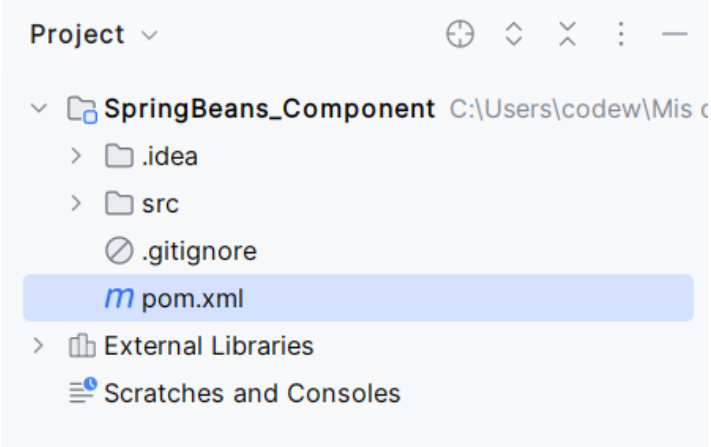
Índice de contenido

Creación de Beans en Spring Framework

1. Declaración de dependencias en Maven

Cuando creamos nuestro proyecto Maven en IntelliJ tendremos por defecto una estructura básica de paquetes y librerías. Entre estos archivos tendremos el archivo de dependencias llamado ***pom.xml*** donde podremos declarar dependencias y propiedades del proyecto.



Nuestro archivo ***pom.xml*** vendrá con una configuración por defecto creada por Maven:

pom.xml (SpringBeans_Component)

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.codewithomar</groupId>
    <artifactId>SpringBeans_Component</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>17</maven.compiler.source>
        <maven.compiler.target>17</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

</project>
```

Para este ejemplo donde crearemos Spring beans necesitaremos declarar dos dependencias:

1. ***spring-context***: Proporciona algunas funcionalidades claves de Spring tales como el *Application Context***,** que gestiona la configuración de la aplicación y la creación de beans. Además, esta dependencia incluye el soporte para *Dependency Injection* (Inyección de dependencias), lo que facilita el desarrollo

2. ***jakarta.annotation-api***: Proporciona anotaciones que se utilizan para proporcionar metadatos sobre el código como la configuración de *@Components* (componentes) en un *IoC Container* (contenedor IoC) y la definición de interceptores para métodos y clases.

Agregando estas dos dependencias a nuestro archivo ***pom.xml*** se podrá ver de esta manera:

pom.xml (SpringBeans_Component)

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.codewithomar</groupId>
    <artifactId>SpringBeans_Component</artifactId>
    <version>1.0-SNAPSHOT</version>

    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>6.0.11</version>
        </dependency>
        <dependency>
            <groupId>jakarta.annotation</groupId>
            <artifactId>jakarta.annotation-api</artifactId>
            <version>2.1.1</version>
        </dependency>
    </dependencies>

    <properties>
        <maven.compiler.source>17</maven.compiler.source>
        <maven.compiler.target>17</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

</project>
```

Dependencias del proyecto

2. Declaración de clase POJO

Un bean se crea a partir de un objeto de una clase POJO (Plain Old Java Object). En palabras simples, una clase java básica con atributos, métodos constructor, getter, setter y toString.

Para nuestro ejemplo usaremos una clase Cliente de la siguiente manera:

```java
package com.codewithomar.beans;

public class Client {
    private String firstName;
    private String lastName;

    public Client() {}

    public Client(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public String getFirstName() {
```

```
15          return firstName;
16      }
17
18      public void setFirstName(String firstName) {
19          this.firstName = firstName;
20      }
21
22      public String getLastName() {
23          return lastName;
24      }
25
26      public void setLastName(String lastName) {
27          this.lastName = lastName;
28      }
29
30      @Override
31      public String toString() {
32          return "Client{" +
33                  "firstName='" + firstName + '\'' +
34                  ", lastName='" + lastName + '\'' +
35                  '}';
36      }
37  }
```

La clase "Client" tiene dos atributos: firstName y lastName. Un constructor vacío y un constructor con los dos atributos como parámetros. Los métodos getter y setter para ambos atributos. Por último, un método toString() por defecto para mostrar los valores de sus dos atributos.

3. IoC utilizando @Component y @Bean

Uno de los pilares y fundamentals del framework Spring es el "*Inversion of Control*" usualmente referido con las siglas "IoC".

Normalmente cuando necesitamos crear objetos de nuestras clases en nuestros programas java es necesario utilizar el operador ***new*** y nosotros mismos como desarrolladores implementamos la lógica detrás de como queremos que estos objetos sean creados a través del código que escribimos.

```
Client client = new Client();
client.setFirstName("Omar");
client.setLastName("Montoya");
System.out.println(client.toString());
```

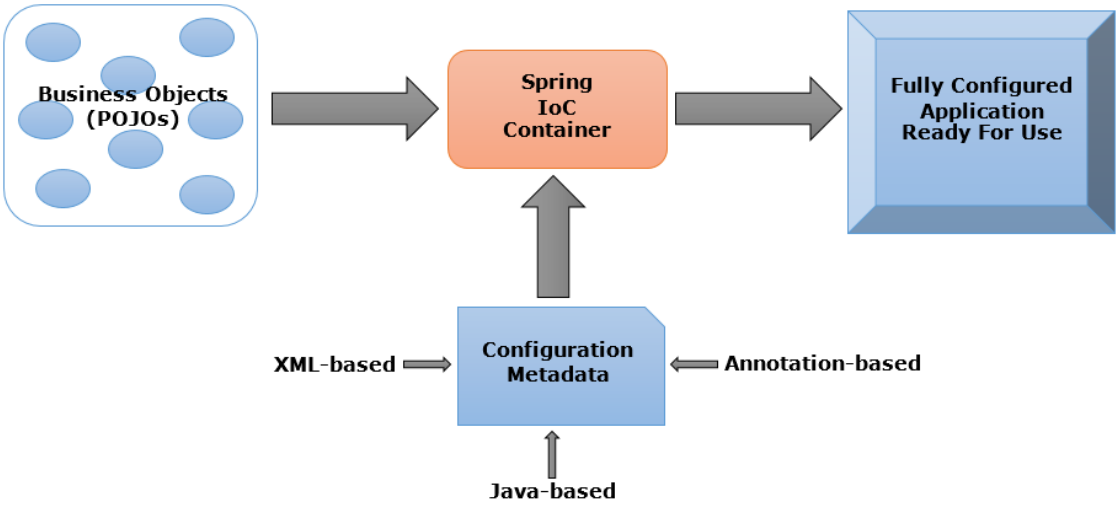
Creación y uso de un objeto "client" fuera de Spring

Cuando utilizamos el framework Spring, la implementación y la lógica de cómo se crean los objetos se abstrae del desarrollador y se delega al framework Spring. En palabras simples, en vez de que el desarrollador declare manualmente toda la lógica sobre cómo se crearan los objetos en nuestro programa (como usualmente se haría en un programa java) esta tarea es delegada a Spring.
Por esta razón, se le conoce como inversión de control. En vez de que el desarrollador tenga el control de los objetos del programa (llamado beans en Spring), Spring es el que tendrá el control de los objetos/beans y el desarrollador se encargará de utilizarlos.

3.1. Arquitectura IoC -> IoC Container

Inversion of Control (IoC) además de ser un principio de diseño también representa una arquitectura dentro del framework Spring.

Esta arquitectura está conformado por tres partes: POJOs, Configuration Metadata y IoC Container.



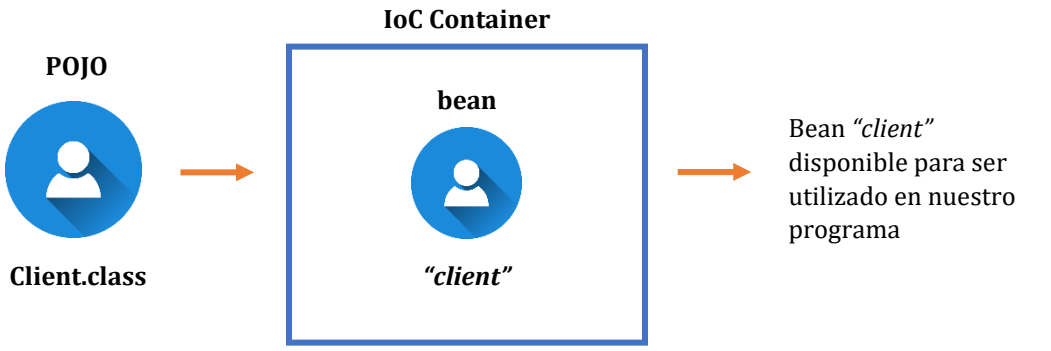
POJOs: Se refieren a las clases de las cuales deseamos crear objetos que sean administrados por Spring. Cuando Spring toma esas clases y crea objetos a partir de estas, pasan a ser consideradas como beans de Spring.

