Observer

```
public interface ObserverRI extends Remote {
    public void update() throws RemoteException;
}



public class ObserverImpl extends UnicastRemoteObject implements ObserverRI {

    private boolean lastObserverState;
    protected SubjectRI subjectRI;


    public ObserverImpl(String id, ObserverGuiClient f, SubjectRI subjectRI) throws
RemoteException {
        super();
        this.state=false;
        this.subjectRI=subjectRI;
        this.subjectRI.attach(this);
    }

    @Override
    public void update() {
        this.state=subjectRI.getState();
    }


public interface SubjectRI  extends Remote {

    public void attach(ObserverRI obsRI) throws RemoteException;
    public void detach(ObserverRI obsRI) throws RemoteException;
    public State getState() throws RemoteException;
    public void setState (State state) throws RemoteException;
    public void notifyAllObservers() throws RemoteException;
}

public class SubjectImpl extends UnicastRemoteObject implements SubjectRI {

    private State subjectState;
    private final ArrayList<ObserverRI> observers = new ArrayList<>();
```

```
    protected SubjectImpl() throws RemoteException {
        super();
        this.subjectState = false;
    }


    @Override
    public void attach(ObserverRI obsRI) {
          this.observers.add(obsRI);
    }

    @Override
    public void detach(ObserverRI obsRI) {
          this.observers.remove(obsRI);
    }
     @Override
     public void notifyAllObservers(){
        for(ObserverRI obs: observers)
            obs.update();
    }
}
```

Factory/Session

```
public interface FactoryRI extends Remote {

    public SessionRI login(String uname, String pw) throws RemoteException;

    public boolean register(String uname, String pw) throws RemoteException;

}

public class FactoryImpl extends UnicastRemoteObject implements FactoryRI {

    //String - username | SessionRI - Session respetiva
    private HashMap<String, SessionRI> sessions;

    private DB db;
```

```
    public FactoryImpl() throws RemoteException {
        super();
        this.db = DB.getInstance();
        this.sessions = new HashMap<>();
    }


    @Override
    public boolean register(String uname, String pw) throws RemoteException {
        return db.registerUser(uname, pw);
    }


    @Override
    public SessionRI login(String uname, String pw) throws RemoteException {
        if(db.existsUser(uname,pw)){
            if(!sessions.containsKey(uname)){
                SessionRI sessionRI = new SessionImpl(db,
          db.getUserByCredentials(uname,pw));
                this.sessions.put(uname, sessionRI);
                return sessionRI;

            }
            //caso já exista alguém logado na session
            else{
                return sessions.get(uname);
            }
        }
        return null;
    }
}

public interface SessionRI extends Remote{

    public void list() throws RemoteException;
}

public class SessionImpl extends UnicastRemoteObject implements SessionRI {

    private DB db;
```

```
    public SessionImpl(DB db, User user) throws RemoteException {
        super();
        this.db = db;
    }

    @Override
    public void list() {

    }


public class Client {

    private SetupContextRMI contextRMI;
    private FactoryRI factoryRI;

    public Client(String[] args) {
        try {
            String registryIP = args[0];
            String registryPort = args[1];
            String serviceName = args[2];
            contextRMI = new SetupContextRMI(this.getClass(), registryIP, registryPort,
new String[]{serviceName});
        } catch (RemoteException e) {
            Logger.getLogger(Client.class.getName()).log(Level.SEVERE, null, e);
        }
    }

 public static void main(String[] args){
        if (args != null && args.length < 3) {
            System.exit(-1);
        } else {
            assert args != null;
            //1. ============ Setup client RMI context ============
            Client client = new Client(args);
            //2. ============ Lookup service ============
            client.lookupService();
            //3. ============ Play with service ============
            client.playService();
        }
    }
```

```
public class Server {
   private SetupContextRMI contextRMI;

   private FactoryRI factoryRI;

 public Server(String[] args){
       try {
          String registryIP   = args[0];
          String registryPort = args[1];
          String serviceName  = args[2];
          contextRMI = new SetupContextRMI(this.getClass(), registryIP, registryPort,
new String[]{serviceName});
       } catch (RemoteException e) {
          Logger.getLogger(this.getClass().getName()).log(Level.SEVERE, null, e);
       }
   }

   public static void main(String[] args) {

       if (args != null && args.length < 3) {
          System.exit(-1);
       } else {
          assert args != null;
          Server srv = new Server(args);
          srv.rebindService();
       }
   }
}
```

RabbitMQ

```
try {
          ConnectionFactory factory = new ConnectionFactory();
          factory.setHost("localhost");

          factory.setUsername("guest");

          factory.setPassword("guest4rabbitmq");
          Connection connection=factory.newConnection();
          Channel channel=connection.createChannel();
```

```
    channel.queueDeclare(Send.QUEUE_NAME, false, false, false, null);

    DeliverCallback deliverCallback = (consumerTag, delivery) -> {
        String message = new String(delivery.getBody(), "UTF-8");
        System.out.println(" [x] Received '" + message + "'");
    };


    channel.basicConsume(Send.QUEUE_NAME, true, deliverCallback, consumerTag -> { });


        String message="Hello World!";
        channel.basicPublish("", QUEUE_NAME, null, message.getBytes(StandardCharsets.UTF_8));


} catch (Exception e){

    e.printStackTrace();
}
```