UNIVERSIDADE FEDERAL DO ESTADO DO RIO DE JANEIRO
CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA

Relatórios Técnicos
do Departamento de Informática Aplicada
da UNIRIO
n°0014/2009

Desenvolvendo web services no BEA Workshop for WebLogic Platform

Leonardo Guerreiro Azevedo
Henrique Prado Sousa
Fernanda Baião
Flávia Santoro

Departamento de Informática Aplicada

UNIVERSIDADE FEDERAL DO ESTADO DO RIO DE JANEIRO
Av. Pasteur, 458, Urca - CEP 22290-240
RIO DE JANEIRO – BRASIL

Projeto de Pesquisa

Grupo de Pesquisa Participante

Patrocínio

PETROBRAS

Relatórios Técnicos do DIA/UNIRIO, No. 0014/2009
Editor: Prof. Sean W. M. Siqueira Agosto, 2009

Desenvolvendo web services no BEA Workshop for WebLogic Platform*

Leonardo Guerreiro Azevedo, Henrique Prado Sousa, Fernanda Baião, Flavia Santoro

Núcleo de Pesquisa e Prática em Tecnologia (NP2Tec)
Departamento de Informática Aplicada (DIA) – Universidade Federal do Estado do Rio de Janeiro (UNIRIO)

azevedo@uniriotec.br, henrique.souza@uniriotec.br, fernanda.baiao@uniriotec.br, flavia.santoro@uniriotec.br

Abstract. Web services is the main technologic for service implementation in a SOA (Service Oriented Architecture) approach. Therefore, it is very important to know how to develop web services in a well known platform. The main goal of this work is to present a methodology for service development, as well as details about how to implement web services and execute query on databases. The BEA Workshop for WebLogic Platform and PostgreSQL were chosen as IDE for service development and database to execute the queries, respectively.

Keywords: SOA, web services, modeling, BEA Workshop for WebLogic Platform, PostgreSQL.

Resumo. Web services é a principal tecnologia para implementação de serviços em uma arquitetura orientada a serviços (SOA – Service Orientede Architecture). Logo, conhecer bem a implementação de web services em uma plataforma amplamente utilizada é intrínseco para a implantação de SOA. Este trabalho tem o objetivo de apresentar uma metodologia para desenvolvimento de serviços, bem como detalhes para a codificação de web services e conexão com banco de dados. Neste trabalho, a ferramenta escolhida para codificação de serviços foi a BEA Workshop for WebLogic Platform e o banco de dados PostgreSQL.

Palavras-chave: SOA, web services, modelagem, BEA Workshop for WebLogic Platform, PostgreSQL.

---

* Trabalho patrocinado pela Petrobras.

Sumário

1 Introdução

1.1 Motivação

A implantação de SOA em uma organização apresenta uma série de desafios, incluindo questões relacionadas a recursos de hardware e software (por exemplo, aplicações de software, dispositivos de hardware, servidores etc), infra-estrutura de ESB (*Enterprise Service Bus*) (tais como, segurança, integração de aplicações com processos de negócio etc), bem como questões relacionadas à modelagem, projeto, implementação e monitoramento e gestão de serviços [Papazoglou *et al.*, 2007].

Web services é a principal tecnologia para desenvolvimento de serviços [Erl, 2005]. Conhecer bem uma tecnologia para desenvolvimento de web services deve fazer parte de qualquer estratégia para implantação de uma arquitetura orientada a serviços (SOA).

1.2 Objetivo

O objetivo deste trabalho é apresentar detalhes da implementação de serviços na ferramenta BEA Workshop for WebLogic Platform[1], bem como detalhes do uso do componente componente Database Control[2] para acesso a banco de dados. Neste trabalho, o banco de dados PostgreSQL[3] foi utilizado.

O componente Database Control permite o acesso simples a bancos de dados relacionais. O Database Control automaticamente traduz consultas enviados ao banco de dados em objetos Java, de modo que os resultados das consultas podem ser facilmente acessados a partir destes objetos.

1.3 Metodologia de desenvolvimento

A metodologia de desenvolvimento aqui apresentada é bottom-up, ou seja, o serviço é gerado a partir da necessidade de acesso ao banco de dados. Em outras palavras, a partir de uma consulta realizada na base de dados, a qual foi solicitada por uma demanda de acesso a dados.

A partir da consulta, é criado um arquivo XSD para representar a estrutura dos elementos a serem retornados, os quais são:

- Tipo complexo representando a estrutura de retorno da consulta;
- Tipo complexo para representar uma lista de registros;
- Variável para o tipo complexo que representa a lista de objetos.

Além disso, é criada uma classe POJO (*Plain Old Java Object*) para representar os objetos retornados pela consulta.

O componente Database Control é utilizado para consultar a base de dados. Este componente retorna um conjunto de objetos Java, os quais devem ser transformados

[1] http://edocs.bea.com/workshop/docs92/platform.html

[2] http://edocs.bea.com/workshop/docs81/doc/en/core/index.html

[3] http://www.postgresql.org/

para um arquivo XML de acordo com a estrutura do arquivo XSD[4]. Este arquivo é retornado ao cliente que invocou o serviço.

O aplicativo SOAPUI[5] é utilizado para testar o WebService. Testes também podem ser realizados utilizando o Test Client, disponível no Workshop for WebLogic Platform, ou mesmo através da implementação de um cliente específico para acessar o WebService implementado. Estas formas de realizar testes também são apresentadas neste trabalho.

1.4 Estrutura do Relatório

O relatório está estruturado em elementos pré-textuais, 5 (cinco) capítulos, e elementos pós-textuais. Na primeira parte há itens como sumário e índices. Em seguida, estão os capítulos que estão brevemente descritos a seguir, e, por fim, os elementos pós-textuais.

O capítulo 1 explica a motivação e o objetivo para este trabalho, bem como apresenta uma descrição em alto nível da metodologia utilizada atualmente para desenvolvimento de serviços.

O capítulo 2 apresenta o passo-a-passo para desenvolvimento de serviços na GDIEP.

O capítulo 3 é dedicado a apresentar possibilidades para testes de serviços.

O capítulo 4 apresenta as conclusões do presente trabalho, listando algumas possibilidades de melhoria na metodologia atual de desenvolvimento.

O capítulo 5 apresenta as referências bibliográficas.

2 Etapas do desenvolvimento de serviços

Este capítulo apresenta o passo-a-passo para desenvolvimento de serviços.

2.1 Softwares utilizados

Os seguintes softwares foram utilizados para a implementação dos serviços.

1. BEA Workshop for WebLogic Platform 9.02
2. JDK 1.5
3. PostgreSQL
4. SOAPUI

2.2 Passo a passo para implementação de um serviço de consulta a dados de Unidade Operativa

A seguir são apresentados os detalhes para desenvolvimento de web services.

