donde incorporaremos datos combinados de las propiedades y de los inquilinos tal como se puede ver en la captura de código (suponiendo que solo esos datos le sirven/interesan al cliente que recibirá la respuesta de nuestra API).



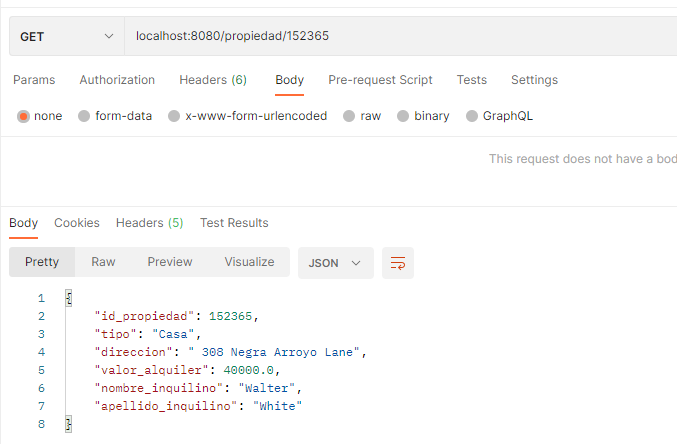
#### PASO 3

Creamos nuestro paquete controller y dentro armamos nuestra clase controladora. Luego creamos un nuevo endpoint, en donde tendremos un objeto propiedad y otro inquilino que unificaremos en un objeto de tipo PropiedadDTO para devolverlo mediante @ResponseBody.



#### PASO 4

Ejecutamos la API desarrollada y probamos realizar con Postman una solicitud y ver si obtenemos como respuesta el objeto DTO que acabamos de crear.



Si todo fue realizado correctamente, obtendremos como respuesta nuestro objeto DTO.

**¿Por qué es importante utilizar DTO?** Porque disminuye la carga de datos que son devueltos desde el servidor hacia el cliente (bajando de esa manera también el tiempo de respuesta) y facilita ampliamente, al cliente que esté consultando, la posibilidad de obtener únicamente los datos que está necesitando, sin tener que hacer luego un «refiltrado» entre todos los datos que pueda recibir.

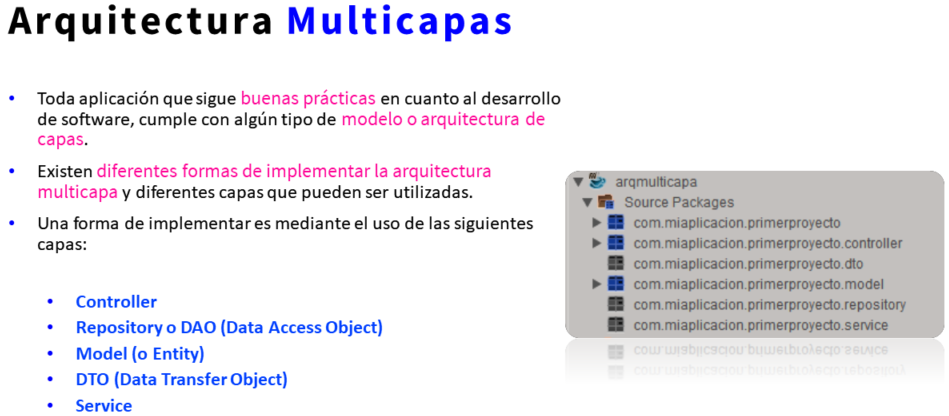
## Arquitectura Multicapas

Toda aplicación que sigue **buenas prácticas de desarrollo de software**, cumple con algún tipo de modelo o **arquitectura de capas**, es decir, una separación entre cada una de las partes con la que interactúa la misma y una forma de comunicación entre ellas.

Existen diversos **modelos o arquitecturas multicapa** que pueden ser implementados o utilizados según el proyecto sobre el cual se esté trabajando, sin embargo, hay algunas estandarizaciones a seguir que se adaptan a la mayoría de los desarrollos realizados en **Java**.

