05

# Trabalhando com a JPA

### Transcrição

Fizemos a mudança no método `login()` para que agora ele atenda apenas requisições do tipo **POST**. Dessa forma, os parâmetros não estão mais sendo passados pela URL e sim pelo corpo da requisição. Apesar de termos melhorado, isso não é o suficiente para garantir que a aplicação está segura, pois os dados ainda estão sendo passados para o servidor.

O problema está na classe `UsuarioDaoImpl`, que concatena código SQL com código Java. Não estamos fazendo nenhum tipo de validação do tipo de informação que o usuário pode passar no formulário. Como vimos em outros cursos da Alura, o ideal é usar a especificação da **JPA** e a especificação do **EntityManager**. Vamos fazer essa mudança.

Pararemos o Tomcat para fazer as modificações na classe `UsuarioDaoImpl`. Inicialmente, os desenvolvedores estavam criando a instância `new ConnectionFactory().getConnection()` para pegar a conexão. Apagaremos essa instância e todo o conteúdo dos métodos `salva()` e `procuraUsuario()`.

```
@Repository
public class UsuarioDaoImpl implements UsuarioDao {

    public void salva(Usuario usuario) {

    }

    public Usuario procuraUsuario(Usuario usuario) {

    }
}
```

Como estamos trabalhando com Spring, pediremos para que ele faça a injeção do `EntityManager`, por meio da anotação `@PersistenceContext`.

```
@Repository
public class UsuarioDaoImpl implements UsuarioDao {

    @PersistenceContext
    private EntityManager manager;

    public void salva(Usuario usuario) {

    }

    public Usuario procuraUsuario(Usuario usuario) {

    }
}
```

No método `salva()`, queremos apenas realizar a persistência dos dados do usuário. Usaremos o objeto do `EntityManager` chamando o método de persistência.

```
@Repository
public class UsuarioDaoImpl implements UsuarioDao {

    @PersistenceContext
    private EntityManager manager;

    public void salva(Usuario usuario) {
        manager.persistence(usuario);
    }

    public Usuario procuraUsuario(Usuario usuario) {

    }
}
```

No método `procuraUsuario()`, precisamos pegar as informações de e-mail e senha que serão passadas no formulário e realizar a *query* de consulta no banco de dados. o método que cria a *query* é o `manager.createQuery()`. Nele nós passamos em String o comando SQL, e como segundo parâmetro o para quem é essa *query*, como faremos para o usuário nós passaremos a classe `Usuario.class`, e será retornado um objeto do tipo `TypedQuery<Usuario>`.

Para evitar misturar código SQL com Java, trabalharemos com **JPQL** de forma orientada a objetos. Dessa forma, criaremos a *query* colocando um identificador para os parâmetros que serão substituídos pelos valores pela própria **JPA**. O usuário buscado, nós associaremos a um *alias* (apelido). O método ficara da seguinte forma:

```
@Repository
public class UsuarioDaoImpl implements UsuarioDao {

    @PersistenceContext
    private EntityManager manager;

    public void salva(Usuario usuario) {
        manager.persistence(usuario);
    }

    public Usuario procuraUsuario(Usuario usuario) {
       TypedQuery<Usuario> usuario = manager
            .createQuery("select u from Usuario u where u.email=:email and u.senha=:senha", Usu
    }
}
```

Estamos separando as informações passadas pelo formulário da *query*. Primeiro estamos executando a *query*, e só depois pegaremos os parâmetros do formulário. Dessa forma conseguiremos proteger a aplicação contra os ataques de injeção de código SQL. Agora precisamos colocar os valores para `:email` e `:senha`.

```
@Repository
public class UsuarioDaoImpl implements UsuarioDao {

    @PersistenceContext
    private EntityManager manager;

    public void salva(Usuario usuario) {
        manager.persistence(usuario);
```

```
    }

    public Usuario procuraUsuario(Usuario usuario) {
       TypedQuery<Usuario> query = manager
            .createQuery("select u from Usuario u where u.email=:email and u.senha=:senha", Usu
        query.setParameter("email", usuario.getEmail());
        query.setParameter("senha", usuario.getSenha());
    }
}
```

Ainda falta retornar o resultado, utilizaremos os recursos do Java 8. Como não podem existir dois usuários com o mesmo valor de e-mail e senha, basta retornarmos o primeiro valor encontrado na lista retornada pelo banco. Mas e se não tiver nenhum usuário para o e-mail e senha? Nesse caso devemos retornar `null`.

Podemos pegar a lista retornada pela *query* com `query.getResultList()`, e utilizar os métodos do Java 8 de `stream().findFirst()` para retornar o primeiro usuário encontrado. Caso não encontre, usaremos o `orElse(null)` para que retorne o valor `null`. O método ficara da seguinte forma:

```
@Repository
public class UsuarioDaoImpl implements UsuarioDao {

    @PersistenceContext
    private EntityManager manager;

    public void salva(Usuario usuario) {
        manager.persistence(usuario);
    }

    public Usuario procuraUsuario(Usuario usuario) {
       TypedQuery<Usuario> query= manager
            .createQuery("select u from Usuario u where u.email=:email and u.senha=:senha", Usu
        query.setParameter("email", usuario.getEmail());
        query.setParameter("senha", usuario.getSenha());

        Usuario usuarioRetornado = query.getResultList().stream().findFirst().orElse(null);
        return usuarioRetornado;
    }
}
```

Vamos fazer um teste, começaremos reinicializando o Tomcat. Tentaremos fazer o primeiro teste que o usuário Alex fez, tentar efetuar a autenticação como se fosse o Fernando. No formulário de *Login*, colocaremos no campo **E-mail** `fernando@gmail.com`, e no campo senha `x' or 'a'='a`. Deu *Usuário não encontrado*, conseguimos proteger a aplicação desse tipo de injeção de código SQL.

Tentaremos agora executar os testes utilizando o `sqlmap`. No Terminal da máquina Linux, colocaremos o comando:

```
sqlmap -u "http://192.168.121.171/alura-shows/login" --dump -T usuario -D owasp --data="email=a
```

O `sqlmap` informou que os parâmetros não são injetáveis. Conseguimos proteger a aplicação contra os ataques de injeção de código SQL.