En nuestro programa el POJO que utilizaremos será la clase "Client". Mediante Spring vamos a crear un bean de esta clase llamado "client". Spring se encargará de la administración y creación de ese bean/objeto y nosotros como desarrolladores podremos utilizarlo en nuestro proyecto.

Configuration Metada: Para poder que Spring cree esos beans de nuestros POJOs es necesario cierta configuración. Esta configuración puede proveérsele a Spring de diferentes maneras. Hablaremos a detalle sobre la configuración de la metadata en el siguiente punto.

IoC Container: Es el que finalmente se encarga de administrar y controlar la creación, configuración y gestión de los beans en Spring. Todos los beans serán almacenados en nuestro IoC Container.

En nuestro proyecto si queremos crear un bean de la clase *"Client"* entonces el IoC Container se encargará de crear ese bean y de almacenarlo para que se encuentre disponible para que nosotros luego podremos usarlo.



3.2. Configuración de la metadata

Nosotros como desarrolladores debemos de comunicarle a Spring cuales son las clases de las que queremos crear beans (objetos) para que él los controle y luego nosotros lo utilicemos.

Para configurar los beans dentro de Spring hay diferentes maneras. Entre estas tenemos:

- Annotation-based: En esta configuración le pasaremos al framework la metadata sobre nuestros beans y las clases que queremos utilizar mediante el uso de anotaciones en nuestras clases java. Hoy día es la manera más utilizada. Es la configuración que utilizaremos en nuestro proyecto.
- XML-based: En esta configuración le pasaremos al framework la metadata sobre nuestros beans y clases que queremos utilizar mediante el uso de un archivo XML. Este es el método que se utilizaba en las primeras versiones de Spring y es un método bastante complicado de implementar. Actualmente no se usa en proyectos nuevos por lo que no se utilizará en este proyecto.

3.3. @Component vs @Bean

Una vez decidido que utilizaremos anotaciones para la configuración de nuestros beans, tendremos dos maneras de implementar esas anotaciones: *@Component* o @Bean

@Component	*@Bean*
Manera sencilla de tener un solo bean de una clase.	Manera sencilla de tener múltiples beans de una misma clase.
Spring administrará por completo la creación y configuración de las instancias de la clase como un bean dentro del IoC Container.	El desarrollador podrá definir métodos que producen beans dentro del IoC Container.
No se necesita ninguna configuración especial para la creación del bean. Abstracción completa.	Se provee una configuración específica o especial a Spring sobre cómo crear el bean.
Solo se desea crear el bean a partir de la clase de manera predeterminada.	Se desea crear el bean a partir de una lógica adicional.

3.4. Implementando beans con @Component

En este tipo de implementación de la configuración de los beans tendremos que usar la anotación ***@Component*** sobre la clase POJO que deseamos que Spring cree el bean.

Utilizando nuestra clase *"Client"* se vería de la siguiente manera:

```
1  package com.codewithomar.beans;
2
3  import org.springframework.stereotype.Component;
4
5  @Component
6  public class Client {
7      ...
8  }
```

Con esta simple anotación (línea 5) Spring reconocerá a la clase Client como un candidato para crear un bean y agregarlo al IoC Container.

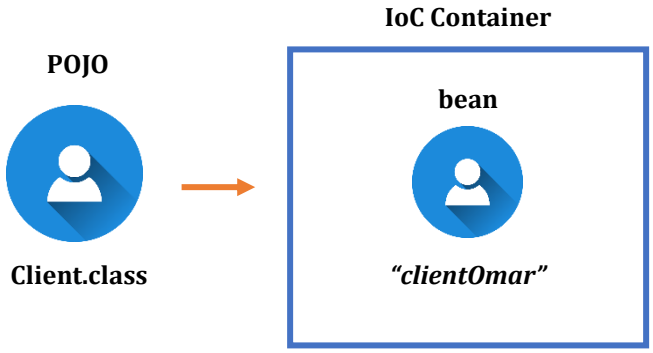
Para que la anotación funcione dentro de nuestro proyecto debemos importar la clase `org.springframework.stereotype.Component` (línea 3).

Por defecto Spring tomará el nombre de la clase como el nombre del bean que va a crear. Como nuestra clase se llama "Client", nuestro bean se llamará "client" (los nombres de los beans se escriben en minúscula).

Si queremos definir un nombre específico para nuestro bean que se creará a partir del component, podremos hacerlo agregando el parámetro "value" a la anotación:

```
1  package com.codewithomar.beans;
2
3  import org.springframework.stereotype.Component;
4
5  @Component(value = "clientOmar")
6  public class Client {
7      ...
8  }
```

De esta manera cuando Spring cree el bean lo llamará *"clientOmar"* y al momento de utilizarlo en nuestro programa si queremos podremos llamarlo por su nombre.



3.4.1. Uso de @PostConstruct

Si bien con la anotación @Component ya le estamos indicando a Spring que queremos crear un bean a partir de esta clase, al momento de la creación de ese bean/objeto el valor de sus atributos se encuentran nulos ya que en ningún momento le asignamos valores.

Para asignarle valores al momento de crear un bean utilizando @Component podemos utilizar otra anotación llamada @PostConstruct.

Con esta anotación podremos ejecutar un método al momento en que se cree un bean de una clase. Podemos utilizar este método para inicializar sus atributos o ejecutar cualquier lógica que deseemos cuando el bean sea creado.

Para utilizarlo lo colocamos arriba del método que deseamos sea ejecutado y dentro de éste método implementamos la lógica que deseamos sea ejecutada:

```
1  package com.codewithomar.beans;
2
3  import jakarta.annotation.PostConstruct;
4  import org.springframework.stereotype.Component;
5
6  @Component(value = "clientOmar")
7  public class Client {
8      ...
9
10     @PostConstruct
11     public void clientInitialize(){
12         this.firstName = "Omar";
13         this.lastName = "Montoya";
14     }
15
16     ...
17 }
```

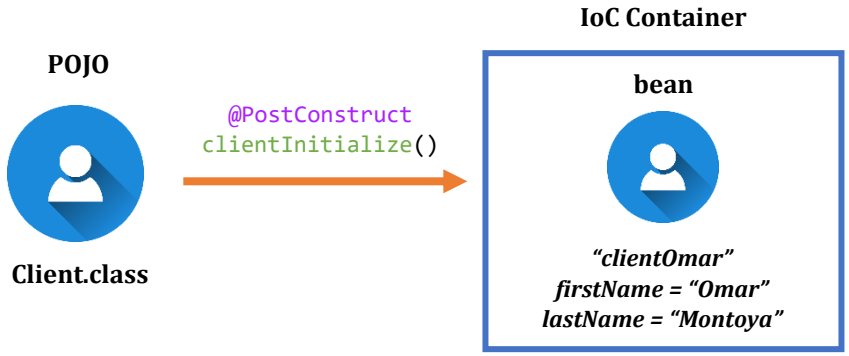
En la línea 10 podemos ver que se utiliza la anotación ***@PostConstruct***

En la línea 11 declaramos el método con acceso *public* y de tipo *void* ya que no retornará ningún valor. El nombre del método no es de importancia ya que puede tener cualquier nombre que deseemos.

En la línea 12 inicializamos el valor del atributo *firstName* con "Omar".

En la línea 13 inicializamos el valor del atributo *lastName* con "Montoya".

De esta manera cuando se cree el bean *"clientOmar"* de esta clase sus dos atributos ya no tendrán valores nulos, sino que tendrán sus valores desde un inicio.



3.4.2. Clase @Configuration cuando utilizamos @Component

Utilizando la anotación ***@Component*** ya le estamos notificando a Spring que queremos crear un bean de esa clase en específico; pero, también Spring necesitará una clase con la anotación ***@Configuration*** donde declararemos todas las configuraciones del ***IoC Container*** de Spring.