A continuación, se especifica en mayor detalle cada una de las capas de una de las arquitecturas multicapas estándar más utilizada para el desarrollo de aplicaciones con **Spring Boot**.

* **Controller:** Es la capa encargada de atender las solicitudes http entrantes, derivarlas a la capa que corresponda, esperar por una respuesta, generarla y transmitirla nuevamente al cliente. Generalmente la capa “Controller” trabaja estrechamente con la capa de “Service”, donde a partir de una request llama a las funciones que necesite de la capa service para generar una response.
* **Repository o DAO (Data Access Object):** Es la capa encargada de la persistencia de los datos, es decir, del resultado de la interacción de modelado entre las clases desarrolladas y las tablas de una base de datos. Permite el acceso a los datos mediante diferentes tecnologías como por ejemplo JDBC o algún ORM como por ejemplo JPA con Hibernate. Cada una de las clases que se encuentren dentro de esta capa deben estar mapeadas/etiquetadas mediante la annotation @Repository.
* **Model (o Entity):**La capa “model” trabaja estrechamente en conjunto con la clase Repository. Cada una de las clases modela un objeto de la vida real y es marcado con la annotation @Entity en caso de que se transforme en una entidad (tabla) en la base de datos. Cada instancia que se haga a una clase entity, en caso que sea persistida, representará una fila en una tabla de la base de datos.
* **DTO (Data Transfer Object):**Esta capa se encarga de contener todas las clases DTO que hayan sido especificadas en un proyecto. Los DTO buscan desacoplar la forma de presentación de los datos (a futuro en el frontend) con respecto a los objetos propiamente dichos de la capa Model.
* **Service:**La capa de “Service”, mejor conocida como lógica de negocio, es la capa donde se especifican todas las funciones u operaciones que sean necesarias y que puedan ofrecer, como dice su nombre, un servicio a la capa controller. La capa service, por ejemplo, puede encargarse de las autenticaciones o de las políticas de autorización que puede tener la aplicación con respecto al acceso a determinadas funciones.

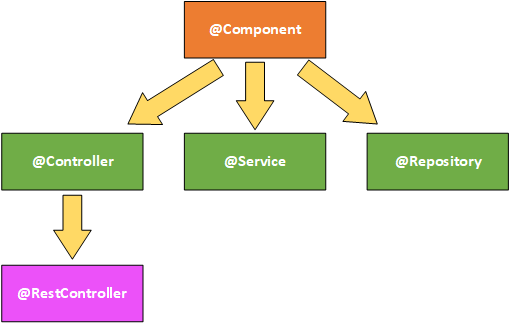


## Arq. Multicapas: @Repository y @Service

## Annotations para capas

Como ya habremos notado a medida que fuimos aprendiendo Spring Boot, el framework se basa principalmente en el uso de **annotations**; es por ello que, al aplicar una arquitectura multicapas, más allá de realizar la correspondiente división de paquetes para mantener ordenada nuestra estructura, es necesario marcar de alguna manera cada una de las clases que pertenezcan a estas capas para que el Framework las reconozca.

**Para ello se vale de 3 annotations:**



Annotations para capas

En la imagen podemos ver @Component, el cual es una annotation para hacer referencia de forma genérica a cualquier annotation de Spring Boot.

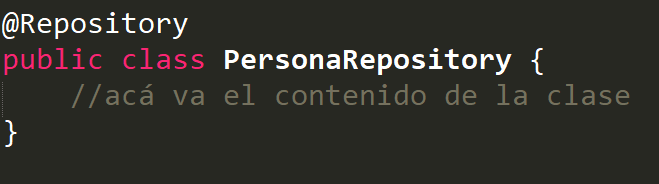
La línea que nos interesa es la que podemos observar en color verde, donde podemos identificar la annotation para la capa Controller. Anteriormente se utilizaba @Controller, sin embargo, tal como se puede ver en la imagen, con las últimas actualizaciones del Framework, esta annotation se convirtió en @RestController, la cual venimos utilizando en clases anteriores.

