02

Injeção de dependências

Dê uma olhada no código da nossa classe `ContaController`. Repare que em todos os métodos, instanciamos a classe `ContaDao`. Poderíamos melhorar um pouco esse código, e instanciar esse objeto no construtor. Por exemplo:

```
class ContaController {
  private ContaDao dao;

  public ContaController() {
    this.dao = new ContaDao();
  }

  // controller continua aqui...
}
```

O nosso código melhorou, pois temos menos código para manter, mas há mais um problema. Repare que continuamos responsáveis pela criação do DAO. A classe que faz uso deste DAO está intimamente ligada com a maneira de instanciação do mesmo, fazendo com que ela mantenha um alto grau de acoplamento. Isso dificulta inclusive a testabilidade dessa classe.

O ideal seria receber dependências sempre pelo construtor da classe. Assim, a dependência fica explícita, fácil de ser trocada, e fácil de ser testada:

```
class ContaController {
  private ContaDao dao;

  public ContaController(ContaDao dao) {
    this.dao = dao;
  }

  // controller continua aqui...
}
```

Mas a pergunta agora é: quem instancia o `ContaController` e passa pra ele o `ContaDao`?

Contâiner de Injeção de Dependências

O padrão de projetos Dependency Injection (DI) (Injeção de dependências), procura resolver esses problemas. A ideia é que a classe não mais resolve as suas dependências por conta própria, mas apenas declara que depende de alguma outra classe. E de alguma outra forma que não sabemos ainda, alguém resolverá essa dependência para nós. Alguém pega o controle dos objetos que liga (ou amarra) as dependências. Não estamos mais no controle, há algum container que gerencia as dependências e amarra tudo. Esse container já existia e já usamos ele sem saber.

Repare que com `@Controller` já definimos que a nossa classe faz parte do Spring MVC, mas a anotação vai além disso. Também definimos que queremos que o Spring controle o objeto. Spring é no fundo um container que dá new para nós e também sabe resolver e ligar as dependências. Por isso, o Spring também é chamado **Container IoC (Inversion of Control) ou Container DI**.

Essas são as principais funcionalidades de qualquer container de inversão de controle/injeção de dependência. O Spring é um dos vários outros containers disponíveis no mundo Java.

O Spring Container sabe criar o objeto, mas também liga as dependências dele. Como o Spring está no controle, ele administra todo o ciclo da vida. Para isso acontecer será preciso definir pelo menos duas configurações: declarar o objeto como componente, e ligar a dependência.

Para receber o DAO no construtor, basta anotar o construtor com `@Autowired`. Dessa forma, o Spring sabe que precisa injetar essa dependência:

```
class ContaController {
  private ContaDao dao;

  @Autowired
  public ContaController(ContaDao dao) {
    this.dao = dao;
  }

  // controller continua aqui...
}
```

Já no nosso DAO, precisamos anotá-lo com `@Repository`, anotação específica para DAOs. Mas repare só que nosso DAO instancia uma `Connection`. Por que não recebê-la pelo construtor e deixar o Spring injetá-la também?

```
@Repository
public class ContaDao {
  private Connection connection;

  public ContaDao(Connection connection) {
    this.connection = connection;
  }

  // DAO continua ..

}
```

Ao usar `@Autowired` no construtor, o Spring tenta descobrir como abrir uma conexão, mas claro que o Container não faz ideia com qual banco queremos nos conectar. Para solucionar isso o Spring oferece uma configuração de XML que define um provedor de conexões. No mundo JavaEE, este provedor é chamado `DataSource` e abstrai as configurações de Driver, URL, etc da aplicação.

Sabendo disso, devemos declarar um `DataSource` no XML do Spring, dentro do `spring-context.xml`:

```
<bean id="mysqlDataSource" class="org.apache.commons.dbcp.BasicDataSource">
<property name="driverClassName" value="org.hsqldb.jdbcDriver"/>
<property name="url" value="jdbc:hsqldb:file:contas.db"/>
<property name="username" value="sa"/>
<property name="password" value=""/>
</bean>
```

Definimos um bean no XML. Um bean é apenas um sinônimo para componente. Ao final, cada bean se torna um objeto administrado pelo Spring. Para o Spring Container, a `mysqlDataSource` , o `ContaDao` e `ContaController` são todos componentes(ou beans) que foram ligados/amarrados. Ou seja, um depende ao outro. O Spring vai criar todos e administrar o ciclo da vida deles. Com a `mysqlDataSource` definida, podemos injetar ela na `ContaDao` para recuperar a conexão JDBC. Para isso não podemos esquecer do `@Autowired` também:

```
@Repository
public class ContaDao {
  private Connection connection;

  @Autowired
  public ContaDao(DataSource ds) {
    try {
      this.connection = ds.getConnection();
    } catch(SQLException e) {
      throw new RuntimeException(e);
    }
  }

  // DAO continua ..

}
```

Pronto! Repare que no nosso projeto o controlador -- depende do --> dao que -- depende do --> datasource. O Spring vai primeiro criar a `mysqlDataSource` , depois o DAO e no final o controlador. Ele é o responsável pela criação de toda a cadeia de dependências.

O padrão de injeção de dependências é muito utilizado, e o Spring facilita a nossa vida. Receba suas dependências sempre pelo construtor, isso é uma boa prática de código!