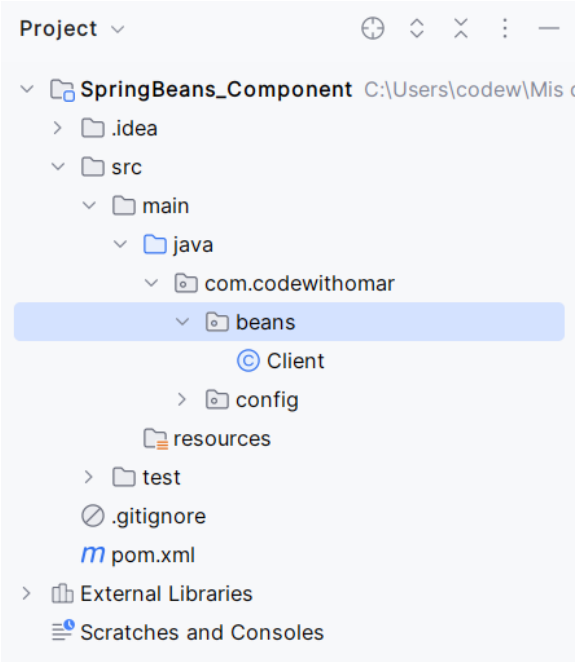
Cuando implementamos ***@Component*** la configuración específica de las creaciones de los beans se abstrae por completo al framework Spring por lo que la configuración que hay que declarar en esta clase ***@Configuration*** es mínima.

```
1  package com.codewithomar.config;
2
3  import org.springframework.context.annotation.ComponentScan;
4  import org.springframework.context.annotation.Configuration;
5
6  @Configuration
7  @ComponentScan(basePackages = "com.codewithomar.beans")
8  public class ProjectConfig {
9  }
```

En la línea 6, tenemos la anotación principal **`@Configuration`**. Esta anotación le indica a Spring que la configuración de nuestro IoC Container se encuentra declarada en la clase **`ProjectConfig`**. Para utilizar esta anotación es necesario importar el paquete: **`org.springframework.context.annotation.Configuration`** (línea 4).

En la línea 7, tenemos la anotación **`@ComponentScan`** para indicarle al **IoC Container** que escanee un paquete y subpaquetes en busca de las clases anotadas con **`@Component`**. Cuando encuentra estas clases, las registra en el **Application Context** como beans. Para utilizar esta anotación es necesario importar el paquete: **`org.springframework.context.annotation.ComponentScan`** (línea 3).

A la anotación **`@ComponentScan`** se le debe especificar donde se encuentran las clases que tienen la anotación **`@Component`**. Para lograr esto se puede utilizar el atributo "**`basePackages`**" al cual se le debe pasar el path/ruta de los paquetes donde se encuentran estas clases. En nuestro proyecto tenemos un paquete llamado "beans" por lo que la ruta que se le pasa al atributo **"`basePackages`**" es `"com.codewithomar.beans"`.

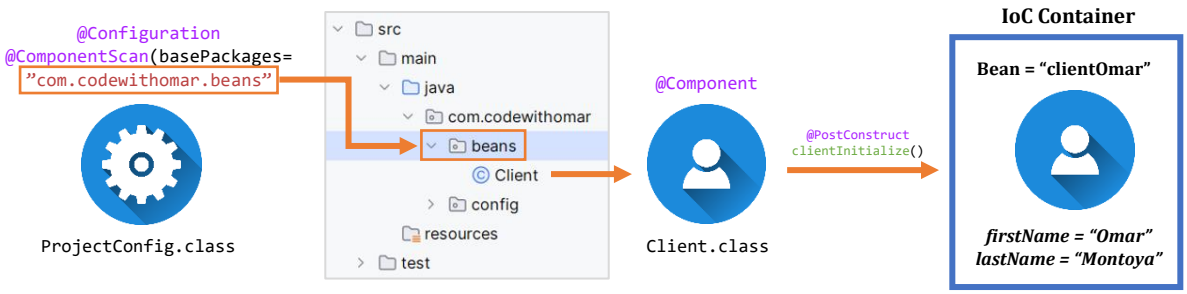


Como usamos **`@Component`** Spring se encarga por completo de la configuración de la creación de los beans por lo que la clase en concreto no tiene ningún código ya que el código en concreto que tiene esa implementación se encuentra dentro de Spring y nosotros como desarrolladores no debemos implementar nada.

3.4.3. Utilización de los beans en el método main

Recapitulando los pasos que hemos seguido:

1. Declaramos nuestras dependencias del proyecto en el pom.xml
2. Creamos nuestra clase POJO y le agregamos la anotación @Component para que Spring sepa que queremos crear beans a partir de esa clase
3. Utilizamos la anotación ***@PostConstruct*** para que se ejecute un método al momento de crear nuestro bean. En nuestro caso lo utilizamos para inicializar dos atributos de tipo String.
4. Creamos una clase ProjectConfig donde usamos la anotación ***@Configuration*** donde se encuentra la configuración del IoC Container. En esta clase, además, tenemos la anotación ***@ComponentScan*** donde le indicaremos al IoC Container donde se encuentran nuestras clases POJOs.



Una vez realizado estos pasos ya tenemos nuestro IoC Container listo para ser utilizado por nuestro programa. Como cualquier programa Java necesitamos declarar una clase con el método main para poder ejecutar el programa. Dentro del método main debemos hacer uso del Application Context para poder utilizar los beans que se encuentran en el IoC Container

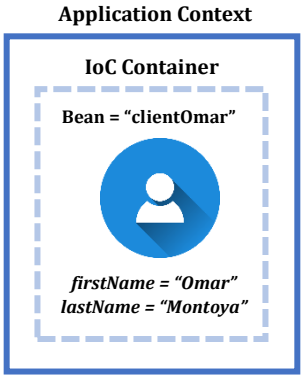
3.4.3.1 Application Context en Spring

El Application Context es un contenedor que contiene y gestiona los beans de una aplicación. Proporciona soporte para la configuración de la aplicación, la inyección de dependencias, la internalización, la gestión de eventos y otros mecanismos que se utilizan para el desarrollo de aplicaciones Java.

El Application Context crea los beans, aplica la configuración definida en la aplicación (En nuestro proyecto utilizamos anotaciones), resolver dependencias entre los beans y proporciona acceso a los beans cuando son necesarios en la aplicación.

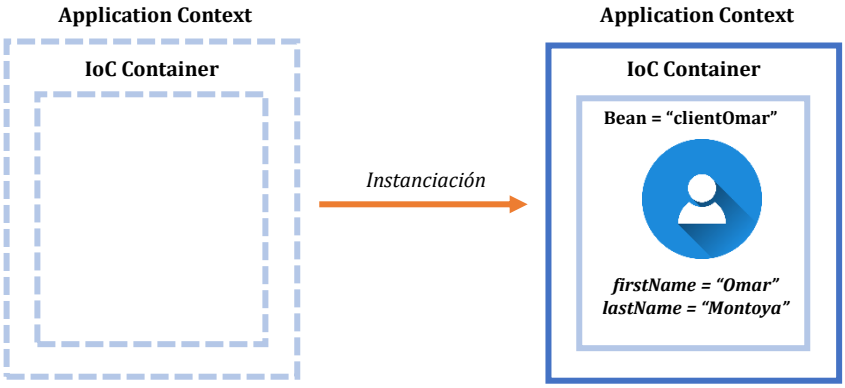
Tal vez te estés preguntando, ¿no se supone que el que hacía todas esas tareas era el IoC Container? La respuesta es sí. Lo que pasa es que el IoC Container es representado por el Application Context. El Application Context abarca varias herramientas entre las cuales se encuentra el IoC Container.

En nuestro proyecto en vez de utilizar directamente el IoC Container, utilizaremos el Application Context que a su vez representará al IoC Container.



3.4.3.2. Instanciación del Application Context

Para poder que Spring busque, inicialice y maneje nuestros Beans (inicialice el IoC container), es necesario crear un objeto del Application Context. Si no creamos un objeto del Application Context, éste no se inicializará y no podremos utilizar nuestros beans en la aplicación.



El *Application Context* se encuentra definido en el paquete de Spring -> `org.framework.context.ApplicationContext`. Sin embargo, `ApplicationContext` es una **interfaz** por lo que no podemos crear un objeto directamente de ella. Por eso debemos crear un objeto de una clase que implemente esa interfaz.