Por otro lado nos encontramos con @Repository (para la capa repository) y @Service (para la capa service). Ambas anotaciones ampliaremos a continuación para que podamos comprender… ¿Cuál es su uso? Y de que manera las vamos a implementar.

## @Repository

**@Repository** es una anotación de Spring que indica que la clase anotada con ella es un repositorio de datos. Con esto no nos referimos a un repositorio como ser GitHub o GitLab, sino a un lugar donde se realiza el **almacenamiento, manejo o administración de los datos** con los que trabajará la aplicación, sea una base de datos, un archivo, etc.

Todas las clases de manejo de persistencia de datos deberán tener antes de su nombre la anotation **@Repository** y por convención, también el nombre de estas clases suele llevar dicha palabra incorporada. Por ejemplo:



Ejemplo de mapeo de @Repository

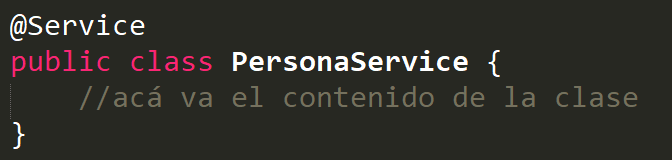
Una ventaja de usar esta anotación es que tiene habilitada la traducción automática de excepciones de persistencia. Cuando se usa un marco de persistencia, por ejemplo: JPA con Hibernate (que veremos más adelante), las excepciones nativas lanzadas dentro de las clases anotadas con @Repository se traducirán automáticamente en subclases de DataAccessExeption de Spring.

## @Service

La capa service, como vimos anteriormente, se encarga de llevar dentro la capa de la lógica de negocio de nuestra aplicación.

Si bien anteriormente estuvimos trabajando directamente en nuestro controller para lograr la devolución de datos y ejecución de métodos, a partir de ahora lo haremos en clases especiales que se van a encontrar en el paquete service. La pregunta ahora es… ¿Cómo vamos a identificar a estas clases que nos van a proporcionar la lógica de usuario? Y la respuesta es, mediante la annotation @Service.

@Service se usa de igual manera que @Repository. Cada clase dentro de nuestro paquete service debe de tener la annotation antes del nombre de la misma, como así también por convención (y buena práctica) se suele incluir la palabra service en el nombre de la clase. Por ejemplo:



Ejemplo de mapeo de @Service

Tanto nuestras clases Service como Repository, tendrán una gran cantidad de métodos que podremos implementar de diferentes maneras dependiendo de las funcionalidades que debamos desarrollar.

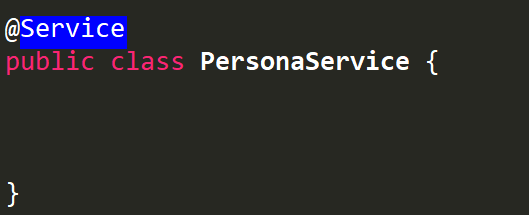
Dado esto, como buena práctica de utilización de capas y división de tareas, tanto la lista completa de métodos que tengamos para nuestras clases service como para nuestras clases repository, no estarán en sus clases concretas (por ejemplo PersonaService o PersonaRepository) sino, que se listarán en una interfaz (asociada a cada clase) que será implementada por cada una de las clases concretas en cuestión.

Dicho esto… ¿Cómo hacemos esto? De la siguiente manera:

#### PASO 1

Haremos el ejemplo con una clase de nuestra capa service (sin embargo, tener en cuenta que se debe aplicar lo mismo en las clases repository).