1. Criar projeto "Web Service Project" em File → New → Other... → Web Service → Web Service Project. Ao avançar, na tela "Web service Project", o target run-

[4] http://www.w3.org/XML/Schema

[5] http://www.soapui.org/

time “BEA Weblogic v9.2” deve ser escolhida. Avançando novamente, em “Project Facets”, a opção “Annotated Web Service Facets” deve estar marcada e a opção “XML Beans ➔ XMLBeans Builder” deve ser marcada.

a. Criar os seguintes pacotes no pacote “src”:

i. Services: classes webservices, por exemplo, br.uniriotec.services

ii. Controls: classes controls, por exemplo, br.uniriotec.controls

iii. Beans: classes POJO (Plain-Old Java Object) correspondente aos objetos lidos do banco, por exemplo, br.uniriotec.beans

2. Elaborar consulta para a tabela (ou conjunto de tabelas) que se deseja consultar, em ferramenta cliente do banco de dados.

a. Por exemplo:

```
select nome, cpf, id from cliente;
```

b. A partir desta consulta, definir os atributos que serão utilizados pelo POJO. Por exemplo:

```
BigDecimal id;
String cpf;
String nome;
```

3. Um arquivo XSD deve ser criado definindo a estrutura dos objetos a serem transferidos na mensagem de resposta do WebService. O componente utilizado para conexão com o banco de dados é o Database Control disponibilizado pelo Workshop. Este componente retorna um array de objetos POJO. Entretanto, no tráfego de mensagens entre WebServices são enviadas mensagens XML. Logo, os objetos POJO devem ser transformados em uma estrutura XML, de acordo com o XSD definido. São especificados três elementos complexos no XSD:

a. Definição da estrutura de cada registro da tabela *cliente*

```
<xs:complexType name="cliente">
      <xs:sequence>
            <xs:element name = "id" type="xs:decimal"  nillable="true"
      minOccurs="0" maxOccurs="1" />
            <xs:element name = "nome" type="xs:string"   nilla-
ble="true" minOccurs="0" maxOccurs="1" />
            <xs:element name = "cpf" type="xs:string" nillable="true"
minOccurs="0" maxOccurs="1" />
      </xs:sequence>
</xs:complexType>
```

b. Definição de uma lista de elementos *cliente*

```
<xs:complexType name="lstCliente">
      <xs:sequence>
            <xs:element name="Cliente" type="cliente" minOccurs="0"
maxOccurs="unbounded" />
      </xs:sequence>
</xs:complexType>
```

c. Definição de variável para a lista de elementos

```
<xs:element name="ClienteList" type="lstCliente" />
```

d. O arquivo XSD produzido correspondente à estrutura do XML que re-

presenta a classe *Cliente*, e deve ser criado no pacote "Schemas", seguindo-se o menu File ➔ New ➔ Other ➔ XML ➔ XML Schema.
Preencher o nome do arquivo com "Cliente.xsd". Após criar o arquivo, o targetNamespace e o nameSpace devem ser ajustados (através da edição do xsd) para o pacote do elemento "http://controls.beans/cliente". Além disso, deve ser criado o atributo xmlns:wld="http://www.bea.com/2002/10/weblogicdata".

```
<?xml version="1.0"?>
<xs:schema targetNamespace="http://controls.beans/cliente"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns="http://controls.beans/cliente" elementFormDefault="qualified"
xmlns:wld="http://www.bea.com/2002/10/weblogicdata"
attributeFormDefault="unqualified">

<xs:complexType name="cliente">
      <xs:sequence>
            <xs:element name = "id" type="xs:decimal"  nillable="true"
      minOccurs="0" maxOccurs="1" />
            <xs:element name = "nome" type="xs:string"
nillable="true" minOccurs="0" maxOccurs="1" />
            <xs:element name = "cpf" type="xs:string" nillable="true"
minOccurs="0" maxOccurs="1" />
      </xs:sequence>
</xs:complexType>
<xs:complexType name="lstCliente">
      <xs:sequence>
            <xs:element name="Cliente" type="cliente" minOccurs="0"
maxOccurs="unbounded" />
      </xs:sequence>
</xs:complexType>
<xs:element name="ClienteList" type="lstCliente" />
</xs:schema>
```

4. No pacote Bean ("br.uniriotec.beans") criar classe POJO para representar o elemento <Unidade Operativa> - Todos os atributos criados são privados com métodos get e set (Figura 1).

```java
package br.uniriotec.beans;

import java.math.BigDecimal;

public class Cliente {
    BigDecimal id;
    String cpf;
    String nome;

    public String getCpf() {
        return cpf;
    }
    public void setCpf(String cpf) {
        this.cpf = cpf;
    }
    public BigDecimal getId() {
        return id;
    }
    public void setId(BigDecimal id) {
        this.id = id;
    }
    public String getNome() {
        return nome;
    }
    public void setNome(String nome) {
        this.nome = nome;
    }

}
```

Figura 1 – Código da classe POJO Cliente

5. A partir do XSD criar arquivo JAR com os tipos de dados.
    a. No pacote "br.uniriotec.controls", clicar com o botão direito no arquivo Cliente.xsd ➔ Web Services ➔ "Generates Types JAR File..."
    b. Escolher o tipo "Apache XML Beans".
    c. O JAR criado fica na biblioteca do projeto, em "...\WebContent\WEB-INF\lib"
6. O próximo passo é criar um objeto Control para acessar a base de dados.
    a. Clicar em File ➔ New ➔ JDBC Control
    b. Nomear o objeto como: <nome do objeto> + DbControl (Figura 2), por exemplo, ClienteDbControl.