En Spring la interfaz ApplicationContext tiene varias implementaciones principales que proporcionan diferentes funcionalidades y características. Algunas de las implementaciones son:

1. **ClassPathXmlApplicationContext**: Esta implementación carga el contexto de la aplicación desde un archivo XML de configuración ubicado en el classpath de la aplicación.
2. **FileSystemXmlApplicationContext**: Similar a ClassPathXmlApplicationContext, pero carga el contexto de la aplicación desde un archivo XML en el sistema de archivos del sistema operativo.
3. **AnnotationConfigApplicationContext**: Esta implementación permite configurar el contexto de la aplicación utilizando clases anotadas con @Configuration.
4. **GenericApplicationContext**: Esta es una implementación genérica de ApplicationContext que no impone ninguna restricción específica de configuración. Puede ser útil en casos donde se necesita un control más personalizado sobre la configuración del contexto.
5. **WebApplicationContext**: Esta es una interfaz extendida de ApplicationContext diseñada específicamente para aplicaciones web. Tiene implementaciones como

XmlWebApplicationContext y AnnotationConfigWebApplicationContext que son adecuadas para la configuración de aplicaciones web.

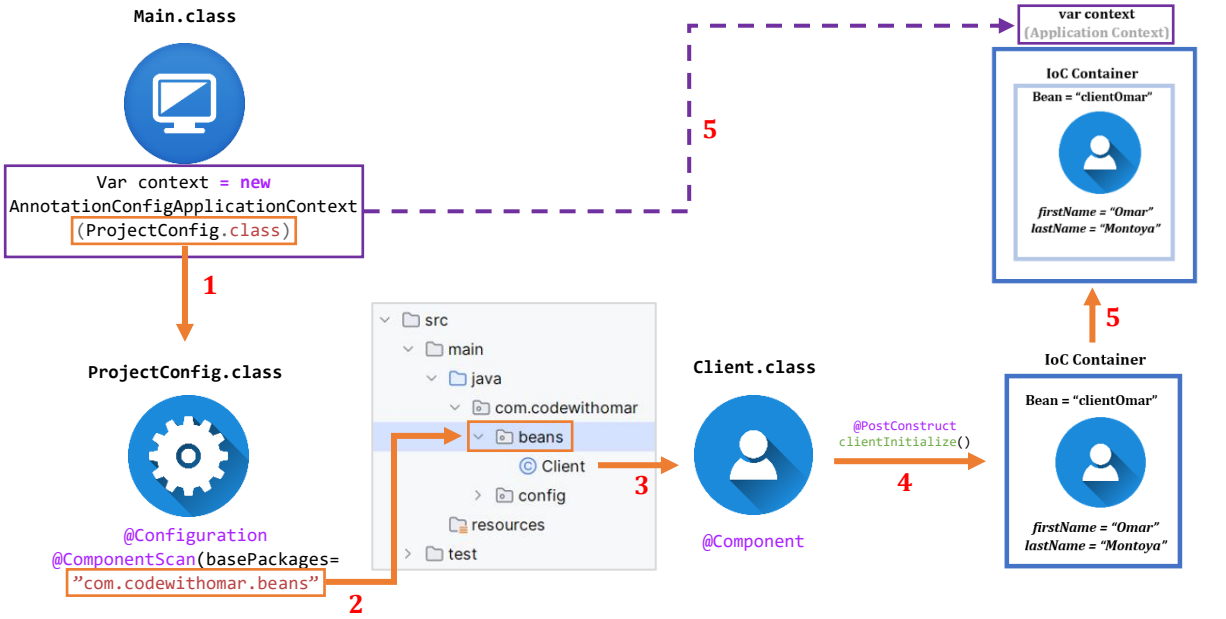
Como has visto en nuestro Proyecto estamos utilizando la configuración en base a las anotaciones. Por lo que la clase que debemos utilizar es -> ***AnnotationConfigApplicationContext***

La creación de un objeto de la clase AnnotationConfigApplicationContext en la clase main puede hacerse de la siguiente manera:

```
1  package com.codewithomar.main;
2
3  import com.codewithomar.config.ProjectConfig;
4  import org.springframework.context.annotation.AnnotationConfigApplicationContext;
5
6  public class Main {
7      public static void main(String[] args) {
8          var context = new AnnotationConfigApplicationContext(ProjectConfig.class);
9      }
10 }
```

En la línea 8, declaramos una variable local "context" del tipo de dato "var". El tipo de dato var fue introducido en Java 10 (Marzo 2018). Luego, nuestra variable local "context" es inicializada con el operador new como un objeto de la clase AnnotationConfigApplicationContext; es decir, hacemos que la variable local "context" sea un objeto de Application Context para poder utilizar nuestros beans que se encontrarán dentro del IoC Container.

Para poder crear el objeto del Application Context, es necesario pasarle como argumento la clase donde se encuentra nuestra configuración (que recordamos debe de ser de tipo Annotation). En nuestro proyecto, tenemos la configuración en la clase "ProjectConfig" por lo que le pasamos `Main.class` "ProjectConfig.class" como el argumento. Para poder utilizar estas declaraciones es necesario importar los paquetes declarados en las líneas 3 y 4.



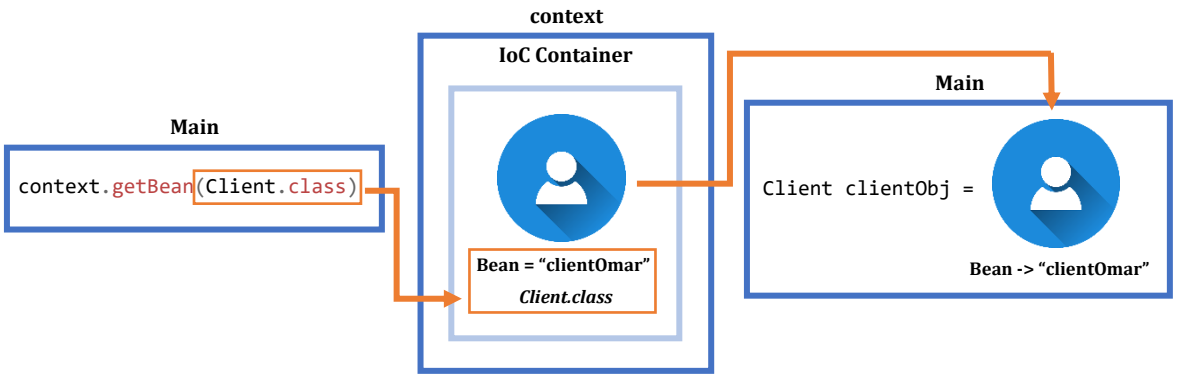
3.4.3.3. Utilización de los beans con el método .getBean()

Una vez creado nuestro objeto Application Context ("context" en nuestro proyecto) podemos utilizar los beans que se encuentren en el IoC Container. Para utilizar estos beans debemos declarar objeto del tipo de la clase correspondiente al bean y en lugar de utilizar el operador new, lo inicializamos utilizando el método .getBean();

```
1  package com.codewithomar.main;
2
3  import com.codewithomar.beans.Client;
4  import com.codewithomar.config.ProjectConfig;
5  import org.springframework.context.annotation.AnnotationConfigApplicationContext;
6
7  public class Main {
8      public static void main(String[] args) {
9          var context = new AnnotationConfigApplicationContext(ProjectConfig.class);
10         Client clientObj = context.getBean(Client.class);
11     }
12 }
```

En la línea 10 tenemos la declaración del objeto "clientObj" del tipo "Client" (clase). Si fuésemos a crear un objeto afuera del contenedor de Spring utilizaríamos el operador new para instanciar la clase e inicializar el objeto. En nuestro caso queremos utilizar el bean que se encuentra dentro de nuestro "context" por lo que usando el método .getBean() podemos acceder a ese bean y asignarlo a un objeto para poder manipularlo.

Al método .getBean() se le pueden pasar diferentes parámetros. En nuestro proyecto necesitamos pasarle solo un parámetro que corresponde al tipo de clase correspondiente al bean (Client.class). Como dentro de nuestro IoC Container podemos tener muchos beans de diferentes clases, Spring necesita que le indiquemos específicamente de qué tipo de clase queremos que nos retorne el bean.



3.4.3.4. Utilización de los métodos del bean

Una vez ejecutado el método .getBean() y asignando el bean a nuestro objeto "clientObj" podremos ejecutar los métodos declarados en la clase "Client".

```
1  package com.codewithomar.main;
2
3  import com.codewithomar.beans.Client;
4  import com.codewithomar.config.ProjectConfig;
5  import org.springframework.context.annotation.AnnotationConfigApplicationContext;
6
7  public class Main {
8      public static void main(String[] args) {
9          var context = new AnnotationConfigApplicationContext(ProjectConfig.class);
10         Client clientObj = context.getBean(Client.class);
11
12         System.out.println("Client first name from Spring context is: " + clientObj.getFirstName());
13         System.out.println("Client last name from Spring context is: " + clientObj.getLastName());
14         System.out.println(clientObj.toString());
15     }
16 }
```

Output por consola

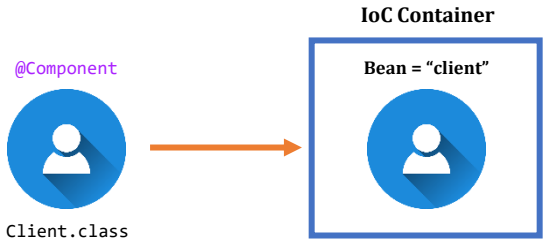
```
"C:\Program Files\Java\jdk-17\bin\java.exe" ...
Client first name from Spring context is: Omar
Client last name from Spring context is: Montoya
Client{firstName='Omar', lastName='Montoya'}

Process finished with exit code 0
```

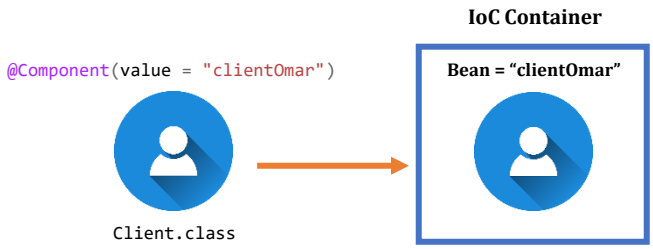
Como podemos ver en las líneas 12, 13 y 14, una vez que le asignamos nuestro bean "clientOmar" al objeto "clientObj" podemos invocar los métodos declarados en nuestra clase POJO .getFirstName(), .getLastName() y .toString(). Recordemos que le asignamos valores a nuestros dos atributos "firstName" y "lastName" con el uso de la anotación @PostConstruct. Si no hubiésemos declarado esta anotación y no hubiésemos creado el método "clientInitialize()" entonces los valores de "firstName" y "lastName" serian nulos.

3.4.3.5. Retornando los nombres de los Beans con .getBeanNamesForType()

Recordemos que cuando utilizamos la anotación @Component por defecto el bean que será creado tendrá el nombre de la clase POJO de cual es creada. En nuestro proyecto la clase se llama "Client" por lo que por defecto el bean tendrá el nombre de "client".



En nuestro proyecto le asignamos un nombre específico al Bean que será creado a partir de la clase "Client" utilizando el argumento @Component(value = "clientOmar"). De esta manera el bean tendrá el nombre de "clientOmar" dentro del IoC Container y del Application Context.



El Application Context tiene un método llamado "getBeanNamesForType" el cual retorna un arreglo de cadenas con los nombres de los beans de un tipo de clase específico. Para especificarle el tipo de clase que deseamos tener los nombres de los beans debemos pasarle el tipo de clase como parámetro.

```
1  package com.codewithomar.main;
2
3  import com.codewithomar.beans.Client;
4  import com.codewithomar.config.ProjectConfig;
5  import org.springframework.context.annotation.AnnotationConfigApplicationContext;
6
7  public class Main {
8      public static void main(String[] args) {
9          var context = new AnnotationConfigApplicationContext(ProjectConfig.class);
10         Client clientObj = context.getBean(Client.class);
11
12         System.out.println("Client first name from Spring context is: " + clientObj.getFirstName());
13         System.out.println("Client last name from Spring context is: " + clientObj.getLastName());
14         System.out.println(clientObj.toString());
15
16         String[] clientBeans = context.getBeanNamesForType(Client.class);
17         for (String beanName : clientBeans) {
18             System.out.println("Bean names: " + beanName);
19         }
20     }
21 }
```

En la línea 16 declaramos un arreglo de tipo String llamado "clientBeans". Al arreglo le asignamos el arreglo de tipo String que retorna el método "getBeanNamesForType(Client.Class)".

Una vez el arreglo "clientBeans" contiene el nombre de todos los beans de la clase "Client" pasamos a imprimirlos en pantalla con el uso de un ciclo for-each (línea 17 y 18).

Output por consola

```
"C:\Program Files\Java\jdk-17\bin\java.exe" ...
Client first name from Spring context is: Omar
Client last name from Spring context is: Montoya
Client{firstName='Omar', lastName='Montoya'}
Bean names: clientOmar

Process finished with exit code 0
```

Podemos ver en la cuarta línea del resultado por consola que nuestro programa imprime el nombre del bean como "clientOmar" confirmando la especificación que le dimos a la anotación @Component en nuestro proyecto.

Debido a que estamos utilizando la anotación @Component Spring solo creará un bean de esta clase. Por esta razón solo se imprime el nombre de un bean ya que dentro de nuestro IoC Container solo tenemos un bean.

En el siguiente punto principal veremos como es la implementación para tener múltiples beans de una misma clase.

3.5. Implementando beans con @Bean

Cuando utilizamos la anotación ***@Bean*** tenemos la posibilidad de crear múltiples beans de una misma clase de manera sencilla. A diferencia de ***@Component***, la implementación de la anotación ***@Bean*** es un poco mas compleja y hay que escribir más código.

3.5.1. Clase POJO con @Bean

A diferencia de la implementación con @Component, cuando utilizamos @Bean no debemos declarar ninguna anotación en la clase POJO a partir de la que queremos crear los beans.

Nuestra clase "Client" con la implementación de @Bean se vería de la siguiente manera

```
1  package com.codewithomar.beans;
2
3  public class Client {
4      private String firstName;
5      private String lastName;
6
7      public Client() {}
8
9      public Client(String firstName, String lastName) {
10         this.firstName = firstName;
11         this.lastName = lastName;
12     }
13
14     public String getFirstName() {
15         return firstName;
16     }
17
18     public void setFirstName(String firstName) {
19         this.firstName = firstName;
20     }
21
22     public String getLastName() {
23         return lastName;
24     }
25
26     public void setLastName(String lastName) {
```

```
27          this.lastName = lastName;
28      }
29
30      @Override
31      public String toString() {
32          return "Client{" +
33                  "firstName='" + firstName + '\'' +
34                  ", lastName='" + lastName + '\'' +
35                  '}';
36      }
37  }
```

Podemos apreciar que en efecto la anotación ***@Component*** no se encuentra declarada y además la anotación ***@PostConstruct*** y el método ***"clientInitialize()"*** tampoco se encuentran declarados. Esto se debe a que la anotación ***@Bean*** en vez de declararse sobre la clase POJO se declarará dentro de la clase ***"ProjectConfig"*** que posee la anotación ***@Configuration***. Además, el método con las indicaciones sobre como crear el bean se declarará de igual manera dentro de la clase ***"ProjectConfig".***

3.5.2. Clase @Configuration cuando utilizamos @Bean

Cuando utilizamos @Bean para la configuración de nuestros beans debemos declarar métodos que creen objetos de nuestros POJOs. Si bien con @Component este proceso se abstraía por completo, con @Bean la abstracción no es tan completa.

Creamos nuestra clase "ProjectConfiguration" y agregamos la anotación @Configuration. Para esta implementación no tenemos que utilizar la anotación @ComponentScan porque no habrá ninguna anotación @Component que escanear.

```
1  package com.codewithomar.config;
2
3  import org.springframework.context.annotation.Configuration;
4
5  @Configuration
6  public class ProjectConfig {
7  }
```

Una vez creada nuestra clase "ProjectConfig" con la anotación @Configuration podemos pasar a implementar nuestras anotaciones @Bean

```
1  package com.codewithomar.config;
2
3  import com.codewithomar.beans.Client;
4  import org.springframework.context.annotation.Bean;
5  import org.springframework.context.annotation.Configuration;
6
7  @Configuration
8  public class ProjectConfig {
```

```
9
10      @Bean(name = "clientOmar")
11      Client client1() {
12          Client client = new Client();
13          client.setFirstName("Omar");
14          client.setLastName("Montoya");
15          return client;
16      }
17
18      @Bean(value = "clientMarlly")
19      Client client2() {
20          Client client = new Client("Marlly", "Guido");
21          return client;
22      }
23
24      @Bean("clientKai")
25      Client client3(){
26          Client client = new Client();
27          client.setFirstName("Kai");
28          return client;
29      }
30
31      @Bean
32      Client client4(){
33          Client client = new Client();
34          return client;
35      }
36  }
```

Primer Bean (Línea 10-16)

Línea 10: Utilizamos la anotación `@Bean` para notificarle a Spring que el método con esta anotación contendrá la configuración que queremos implementar para crear el bean de una clase. Adicionalmente le agregamos el argumento `(name = "clientOmar")` para asignarle un nombre a este bean.