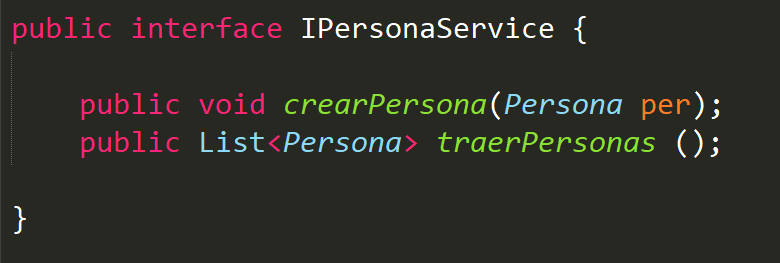
Vamos a crear una clase llamada PersonaService, dentro del paquete (capa) service de nuestro proyecto y la vamos a dejar vacía.



#### PASO 2

Crearemos una interfaz llamada IPersonaService (la I delante indica que es la interfaz asociada a nuestra clase creada anteriormente PersonaService).

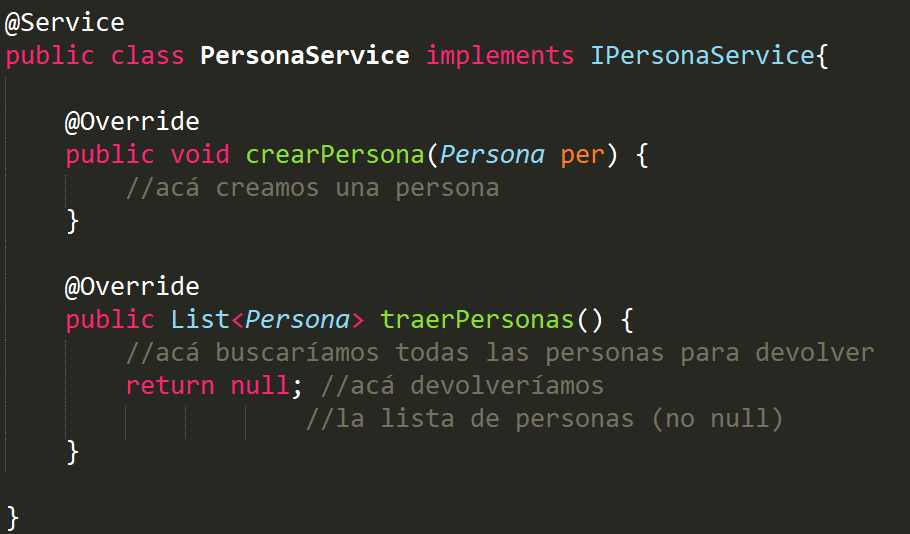
Dentro de esta interfaz vamos a declarar dos métodos: **crearPersona** y **traerPersonas**, pero SIN ESTABLECER SU IMPLEMENTACIÓN (tal y como se lleva a cabo en todas las interfaces):



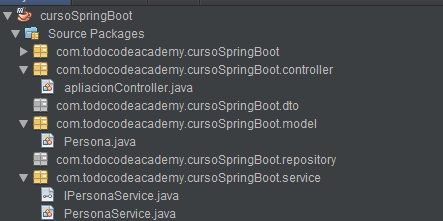
#### PASO 3

Una vez terminada nuestra interfaz, vamos a ir a nuestra clase concreta PersonaService y vamos a implementar la interfaz anteriormente creada.

Una vez implementada, podremos pasar a detallar la lógica en cuestión de cada uno de los métodos.



Y nuestra estructura de clases dentro de nuestro proyecto quedaría de esta manera:



## Inversión de Control (IoC)

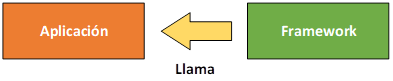
Un aspecto clave y muy característico de **Spring** como **framework** es la utilización de la **inversión de control**, o mejor conocida como**IoC**, por sus siglas en inglés: **Inversion Of Control**.

La **inversión de control** es un concepto en el cual, en lugar de que el programador a través de la aplicación lleve el control del flujo de la misma, sea el framework quien lo haga. Tal como lo dice su nombre, los roles de control se invierten.



Flujo normal de trabajo

En la imagen anterior podemos ver un **flujo normal de trabajo**, donde la app, a través de las líneas de código que especifique el programador, se encarga de llamar a las diferentes funcionalidades de un framework. Sin embargo, en la **inversión de control**, este proceso es diferente, en donde el framework se encarga de llamar a la aplicación.