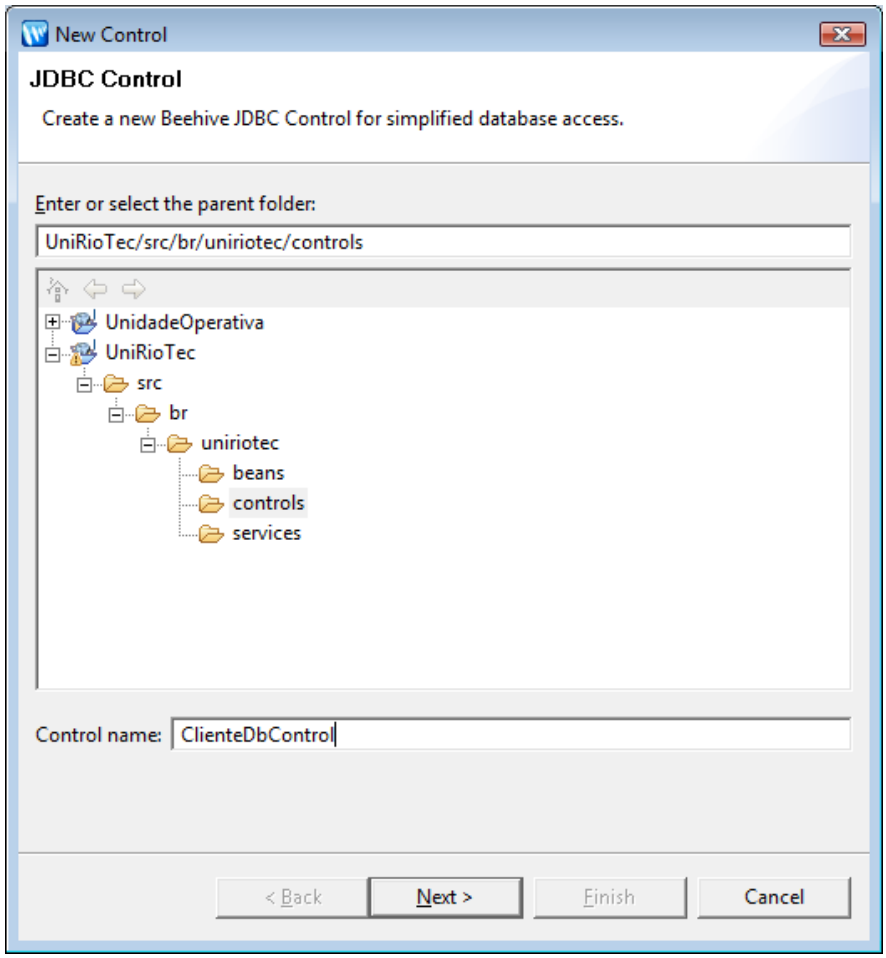

Figura 2 – Janela de criação do controle JBDC

Na criação do objeto Control, será solicitada a associação a um DataSource. O DataSource deve estar associado a um servidor, que por sua vez deve estar associado a um domínio. Deverão ser criados e associados, em ordem: o domínio, o servidor, o DataSource e por fim, o Control. Caso algum destes componentes não esteja criado, siga os passos 6.1, 6.2 e 6.3, e ao concluir estes passos, retorne para executar a letra o passo a seguir (c). Senão, se o domínio, servidor e DataSource já estiverem configurados, continue no passo seguinte (c).

c. Definir a conexão a ser utilizada, clicando em Browser (Figura 3).

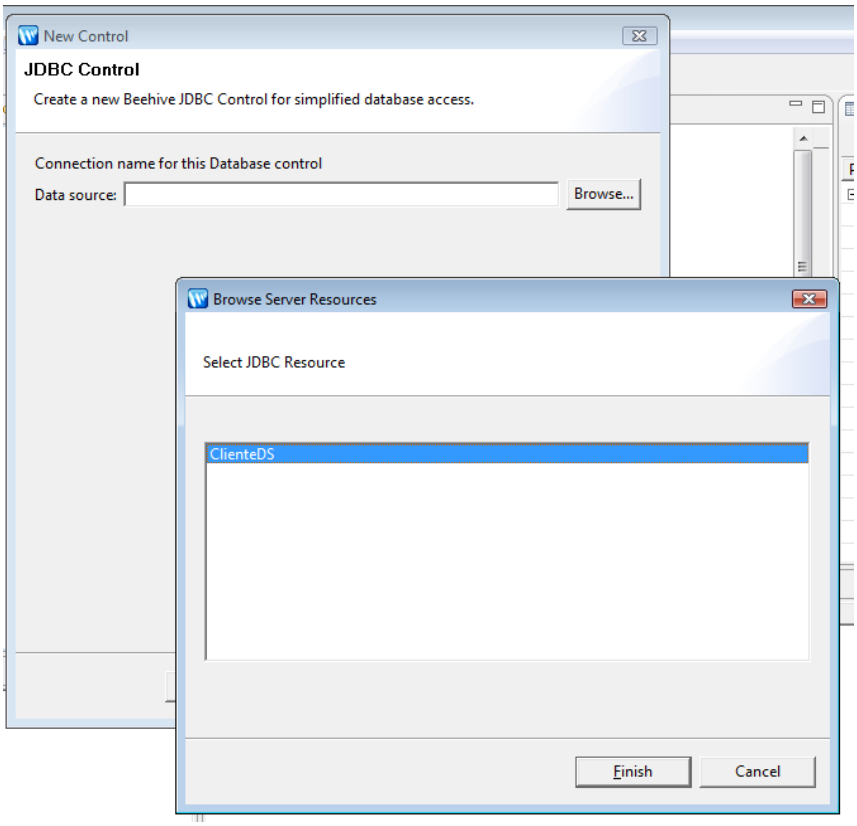

Figura 3 – Janela para definição do data source

d. Substituir o código gerado pela SQL elaborada anteriormente.

```
@JdbcControl.SQL(statement="select  nome, " +
                           "     cpf," +
                           "     id" +
                           "  FROM cliente" +
                           "  Where 1=1 " +
                           "  {sql: p_where} " +
                           "  {sql: p_groupby} " +
                           "  {sql: p_having} " +
                           "  {sql: p_orderby} ")
```

IMPORTANTE: O nome dos campos de retorno da consulta (projeção da consulta) devem ser iguais aos nomes dos atributos da classe POJO criada (Figura 1). O Database Control utiliza estes nomes para fazer introspecção e transformar os registros lidos do banco de dados em objetos da classe Java. Logo, por exemplo, se na tabela "Cliente" o nome da coluna id fosse "cliente_id" então na SQL deveria ser utilizado um alias para que fosse retornado id, por exemplo, "cliente_id as id".

e. Na consulta podem ser utilizados os parâmetros: {sql: } para código SQL e {var} para variável, quando for o caso.

```
                           "  {sql: p_where} " +
                           "  {sql: p_groupby} " +
                           "  {sql: p_having} " +
                           "  {sql: p_orderby} ")
```

f. Criar método que retorna um array de elementos.

```
Cliente[] getCliente(String p_where,
                                    String p_groupby,
                                    String p_having,
                                    String   p_orderby)   throws
SQLException;
```

g. É necessário importar a classe Cliente e SQLException.

```
import java.sql.SQLException;
import br.uniriotec.beans.Cliente;
```

h. A classe resultante é apresentada na Figura 4.

```
package br.uniriotec.controls;

import org.apache.beehive.controls.system.jdbc.JdbcControl;
import org.apache.beehive.controls.api.bean.ControlExtension;
import java.sql.SQLException;
import br.uniriotec.beans.Cliente;

@ControlExtension
@JdbcControl.ConnectionDataSource(jndiName = "ClienteDS")
public interface ClienteDbControl extends JdbcControl {

    @JdbcControl.SQL(statement="select  nome, " +
                    "     cpf," +
                    "     id" +
                    "  FROM cliente" +
                    "  Where 1=1 " +
                    "  {sql: p_where} " +
                    "  {sql: p_groupby} " +
                    "  {sql: p_having} " +
                    "  {sql: p_orderby} ")
    Cliente[] getCliente(String p_where,
                String p_groupby,
                String p_having,
                String p_orderby) throws SQLException;

    static final long serialVersionUID = 1L;
}
```

Figura 4 – Classe ClienteDbControl

6.1 Um domínio pode ser criado utilizando o "BEA Web Logic Cofiguration Wizard", que pode ser acessado através do menu padrão "BEA Products/Tools/Configuration Wizard" ou no wizard disponibilizado ao iniciar um servidor no Workshop.
Para criar o domínio através do "BEA WebLogic Cofiguration Wizard" faça:

a. Avançar na primeira tela (Figura 5).

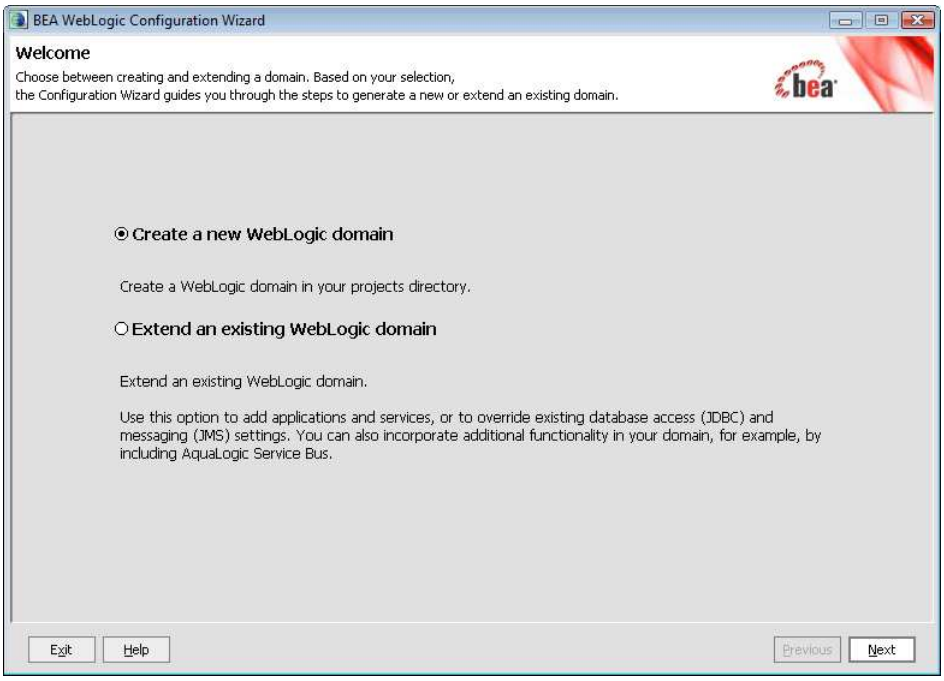

Figura 5 – Janela para criação de um novo domínio

b. Selecionar o template padrão e avançar (Figura 6).

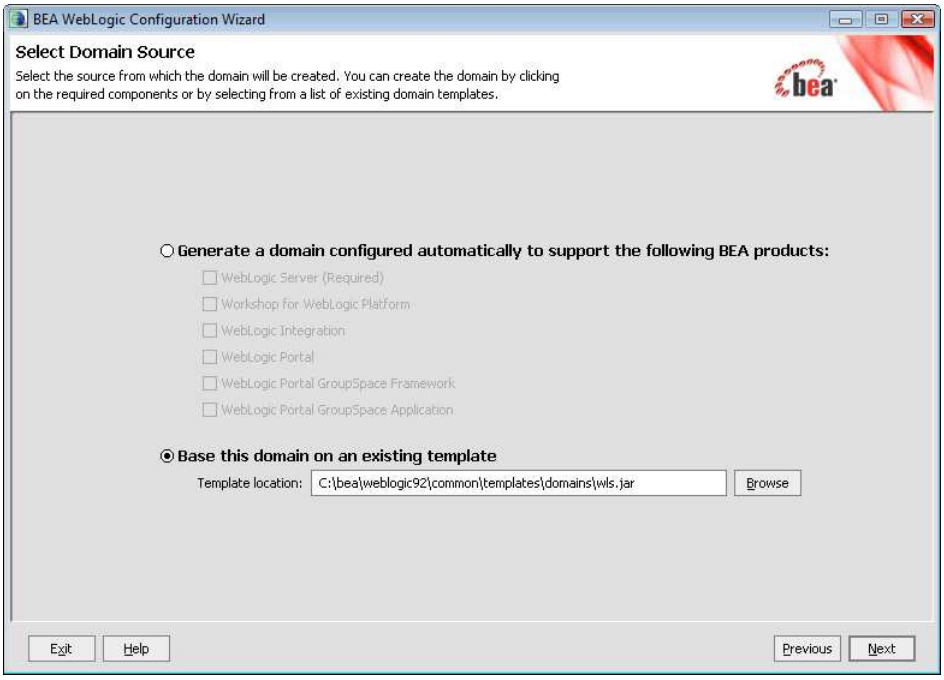

Figura 6 – Janela para seleção do template padrão

c. Inserir senha padrão (weblogic) e clicar em "Next" (Figura 7).

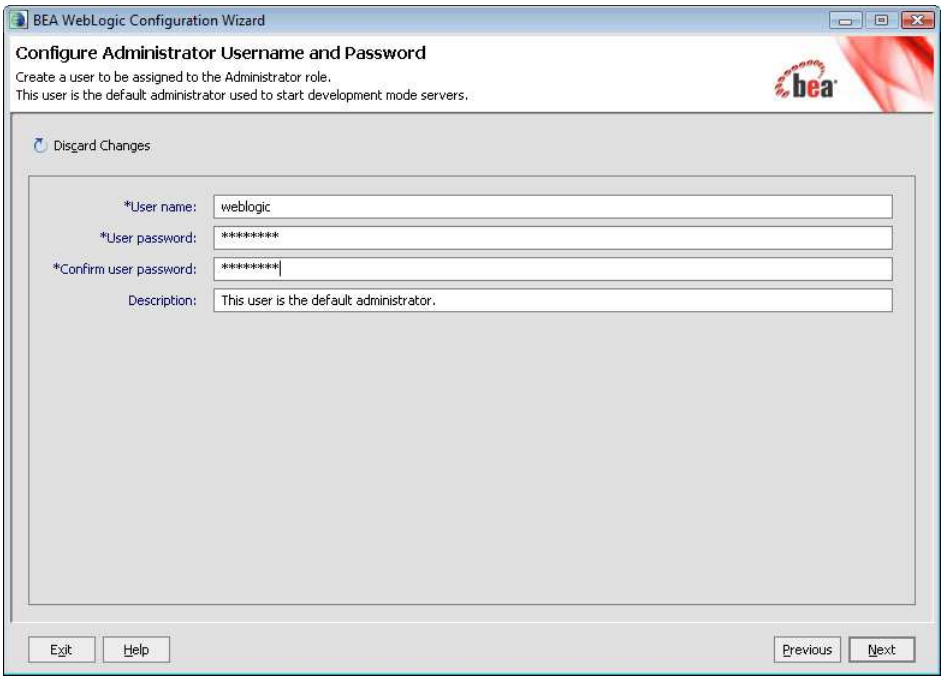

Figura 7 – Janela para configuração de nome de usuário e senha

d. Inserir um nome para o domínio e clicar em "Create" (Figura 8).

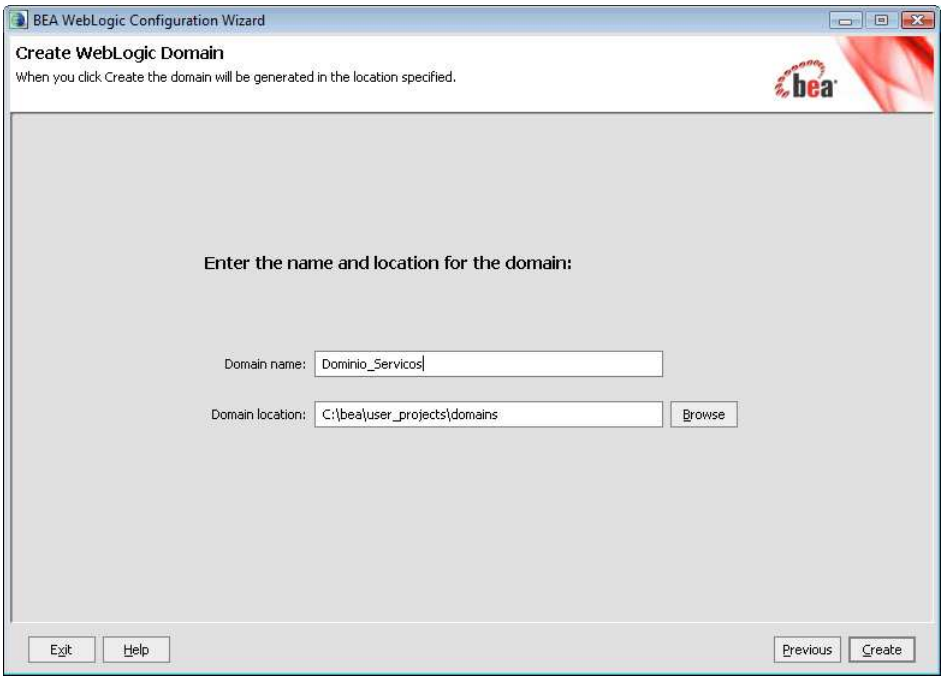

Figura 8 – Janela de configuração do domínio

e. Os domínios são criados por padrão no diretório "C:\bea\user_projects\domains", ou seja, neste diretório será criado o domínio Dominio_Servicos. Vá até este diretório e certifique-se que ele foi criado.

Para saber mais sobre domínios e criação de domínios, acessar http://edocs.bea.com/platform/docs81/confgwiz/intro.html#1052972.

6.2 Para criar o servidor faça:

a. No BEA Workshop for WebLogic Platform, abrir a janela de criação do servidor em File → New → Server, escolher o servidor BEA WebLogic v9.2 Server e clicar em "Next" (Figura 9).

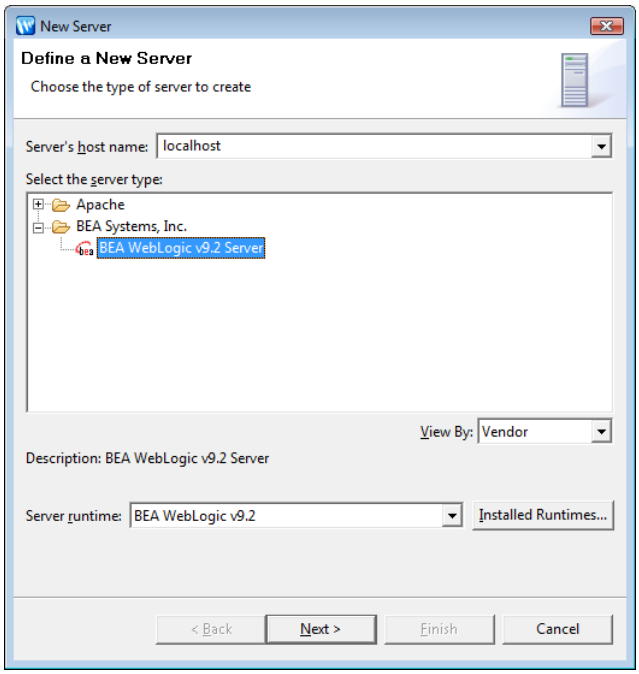

Figura 9 – Janela para criação de novo servidor

b. Escolher o domínio para instalar o servidor. Por padrão, os domínios estão instalados em C:\bea\user_projects\domains. Ex: O domínio criado na sessão anterior está em C:\bea\user_projects\domains\Dominio_Servicos. Após inserir o domínio, clicar em "Next" (Figura 10).

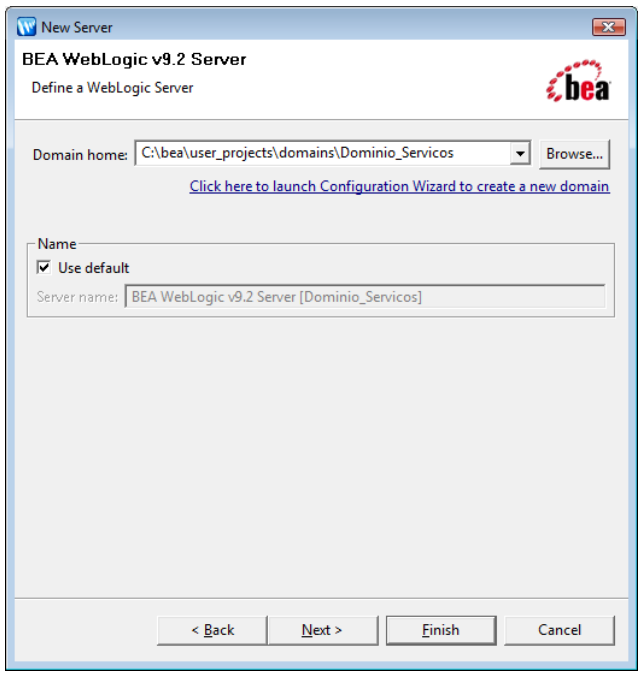

Figura 10 – Janela para configuração do domínio

c. Adicionar o projeto que está sendo desenvolvido para configuração do mesmo no servidor e clique no botão *Finish* (Figura 11).

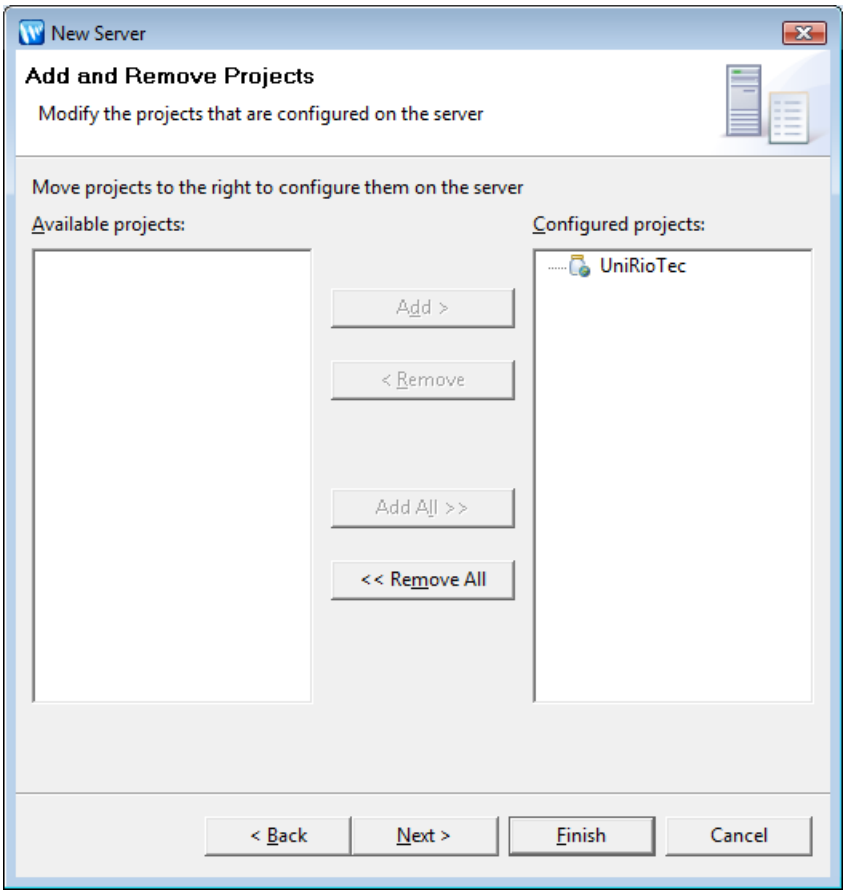

Figura 11 – Janela para adição de projetos para configuração no servidor

6.3 A configuração do DataSource pode depender do banco de dados a ser usado. Abaixo segue os passos de uma configuração genérica que pode não funcionar com todos os bancos. Para maiores informações, verificar a documentação específica da versão que pretende usar. Na sessão 6.3 encontra-se o passo a passo para a criação do DataSource para o Postgres.

a. Obter o driver JDBC do banco que será utilizado.

b. Copiar o driver JDBC para C:\bea\weblogic92\server\lib.

c. Abrir para edição o arquivo "startWebLogic.cmd" (com o notepad, por exemplo) que se encontra na pasta Bin, dentro da pasta do domínio que está sendo utilizado. Ex: C:\bea\user_projects\domains\Dominio_Servicos**bin**\startWebLogic.cmd

d. Incluir o caminho para o diretório onde o driver JDBC do banco foi salvo na linha onde é referenciada a variável CLASSPATH, por exemplo:

```
set
SAVE_CLASSPATH=%CLASSPATH%;C:\bea\weblogic92\server\lib\postgresql-8.3-604.jdbc3.jar
```

e. Salvar o arquivo e executá-lo.

Obs: Cada domínio criado possui um "startWebLogic.cmd " distinto. Para acessar o console do servidor, é necessário que este arquivo seja executado. Através do menu de instalação padrão, é possível encontrar o atalho para o "WebLogic Server 9.2", que ao ser executado, abrirá uma janela de console semelhante . Entretanto, esse atalho NÃO aponta para o arquivo do domínio que foi criado pelo usuário. Portanto sempre execute o "startWebLogic.cmd" de dentro da pasta do domínio desejado.

f. Quando o servidor permanecer em "RUNNING MODE", ou seja, quando for exibida a mensagem "<Server started in RUNNIN mode>", executar o browser e abrir o console na URL http://localhost:7001/console, com o nome de usuário padrão weblogic e senha weblogic.

g. Na janela principal do console, clicar em "Lock and Edit" (Figura 12).

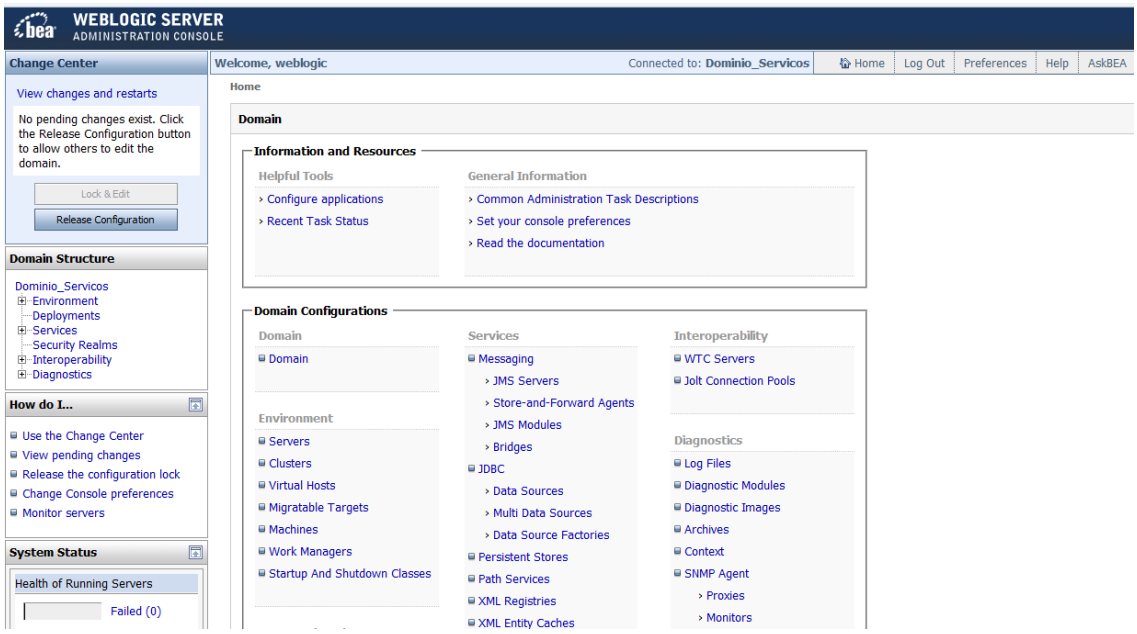

Figura 12 – Tela de configuração do domínio criado Dominio_Servicos

h. Abrir a configuração do DataSource dentro do frame Domain Configurations, em Services/JDBC/Data Sources.

i. Na janela que abrir, clicar em New.

j. Inserir um nome para o DataSource em "Name". Copiar o mesmo nome e inserir em "JNDI Name". Selecionar o tipo do banco de dados em "DataBase Type" e selecionar o driver em "DataBase Driver" (Figura 13). Por fim, clicar em "Next".

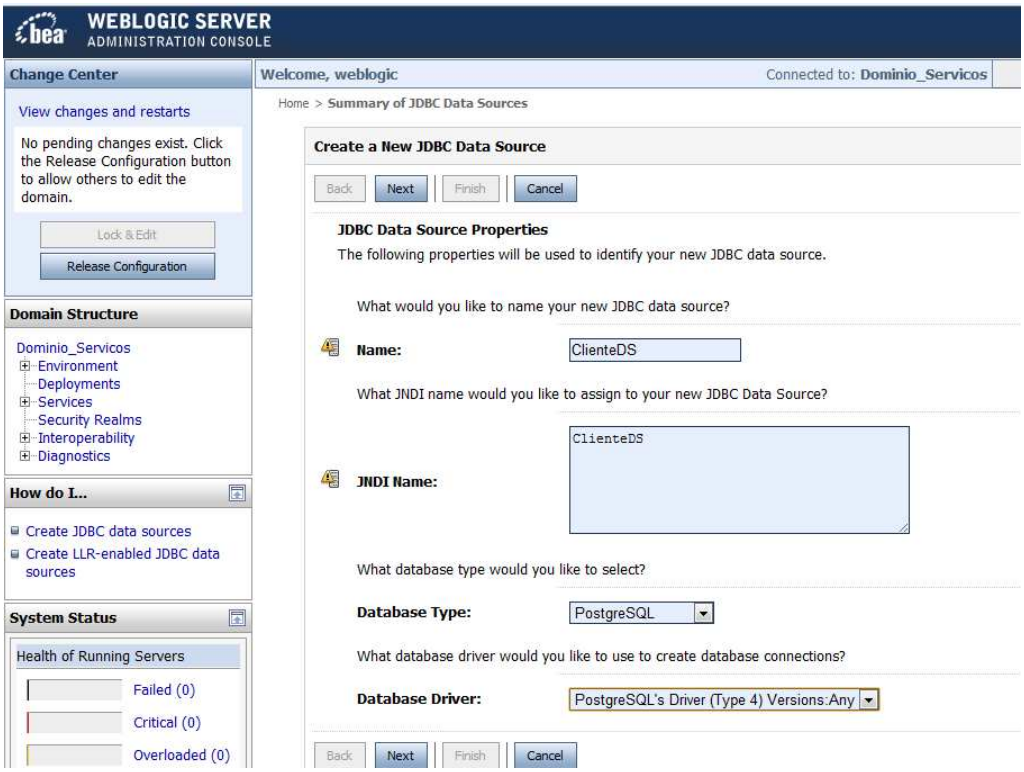

Figura 13 – Tela de criação do DataSource

k. Na janela "Transaction Options" clicar em "Next".

l. Preencher as informações de conexão com o banco de dados (Figura 14) e clicar em "Next".

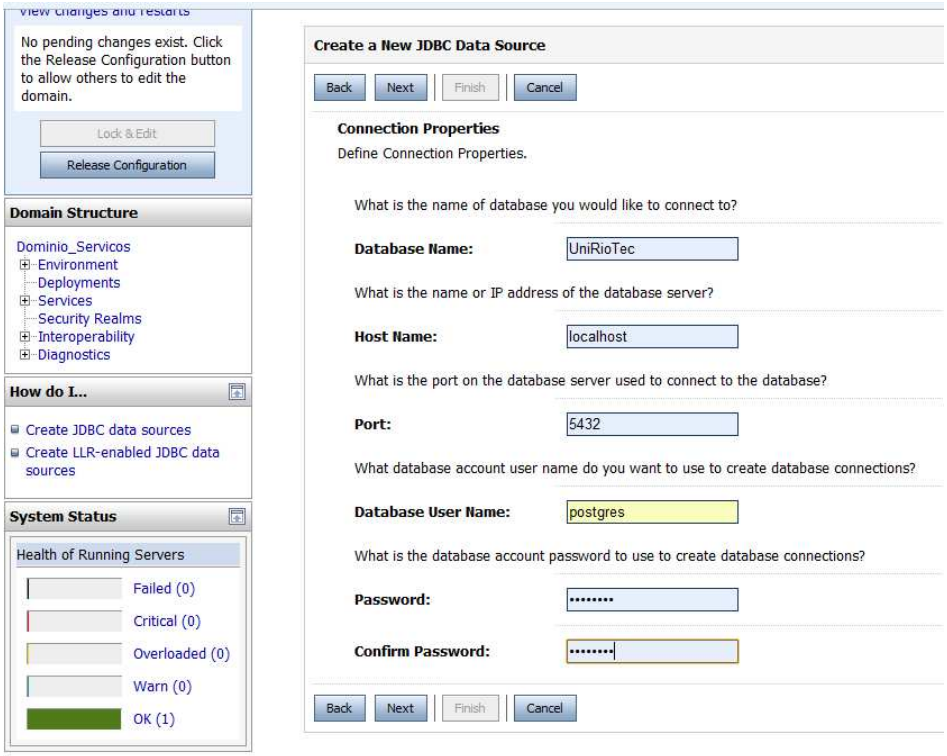

Figura 14 – Configuração da conexão com o banco de dados

m. Inserir os parâmetros de configuração do driver de conexão com o

banco. Testar a configuração clicando em "Test Configuration" (Figura 15). Caso ocorra algum erro, rever as configurações. Clicar em Next e depois em Finish.

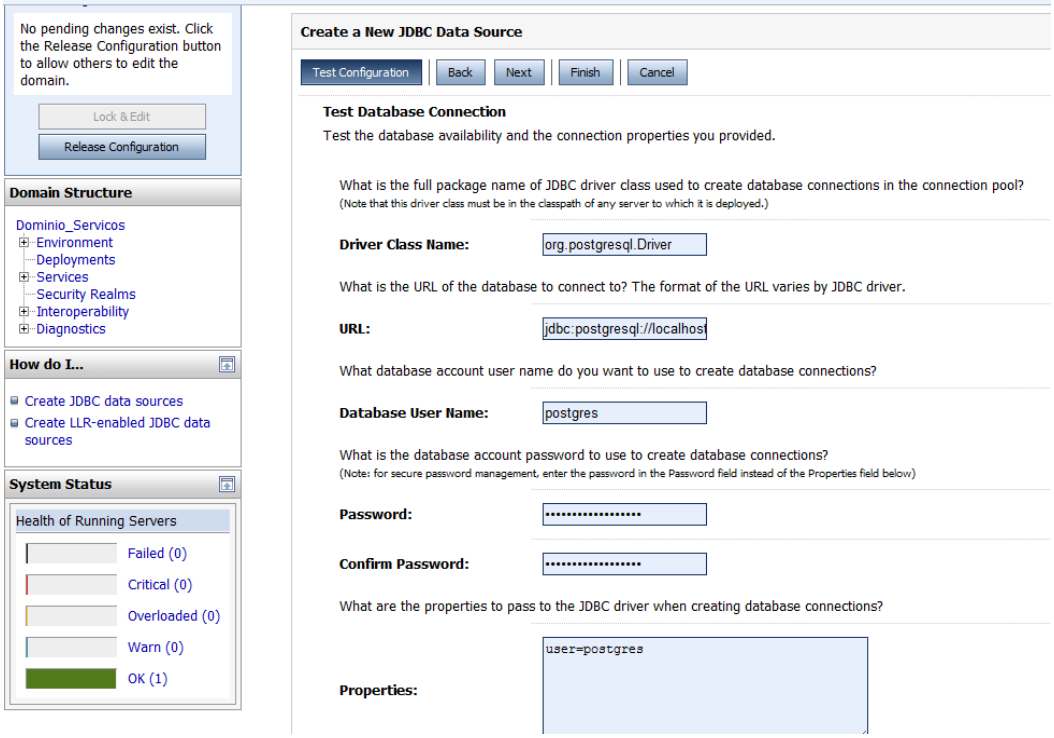

Figura 15 – Teste da configuração do Datasource

n. Para finalizar, clicar em "Activate changes" para aplicar a nova configuração (Figura 16).