Línea 11: Declaramos un método llamado "`client1()`" con valor de retorno "`Client`" ya que nuestro método deberá retornarnos un objeto de la clase POJO, en nuestro caso la clase "Client".

Línea 12: Declaramos e instanciamos la clase "`Client`" para crear el objeto "`client`".

Línea 13 y 14: Utilizamos los métodos setter de ambos atributos e inicializamos cada uno con los valores.

Línea 15: Como declaramos en la línea 11 que nuestro método retornará un objeto del tipo "`Client`" debemos retornar ese objeto que acabamos de crear.

Segundo Bean (Línea 18-22)

Línea 18: Utilizamos la anotación `@Bean` y le agregamos el argumento `(value = "clientMarlly")` para asignarle un nombre a este bean. Los parámetros "`name`" y "value" del atributo `@Bean` pueden ser utilizados para nombrar a los beans por igual.

Línea 20: Recordemos que nuestra clase POJO tiene dos métodos constructores declarados. El método constructor por defecto vacío sin ningún parámetro y el constructor con dos parámetros de los atributos firstName y lastName. En este bean utilizamos el constructor con los dos parámetros para inicializar sus valores.

Línea 21: retornamos el objeto "`client`" creado en este método

Tercer Bean (Línea 24-29)

Línea 24: Utilizamos la anotación `@Bean` y le agregamos el argumento `("clientKai")` para asignarle un nombre a este bean. Podemos ver que ni siquiera debemos utilizar "`value =`" o "`name =`" para asignarle un valor a nuestro bean.

Línea 26: Utilizamos el constructor por defecto para declarar e inicializar el objeto "`client`".

Línea 27: Utilizamos el método setter del atributo "`firstName`". Como no estamos utilizando el setter del atributo "`lastName`" este tendrá un valor nulo.

Línea 28: retornamos el objeto creado en este método.

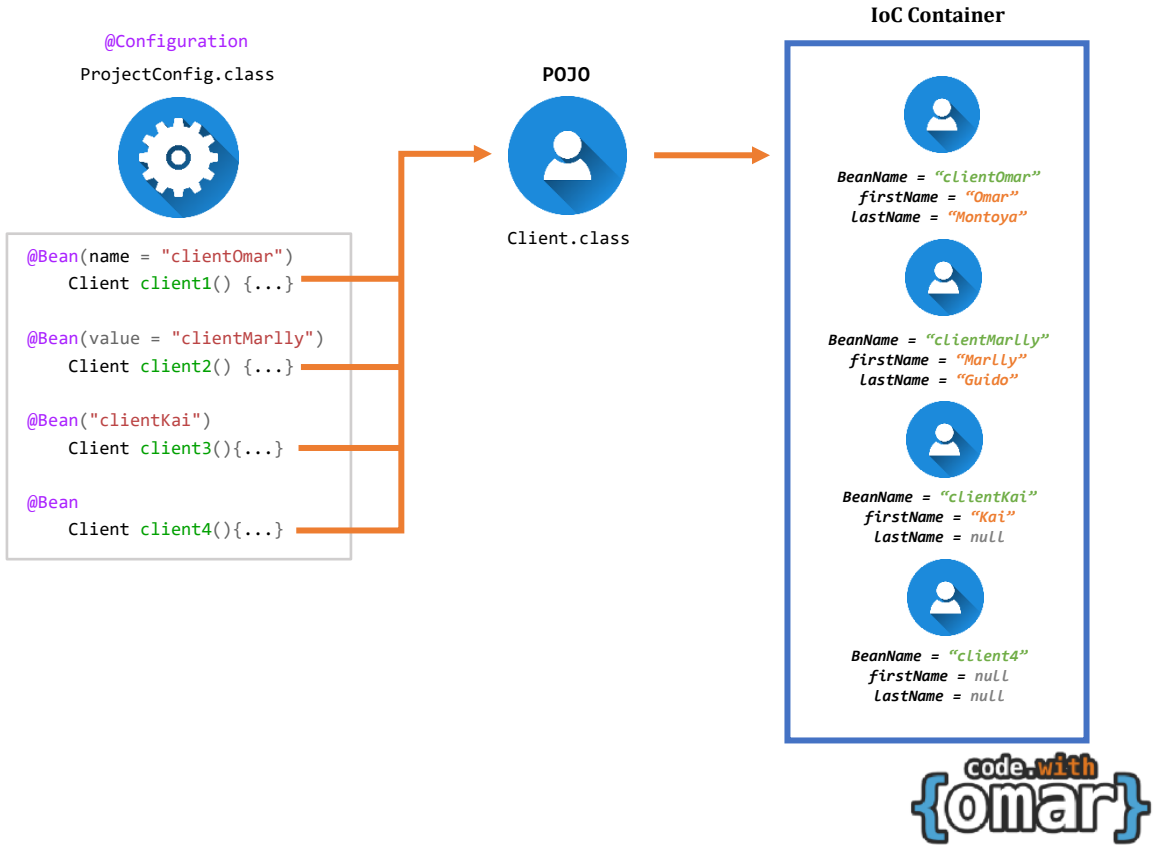
Cuarto Bean (Línea 31-35)

Línea 31: Utilizamos la anotación `@Bean` y esta vez no le agregamos ningún argumento. De esta manera, el bean que será creado a partir de este método tendrá el nombre del método en el que es declarado; es decir, si el método se llama "client4()" (Línea 32) entonces el nombre del cuarto bean será "client4".

Línea 33: Declaración e inicialización del objeto "client".

Línea 34: Retornamos el objeto creado en este método. Cabe destacar que como no utilizamos ningún método setter de ningún atributo ambos tendrán valores nulos.

Diagrama IoC Container con los beans utilizando @Bean



Conclusión

Podemos apreciar que con la implementación de la anotación @Bean tenemos mas libertad y variedad en la definición de cada bean que queremos crear. En nuestro ejemplo solo lo utilizamos para crear los objetos de la clase, pero podrían tener alguna lógica específica para crear los beans a partir de una condición o ejecutar otros métodos no necesariamente relacionados con la instanciación de objetos.

También es una manera bastante sencilla de crear varios beans de una misma clase POJO. Con la anotación @Component también se podría crear varios beans de la misma clase sin embargo tiene una implementación más compleja que la de @Bean.

3.5.3. Utilización de los beans en el método main

Creamos una clase "***Main***" donde declaramos el método main de nuestro proyecto. Dentro del método main declaramos una variable de tipo "`var`" llamada "`context`" donde le asignaremos el ***Application Context***. A pesar de que ***@Bean*** es distinto a ***@Component***, ambos son configuraciones del tipo Annotation por lo que también usaremos la clase "***AnnotationConfigApplicationContext"*** y le pasaremos como argumento la clase donde tenemos nuestra configuración (*`ProjectConfig.Class`*)

```
1  package com.codewithomar.main;
2
3  import com.codewithomar.config.ProjectConfig;
4  import org.springframework.context.annotation.AnnotationConfigApplicationContext;
5
6  public class Main {
7      public static void main(String[] args) {
8          var context = new AnnotationConfigApplicationContext(ProjectConfig.class);
9      }
10 }
```

3.5.3.1. Retornando los nombres de los beans con el método .getBeanNamesForType()

Una vez declarado nuestro Application Context en la variable "context" podemos verificar los nombres de los beans en nuestro IoC Container utilizando el método .getbeanNamesForType(). Recordemos que este método retorna un arreglo de tipo String con todos los nombres de los beans de un tipo de clase en específico.

Declaramos un arreglo de tipo *String* llamado "clientBeans" donde guardaremos los nombres de los beans que retornará el método .getbeanNamesForType(). Luego lo imprimimos en pantalla con un ciclo for-each.

```
1  package com.codewithomar.main;
2
3  import com.codewithomar.beans.Client;
4  import com.codewithomar.config.ProjectConfig;
5  import org.springframework.context.annotation.AnnotationConfigApplicationContext;
6
7  public class Main {
8      public static void main(String[] args) {
9          var context = new AnnotationConfigApplicationContext(ProjectConfig.class);
```

```
10
11          String[] clientBeans = context.getBeanNamesForType(Client.class);
12          for (String beanName : clientBeans) {
13              System.out.println("Bean name: " + beanName);
14          }
15
16      }
17  }
```

Output por consola

```
"C:\Program Files\Java\jdk-17\bin\java.exe" ...
Bean name: clientOmar
Bean name: clientMarlly
Bean name: clientKai
Bean name: client4

Process finished with exit code 0
```

Con la impresión por consola del ciclo for-each podemos confirmar que dentro de nuestro Application Context en el IoC Container se encuentran disponibles los cuatro beans que declaramos dentro de nuestra clase "ProjectConfig". Además, podemos confirmar que fueron guardados con los nombres que le asignamos utilizando los diferentes argumentos a la anotación @Bean (a excepción del cuarto bean que se guardó con el nombre por defecto).

3.5.3.2. Utilización de los beans con el método .getBean()

Al igual que con @Component, para poder utilizar nuestros beans debemos asignarlo a una variable local en el método main. Para lograr esto debemos hacer uso del método .getBean(). Sin embargo, cuando utilizamos @Bean surge un inconveniente a la hora de utilizar este método.

Cuando utilizamos @Component y le pedíamos al método .getBean() que nos retornara el bean del tipo "Client.class" disponible en el IoC Container no debíamos especificarle cual ya que solo existía un (1) bean de ese tipo. Sin embargo, en este proyecto tenemos 4 beans del tipo "Client.class". ¿Qué pasará si ejecutamos el mismo método pidiéndole que nos retorne un bean del tipo "Client.class"?

```
1   package com.codewithomar.main;
2
3   import com.codewithomar.beans.Client;
4   import com.codewithomar.config.ProjectConfig;
5   import org.springframework.context.annotation.AnnotationConfigApplicationContext;
6
7   public class Main {
8       public static void main(String[] args) {
9           var context = new AnnotationConfigApplicationContext(ProjectConfig.class);
10
11          String[] clientBeans = context.getBeanNamesForType(Client.class);
12          for (String beanName : clientBeans) {
```

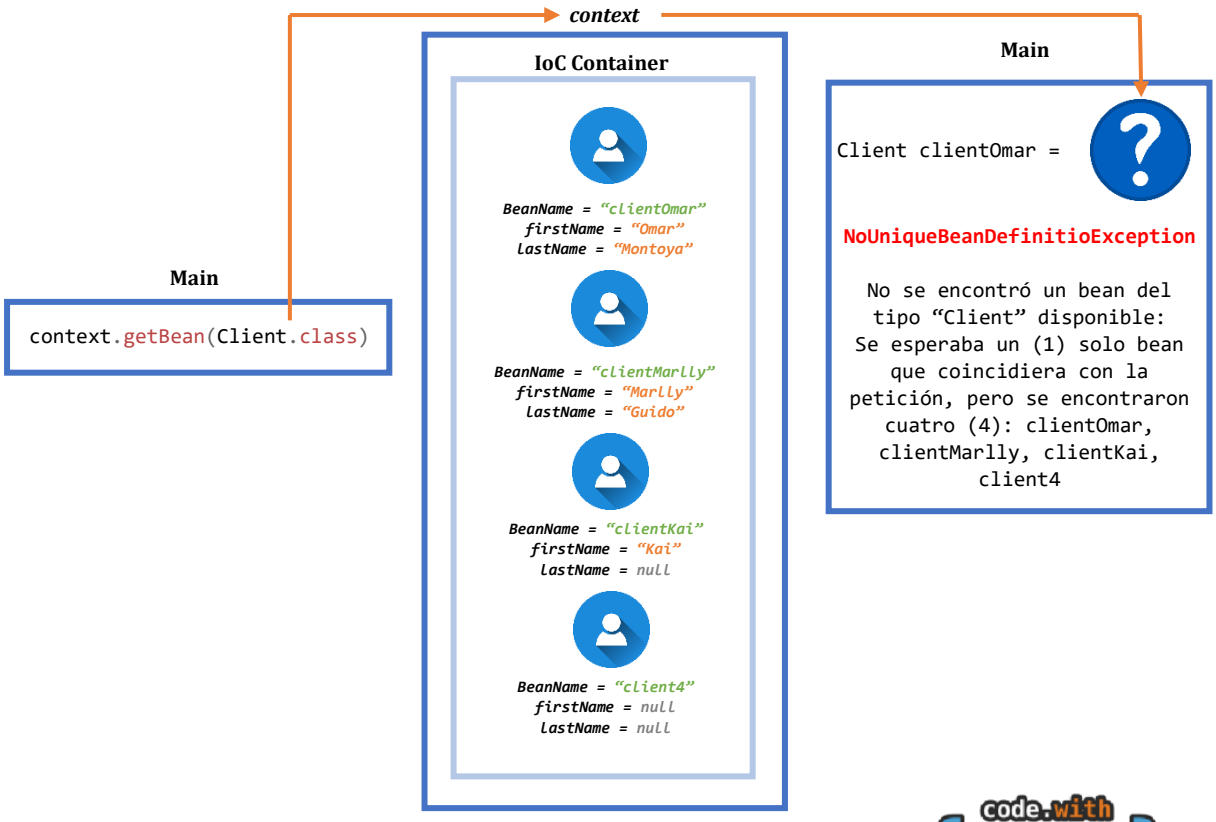
```
13              System.out.println("Bean name: " + beanName);
14          }
15
16          Client clientOmar = context.getBean(Client.class);
17      }
18  }
```

Output por consola

```
"C:\Program Files\Java\jdk-17\bin\java.exe" ...
Bean name: clientOmar
Bean name: clientMarlly
Bean name: clientKai
Bean name: client4
Exception in thread "main"
org.springframework.beans.factory.NoUniqueBeanDefinitionException: No qualifying
bean of type 'com.codewithomar.beans.Client' available: expected single matching
bean but found 4: clientOmar,clientMarlly,clientKai,client4
        at ...

Process finished with exit code 1
```

Cuando ejecutamos la línea 16 del método main se ejecuta la excepción NoUniqueBeanDefinitionException. Esto nos indica que dentro de nuestro IoC container se encuentra más de un bean del mismo tipo y no estamos especificando a Spring cuál de todos ellos es el que queremos que retorne por lo que se produce una ambigüedad de como Spring debe decidir.



3.5.3.3. Soluciones a NoUniqueBeanDefinitionException

Para solucionar esta excepción tenemos dos opciones:

1. Especificarle al método .getBean() el nombre del bean que queremos que nos retorne junto con el tipo de bean.
2. Utilizar la anotación @Primary para que Spring tenga un bean por defecto a utilizar en caso de que no se especifique cuál debemos usar.

Solución 1: .getBean(String name, Class<T> requiredType)

Podemos pasarle dos argumentos al método .getBean() donde el primero será el nombre del bean que queremos utilizar y el segundo será el nombre de la clase del tipo de bean queremos usar.

```
1  package com.codewithomar.main;
2
3  import com.codewithomar.beans.Client;
4  import com.codewithomar.config.ProjectConfig;
5  import org.springframework.context.annotation.AnnotationConfigApplicationContext;
6
7  public class Main {
8      public static void main(String[] args) {
9          var context = new AnnotationConfigApplicationContext(ProjectConfig.class);
10
11         String[] clientBeans = context.getBeanNamesForType(Client.class);
12         for (String beanName : clientBeans) {
13             System.out.println("Bean name: " + beanName);
14         }
15
16         Client clientOmar = context.getBean("clientOmar", Client.class);
17         System.out.println("\nBean clientOmar values: " + clientOmar.toString());
18     }
19 }
```

En la línea 16 tenemos `context.getBean("clientOmar", Client.class)` done le pasamos el nombre "clientOmar" como parámetro para que Spring sepa que ese bean en específico es el que queremos que nos retorne y Client.class porque hay que especificar de que tipo es el bean que estamos buscando.