Flujo mediante inversión de control

Para poder aplicar este concepto, el framework tiene que entender qué es lo que hace la aplicación o de qué manera funciona. Para ello, el framework le requiere a la aplicación una serie de datos que puede estar expresada de diferentes maneras dependiendo del lenguaje de programación, en **Java**, por ejemplo, lo expresamos mediante las **Annotations**.

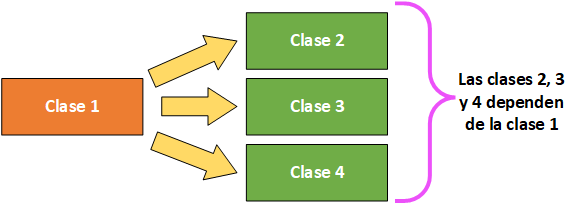
Uno de los principales “problemas” que trata la **IoC** es la creación o instanciación innecesaria de objetos mediante la sentencia new; mediante la IoC, el framework se asegura de crear los objetos (generalmente respetando el patrón de diseño Singleton que establece la utilización de una sola instancia por clase) y los pone a disposición de la aplicación mediante otro concepto muy importante conocido como **Inyección de Dependencias.**

## ¿Qué es la inyección de dependencias?

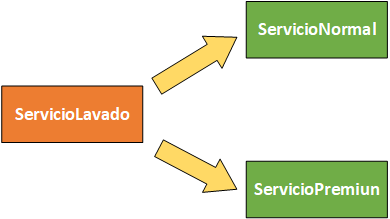
La inyección de dependencias, o mejor conocida como DI (dependency inyection por sus siglas en inglés), es un patrón de diseño que está orientado al manejo de los objetos de una aplicación. Su principal objetivo es el de mantener las capas de una aplicación lo más desacopladas posible entre sí.

Para poder lograr esto, la inyección de dependencias permite que cada una de las partes del programa que se esté desarrollando sea independiente y que no se comuniquen entre si mediante instancias, sino mediante interfaces.

Se entiende entonces que la inyección de dependencias busca desacoplar lo máximo posible la relación entre clases o capas, pero… ¿Qué es una dependencia? Una dependencia es una relación que puede existir entre una o varias clases, donde generalmente una (o varias) dependen de otra principal.

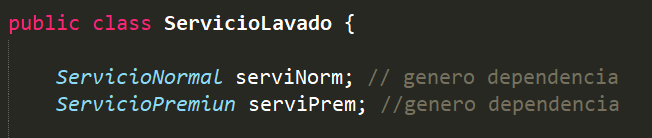
Ejemplo de dependencia entre clases

¿Cómo se representa esto a nivel código? Supongamos que tenemos el modelado de un lavadero de autos, donde existe una clase llamada **ServicioLavado** de la cual dependen otras dos clases, **ServicioNormal** y **ServicioPremiun**, tal como puede verse gráficamente a continuación:



Ejemplo de dependencia entre clases

Como vemos, ServicioNormal y ServicioPremiun dependen fuertemente de ServicioLavado. Esto, a nivel código podríamos verlo reflejado de la siguiente manera:



Ejemplo de dependencia entre clases (código)

Como se puede ver en ambas imágenes, tanto **ServicioNormal** como **ServicioPremiun** dependen de **ServicioLavadado** y es ésta clase quien tiene la responsabilidad de inicializar a ambos servicios en su constructor, sin embargo, si aplicáramos inyección de dependencias, podríamos delegar esta responsabilidad que tiene **ServicioLavado** a otra clase, como por ejemplo, la **clase main** que tengamos en el proyecto.

