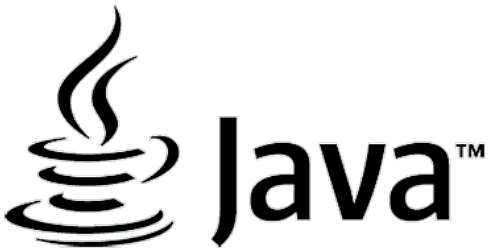
Sistemas Distribuídos
RELATÓRIO TRABALHO PRÁTICO
Grupo 2

Rui Soares - 46022
Francisco Reis - 45550
Bruno Bento - 47583

Maio 2024



1 Introdução

O objetivo deste trabalho é desenvolver uma aplicação web que permita a gestão dos serviços oferecidos por uma escola profissional de Teatro. A aplicação será baseada na arquitetura de Web Services `RESTful`, utilizando a framework `Spring` em Java, o que garante uma abordagem modular e escalável. Para a persistência de dados, será implementado um banco de dados relacional, gerenciado por um Sistema de Gestão de Base de Dados (SGBD).

O relatório abordará os seguintes tópicos: uma visão geral do modelo de dados utilizado, a descrição detalhada da arquitetura da aplicação e suas funcionalidades, e uma explicação das decisões tomadas em relação à gestão de sessões e segurança.

2 Modelo de Dados

O modelo de dados utilizado é o modelo relacional, implementado através de `SQL`, permitindo a organização e manipulação eficiente dos dados num sistema de gestão de base de dados (SGBD) relacional.

A nossa Base de Dados não é nada mais nada menos que um Servidor fisicamente instalado na nossa Universidade. Para aceder ao mesmo, de modo a fazer a conexão à nossa Base de Dados, foi necessário utilizar a VPN da UBI. Depois de configurados estes passos, a Base de Dados estava pronta a ser utilizada devidamente pela nossa Aplicação.

No código da Aplicação, a classe `DatabaseConnection` demonstra a implementação de conexão `JDBC` (*Java DataBase Connection*) com a base de dados `SQL Server`. Ela configura os parâmetros de conexão, carrega o driver `JDBC`, tenta estabelecer a conexão, e assegura o encerramento da conexão após o uso (lidando com possíveis exceções).

Explicando o código referente à conexão passo a passo:

São importadas classes necessárias para gerir a conexão à `BD`. Inicializada a classe e o seu construtor, são definidas variáveis para configurar a conexão:

- `server`: Endereço IP do servidor `SQL`.
- `database`: Nome da `BD`.
- `username`: Nome de utilizador.
- `password`: Password de acesso.
- `driver`: Classe do driver `JDBC` (*Java DataBase Connection*).

São definidas também as variáveis `connection` e `statement` (do tipo `Connection` e `Statement`, respetivamente).

De seguida, no método `getConnection()` (do tipo `Connection`) a variável `connectionUrl` utiliza o método `String.format()` para criar uma string de conexão à `BD`. A string é formada pela correta concatenação das variáveis definidas acima (`server`, `database`, `username`, `password`). Cada placeholder `%s` será substituído pelos valores das variáveis fornecidas pela ordem apresentada, devidamente separadas por ponto e vírgula.

Seguidamente é iniciada a conexão (com a variável do tipo `Connection`). Dentro de um bloco `try-catch`, é carregada a classe do driver `JDBC` (`Class.forName(driver)`) e estabelecida a conexão usando a string de conexão acima referida, dando-se de seguida o tratamento das exceções (se o driver não for encontrado, ou se a conexão falhar). O método `closeConnection()` recebe o objeto `connection` como parâmetro e tenta fechar a conexão (tratando a possível exceção de encerramento da conexão).

Mais métodos são definidos nesta classe, e serão devidamente abordados nos próximos capítulos.

3 Arquitetura e Funcionalidades

"A aplicação foi desenvolvida com a **framework Spring**, que simplifica a criação de aplicações Java robustas e escaláveis, facilitando a configuração e execução da aplicação, permitindo a criação de projetos com um mínimo de configuração inicial e com as devidas dependências necessárias. A arquitetura **RESTful** foi utilizada para a criação de **APIs** que seguem os princípios do **REST** (*Representational State Transfer*), permitindo a comunicação entre sistemas através de requisições HTTP, via **endpoints** que estão devidamente definidos na classe **WebController** para mapear as requisições HTTP aos métodos da classe em questão.

