Guia da Arquitetura RMI: Os 4 Papéis

Quando você constrói um sistema RMI, você está, na verdade, montando um time com 4 tipos de "jogadores" (arquivos). Cada um tem uma função muito específica.
Aqui estão os 4 papéis e como identificá-los:

1. **O "Contrato" (A Interface Remota)**
2. **O "Servidor" (A Implementação Real)**
3. **Os "Dados" (As Classes Transportáveis)**
4. **O "Cliente" (O Consumidor do Serviço)**

Papel 1: O "Contrato" (A Interface Remota)

É aqui que tudo começa. O Contrato é o "cardápio" do seu restaurante. Ele diz ao mundo quais métodos o seu servidor oferece, mas não como eles são feitos.

- **Como identificar:** public interface ... extends java.rmi.Remote
- **Exemplo:** ServicoLoja.java (ou Servico.java do leilão [cite: Leilao/Servico.java]).

Regras-Chave:

1. **Deve ser uma interface**, não uma class.
2. **Deve estender java.rmi.Remote**. Isso é uma "etiqueta" que diz ao Java que esta interface pode ser chamada pela rede.
3. **Todos os métodos** na interface **DEVEM** declarar que lançam throws java.rmi.RemoteException. Isso é obrigatório, pois qualquer chamada de rede pode falhar.

O que vai dentro?

- **Apenas assinaturas de métodos.** Nada de lógica, nada de if, for, while.
- É o "O QUÊ", não o "COMO".

```
// Exemplo de "Contrato"
import java.rmi.Remote;
import java.rmi.RemoteException;

// REGRA 1: É uma interface
// REGRA 2: Extende Remote
public interface ServicoLoja extends Remote {

   // REGRA 3: Todos os métodos lançam RemoteException
   public RespostaPedido fazerPedido(String nomeCliente,
                                     String idProduto,
                                     int quantidade,
                                     Categoria categoria)
                                     throws RemoteException;
```

```
    // Você pode ter quantos métodos quiser
    public int consultarEstoque(String idProduto) throws RemoteException;
}
```

Papel 2: O "Servidor" (A Implementação Real)

Este é o "cérebro" do sistema. É a "cozinha" do restaurante. Ele pega os pedidos do "cardápio" (o Contrato) e executa a lógica de verdade.

- **Como identificar:** public class ... extends UnicastRemoteObject implements [SeuContrato]
- **Exemplo:** ServidorLoja.java (ou ServidorLeilao.java [cite: Leilao/ServidorLeilao.java]).

Regras-Chave:

1. **Deve ser uma class**.
2. **Deve estender java.rmi.server.UnicastRemoteObject**. Isso transforma sua classe em um "servidor" real que pode escutar na rede.
3. **Deve implementar o "Contrato"** (Papel 1). Ex: implements ServicoLoja.
4. Deve ter um **construtor** que chama super() e lança RemoteException.
5. Deve ter um método **main()** para iniciar o Registry (o "catálogo telefônico" do RMI) e se registrar nele usando registry.rebind(...).

O que vai dentro?

- **TODA a lógica de negócios.**
- As filas (PriorityBlockingQueue), os Semaphores, as threads (ProcessadorPedidos), o ConcurrentHashMap do estoque... tudo isso vive aqui.
- É o "COMO" o trabalho é feito.

```
// Exemplo de "Servidor"
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
// ... (imports de concorrência)

// REGRA 1: É uma classe
// REGRA 2: Extende UnicastRemoteObject
// REGRA 3: Implementa o Contrato (ServicoLoja)
public class ServidorLoja extends UnicastRemoteObject implements ServicoLoja {

    // Toda a lógica vive aqui:
    private final PriorityBlockingQueue<RequisicaoPedido> filaPedidos;
    private final ProcessadorPedidos processador;
```

```
    // ... (estoque, semáforo, etc.)

    // REGRA 4: Construtor obrigatório
    public ServidorLoja() throws RemoteException {
        super(); // Importante!
        // Inicializa a lógica interna
        this.filaPedidos = new PriorityBlockingQueue<>();
        // ... (etc.)
        this.processador = new ProcessadorPedidos(filaPedidos, ...);
        new Thread(processador).start();
    }

    // Implementação real dos métodos do Contrato
    @Override
    public RespostaPedido fazerPedido(String nomeCliente, String idProduto, int qtde,
Categoria cat)
            throws RemoteException {
        // A lógica do servidor (enfileirar e esperar)
        RequisicaoPedido req = new RequisicaoPedido(nomeCliente, idProduto, qtde, cat);
        filaPedidos.put(req);
        // ... (espera a resposta da SynchronousQueue)
        return resposta; // (simplificado)
    }

    @Override
    public int consultarEstoque(String idProduto) throws RemoteException {
        // ... (lógica para consultar o estoque)
        return 0;
    }

    // REGRA 5: O main() para se registrar
    public static void main(String[] args) {
        try {
            Registry registry = LocateRegistry.createRegistry(1099);
            // "Publica" este servidor no catálogo com o nome "LojaService"
            registry.rebind("LojaService", new ServidorLoja());
            System.out.println("Servidor da Loja no ar!");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Papel 3: Os "Dados" (Objetos Transportáveis)

Quando o Cliente chama fazerPedido(...), ele precisa enviar os dados (como Categoria.BASIC). Quando o Servidor responde, ele precisa enviar dados de volta (o RespostaPedido).
Esses objetos são os "pacotes" ou "caixas" que viajam pela rede. Para o Java saber como "empacotar" e "desempacotar" sua classe, ela precisa ser Serializable.

- **Como identificar:** public class ... implements java.io.Serializable
- **Exemplo:** RespostaPedido.java, Categoria.java (Enums também!), Cliente.java (do leilão [cite: Leilao/Cliente.java]).

Regras-Chave:

1. **Deve ser uma class ou enum**.
2. **Deve implementar java.io.Serializable**. Isso é uma "etiqueta" que diz ao Java: "Eu autorizo que esta classe seja transformada em bytes e enviada pela rede".
3. **Todos os atributos (campos) DENTRO da classe** também devem ser serializáveis (tipos primitivos como int, boolean, String ou outras classes que também sejam Serializable).

O que vai dentro?

- **Apenas dados (atributos) e métodos para acessá-los (getters/setters).**
- **NÃO** coloque lógica de negócios complexa, threads, filas, etc., aqui. É só uma "caixa" de dados.

```
// Exemplo de "Dados"
import java.io.Serializable;

// REGRA 1: É uma classe
// REGRA 2: Implementa Serializable
public class RespostaPedido implements Serializable {

   // REGRA 3: Todos os campos são serializáveis
   // (boolean, String são serializáveis por padrão)
   private final boolean sucesso;
   private final String codigoPedido;
   private final String mensagemErro;

   // Construtores, getters, etc.
   public RespostaPedido(boolean sucesso, ...) { ... }
   public boolean getSucesso() { ... }
   public String getCodigoPedido() { ... }
}
```

Papel 4: O "Cliente" (A Aplicação Consumidora)

Este é o programa que o usuário final executa. Ele não sabe onde o servidor está (poderia estar em outro país). Ele só conhece o "Contrato" (o cardápio) e o nome do serviço no "catálogo" (Registry).

- **Como identificar:** É uma classe normal com main() que usa LocateRegistry.getRegistry(...) e registry.lookup(...).
- **Exemplo:** ClienteLoja.java (ou ClienteLeilao.java [cite: Leilao/ClienteLeilao.java]).

Regras-Chave:

1. É uma classe normal. **Não estende nem implementa** nada de UnicastRemoteObject ou Remote.
2. Usa LocateRegistry.getRegistry(...) para se conectar ao "catálogo telefônico" do RMI.
3. Usa registry.lookup("NomeDoServico") para "pedir" o serviço.
4. **Importante:** Ele faz o *cast* (conversão) do resultado do lookup para o **"Contrato" (Papel 1)**, não para o Servidor. O Cliente nunca conhece a classe ServidorLoja, ele só conhece a interface ServicoLoja.

O que vai dentro?

- Lógica de interface do usuário (ex: Scanner, menus, etc.).
- A chamada dos métodos remotos.

```
// Exemplo de "Cliente"
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

// REGRA 1: Classe normal
public class ClienteLoja {
   public static void main(String[] args) {
      try {
         // REGRA 2: Encontra o catálogo
         Registry registry = LocateRegistry.getRegistry("localhost", 1099);

         // REGRA 3: Procura o serviço pelo NOME
         // REGRA 4: Faz o cast para a INTERFACE (Contrato), não para a Classe Servidor
         ServicoLoja servico = (ServicoLoja) registry.lookup("LojaService");

         // Agora, usa o serviço como se fosse um objeto local!
         // O RMI cuida de toda a mágica da rede por baixo dos panos.
         RespostaPedido resp = servico.fazerPedido("Cliente Ana", "ProdutoA", 10,
Categoria.BASIC);

         if (resp.getSucesso()) {
```

```
            System.out.println("Sucesso! Cód: " + resp.getCodigoPedido());
        } else {
            System.out.println("Falha: " + resp.getMensagemErro());
        }

    } catch (Exception e) {
        e.printStackTrace();
    }
  }
}
```

Resumo: Quem Chama o Quê? (A História Completa)

1. **Início:** O **Servidor** (Papel 2) é executado. Ele cria o Registry (catálogo) e se registra: registry.rebind("LojaService", ...).
2. **Conexão:** O **Cliente** (Papel 4) é executado. Ele encontra o Registry e pede por "LojaService": registry.lookup("LojaService").
3. **Mágica do RMI:** O RMI entrega ao Cliente um "Stub" (um objeto fantasma) que *se parece* com o **Contrato** (Papel 1).
4. **Chamada:** O Cliente chama um método, ex: servico.fazerPedido(...).
5. **Empacotamento:** O RMI vê que Categoria.BASIC é um **Dado** (Papel 3, Serializable) e o "empacota" (serializa) para a rede.
6. **Execução:** O **Servidor** (Papel 2) recebe a chamada, executa a lógica real (filas, threads, etc.) e cria um RespostaPedido.
7. **Retorno:** O RMI vê que RespostaPedido é um **Dado** (Papel 3, Serializable), o "empacota" e envia de volta para o Cliente.
8. **Fim:** O Cliente recebe o objeto RespostaPedido como se o método tivesse sido executado em sua própria máquina.

Espero que este guia ajude a solidificar como as partes se encaixam!