Ahora bien, **¿Cómo se aplica la inyección de dependencias?** Existen diferentes maneras de hacerlo, sin embargo hay 3 que son las más comunes:

* Mediante un **constructor**
* Mediante un **setter**
* Mediante la Annotation **@Autowired**

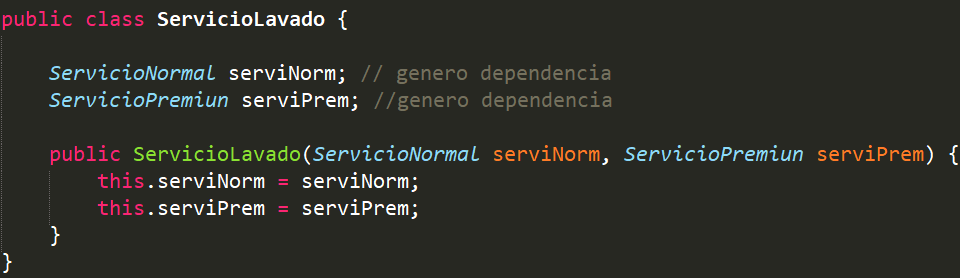
## Inyección de dependencias mediante un Constructor

En la **inyección de dependencias mediante un constructor**, es el propio constructor de una clase el encargado de inyectar la dependencia.

En el ejemplo de la imagen anterior, se podía observar como el constructor creaba las instancias de ambos tipos de servicios dentro de él; en la **inyección de dependencias mediante un constructor**, éste únicamente recibirá como parámetros los objetos ya creados y los asignará según corresponda.

***Nota:****Generalmente cuando creamos constructores con parámetros de forma automática mediante IDEs como Netbeans, ya se nos proporciona de igual manera la inyección de dependencias sin la necesidad de tener que implementarla manualmente.*

Un ejemplo de esto puede verse a continuación:

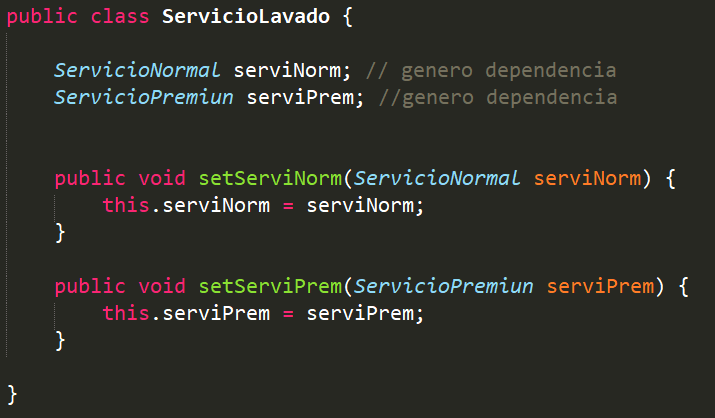
Ejemplo de inyección de dependencias mediante un constructor

## Inyección de dependencias mediante un Setter

Los métodos getter y setter nos permiten obtener o setear valores a los atributos de los objetos que sean creados, de igual manera los métodos set nos permiten inyectar dependencias, donde a partir de la recepción de un objeto como un parámetro este se asigna.

***Nota:****Generalmente cuando creamos setters de forma automática mediante IDEs como Netbeans, ya se nos proporciona de igual manera la inyección de dependencias sin la necesidad de tener que implementarla manualmente.*

Un ejemplo de esto puede verse a continuación:

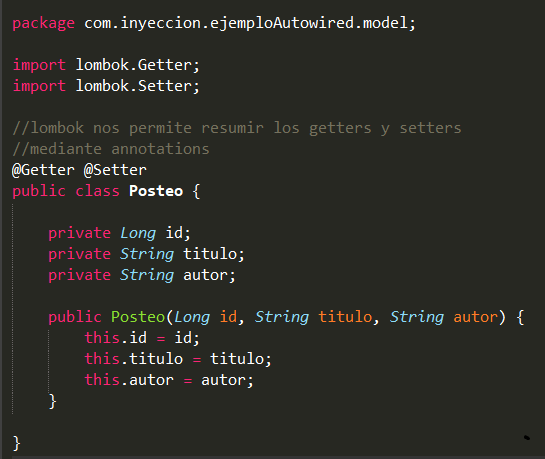
Ejemplo de inyección de dependencias mediante setters