A estrutura em termos de código está dividida em 3 pastas:

Controller:

Contém uma das classes principais, a **WebController**, tem como função controlar as requisições web e direcionar para os serviços apropriados. Além desta, tem as classes **FilterConfig** e **Tfil**, utilizadas para tratar a gestão de sessões na aplicação (serão detalhadas, portanto, no próximo capítulo).

Model:

Contém as classes que representam as entidades do domínio da aplicação. Estas classes são usadas para mapear os dados da base de dados para objetos Java, e vice-versa. As classes são: **Course**, **Student**, **Professor**, **User**, **School**, **Discipline**,

que têm os seus atributos, métodos construtores, **getters**, **setters** e **toString()** devidamente definidos para entidade; além das classes de conexão com a BD: **DatabaseConnection**, **CourseDB**, **DisciplineDB**, **ProfessorDB**, **StudentDB**.

Resources(View):
Contém os arquivos que definem a interface do utilizador: as páginas HTML e o arquivo CSS que as estiliza. Estes arquivos são responsáveis pela apresentação dos dados e pela interação do utilizador com a aplicação."

Feita a explicação da estrutura base da aplicação, iremos detalhar agora a explicação de cada classe do código:

DatabaseConnection - Já referida no capítulo anterior, proseguiremos a explicação dos restantes métodos, sendo que todos eles estabelecem a comunicação com a BD (`connection.createStatement()`), definem e executam uma consulta SQL, e armazenam os resultados da consulta num `ResultSet` para posterior processamento (`statement.executeQuery(query)`):

`getCountUsers`: Este método conta todos os utilizadores registados na tabela `Users`, executando a consulta SQL para os obter, iterando sobre as linhas do conjunto de resultados (`ResultSet`) através de um ciclo `while` e do método `next()` para seguir para a linha seguinte do conjunto. Dentro dele é incrementado o contador (variável `count`) para cada utilizador encontrado.

`getUsers`: De forma semelhante ao método anterior, este método exibe uma lista de todos os utilizadores registados na tabela `Users`, criando objetos do tipo `User` para cada utilizador encontrado na BD, retornando a lista.

`userstoString`: Este método converte as informações dos utilizadores para uma representação em string, isto é, formata os dados encontrados (dados esses encontrados e iterados da mesma forma que os dois métodos anteriores) dos utilizadores devidamente numa String (usando métodos como `substring()` e `toUpperCase()`), e a concatenação das informações de forma correta (usando "—" a separar as mesmas).

`addUser`: Este método recebe os parâmetros definidos para o utilizador na classe `User`, conecta-se à BD, sendo os valores dos parâmetros inseridos dinamicamente, nos locais correspondentes. Após a execução da atualização, o método recupera a chave gerada pela BD, que é o ID do novo User. Se a chave gerada for obtida com sucesso, ela é retornada; caso contrário, uma exceção `SQLException` é

lançada.

`deleteUser`: Este método, de forma semelhante ao de adicionar, exclui um utilizador dado um determinado ID da base de dados. Se o update não tiver sido executado devidamente imprime a mensagem de erro. Se a conexão com a BD não ocorreu de forma correta, lança a exceção `SQLException`.

`updateUser`: Este método recebe os parâmetros definidos pelo utilizador na classe `User` (tal como o `addUser`), conecta-se à BD e constrói dinamicamente a consulta SQL com base nos parâmetros fornecidos, de modo a atualizar corretamente as ifnormações do utilizador em questão na BD, caso estes campos não estejam vazios, pois caso contrário não seria uma atualização.

`checkUser`: Verifica se dado o email e a password, o utilizador existe na BD. O resultado da consulta SQL está armazenado em `ResultSet`. Se esta variável tiver pelo menos uma linha, o utilizador foi encontrado e `"true"` é retornado. Caso contrário retorna `"false"`.

`checkType`: Semelhante ao anterior, verifica o tipo de utilizador (Admin, Professor ou Aluno) dados o email e a password do utilizador. Se `ResultSet` tiver pelo menos uma linha, usamos o `getString()` para retirar o tipo do utilizador da linha.

`getMaxUserId`: Retorna o ID máximo presente na tabela de `Users`, ou seja, o ID do último utilizador inscrito, usando a função `MAX` na consulta SQL. Da mesma forma que no método anterior, se `ResultSet` tiver pelo menos uma linha, usamos `getInt()` para retirar o ID máximo (caso contrário retorna 0).

`getUserById`: Recebe como parâmetro o ID e cria um objeto `User` com os parâmetros do Utilizador do ID fornecido. A consulta pelos dados e a iteração pelas linhas do resultado da consulta é semelhante ao método `getUsers()`.

Por último, na função `main`, são criados objetos do tipo `DatabaseConnection` e `Connection` respetivamente, para iniciar e representar por um objeto, respetivamente, a conexão. Por fim é fechada a conexão à DB através do `closeConnection()`.

Vamos agora à classe **CourseDB**. Esta classe é responsável por gerir a persistência de dados relacionados aos cursos da nossa Escola na BD, ao invés da **DatabaseConnection**, que tratava os dados dos Utilizadores. Os seus métodos, como tal, foram implementados de forma semelhante aos métodos da classe **DatabaseConnection**. Definidas as variáveis **connection** e **statement**, e o construtor

da classe com estes parâmetros, passamos aos métodos:

`getCourses()`: Devolve a lista de cursos registados na base de dados, criando um novo objeto **Course** por cada curso encontrado na Base de Dados. O Curso tem como parâmetros o ID, nome, descrição, preço, duração e tempo-normal.

`addCourse()`: Adiciona um Curso à Base de dados conforme os parâmetros acima referidos. Implementado de forma muito semelhante ao `addUser()` presente na classe anteriormente referida.

`updateCourse()`: Atualiza as informações do Curso dados os parâmetros referidos, da mesma forma que o método `updateUser()`.

`deleteCourse()`: Dado o ID do Curso, exclui o curso da tabela **Courses** e também remove todas as referências ao curso nas tabelas **Student_Course** e **Course_Discipline**.

`addStudentCourse()`: Dados o ID do Estudante e o ID do Curso, estabelece uma nova relação entre um estudante e um curso na tabela **Student_Course**.

`getDisciplinesByCourse()`: Retorna uma lista de disciplinas associadas a um curso específico, realizando um join entre as tabelas **Discipline** e **Course_Discipline** para só nessa nova tabela fazer a consulta SQL. Encontrada a disciplina, é criado o objeto do tipo **Disciplina** e é adicionado à lista de disciplinas.

`getCourse()`: Dado o ID do Curso retorna as informações todas do Curso (os seus parâmetros) em formato de String, através do método `toString()`.

Estes métodos foram implementados com a palavra-chave **synchronized** para garantir que apenas uma thread possa aceder ao método por vez afim de evitar problemas de concorrência, como a leitura e escrita simultânea de dados na BD, que podem resultar em inconsistências e erros.

Por último, na função `main`, são criados objetos do tipo **DatabaseConnection** e **Connection** respetivamente, para iniciar e representar por um objeto, respetivamente, a conexão. É criado o objeto **courseDB** com o mesmo objetivo (com o parâmetro **connection**). São listados os Cursos e respetivas disciplinas correspondentes ao Curso em questão. Por fim é fechada a conexão à DB através do `closeConnection()`.

Na classe **DisciplineDB** - semelhante à **CourseDB**, mas relacionado às dis-

ciplinas de cada curso - , definidas as variáveis **connection** e **statement**, e o construtor da classe com estes parâmetros, passamos aos métodos:

`addProfessorDiscipline()`: Adiciona uma associação entre um professor e uma disciplina na tabela **dbo.Professor_Discipline**, dados o **Id** do Professor e o **Id** do Curso.

`addProfessor()`: Dados os parâmetros definidos na classe **Professor**, adiciona um novo professor na tabela **dbo.Professor** com os dados fornecidos.

`updateProfessor()`: Atualiza os dados de um professor na tabela **dbo.Professor** com base nos parâmetros do professor.

`deleteProfessor()`: Remove um professor e suas associações de disciplinas das tabelas **dbo.Professor** e **dbo.Professor_Discipline**, dado **ID** do professor em questão.

`getProfessorsByDiscipline()`: Dado o **Id** de uma disciplina como parâmetro, retorna uma lista de professores associados à disciplina específica, consultando as tabelas **dbo.Professor** e **dbo.Professor_Discipline**.

`getUserProfessors()`: Retorna uma lista de utilizadores que são professores, consultando as tabelas **dbo.Users** e **dbo.Professor**.

Estes métodos, tal como na classe **CourseDB**, foram implementados com a palavra-chave "**synchronized**"para garantir que apenas uma thread possa aceder ao método por vez afim de evitar problemas de concorrência, como a leitura e escrita simultânea de dados na BD, que podem resultar em inconsistências e erros.

Por último, na função **main**, são criados objetos do tipo **DatabaseConnection** e **Connection** respetivamente, para iniciar e representar por um objeto, respetivamente, a conexão. É criado o objeto **professorDB** com o mesmo objetivo(com o parâmetro **connection**). São listados os Professores e respetivas disciplinas correspondentes ao Professor em questão através do método **toString()**. Por fim é fechada a conexão à BD através do **closeConnection()**.

Passando à classe **StudentsDB** - semelhante às duas anteriores - , definidas as variáveis **connection** e **statement**, e o construtor da classe com estes parâmetros, passamos aos métodos:

`getStudents()`: Devolve uma lista de todos os estudantes presentes na tabela

Student do banco de dados.

`addStudent()`: Adiciona um novo estudante à tabela **Student** com os valores especificados para **id_user** e **entryYear**, definidos como parâmetros.

`updateStudent()`: Atualiza as informações de um estudante existente na tabela **Student** com os novos valores para o **id**, **id_user** e **entryYear**, dados inicialmente como parâmetros.

`deleteStudent()`: Exclui um estudante da tabela **Student** e suas associações na tabela **Student_Course**, com base no **id** do estudante.

`addStudentCourse()`: Adiciona uma associação entre um estudante e um curso na tabela **Student_Course** com base nos **studentId** e **courseId** fornecidos como parâmetros.

`getStudentsByCourse()`: Retorna uma lista de estudantes inscritos num curso específico, identificado pelo **courseId** dado como parâmetro.

`getUserStudentsByCourse()`: Retorna uma lista de utilizadores que são estudantes inscritos num curso específico, identificado pelo **courseId**.

`deleteStudent_ProfessorByIdUser()`: Exclui um utilizador e todas as suas associações como estudante e professor, bem como os registos nas tabelas **Student_Course**, **Student**, **Professor_Discipline**, **Professor**, e **Users**, com base no **id_user** dado como parâmetro.

Estes métodos, tal como nas classes anteriores, foram implementados com a palavra-chave "**synchronized**"para garantir que apenas uma thread possa aceder ao método por vez afim de evitar problemas de concorrência, como a leitura e escrita simultânea de dados na BD, que podem resultar em inconsistências e erros.

Por último, na função **main**, são criados objetos do tipo **DatabaseConnection** e **Connection** respetivamente, para iniciar e representar por um objeto, respetivamente, a conexão. É criado o objeto **studentDB** com o mesmo objetivo(com o parâmetro **connection**). São listados os alunos e os alunos correspondentes a um curso através do método `toString()`. Por fim é fechada a conexão à BD através do `closeConnection()`.

Por fim, foram implementadas as classes: **Course**, **Discipline**, **Professor**, **Student**, **School**, **User**; com os parâmetros correspondentes, os métodos cons-

trutores, os getters e setters, e o `toString()`, devidamente implementados. Falar também da classe **Application** que serve para inicializar e correr a aplicação em **Spring Boot**.

Passando à classe **WebController**. A classe **WebController** é uma parte essencial de uma aplicação **Spring Boot**, responsável por gerir as requisições HTTP e retornar as respostas apropriadas. Ela atua como o controlador na arquitetura **MVC** (**Model-View-Controller**) usada nesta aplicação (View foi acima referido como "resources"), onde recebe as entradas do utilizador, interage com o modelo (dados) e seleciona a view (interface) para exibir a resposta ao utilizador, através de **Endpoints**.

Endpoints são pontos de acesso aos recursos de uma aplicação web. Cada método na classe **WebController** é mapeado para um **endpoint** específico através de anotações como `@GetMapping` e `@PostMapping`: `@GetMapping` mapeia requisições HTTP GET para métodos específicos. É utilizado para recuperar dados sem alterar o estado do servidor. Por exemplo, a listagem de cursos ou a exibição do formulário de registo na página. `@PostMapping` mapeia requisições HTTP POST para métodos específicos. É utilizado para enviar dados ao servidor, geralmente para criar ou atualizar recursos. Por exemplo, o registo de novos cursos ou utilizadores.

Segue a explicação da classe por tópicos, sendo cada tópico correspondente a um @:

`@GetMapping("/")`
Endpoint: Home
Descrição: Este método retorna a página inicial. Ele conecta-se à **BD**, obtém a lista de cursos e adiciona-os ao objeto `ModelAndView` que representa a view "home".

`@GetMapping("/professors")`
Endpoint: Professores
Descrição: Este método retorna a página que lista todos os professores. Conecta-se à **BD**, obtém a lista de professores e os adiciona ao objeto `ModelAndView` que representa a view "professors".

`@GetMapping("/register")`
Endpoint: Registo
Descrição: Este método retorna a página de registo. Ele simplesmente retorna

o nome da view "register".

`@GetMapping("/registerDiscipline")`
Endpoint: Registo de Disciplina
Descrição: Este método retorna a página de registo de disciplina. Ele retorna o nome da view "registerDiscipline".

@GetMapping("/registerCourse")
Endpoint: Registo de Curso
Descrição: Este método retorna a página de registo de curso. Ele retorna o nome da view "**registerCourse**".

@PostMapping("/registerCourse")
Endpoint: Registo de Curso (POST)
Descrição: Este método lida com o envio do formulário de registo de curso. Ele recebe os parâmetros do formulário, conecta-se à BD e adiciona um novo curso, redirecionando para a página do administrador.

@PostMapping("/registerDiscipline")
Endpoint: Registo de Disciplina (POST)
Descrição: Este método lida com o envio do formulário de registo de disciplina. Recebe os parâmetros do formulário, conecta-se à BD e adiciona uma nova disciplina, redirecionando para a página do administrador.

@PostMapping("/registerAdmin")
Endpoint: Registo de Administrador
Descrição: Este método lida com o envio do formulário de registo de administrador. Recebe os parâmetros do formulário, conecta-se à BD e adiciona um novo administrador, redirecionando para a página do administrador.

@PostMapping("/registerStudent")
Endpoint: Registo de Estudante
Descrição: Este método lida com o envio do formulário de registo de estudante. Recebe os parâmetros do formulário, conecta-se à BD, adiciona um novo utilizador estudante e redireciona para a página do administrador.

@PostMapping("/registerProfessor")
Endpoint: Registo de Professor
Descrição: Este método lida com o envio do formulário de registo de professor. Recebe os parâmetros do formulário, conecta-se à BD, adiciona um novo professor e redireciona para a página do administrador.

@GetMapping("/login")
Endpoint: Login
Descrição: Este método retorna a página de login. Ele retorna o nome da view "**login**".

@PostMapping("/login") Endpoint: **Login** (**POST**) Descrição: Este método lida com o envio do formulário de *login*. Verifica o tipo de **user** (**Admin**, **Professor** ou **Aluno**) e redireciona para a página correspondente, ou retorna à página de *login* se as credenciais forem inválidas.

@GetMapping("/admin") Este método: Cria uma instância de **DatabaseConnection** e estabelece a conexão com a BD. Obtém listas de cursos (**CourseDB**), disciplinas (**DisciplineDB**), e **Users** (**db.getUsers()**). Filtra os **users** em três listas distintas baseadas no tipo: **administradores**, **professores** e **alunos**. Obtém a lista de **professores** (**ProfessorDB**). Calcula e armazena o total de **alunos** e **professores** em strings. Identifica o curso com o maior número de **alunos** e armazena essa informação. Determina o curso com o maior preço e armazena essa informação. Cria um objeto **ModelAndView** para a view "admin". Adiciona os dados recuperados (cursos, disciplinas, **users**) e as estatísticas calculadas ao **ModelAndView**. Retorna o objeto **ModelAndView** que será usado pela *framework* para renderizar a página "admin"com os dados e estatísticas fornecidos.

@PostMapping("/deleteUser") Função: Elimina um **utilizador**: Loga o **ID** recebido. Cria uma conexão com a BD. Cria uma instância de **StudentDB**. Chama o método *deleteStudent_ProfessorByIdUser(id).* Redireciona para "/admin".

@PostMapping("/deleteCourse") Função: Elimina um **curso**: Loga o **ID** recebido. Cria uma conexão com a BD. Cria uma instância de **CourseDB**. Chama o método *deleteCourse(id).* Redireciona para "/admin".

`@PostMapping("/deleteDiscipline")` Função: Elimina uma disciplina: Loga o ID recebido. Cria uma conexão com a BD. Cria uma instância de **DisciplineDB**. Chama o método **deleteDiscipline**(id). Redireciona para "/admin".

`@PostMapping("/editUser")` Função: Edita um utilizador: Loga os parâmetros recebidos. Cria uma conexão com a BD. Tenta atualizar o utilizador com **db.updateUser**(id, name, age, userType, email, password). Redireciona para "/admin".

`@PostMapping("/editDiscipline")` Função: Edita uma disciplina: Loga os parâmetros recebidos. Cria uma conexão com a BD. Tenta atualizar a disciplina com **disciplineDB.updateDiscipline**(id, name, description, schedule). Redire-

ciona para "/admin".

`@PostMapping("/editCourse")` Função: Edita um curso: Loga os parâmetros recebidos. Cria uma conexão com a BD. Tenta atualizar o curso com **courseDB.updateCourse**(id, name, description, price, duration, normalTime). Redireciona para "/admin".

`@PostMapping("/addAlunoToCourse")` Função: Adiciona um aluno a um curso: Loga os IDs recebidos. Cria uma conexão com a BD. Tenta adicionar o aluno ao curso com **courseDB.addStudentCourse**(studentId, courseId). Redireciona para "/admin".

`@PostMapping("/addDisciplineToCourse")` Função: Adiciona uma disciplina a um curso: Loga os IDs recebidos. Cria uma conexão com a BD. Tenta adicionar a disciplina ao curso com **disciplineDB.addCourseDiscipline**(courseId, disciplineId). Redireciona para "/admin".

`@PostMapping("/addProfToDiscipline")` Função: Adiciona um professor a uma disciplina: Loga os IDs recebidos. Cria uma conexão com a BD. Tenta adicionar o professor à disciplina com **professorDB.addProfessorDiscipline**(professorId, disciplineId). Redireciona para "/admin".

`@PostMapping("/courseDetails")` Função: Exibe os detalhes de um curso: Loga o ID do curso recebido. Cria uma conexão com a BD. Recupera o curso, disciplinas associadas e alunos inscritos. Cria um **ModelAndView** com esses dados. Retorna o **ModelAndView** para exibir os detalhes do curso.

4 Gestão de sessões e Segurança

A nossa aplicação permite que utilizadores apenas entrem nas páginas inicial (**home**) e na de login via URL. as outras páginas correspondentes só podem ser acedidas por meio da aplicação, assegurando a devida gestão de sessões e segurança da melhor forma. Por exemplo, a página de registo não pode ser acedida diretamente por: localhost:8080/register. Apenas abrindo a página inicial da nossa escola, a página de registo (assim como as dos Cursos, dos Alunos, etc) poderá ser acedida.

Em termos de código, a gestão de sessões e segurança da nossa aplicação foram garantidas através da implementação de duas classes, a **Tfil** e a **FilterConfig**.

A classe **Tfil** é um filtro servlet configurado para interceptar requisições que correspondem aos padrões de URL especificados no atributo **urlPatterns** da anotação **@WebFilter**. A função principal desta classe é verificar se a requisição está vindo de um referenciador válido e, caso contrário, redirecionar o usuário.

Componentes da Classe: - Anotação **@WebFilter**: Define os padrões de URL que o filtro irá interceptar. - **Logger**: Utiliza **Logger** para registrar informações e eventos. - Método **init**: Executado quando o filtro é inicializado. - Método **doFilter**: Intercepta cada requisição e resposta. - Converte **servletRequest** e **servletResponse** para **HttpServletRequest** e **HttpServletResponse**. - Obtém o referenciador (**Referer**) e o nome do servidor da requisição. - Loga o URI da requisição e o referenciador. - Verifica se o referenciador é nulo ou não contém o nome do servidor. - Se sim, redireciona para a página inicial (/). - Se não, continua o filtro com **filterChain.doFilter**. - Método **destroy**: Executado quando o filtro é destruído.

Esta classe garante que a requisição vem de uma página interna do mesmo servidor. Impede acessos diretos não autorizados, protegendo contra ataques de **CSRF** (**Cross-Site Request Forgery**). Reforça a segurança ao redirecionar os utilizadores que tentam acessar URLs protegidas sem um referenciador válido. Evita que usuários externos ou scripts maliciosos acessem áreas restritas da aplicação. A implementação do filtro contribui significativamente para a segurança da aplicação ao garantir que apenas requisições válidas e autorizadas sejam processadas, mantendo a integridade das sessões e protegendo contra ataques comuns na web.

A classe **FilterConfig** influencia a gestão de **sessões** e **segurança** da aplicação ao permitir a configuração e o registo de filtros que podem controlar o acesso a recursos específicos da aplicação, como áreas de **administração** e funcionalidades sensíveis. O filtro **Tfil**, registado para diferentes padrões de **URL**, pode implementar lógicas de segurança, como verificação de **autorização**, validação de **sessão**, prevenção de ataques **CSRF**, entre outros, contribuindo para a proteção e segurança da aplicação.

Componentes da Classe: - Anotação **@Bean**: Indica que o método **loggingFilter()** retorna um objeto que será gerenciado pelo container **Spring** como um bean. - Método **loggingFilter()**: - Retorna um objeto **FilterRegistrationBean¡Tfil¿**, que é uma classe fornecida pelo **Spring Framework** para registrar filtros **Servlet** de forma programática. - Configura o filtro **Tfil** para os padrões de URL especificados usando o método **addUrlPatterns()**. - Cada padrão de **URL** representa uma área específica da aplicação: **administração**, registo, es-

tudante, professor, curso, disciplina. - Instância de **FilterRegistrationBean**: É criada para encapsular o filtro **Tfil** e as configurações associadas a ele. Permite a definição de parâmetros, como padrões de URL, ordem de execução, entre outros. - Filtro **Tfil**: É instanciado e configurado para processar as requisições que correspondem aos padrões de URL especificados. A lógica do filtro pode incluir verificações de segurança, controle de acesso, registro de logs, entre outros.

5 Conclusão - Descrição da interface

Em conclusão do relatório do nosso trabalho, faremos uma explicação da interface do utilizador ao abrir a nossa aplicação.

Ao entrar no website da nossa escola de teatro, o utilizador tem algumas informações na landing page, como uma pequena descrição da escola e uma lista dos cursos que oferece. É possível clicar em cada curso para ver a sua respetiva página. Há também uma opção 'Professores' que leva à página onde são listados todos os professores da escola. No canto superior direito encontra-se a opção 'Login'. Ao fazer login com uma conta válida, dependendo do tipo de utilizador a app redireciona para as respetivas páginas. As páginas de Aluno e Professor são páginas simples com informações relativas ao utilizador em questão. A página 'Admin' tem controlo sobre praticamente toda a WebApp, lista Administradores, Professores e Alunos, e tem as opcões adicionar, remover ou editar para cada um dos tipos de utilizador. Lista as Disciplinas da escola, permitindo adicionar novas disciplinas ou editar/remover as existentes. Além disso, lista os Cursos disponíveis permitindo também, adicionar, editar e remover, e ainda adiconar um aluno ou uma disciplina ao respetivo curso. Clicando em cada curso a WebApp redireciona para a página do curso em questão. Finalmente a página de Admin tem ainda uma secção 'Estatísticas' que lista o 'Total de alunos', 'Total de professores', 'Curso com mais alunos' e 'Curso mais caro'.

Em suma, este trabalho serviu para todos os membros do grupo aprofundarem e relacionarem corretamente conhecimentos em Java, da framework Spring, em requisições HTTP, em arquitetura RestFUL, em SQL e modelos relacionais de bases de dados, e também em gestão de logins e registos de forma segura numa aplicação. A correta implementação e correlação de todos estes conceitos permitiu chegar a um resultado final que satistaz os 3 membros do grupo.

Departamento de
Informática