# GUÍA de ANOTACIONES

## Introducción

El lenguaje de programación Java proporcionó soporte para anotaciones desde Java 5.0 en adelante. Los principales frameworks de Java adoptaron rápidamente anotaciones, y Spring Framework comenzó a usar anotaciones de la versión 2.5. Debido a la forma en que se definen, las anotaciones proporcionan mucho contexto en su declaración.

Antes de las anotaciones, el comportamiento de Spring Framework se controlaba en gran medida a través de la configuración XML. Hoy, el uso de anotaciones nos proporciona capacidades tremendas en la forma en que configuramos los comportamientos de Spring Framework.

A continuación, se listarán las anotaciones disponibles en Spring Framework.

## Anotaciones Core Spring Framework

### @Required

Esta anotación se aplica a los métodos de "setters" de beans. Si se considera un escenario en el se que necesita hacer cumplir una propiedad requerida. La anotación **@Required** indica que el bean afectado debe llenarse en el momento de la configuración con la propiedad requerida. De lo contrario, se genera una excepción de tipo ***BeanInitializationException.***

### @Autowired

Esta anotación se aplica a campos, métodos de "setters" y constructores. La anotación **@Autowired** inyecta la dependencia del objeto implícitamente.

Cuando se usa **@Autowired** en los campos y se pasen los valores de los campos con el nombre de la propiedad, Spring asignará automáticamente los campos con los valores que se pasan.

```
@Autowired
```

Incluso se puede usar **@Autowired** en propiedades privadas, como se muestra a continuación.

```
1  public class Customer {
2      @Autowired
       private Person person;
```

```
3         private int type;
4     }
5
```

Cuando usa @Autowired en métodos “setter”, Spring intenta realizarlo mediante Type autowiring en el método. Le está indicando a Spring que debe iniciar esta propiedad utilizando un método de “setter” donde se puede agregar su código personalizado, como inicializar cualquier otra propiedad con esta propiedad.

```
1
2     public class Customer {
3         private Person person;
4         @Autowired
5         public void setPerson (Person person) {
6             this.person=person;
7         }
8     }
```

Si se considera un escenario en el que necesita una instancia de clase A, pero no almacena A en el campo de la clase, este simplemente usa A para obtener una instancia de B, y está almacenando B en este campo. En este caso, **@Autowired** automáticamente del método setter se adaptará mejor. No tendrá campos no utilizados a nivel de clase.

Cuando se usa **@Autowired** en un constructor, la inyección del constructor ocurre en el momento de la creación del objeto. Le dice al constructor que se conecte automáticamente cuando se usa como un bean. Una cosa a tener en cuenta aquí es que solo un constructor de cualquier clase de bean puede llevar la anotación **@Autowired.**

```
1
2     @Component
3     public class Customer {
4         private Person person;
5         @Autowired
6         public Customer (Person person) {
7             this.person=person;
8         }
9     }
```

*Nota:* A partir de Spring 4.3, **@Autowired** se convirtió en opcional en las clases con un solo constructor. En el ejemplo anterior, Spring aún inyectaría una instancia de la clase Person si omitiera la anotación **@Autowired.**

## @Qualifier

Esta anotación se usa junto con la anotación **@Autowired**. Cuando necesita más control del proceso de inyección de dependencia, se puede utilizar **@Qualifier**. **@Qualifier** se puede especificar en argumentos de constructor individuales o parámetros de método. Esta anotación se utiliza para evitar la confusión que ocurre cuando crea más de un bean del mismo tipo y se desea conectar solo uno de ellos con una propiedad.

Considere un ejemplo donde una interfaz BeanInterface es implementada por dos beans, BeanB1 y BeanB2.

```
@Component
public class BeanB1 implements BeanInterface {
    //
}
@Component
public class BeanB2 implements BeanInterface {
    //
}
```

Ahora, si BeanA automatiza esta interfaz, Spring no sabrá cuál de las dos implementaciones inyectar.

Una solución a este problema es el uso de la anotación **@Qualifier**.

```
@Component
public class BeanA {
    @Autowired
    @Qualifier("beanB2")
    private IBean dependency;
    ...
}
```

Con la anotación **@Qualifier** agregada, Spring ahora sabrá qué bean conectar automáticamente, donde beanB2 es el nombre de BeanB2.

## @Configuration

Esta anotación se usa en clases que definen beans**. @Configuration** es un análogo para un archivo de configuración XML: se configura mediante clases Java. Una clase Java anotada con **@Configuration** es una configuración en sí misma y tendrá métodos para crear instancias y configurar las dependencias.

Aquí hay un ejemplo:

```
@Configuartion
public class DataConfig {
    @Bean
    public DataSource source() {
        DataSource source = new OracleDataSource();
        source.setURL();
        source.setUser();
        return source;
    }
    @Bean
    public PlatformTransactionManager manager() {
                PlatformTransactionManager manager = new
BasicDataSourceTransactionManager();
        manager.setDataSource(source());
        return manager;
    }
}
```

## @ComponentScan

Esta anotación se usa con la anotación **@Configuration** para permitir que Spring conozca los paquetes para buscar componentes anotados**. @ComponentScan** también se utiliza para especificar paquetes base usando basePackageClasses o basePackage son atributos para escanear. Si no se definen paquetes específicos, el escaneo se realizará desde el paquete de la clase que declara esta anotación.

## @Bean

Esta anotación se utiliza a nivel de método. La anotación **@Bean** funciona con @Configuration para crear beans Spring. Como se mencionó anteriormente, *@Configuration* tendrá métodos para crear instancias y configurar dependencias. Dichos métodos serán anotados con **@Bean**. El método anotado con esta anotación funciona como la ID del bean, y crea y devuelve el bean real.

Aquí hay un ejemplo:

```
1
2     @Configuration
3     public class AppConfig {
4         @Bean
5         public Person person() {
6             return new Person(address());
7         }
8         @Bean
9         public Address address() {
10            return new Address();
11        }
12    }
13
```

## @Lazy

Esta anotación se usa en clases de componentes. De forma predeterminada, todas las dependencias con conexión automática se crean y configuran al inicio. Pero si desea inicializar un bean "como una carga diferida", se puede usar la anotación **@Lazy** sobre la clase. Esto significa que el bean se creará e inicializará solo cuando se solicite por primera vez. También puede usar esta anotación en las clases de **@Configuration**. Esto indica que todos los métodos de **@Bean** dentro de esa **@Configuración** deben inicializarse "como una carga diferida".

## @Value

Esta anotación se utiliza en los niveles de campo, parámetro de constructor y parámetro de método. La anotación **@Value** indica una expresión de valor predeterminado para el campo o parámetro para inicializar la propiedad. Como la anotación **@Autowired** le dice

a Spring que inyecte un objeto en otro cuando carga el contexto de su aplicación, también puede usar la anotación **@Value** para inyectar valores de un archivo de propiedades en el atributo de un bean. Admite marcadores de posición # {…} y $ {…}.

# Spring Framework Stereotype Annotations

## @Component

Esta anotación se usa en clases para indicar un componente Spring. La anotación **@Component** marca la clase Java como un bean o componente para que el mecanismo de exploración de componentes de Spring pueda agregarla al contexto de la aplicación.

## @Controller

La anotación **@Controller** se usa para indicar que la clase es un controlador Spring. Esta anotación se puede utilizar para identificar controladores para Spring MVC o Spring WebFlux.

## @Service

Esta anotación se usa en una clase. **@Service** marca una clase Java que realiza algún servicio, como ejecutar lógica de negocios, realizar cálculos y llamar a API externas. Esta anotación es una forma especializada de la anotación **@Component** destinada a ser utilizada en la capa de servicio.

## @Repository

Esta anotación se utiliza en clases Java que acceden directamente a la base de datos. La anotación **@Repository** funciona como un marcador para cualquier clase que cumpla la función de repositorio u Objeto de acceso a datos.

Esta anotación tiene una función de traducción automática. Por ejemplo, cuando ocurre una excepción en el **@Repository**, hay un controlador para esa excepción y no es necesario agregar un bloque try-catch.

# Spring Boot Annotations

## @EnableAutoConfiguration

Esta anotación generalmente se coloca en la clase de aplicación principal. La anotación **@EnableAutoConfiguration** define implícitamente un "paquete de búsqueda" base. Esta anotación le dice a Spring Boot que comience a agregar beans según la configuración de classpath, otros beans y varias configuraciones de propiedades.

### @SpringBootApplication

Esta anotación se utiliza en la clase de aplicación al configurar un proyecto Spring Boot. La clase anotada con **@SpringBootApplication** debe mantenerse en el paquete base. Lo único que hace **@SpringBootApplication** es un escaneo de componentes. Pero escaneará solo sus subpaquetes. Como ejemplo, si coloca la clase anotada con **@SpringBootApplication** en el ejemplo com, entonces **@SpringBootApplication** escaneará todos sus subpaquetes, como com.mvit.a, com.apps.b y com.apps.a.x.

**@SpringBootApplication** es una anotación conveniente que agrega todo lo siguiente:

@Configuración

@EnableAutoConfiguration

@ComponentScan

## Spring MVC and REST Annotations

### @Controller

Esta anotación se utiliza en clases Java que desempeñan el papel de controlador en su aplicación. La anotación **@Controller** permite la autodetección de clases de componentes en el classpath y las definiciones de bean de registro automático para ellas. Para habilitar la detección automática de dichos controladores anotados, puede agregar el escaneo de componentes a su configuración. La clase Java anotada con **@Controller** es capaz de manejar múltiples asignaciones de solicitudes.

Esta anotación se puede usar con Spring MVC y Spring WebFlux.

### @RequestMapping

Esta anotación se usa tanto a nivel de clase como de método. La anotación **@RequestMapping** se utiliza para asignar solicitudes web a clases de manejador y métodos de manejador específicos. Cuando **@RequestMapping** se usa en el nivel de clase, crea un URI base para el que se usará el controlador. Cuando esta

anotación se utiliza en los métodos, le dará el URI en el que se ejecutarán los métodos del controlador. A partir de esto, puede inferir que la asignación de solicitud a nivel de clase seguirá siendo la misma, mientras que cada método de controlador tendrá su propia asignación de solicitud.

A veces es posible que desee realizar diferentes operaciones en función del método HTTP utilizado, aunque el URI de la solicitud puede permanecer igual. En tales situaciones, puede usar el atributo de método de **@RequestMapping** con un valor de método HTTP para limitar los métodos HTTP para invocar los métodos de su clase.

Aquí hay un ejemplo básico de cómo funciona un controlador junto con las asignaciones de solicitudes:

```
1
2   @Controller
3   @RequestMapping("/welcome")
4   public class WelcomeController {
5       @RequestMapping(method = RequestMethod.GET)
6       public String welcomeAll() {
7           return "welcome all";
8       }
    }
```

En este ejemplo, solo el método welcomeAll () maneja las solicitudes GET a / welcome.

Esta anotación también se puede usar con Spring MVC y Spring WebFlux.

## @CookieValue

Esta anotación se utiliza a nivel de parámetro de método. **@CookieValue** se usa como argumento de un método de mapeo de solicitud. La cookie HTTP está vinculada al parámetro **@CookieValue** para un nombre de cookie determinado. Esta anotación se utiliza en el método anotado con **@RequestMapping**.

Si se que el siguiente valor de cookie se recibe con una solicitud HTTP:

```
1   JSESSIONID=418AB76CD83EF94U85YD34W
```

Para obtener el valor de la cookie, use **@CookieValue** de esta manera:

```
1   @ReuestMapping("/cookieValue")
2       public void getCookieValue(@CookieValue "JSESSIONID" String cookie){
3   }
```

## @CrossOrigin

Esta anotación se usa tanto a nivel de clase como de método para habilitar solicitudes de origen cruzado. En muchos casos, el host que sirve JavaScript será diferente del host que sirve los datos. En tal caso, Cross-Origin Resource Sharing (CORS) permite la comunicación entre dominios. Para habilitar esta comunicación, solo necesita agregar la anotación **@CrossOrigin**.

De forma predeterminada, la anotación **@CrossOrigin** permite todo el origen, todos los encabezados, los métodos HTTP especificados en la anotación **@RequestMapping** y una duración máxima de 30 min. Puede personalizar el comportamiento especificando los valores de atributo correspondientes.

A continuación se muestra un ejemplo del uso de **@CrossOrigin** en los niveles de método controlador y controlador:

```
1
2     @CrossOrigin(maxAge = 3600)
3     @RestController
4     @RequestMapping("/account")
5     public class AccountController {
6         @CrossOrigin(origins = "http://ejemplo.com")
7         @RequestMapping("/message")
8         public Message getMessage() {
9             // ...
10        }
11        @RequestMapping("/note")
12        public Note getNote() {
13            // ...
14        }
    }
```

En este ejemplo, los métodos getExample () y getNote () tendrán una edad máxima de 3600 segundos. Además, getExample () solo permitirá solicitudes de origen cruzado de http://ejemplo.com, mientras que getNote () permitirá solicitudes de origen cruzado de todos los hosts.

# @RequestMapping (variantes)

Spring Framework 4.3 introdujo las siguientes variantes de nivel de método de la anotación **@RequestMapping** para expresar mejor la semántica de los métodos anotados. El uso de estas anotaciones se ha convertido en el método estándar para definir los puntos finales. Actúan como envoltorios para **@RequestMapping**.

Estas anotaciones se pueden usar con Spring MVC y Spring WebFlux.

## @GetMapping

Esta anotación se utiliza para asignar solicitudes HTTP GET a métodos de controlador específicos. **@GetMapping** es una anotación compuesta que actúa como un acceso directo para **@RequestMapping** (method = RequestMethod.GET).

## @PostMapping

Esta anotación se utiliza para asignar solicitudes HTTP POST a métodos de controlador específicos. **@PostMapping** es una anotación compuesta que actúa como un acceso directo para **@RequestMapping** (method = RequestMethod.POST).

## @PutMapping

Esta anotación se utiliza para mapear solicitudes HTTP PUT en métodos de manejador específicos. **@PutMapping** es una anotación compuesta que actúa como un acceso directo para **@RequestMapping** (method = RequestMethod.PUT).

## @PatchMapping

Esta anotación se usa para mapear solicitudes HTTP PATCH en métodos de manejador específicos. **@PatchMapping** es una anotación compuesta que actúa como un acceso directo para **@RequestMapping** (method = RequestMethod.PATCH).

## @DeleteMapping

Esta anotación se usa para asignar solicitudes HTTP DELETE a métodos de controlador específicos. **@DeleteMapping** es una anotación compuesta que actúa como un acceso directo para **@RequestMapping** (method = RequestMethod.DELETE).

## @ExceptionHandler

Esta anotación se usa a nivel de método para manejar excepciones a nivel de controlador. La anotación **@ExceptionHandler** se usa para definir la clase de excepción que atrapará. Puede usar esta anotación en los métodos que deben invocarse para manejar una excepción. Los valores de **@ExceptionHandler** se pueden establecer en una matriz de tipos de excepción. Si se produce una excepción que coincide con uno de los tipos de la lista, se invocará el método anotado con el **@ExceptionHandler** correspondiente.

## @InitBinder

Esta anotación es una anotación de nivel de método que desempeña el papel de identificar los métodos que inicializan WebDataBinder, un DataBinder que vincula el parámetro de solicitud a los objetos JavaBean. Para personalizar el enlace de datos de parámetros de solicitud, puede usar métodos anotados **@InitBinder** dentro del controlador. Los métodos anotados con **@InitBinder** incluyen todos los tipos de argumentos que admiten los métodos de controlador.

Se llamarán los métodos anotados de **@InitBinder** para cada solicitud HTTP si no especifica el elemento de valor de esta anotación. El elemento de valor puede ser un nombre de formulario único o múltiple o parámetros de solicitud a los que se aplica el método de enlace de inicio.

## @Mappings and @Mapping

Esta anotación se usa en campos. La anotación **@Mapping** es una metaanotación que indica una anotación de mapeo web. Al asignar diferentes nombres de campo, debe configurar el campo de origen en su campo de destino, y para hacerlo, debe agregar la anotación **@Mappings**. Esta anotación acepta una matriz de **@Mapping** que tiene los campos fuente y destino.

## @MatrixVariable

Esta anotación se utiliza para anotar argumentos del método del controlador de solicitudes para que Spring pueda inyectar los bits relevantes de un URI de matriz. Las variables de matriz pueden aparecer en cualquier segmento, cada una separada por un punto y coma. Si una URL contiene variables de matriz, el patrón de mapeo de solicitud debe representarlas con una plantilla de URI. La anotación **@MatrixVariable** garantiza que la solicitud coincida con las variables de matriz correctas del URI.

## @PathVariable

Esta anotación se utiliza para anotar argumentos del método del controlador de solicitudes.

La anotación **@RequestMapping** se puede usar para manejar cambios dinámicos en el URI donde un cierto valor de URI actúa como parámetro. Puede especificar este parámetro usando una expresión regular. La anotación **@PathVariable** se puede usar para declarar este parámetro.

## @RequestAttribute

Esta anotación se utiliza para vincular el atributo de solicitud a un parámetro de método de controlador. Spring recupera el valor del atributo nombrado para completar el parámetro anotado con **@RequestAttribute**. Mientras que la **@RequestParamannotation** se usa para vincular los valores de los parámetros de una cadena de consulta, **@RequestAttribute** se usa para acceder a los objetos que se han poblado en el lado del servidor.

## @RequestBody

Esta anotación se utiliza para anotar argumentos del método del controlador de solicitudes. La anotación **@RequestBody** indica que un parámetro de método debe estar vinculado al valor del cuerpo de la solicitud HTTP. El HttpMessageConveter es responsable de convertir del mensaje de solicitud HTTP a objeto.

## @RequestHeader

Esta anotación se utiliza para anotar argumentos del método del controlador de solicitudes.

La anotación **@RequestHeader** se usa para asignar el parámetro del controlador para solicitar el valor del encabezado. Cuando Spring asigna la solicitud**, @RequestHeader** verifica el encabezado con el nombre especificado dentro de la anotación y vincula su valor al parámetro del método del controlador. Esta anotación le ayuda a obtener los detalles del encabezado dentro de la clase de controlador.

## @RequestParam

Esta anotación se utiliza para anotar argumentos del método del controlador de solicitudes.

A veces obtienes los parámetros en la URL de la solicitud, principalmente en las solicitudes GET. En ese caso, junto con la anotación @RequestMapping, puede usar la anotación **@RequestParam** para recuperar el parámetro URL y asignarlo al argumento del método. La anotación **@RequestParam** se usa para vincular parámetros de solicitud a un parámetro de método en su controlador.

## @RequestPart

Esta anotación se utiliza para anotar argumentos del método del controlador de solicitudes.

La anotación **@RequestPart** se puede usar en lugar de **@RequestParam** para obtener el contenido de una multiparte específica y vincularlo al argumento del método anotado con **@RequestPart**. Esta anotación tiene en cuenta el encabezado "Content-Type" en la multiparte (parte de solicitud).

## @ResponseBody

Esta anotación se utiliza para anotar métodos de manejo de solicitudes. La anotación **@ResponseBody** es similar a la anotación **@RequestBody**. La anotación **@ResponseBody** indica que el tipo de resultado debe escribirse directamente en el cuerpo de la respuesta en cualquier formato que especifique, como JSON o XML. Spring convierte el objeto devuelto en un cuerpo de respuesta mediante el HttpMessageConveter.

## @ResponseStatus

Esta anotación se utiliza en métodos y clases de excepción. **@ResponseStatus** marca un método o clase de excepción con un código de estado y un motivo que debe devolverse. Cuando se invoca el método del controlador, el código de estado se establece en la respuesta HTTP que anula la información de estado proporcionada por cualquier otro medio. Una clase de controlador también se puede anotar con **@ResponseStatus**, que luego se hereda con todos los métodos **@RequestMapping**.

## @ControllerAdvice

Esta anotación se aplica a nivel de clase. Como se explicó anteriormente, para cada controlador, puede usar **@ExceptionHandler** en un método que se llamará cuando ocurra una excepción dada. Pero esto maneja solo aquellas excepciones que ocurren dentro del controlador en el que está definido. Para superar este problema, ahora puede usar la anotación **@ControllerAdvice**. Esta anotación se utiliza para definir los métodos **@ExceptionHandler, @InitBinder y @ModelAttribute** que se aplican a todos los métodos **@RequestMapping**. Por lo tanto, si define la anotación **@ExceptionHandler** en un método en una clase **@ControllerAdvice**, se aplicará a todos los controladores.

## @RestController

Esta anotación se usa a nivel de clase. La anotación **@RestController** marca la clase como un controlador donde cada método devuelve un objeto de dominio en lugar de una vista. Al anotar una clase con esta anotación, ya no necesita agregar **@ResponseBody** a todos los métodos RequestMapping. Significa que ya no usa resuelve vistas ni envía HTML en respuesta. Simplemente envía el objeto de dominio como una respuesta HTTP en el formato que entienden los consumidores, como JSON.

### @RestControllerAdvice

Esta anotación se aplica a las clases Java. **@RestControllerAdvice** es una anotación de conveniencia que combina **@ControllerAdvice** y **@ResponseBody**. Esta anotación se usa junto con la anotación **@ExceptionHandler** para manejar las excepciones que ocurren dentro del controlador.

### @SessionAttribute

Esta anotación se utiliza a nivel de parámetro de método. La anotación **@SessionAttribute** se usa para vincular el parámetro del método a un atributo de sesión. Esta anotación proporciona un acceso conveniente a los atributos de sesión existentes o permanentes.

### @SessionAttributes

Esta anotación se aplica a nivel de tipo para un controlador específico. La anotación **@SessionAtrributes** se usa cuando desea agregar un objeto JavaBean en una sesión. Esto se utiliza cuando desea mantener el objeto en sesión por un período breve. **@SessionAttributes** se usa junto con **@ModelAttribute**.

Se toma el siguiente ejemplo:

```
1
2   @ModelAttribute("person")
3   public Person getPerson() {}
4   // within the same controller as above snippet
5   @Controller
6   @SeesionAttributes(value = "person", types = {
7       Person.class
8   })
    public class PersonController {}
```

El nombre de **@ModelAttribute** se asigna a **@SessionAttributes** como un valor. **@SessionAttributes** tiene dos elementos. El elemento de valor es el nombre de la sesión en el modelo y el elemento de tipos es el tipo de atributos de sesión en el modelo.

# Spring Cloud Annotations

### @EnableConfigServer

Esta anotación se usa a nivel de clase. Al desarrollar un proyecto con una serie de servicios, debe tener una manera centralizada y directa para configurar y recuperar las configuraciones de todos los servicios que va a desarrollar. Una ventaja de usar un servidor de configuración centralizado es que no necesita cargar con la carga de recordar dónde se distribuye cada configuración entre los componentes múltiples y distribuidos.

Puede usar la anotación **@EnableConfigServer** de Spring Cloud para iniciar un servidor de configuración con el que las otras aplicaciones puedan comunicarse.

## @EnableEurekaServer

Esta anotación se aplica a las clases Java. Un problema que puede encontrar al descomponer su aplicación en microservicios es que a cada servicio le resulta difícil conocer la dirección de todos los demás servicios de los que depende. Llega el servicio de descubrimiento que se encarga de rastrear las ubicaciones de todos los demás microservicios.

Eureka de Netflix es una implementación de un servidor discovery y Spring Boot proporciona la integración. Spring Boot ha facilitado el diseño de un servidor Eureka simplemente anotando la clase de entrada con **@EnableEurekaServer**.

## @EnableDiscoveryClient

Esta anotación se aplica a las clases Java. Para indicarle a cualquier aplicación que se registre en Eureka, solo necesita agregar la anotación **@EnableDiscoveryClient** al punto de entrada de la aplicación. La aplicación que ahora está registrada con Eureka utiliza la abstracción de Spring Cloud Discovery Client para interrogar al registro para su propio host y puerto.

## @EnableCircuitBreaker

Esta anotación se aplica a las clases Java que pueden actuar como disyuntores. El patrón del disyuntor puede permitir que un microservicio continúe funcionando cuando falla un servicio relacionado, evitando que la falla caiga en cascada. Esto también le da al servicio fallido un tiempo para recuperarse.

La clase anotada con **@EnableCircuitBreaker** monitoreará, abrirá y cerrará el interruptor de circuito.

## @HystrixCommand

Esta anotación se utiliza a nivel de método. La biblioteca Hystrix de Netflix proporciona la implementación de un patrón de disyuntor. Cuando aplica el interruptor de circuito a un método, Hystrix observa las fallas del método. Una vez que las fallas se acumulan hasta un umbral, Hystrix abre el circuito para que las llamadas posteriores también fallen. Ahora, Hystrix redirige las llamadas al método, y se pasan a los métodos de reserva especificados.

Hystrix busca cualquier método anotado con la anotación **@HystrixCommand** y lo envuelve en un proxy conectado a un interruptor de circuito para que Hystrix pueda monitorearlo.

Se toma el siguiente ejemplo:

```
1
2     @Service
3     public class BookService {
4         private final RestTemplate restTemplate;
5         public BookService(RestTemplate rest) {
6             this.restTemplate = rest;
7         }
8         @HystrixCommand(fallbackMethod = "newList") public String bookList() {
9             URI uri = URI.create("http://localhost:8081/recommended");
10            return this.restTemplate.getForObject(uri, String.class);
11        }
12        public String newList() {
13            return "Cloud native Java";
14        }
    }
```

Aquí, **@HystrixCommand** se aplica al método original bookList (). La anotación @HystrixCommand tiene newList como método alternativo. Entonces, por alguna razón, si Hystrix abre el circuito en bookList (), tendrá una lista de libros de marcador de posición lista para los usuarios.

# Spring Framework DataAccess Annotations

## @Transactional

Esta anotación se coloca antes de una definición de interfaz, un método en una interfaz, una definición de clase o un método público en una clase. La mera presencia de **@Transactional** no es suficiente para activar el comportamiento transaccional. **@Transactional** son simplemente metadatos que pueden ser consumidos por alguna infraestructura de tiempo de ejecución. Esta infraestructura utiliza los metadatos para configurar los beans apropiados con comportamiento transaccional.

La anotación además admite configuraciones como:

- El tipo de propagación de la transacción.
- El nivel de aislamiento de la transacción.
- Un tiempo de espera para la operación envuelta por la transacción.

Una marca de solo lectura: una pista para el proveedor de persistencia de que la transacción debe ser de solo lectura Las reglas de reversión para la transacción.

# Cache-Based Annotations

## @Cacheable

Esta anotación se usa en métodos. La forma más sencilla de habilitar el comportamiento de la memoria caché para un método es anotarlo con **@Cacheable** y parametrizarlo con el nombre de la memoria caché donde se almacenarían los resultados.

```
1    @Cacheable("addresses")
2    public String getAddress(Book book){...}
```

En el fragmento anterior, el método getAddress está asociado con las direcciones con nombre de caché. Cada vez que se llama al método, se verifica el caché para ver si la invocación ya se ha ejecutado y no tiene que repetirse.

## @CachePut

Esta anotación se usa en métodos. Siempre que necesite actualizar el caché sin interferir en la ejecución del método, puede usar la anotación **@CachePut**. Es decir, el método siempre se ejecutará y el resultado se almacenará en caché.

```
1    @CachePut("addresses")
2    public String getAddress(Book book){...}
```

Se desaconseja utilizar **@CachePut** y **@Cacheable** en el mismo método, ya que el primero fuerza la ejecución para ejecutar una actualización de caché, el segundo hace que la ejecución del método se omita al usar el caché.

## @CacheEvict

Esta anotación se usa en métodos. No es que siempre desee llenar el caché con más y más datos. A veces, es posible que desee eliminar algunos datos de caché para poder llenar el caché con algunos valores nuevos. En tal caso, use la anotación **@CacheEvict**.

```
1    @CachePut("addresses")
2    public String getAddress(Book book){...}
```

Aquí, se usa un elemento adicional, allEntries, junto con el nombre del caché que se vaciará. Se establece en verdadero para que borre todos los valores y se prepare para contener nuevos datos.

### @CacheConfig

Esta anotación es una anotación de nivel de clase. La anotación **@CacheConfig** ayuda a optimizar parte de la información de la memoria caché en un solo lugar. Colocar esta anotación en una clase no activa ninguna operación de almacenamiento en caché. Esto le permite almacenar la configuración de la memoria caché a nivel de clase para que no tenga que declarar cosas varias veces.

## Task Execution and Scheduling Annotations

### @Scheduled

Esta anotación es una anotación de nivel de método. La anotación **@Scheduled** se usa en métodos junto con los metadatos de activación. Un método con **@Scheduled** debe tener un tipo de retorno nulo y no debe aceptar ningún parámetro.

Hay diferentes formas de usar la anotación **@Scheduled**:

```
1  @Scheduled(fixedDelay=5000)
2  public void doSomething() {
3      // something that should execute periodically
```

En este caso, la duración entre el final de la última ejecución y el inicio de la próxima ejecución es fija. Las tareas siempre esperan hasta que finalice la anterior.

```
1  @Scheduled(fixedRate=5000)
2  public void doSomething() {
3      // something that should execute periodically
4  }
```

En este caso, el comienzo de la ejecución de la tarea no espera la finalización de la ejecución anterior.

```
1  @Scheduled(initialDelay=1000,fixedRate=5000)
2  public void doSomething() {
3     // something that should execute periodically after an initial delay
```

La tarea se ejecuta inicialmente con un retraso y luego continúa con la velocidad fija especificada.

### @Async

Esta anotación se utiliza en métodos para ejecutar cada método en un hilo separado. La anotación **@Async** se proporciona en un método para que la invocación de ese método se produzca de forma asincrónica. A diferencia de los métodos anotados con **@Scheduled**, los métodos anotados con **@Async** pueden tomar argumentos. Serán

invocados de la manera normal por los llamantes en tiempo de ejecución en lugar de por una tarea programada.

**@Async** se puede usar tanto con métodos de tipo de retorno nulo como con métodos que devuelven un valor. Sin embargo, los métodos con valores de retorno deben tener un valor de retorno de tipo futuro.

# Spring Framework Testing Annotations

## @BootstrapWith

Esta anotación es una anotación de nivel de clase. La anotación **@BootstrapWith** se usa para configurar cómo se arranca Spring TestContext Framework. Esta anotación se utiliza como metadatos para crear anotaciones compuestas personalizadas y reducir la duplicación de la configuración en un conjunto de pruebas.

## @ContextConfiguration

Esta anotación es una anotación de nivel de clase que define los metadatos utilizados para determinar qué archivos de configuración usar para cargar el ApplicationContext para su prueba. Más específicamente, **@ContextConfiguration** declara las clases anotadas que se utilizarán para cargar el contexto. También puede decirle a Spring dónde ubicar el archivo.

```
1 @ContextConfiguration(locations={"example/test-context.xml", loader =
  Custom ContextLoader.class})
```

## @WebAppConfiguration

Esta anotación es una anotación de nivel de clase. **@WebAppConfiguration** se utiliza para declarar que el ApplicationContext cargado para una prueba de integración debe ser un WebApplicationContext. Esta anotación se utiliza para crear la versión web del contexto de la aplicación. Es importante tener en cuenta que esta anotación debe usarse con la anotación **@ContextConfiguration**. La ruta predeterminada a la raíz de la aplicación web es src / main / webapp. Puede anularlo pasando una ruta diferente a <span class = “theme: classic lang: default decode: true crayon-inline”> @ WebAppConfiguration.

## @Timed

Esta anotación se usa en métodos. La anotación **@Timed** indica que el método de prueba anotado debe finalizar su ejecución en el período de tiempo especificado (en

milisegundos). Si la ejecución excede el tiempo especificado en la anotación, la prueba falla.

```
1    @Timed(millis=10000)
2    public void testLongRunningProcess() {  ... }
```

En este ejemplo, la prueba fallará si excede los 10 segundos de ejecución.

## @Repeat

Esta anotación se utiliza en métodos de prueba. Si desea ejecutar un método de prueba varias veces seguidas automáticamente, puede usar la anotación **@Repeat**. El número de veces que se debe ejecutar el método de prueba se especifica en la anotación.

```
1    @Repeat(10)
2    @Test
3    public void testProcessRepeatedly() {  ... }
```

En este ejemplo, la prueba se ejecutará 10 veces.

## @Commit

Esta anotación se puede usar como anotación de nivel de clase o de nivel de método. Después de la ejecución de un método de prueba, la transacción del método de prueba transaccional se puede confirmar utilizando la anotación **@Commit**. Esta anotación transmite explícitamente la intención del código. Cuando se usa a nivel de clase, esta anotación define el compromiso para todos los métodos de prueba dentro de la clase. Cuando se declara como una anotación a nivel de método, **@Commit** especifica la confirmación para métodos de prueba específicos que anulan la confirmación a nivel de clase.

## @RollBack

Esta anotación se puede usar como anotación tanto a nivel de clase como a nivel de método. La anotación **@RollBack** indica si la transacción de un método de prueba transaccional debe revertirse después de que la prueba complete su ejecución. Si esto es cierto, **@Rollback** (verdadero), la transacción se revierte. De lo contrario, la transacción se confirma. **@Commit** se utiliza en lugar de **@RollBack** (falso).

Cuando se usa a nivel de clase, esta anotación define la reversión de todos los métodos de prueba dentro de la clase.

Cuando se declara como una anotación de nivel de método, **@RollBack** especifica la reversión para métodos de prueba específicos que anulan la semántica de reversión de nivel de clase.

## @DirtiesContext

Esta anotación se usa como anotación tanto a nivel de clase como a nivel de método. **@DirtiesContext** indica que Spring ApplicationContext ha sido modificado o dañado de alguna manera y debería cerrarse. Esto activará la recarga del contexto antes de la ejecución de la próxima prueba. ApplicationContext se marca como sucio antes o después de cualquier método anotado, así como antes o después de la clase de prueba actual.

La anotación @DirtiesContext admite los modos BEFORE_METHOD, BEFORE_CLASS y BEFORE_EACH_TEST_METHOD para cerrar ApplicationContext antes de una prueba.

*Nota:* Evitar el uso excesivo de esta anotación. Es una operación costosa y si se abusa de ella, realmente puede ralentizar su conjunto de pruebas.

## @BeforeTransaction

Esta anotación se utiliza para anotar métodos nulos en la clase de prueba. Los métodos anotados de **@BeforeTransaction** indican que deben ejecutarse antes de que cualquier transacción comience a ejecutarse. Eso significa que el método anotado con **@BeforeTransaction** debe ejecutarse antes de cualquier método anotado con **@Transactional**.

## @AfterTransaction

Esta anotación se utiliza para anotar métodos nulos en la clase de prueba. Los métodos anotados de **@BeforeTransaction** indican que deben ejecutarse antes de que cualquier transacción comience a ejecutarse. Eso significa que el método anotado con **@BeforeTransaction** debe ejecutarse antes de cualquier método anotado con **@Transactional**.

## @Sql

Esta anotación se puede declarar en una clase de prueba o método de prueba para ejecutar scripts SQL en una base de datos. La anotación **@Sql** configura la ruta del recurso a los scripts SQL que deben ejecutarse en una base de datos dada antes o después de un método de prueba de integración. Cuando **@Sql** se usa a nivel de método, anulará cualquier @Sqldefinido en el nivel de clase.

## @SqlConfig

Esta anotación se usa junto con la anotación **@Sql**. La anotación **@SqlConfig** define los metadatos que se utilizan para determinar cómo analizar y ejecutar scripts SQL

configurados a través de la anotación **@Sql**. Cuando se usa a nivel de clase, esta anotación sirve como configuración global para todos los scripts SQL dentro de la clase de prueba. Pero cuando se usa directamente con el atributo de configuración de **@Sql**, **@SqlConfig** sirve como una configuración local para los scripts SQL declarados.

## @SqlGroup

Esta anotación se usa en métodos. La anotación **@SqlGroup** es una anotación de contenedor que puede contener varias anotaciones **@Sql**. Esta anotación puede declarar anotaciones anidadas de **@Sql**.

Además, **@SqlGroup** se utiliza como una metaanotación para crear anotaciones compuestas personalizadas. Esta anotación también se puede usar junto con anotaciones repetibles, donde **@Sql** se puede declarar varias veces en el mismo método o clase.

## @SpringBootTest

Esta anotación se utiliza para iniciar el contexto Spring para las pruebas de integración. Esto mostrará el contexto completo de autoconfiguración.

## @DataJpaTest

La anotación **@DataJpaTest** solo proporcionará la configuración automática requerida para probar Spring Data JPA utilizando una base de datos en memoria como H2.

Esta anotación se usa en lugar de **@SpringBootTest**.

## @DataMongoTest

**@DataMongoTest** proporcionará una configuración automática mínima y un MongoDB integrado para ejecutar pruebas de integración con Spring Data MongoDB.

## @WebMVCTest

**@WebMVCTest** mostrará un contexto de servlet simulado para probar la capa MVC. Los servicios y componentes no se cargan en el contexto. Para proporcionar estas dependencias para la prueba, normalmente se usa la anotación **@MockBean**.

## @AutoConfigureMockMVC

La anotación **@AutoConfigureMockMVC** funciona de manera muy similar a la anotación **@WebMVCTest**, pero se inicia el contexto completo de Spring Boot.

## @MockBean

**Crea e inyecta un Moc**kito Mock para la dependencia dada.

## @JsonTest

Limitará la configuración automática de Spring Boot a los componentes relevantes para el procesamiento de JSON.