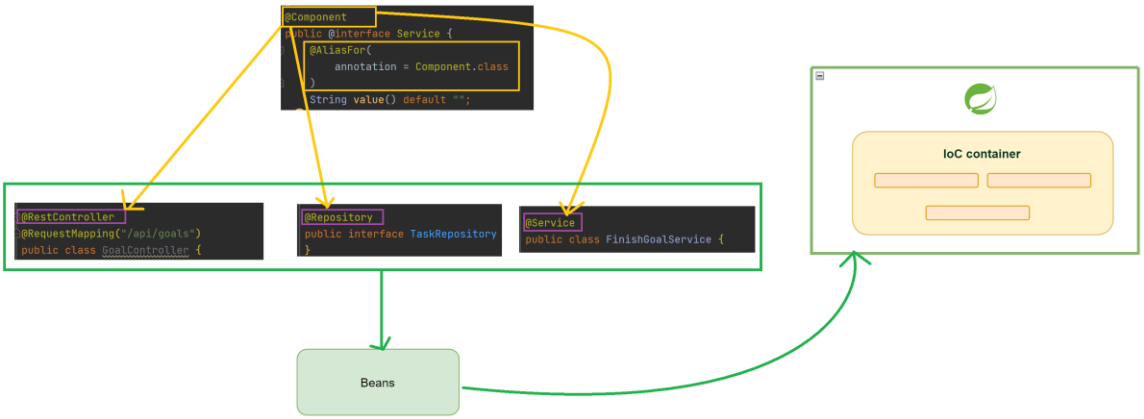
Beans

Bean é uma palavra muita usada no contexto do **Spring**, mas o que de fato é um **Bean**? A resposta é simples, um **Bean é um objeto qualquer que é gerenciado pelo Spring.**

Com *"gerenciado"* estamos dizendo que **é responsabilidade do Spring tomar conta daquele objeto**, então **encontrar esse objeto, instancia-lo, configura-lo e até mesmo injetar ele em outro Bean** são ações feitas pelo **Spring** sem uma necessidade de intervenção nossa.

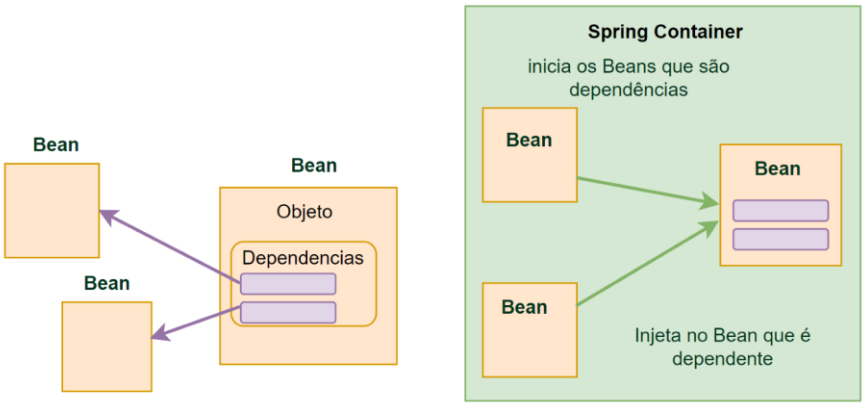
Para dizer que um **objeto** nosso é um **Bean**, basta apenas dizer que ele é um componente do **Spring** através de **anotações nas classes,** existem **anotações especializadas como o controller por exemplo**, esse tipo de anotação garante algumas features mais especializadas para o tipo do componente, mas não deixam de ser **Beans**:



Agora todos os **Beans** são conhecidos pelo Spring e gerenciados por ele, isso traz alguns poderes interessantes para gente, como usufluir do **IoC container**.

IoC Container

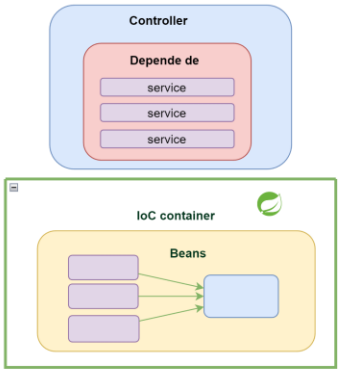
O **IoC container** do Spring é uma implementação do padrão de **inversão de controle e injeção de dependências**. De maneira simples, esse padrão diz que as **dependências de uma classe devem ser controladas por "alguém" de fora da classe** *(daí vem a inversão de controle)*, **esse "alguém" de fora da classe deve injetar essas dependências dentro dela** *(daí vem a injeção de dependência).*



É assim que o container gerencia os beans, como ele tem conhecimento deles ele pode injetar um bean onde existe dependência dele.

Pontos de injeção de dependências

O Container do **Spring** conhece os **Beans** e pode injeta-los automaticamente, mas nem tão automaticamente assim. Existem formas para dizermos onde cada **Bean** deve ser injetado, a injeção pode acontecer através do **construtor de uma classe (um jeito bom)** ou através da anotação **@Autowired**:




```java
@RestController
@RequestMapping("/api/goals")
public class GoalController {

    private final UserCrudService userService;
    private final GoalCrudService goalService;
    private final FinishGoalService finishingGoalService;

    public GoalController(UserCrudService userService, GoalCrudService goalService,
                          FinishGoalService finishingGoalService) {
        this.userService = userService;
        this.goalService = goalService;
        this.finishingGoalService = finishingGoalService;
    }
```

```java
@RestController
@RequestMapping("/api/goals")
public class GoalController {

    @Autowired
    private UserCrudService userService;

    @Autowired
    private GoalCrudService goalService;

    @Autowired
    private FinishGoalService finishingGoalService;
```

Se sua classe tiver **mais de um construtor** você pode anotar o construtor padrão pro Spring utilizar com ***@Autowired,*** a ***anotação*** **também pode ser usada em setters**.