## Inyección de Dependencias mediante @Autowired

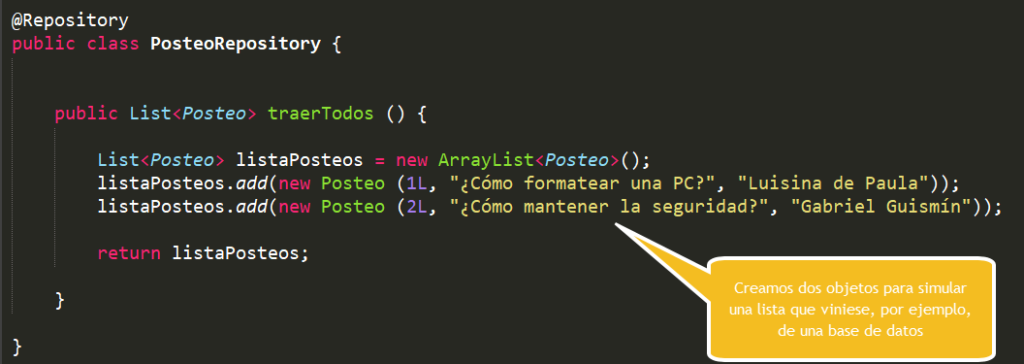
Así como Java permite realizar inyección de dependencias de forma genérica mediante setters o constructores, Spring como framework ofrece una annotation para hacerlo. Ésta es conocida como @Autowired.

Cuando se trabaja con una arquitectura multicapas donde es posible de que exista una dependencia entre los objetos de las mismas, @Autowired es de gran ayuda. A continuación, vamos a ver un ejemplo de su uso en un modelo de capas implementado con Spring Boot.

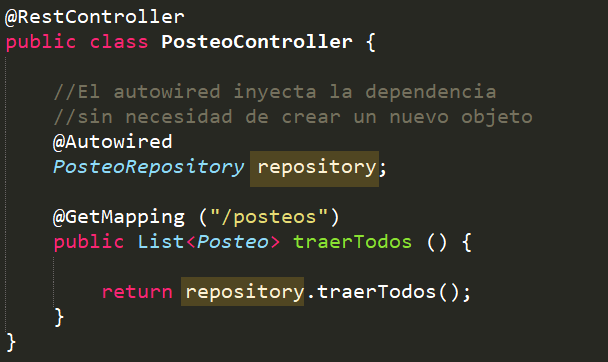
Supongamos que tenemos un sistema creado con Spring Boot para un blog, en donde tenemos una clase llamada Posteo, con los atributos id, titulo y autor.

Clase Posteo

Por otro lado, contamos también con una clase llamada PosteoRepository donde contamos con un método que nos devuelve la lista de todos los posteos existentes.

Clase PosteoRepository

Por último, contamos también con un controller que recibe una solicitud GET para recibir todos los posteos mediante el path «/posteos».



Como podemos ver, tenemos bien divididas las 3 tareas:

* Controller se encarga de recibir nuestra solicitud.
* Nuestra clase Posteo, es el model que necesitamos para trabajar con objetos de ese tipo.
* Y PosteoRepository cumple como si fuese una «base de datos» que nos ofrece los distintos posteos con los que contamos.

Ahora, como podemos ver en el controller, ya no es esta capa la encargada de traer todos los posteos y devolverlos, sino que es el repository quien lo hace… ¿Y cómo logramos esto? Creando un objeto de PosteoRepository SIN INSTANCIARLO, sino que lo mapeamos mediante @Autowired. @Autowired en este caso se va a encargar de crear la instancia necesaria y delegar la responsabilidad de traer los datos al repositorio, para que desde el método «traerTodos» de nuestro controller, podamos acceder a los datos que nos brinda.

De esta manera, una vez que levantemos nuestra aplicación y se haga la solicitud en cuestión, recibiremos como respuesta los posteos que teníamos en el repository:

