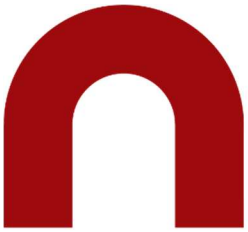

Instituto Superior de Engenharia

Politécnico de Coimbra

Relatório de Meta2 – Programação Distribuída

Daniel Albino - 2020134077

Miguel Neves -2020146521

Nuno Domingues – 2020109910

Conteúdo

1. Introdução

Neste breve relatório, estão especificados detalhes relativos à meta 2 do trabalho prático de programação distribuída, que essencialmente, foca-se especificação do um serviço remoto RMI e da API REST, no contexto do trabalho.

2. RMI

Seguindo o trabalho desenvolvido para a meta 1 do trabalho prático, foi adicionado um módulo RMI, ao servidor que fornece um serviço remoto.

Para isso foi definido no servidor, uma classe rmiService que estende da classe UnicastRemoteObject e implementa uma interface RemoteInterface, onde contém os métodos do serviço.

```
public class rmiService extends UnicastRemoteObject implements RemoteInterface {
    2 usages
    public static final String SERVICE_NAME = "SHOW_SERVICE_";
```

Figura 1 - Classe rmiService

```java
import java.rmi.Remote;
import java.rmi.RemoteException;

3 usages  1 implementation
public interface RemoteInterface extends Remote {

    1 implementation
    public void addListener (_NotificationListeners listener) throws java.rmi.RemoteException;

    1 implementation
    public void removeListener (_NotificationListeners listener) throws java.rmi.RemoteException;

    1 implementation
    public String getServers() throws RemoteException;

}
```

Figura 2 – RemoteInterface

Depois de definidas do “Main” do servidor é instanciado um objeto rmiService, registando esse serviço na máquina em que está a correr.

```java
try {
    LocateRegistry.createRegistry(Registry.REGISTRY_PORT);
} catch (RemoteException e) {
    System.out.println("Registry provavelmente ja' em execucao na maquina local!");
}

System.out.println("Servico GetRemoteFile criado e em execucao!");

Naming.bind( name: "rmi://"+ InetAddress.getLocalHost().getHostAddress() + "/" + SERVICE_NAME+port,  obj: this);

System.out.println("Servico " + SERVICE_NAME + " registado no registry...");
```

Figura 3 - Registo e "bind" do serviço

```java
@Override
public void addListener(_NotificationListeners listener) throws RemoteException {
    System.out.println ("[+] Adding listener: " + listener);
    listeners.add(listener);
}

@Override
public void removeListener(_NotificationListeners listener) throws RemoteException {
    System.out.println ("[-] Removing listener: " + listener);
    listeners.remove(listener);
}

@Override
public synchronized String getServers() throws RemoteException {
    return listServer.getListServer().toString();
}
```

Figura 4 - Implementação dos métodos do Remoteinterface

Ainda temos implementado uma outra aplicação autónoma que essencialmente serve de “listener”, que permite notificações assíncronas quando o servidor notificar os mesmos.

```java
public void notifyListeners(NotifyCode notifyCode, String phrase) {
    System.out.println("[ ] Notifying changes...");
    for (_NotificationListeners listener : listeners)
        try {
            switch (notifyCode) {
                case CLIENTUDP -> listener.ClientReceivedUDP(phrase);
                case CLIENTTCP -> listener.ClientConnectionTCP(phrase);
                case LOSTCONNECTIONTCP -> listener.LostConnectionTCP(phrase);
                case LOGINCLIENT -> listener.LoginClient(phrase);
                case LOGOUTCLIENT -> listener.LogoutClient(phrase);
            }
        } catch (RemoteException e) {
            throw new RuntimeException(e);
        }
}
```

Figura 5 - função de notificação do servidor

```java
import java.rmi.Remote;
import java.rmi.RemoteException;

8 usages
public interface _NotificationListeners extends Remote {

    1 usage
    public void ClientReceivedUDP(String phrase) throws RemoteException;

    1 usage
    public void ClientConnectionTCP(String phrase) throws RemoteException;

    1 usage
    public void LostConnectionTCP(String phrase) throws RemoteException;

    1 usage
    public void LoginClient(String phrase) throws RemoteException;

    1 usage
    public void LogoutClient(String phrase) throws RemoteException;
}
```

Figura 6 - Interface dos listerners

Essencialmente os listeners fazem um “Naming.lookup” no serviço do servidor e são adicionados com listeners do servidor.

```
public void init() throws RemoteException {
    try {

        String registration = "rmi://" + this.register +"/"+ SERVICE_NAME+porto;

        System.out.println("Registo:"+registration);

        Remote remoteService = Naming.lookup(registration);
        remote = (RemoteInterface) remoteService;



        keepGoing = true;
        th = new threadListserver(keepGoing,remote);
        th.start();

        String servers = remote.getServers();
        System.out.println("Servers:"+servers);



        remote.addListener(this);
```

Figura 7 - "lookup" do listener

3. API Rest

Para a utilização da API Rest (Spring Boot), implementamos os modelos, serviços e controladores, para uma base de dados relativa ao trabalho prático.

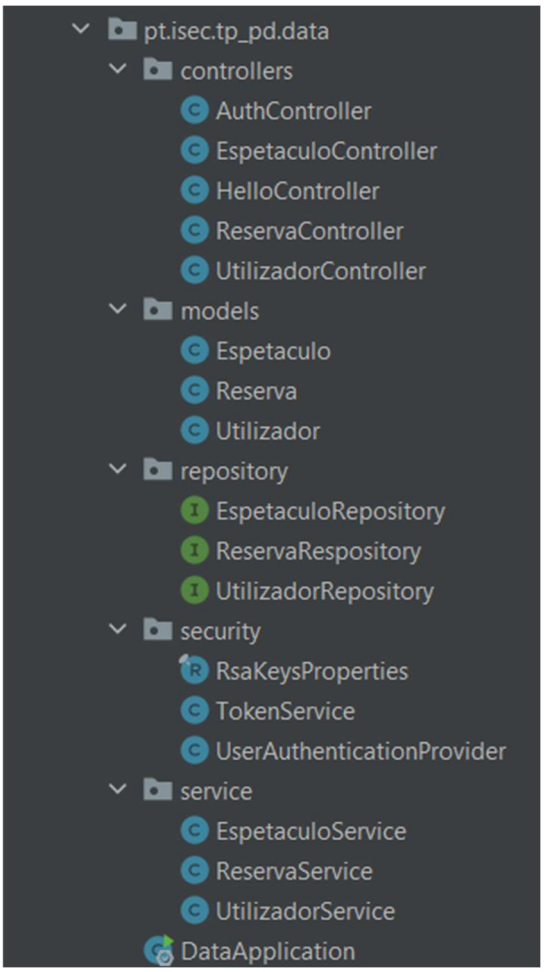

Figura 8 - API Rest

No login, se bem-sucedido, o utilizador fica autenticado

```
@Bean
public SecurityFilterChain loginFilterChain(HttpSecurity http) throws Exception {
    return http
            .csrf(csrf -> csrf.disable())
            .securityMatcher( ...patterns: "/login")
            .authorizeHttpRequests(auth -> auth.anyRequest().authenticated())
            .httpBasic(Customizer.withDefaults())
            .sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
            .build();
}
```

Figura 9 - LoginFilter

Qualquer utilizador(autenticado/não autenticado) que aceda à URL:
“http://localhost/espetaculo/all” , será redirecionado para a mesma.

```
@Bean
public SecurityFilterChain unauthenticatedFilterChain(HttpSecurity http) throws Exception {
    return http
            .csrf(csrf -> csrf.disable())
            .securityMatcher( ...patterns: "/hello", "/hello/**","/espetaculo/all")
            .authorizeHttpRequests(auth -> auth.anyRequest().permitAll())
            .sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
            .build();
}
```

Figura 10 - Não-autenticado fllter

Na autenticação, damos dois tipos de Autoridade: “ADMIN” ou “CLIENT”, consoante as credenciais introduzidas.

Caso um utilizador não exista na base de dados é lançada uma exceção BadCredentials.

```java
@Override
public Authentication authenticate(Authentication authentication) throws AuthenticationException {
    String username = authentication.getName();
    String password = authentication.getCredentials().toString();
    List<Utilizador> utilizadors=null;

    System.out.println("user:"+username);
    if (username.equals("admin") && password.equals("admin")) {
        List<GrantedAuthority> authorities = new ArrayList<>();
        authorities.add(new SimpleGrantedAuthority( role: "ADMIN"));
        return new UsernamePasswordAuthenticationToken(username, password, authorities);
    }else if((utilizadors=repo.findByUsernameWhere(username,password))!=null){
        System.out.println(utilizadors.toString());
        if(utilizadors.size()!=0) {
            List<GrantedAuthority> authorities = new ArrayList<>();
            authorities.add(new SimpleGrantedAuthority( role: "CLIENT"));
            return new UsernamePasswordAuthenticationToken(username, password, authorities);
        }else{
            throw new BadCredentialsException("Bad Credentials");
        }
    }
    return null;
}
```

Figura 11 – Autenticação

3.1. Controladores

No Controlador de utilizador, temos:

@GetMapping: URL: “http://localhost/utilizadores/all” → retorna todos os utilizadores, caso o utilizador esteja autenticado como “ADMIN”.

@PostMapping: URL: “http://localhost/utilizadores” com um corpo em JSON(“username”,”password”,…) → Cria um novo utilizador, caso o utilizador esteja autenticado como “ADMIN”.

@DeleteMapping: URL: “http://localhost/utilizadores/{id}” (com o parâmetro da URL) → Apaga um utilizador através do id do utilizador, caso o utilizador esteja autenticado como “ADMIN”.

No controlador da reserva, temos:

@GetMapping: URL: “http://localhost/reserva/alluser” → retorna as reservas feitas de um utilizador

É possível passar parâmetros no URL, para restringir a pesquisa por reservas pagas e não pagas.

No controlador do espetáculo, temos:

@GetMapping: URL: “http://localhost/espetaculo/all” → retorna os espetáculos.

Não necessita de haver qualquer tipo de autenticação.

É possível, ainda passar parâmetros no URL, para restringir a pesquisa por data de início e fim.

Nota: Todas estas ações podem ser verificadas com o Postman.