Luego en la línea 17 imprimimos por consola los valores de los atributos de nuestro bean "clientOmar"

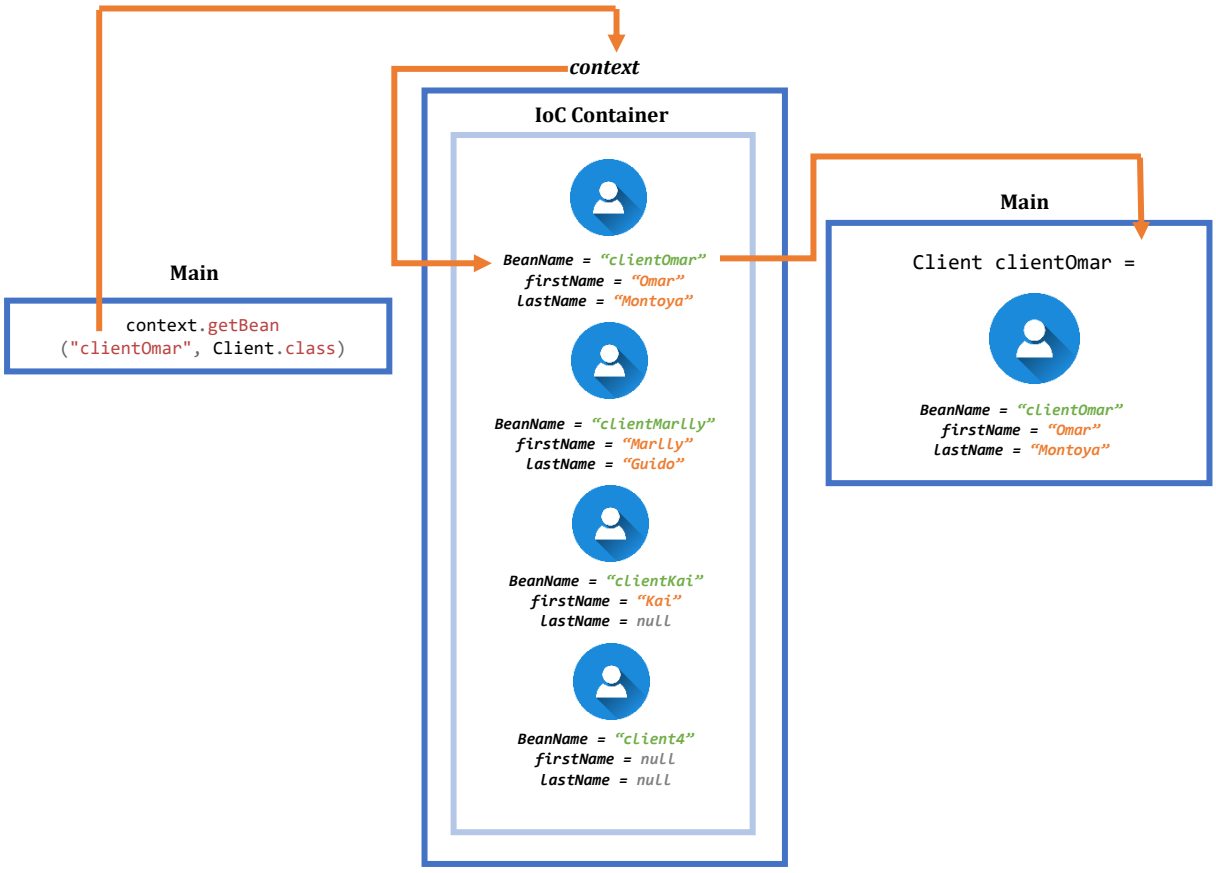
Output por consola

```
"C:\Program Files\Java\jdk-17\bin\java.exe" ...
Bean name: clientOmar
Bean name: clientMarlly
Bean name: clientKai
Bean name: client4

Bean clientOmar values: Client{firstName='Omar', lastName='Montoya'}

Process finished with exit code 0
```

Podemos notar que la excepción ya no se ejecuta ya que esta vez si le estamos indicando específicamente cual bean nos debe retornar. De igual manera nos imprime los valores de los atributos que definimos en la clase "ProjectConfig".



De esta manera podríamos también utilizar todos los beans de nuestro IoC Container e imprimir sus atributos por consola:

```java
package com.codewithomar.main;

import com.codewithomar.beans.Client;
import com.codewithomar.config.ProjectConfig;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class Main {
    public static void main(String[] args) {
        var context = new AnnotationConfigApplicationContext(ProjectConfig.class);

        String[] clientBeans = context.getBeanNamesForType(Client.class);
        for (String beanName : clientBeans) {
            System.out.println("Bean name: " + beanName);
        }

```

```
16          Client clientOmar = context.getBean("clientOmar", Client.class);
17          Client clientMarlly = context.getBean("clientMarlly", Client.class);
18          Client clientKai = context.getBean("clientKai", Client.class);
19          Client client4 = context.getBean("client4", Client.class);
20
21          System.out.println("\nBean clientOmar values: " + clientOmar.toString());
22          System.out.println("\nBean clientMarlly values: " + clientMarlly.toString());
23          System.out.println("\nBean clientKai values: " + clientKai.toString());
24          System.out.println("\nBean client4 values: " + client4.toString());
25      }
26  }
```

Output por consola

```
"C:\Program Files\Java\jdk-17\bin\java.exe" ...
Bean name: clientOmar
Bean name: clientMarlly
Bean name: clientKai
Bean name: client4

Bean clientOmar values: Client{firstName='Omar', lastName='Montoya'}

Bean clientMarlly values: Client{firstName='Marlly', lastName='Guido'}

Bean clientKai values: Client{firstName='Kai', lastName='null'}

Bean client4 values: Client{firstName='null', lastName='null'}

Process finished with exit code 0
```

Solución 2: Anotación @Primary

La anotación ***@Primary*** debe hacerse dentro de la clase "***ProjectConfig***" donde tenemos declarados las anotaciones ***@Bean***. Esta anotación debe hacerse solo en un bean por tipo de clase. Específicamente en el que queremos que sea el que Spring seleccione por defecto si no se le es especificado ningún bean de esa clase.

Con esta simple anotación extra ya es suficiente para que no exista ambigüedad:

```
1   package com.codewithomar.config;
2
3   import com.codewithomar.beans.Client;
4   import org.springframework.context.annotation.Bean;
5   import org.springframework.context.annotation.Configuration;
6   import org.springframework.context.annotation.Primary;
7
8   @Configuration
9   public class ProjectConfig {
```

```
10
11      @Bean(name = "clientOmar")
12      @Primary
13      Client client1() {...}
14
15      @Bean(value = "clientMarlly")
16      Client client2() {...}
17
18      @Bean("clientKai")
19      Client client3(){...}
20
21      @Bean
22      Client client4(){...}
23  }
```

En nuestro proyecto elegimos al bean "*clientOmar*" como nuestro bean principal. De esta manera si no especificamos cual bean queremos Spring sabrá que debe elegir éste bean para retornarlo.

Una vez escrita la anotación ***@Primary*** ya podemos utilizar el método .getBean() en la clase main sin especificar un nombre y no ocurrirá la excepción NoUniqueBeanDefinitionException:

```
1   package com.codewithomar.main;
2
3   import com.codewithomar.beans.Client;
4   import com.codewithomar.config.ProjectConfig;
5   import org.springframework.context.annotation.AnnotationConfigApplicationContext;
6
7   public class Main {
8       public static void main(String[] args) {
9           var context = new AnnotationConfigApplicationContext(ProjectConfig.class);
10
11          String[] clientBeans = context.getBeanNamesForType(Client.class);
12          for (String beanName : clientBeans) {
13              System.out.println("Bean name: " + beanName);
14          }
15
16          Client clientOmar = context.getBean(Client.class);
17
18          System.out.println("\nBean clientOmar values: " + clientOmar.toString());
19      }
20  }
```

Output por consola

```
"C:\Program Files\Java\jdk-17\bin\java.exe" ...
Bean name: clientOmar
Bean name: clientMarlly
Bean name: clientKai
Bean name: client4
```

```
Bean clientOmar values: Client{firstName='Omar', lastName='Montoya'}

Process finished with exit code 0
```

A pesar no especificar el nombre del bean en la línea 16 en el método .getBean(), Spring sabe que el bean que debe retornar es el que está anotado con ***@Primary***.