Configurando Beans

Muitas vezes as configurações padrões de um Bean não atendem a nossa necessidade, por isso **podemos sobrescrever a configuração de um Bean**.

Já vi Beans sendo injetados diretamente em uma classe com **@Bean**, mas o jeito mais limpo de sobrescrever um Bean é usando uma classe de **configuração,** esse tipo de classe é anotada com **@Configuration** e tem como único propósito **fornecer uma instância de um Bean configurado da maneira que quisermos:**

```java
@Configuration
public class CorsConfig {

    @Bean
    public WebMvcConfigurer corsConfigure(){

        return addCorsMappings(corsRegistry) -> {
                corsRegistry.addMapping( pathPattern: "/**")
                        .allowedMethods("GET", "POST","PUT","DELETE")
                        .allowedHeaders("*")
                        .allowedOrigins("*");
        } ;
    }
}
```

```java
@Configuration
public class SwaggerConfig {

    @Bean
    public Docket docket(){
        return new Docket(DocumentationType.SWAGGER_2)
                .select() ApiSelectorBuilder
                .apis(RequestHandlerSelectors.basePackage("br.com.todo"))
                .paths(PathSelectors.ant( antPattern: "/**"))
                .build() Docket
                .apiInfo(apiInfo())
                .ignoredParameterTypes(User.class)
                .globalOperationParameters(Arrays.asList(
                        new ParameterBuilder()
                                .name("Authorization")
                                .description("Header para token jwt")
                                .modelRef(new ModelRef( type: "string"))
                                .parameterType("header")
                                .required(false)
                                .build()
                ));
    }
```

Ambiguidade de Beans

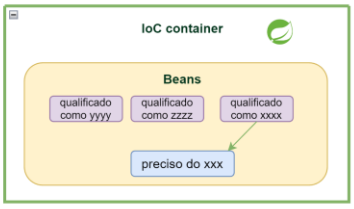
As vezes precisamos de **mais de uma instância de um determinado Bean**, por exemplo, para usar mais de um **DataSource numa aplicação eu preciso de duas instâncias de DataSource**. Isso **gera ambiguidade**, pois o Spring não vai saber qual **Bean** ele tem que injetar, se é um ou outro.

Uma das maneiras de desambiguizar os **Beans** é com a anotação **@Primary**, então se eu tiver **duas instâncias de um Bean** específico, eu posso **definir qual deles é o primário que o Spring sempre deve utilizar**:

```java
@Configuration
public class DataSourceConfig {

    @Primary
    @Bean
    @ConfigurationProperties(prefix = "spring.datasource")
    public DataSource springBatchDataSource(){
        return DataSourceBuilder.create().build();
    }

    @Bean
    @ConfigurationProperties(prefix = "app.datasource")
    public DataSource appDataSource(){
        return DataSourceBuilder.create().build();
    }

}
```

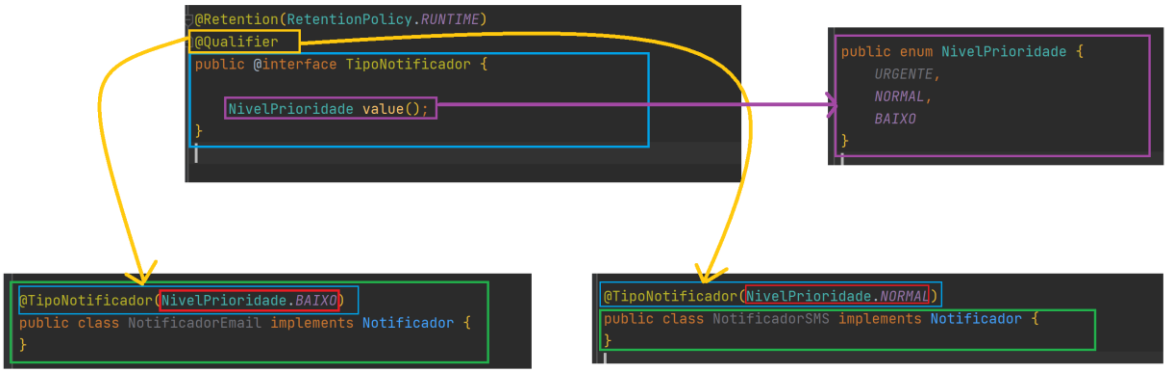
A anotação de configuração de propriedades é o que indica quais valores o Bean de datasource vai utilizar do application properties.

Outra maneira de desambiguar Beans é utilizando o **@Qualifier**, onde você **qualifica um Bean com algum nome** e no **ponto de injeção você especifica o qualificador daquela injeção**:



Um problema do **"Qualifier"** é que dependendo do tamanho do projeto, caso você queira mudar um nome de qualificador, pode ser que se torne algo chato. Pensando nisso é importante **abstrair essa configuração criando anotações customizadas**.

A **anotação customizada** vai fazer o **papel de Qualifier**, recebendo **o valor de um enum como assinatura**. Agora os **demais Beans que vão implementar a anotação customizada serão Qualifiers** também, mas com seus **próprios valores de Enum**:



Agora basta **definir no ponto de injeção qual será o qualificador usado através também da anotação customizada**. O maior ganho nessa abordagem é ter o nome dos qualificadores centralizados no Enum para uma possível refatoração em massa.