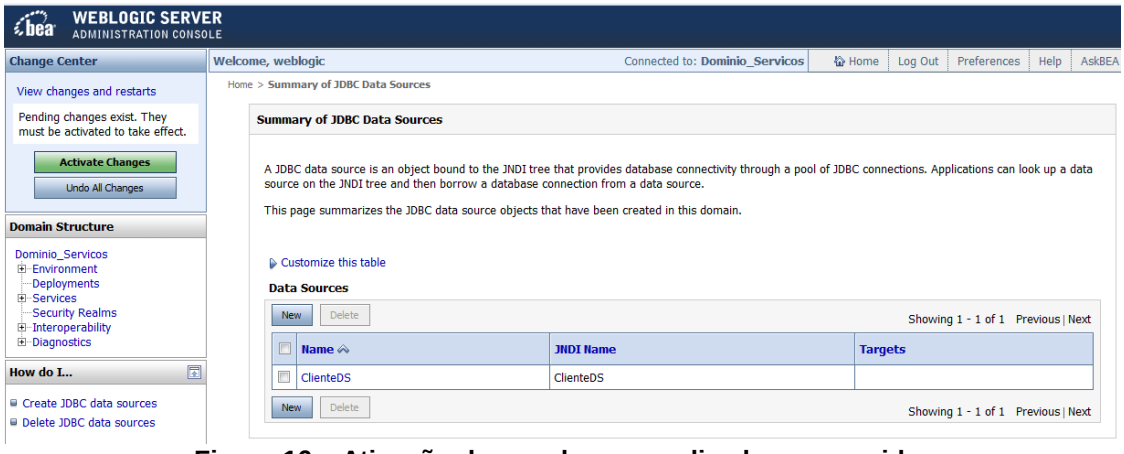

Figura 16 – Ativação das mudanças realizadas no servidor

7. Criar serviço com o padrão de nomenclatura <nome do objeto>+ Service, por exemplo, ClienteService.
    a. No pacote "br.uniriotec.services" clicar em "New ➔ WebLogic Web Service" (Figura 17).

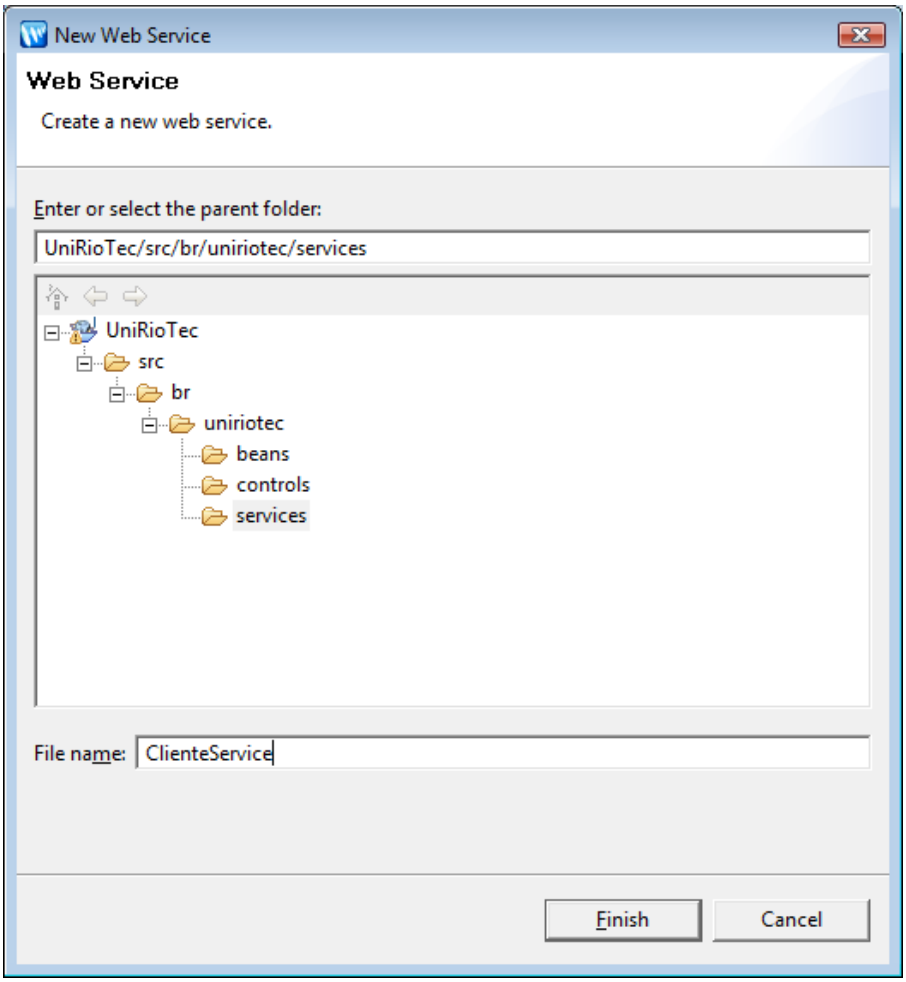

Figura 17 – Criação do web service

b. Incluir referência para o objeto Control criado anteriormente.

```
@Control
ClienteDbControl clienteDbControl;
```

c. É necessário importar ClienteDbControl e beehive.controls.api.bean.Control.

```
import org.apache.beehive.controls.api.bean.Control;
import br.uniriotec.controls.ClienteDbControl;
```

d. Criar WebMethod para retornar o elemento que representa o Array de objetos (Figura 18).
Na implementação deste método, a rotina verificarStringNull é utilizada para verificar se a string recebida como parâmetro é nula (Figura 19).
O método parseCliente2XML é responsável por transformar cliente POJO em cliente XML (Figura 20). O XML dos clientes é o que será retornado para o consumidor do serviço.
O método parseCliente2XML invoca o método parsePOJO2XML, o qual é um método genérico capaz de transformar uma classe POJO em elementos de um XML, seguindo o padrão de nomes especificado (Figura 21). Observe que, neste método, existe um teste se ele contém a palavra

"uniriotec". Isto é usado para garantir que o método só trate classes definidas no namespace "uniriotec".

```
WebService
ublic class ClienteService {

    @Control
    ClienteDbControl clienteDbControl;

    @WebMethod
    public ClienteListDocument getCliente(String p_cpf)
                throws Exception{

        verificarStringNull(p_cpf, "p_cpf");


        String where = "";
        Cliente[] cliente = null;

        where="AND CPF = '" + p_cpf + "' ";
        cliente = clienteDbControl.getCliente(where, "", "", "");

        ClienteListDocument xmlCliente = parseCliente2XML(cliente);


        return xmlCliente;
    }
```

Figura 18 – Método getCliente

```
/** Verifica se uma string é nula e levanta uma exceção caso seja.
  * <p>
   * Uma das verificações que todo serviço deve fazer é se os parâmetros de
   * entrada são válidos. Este método verifica se a string passada como
   * parâmetro é nula. Caso seja nula, ele levanta uma exceção.
   * <p>
   *
   * @param str string que se deseja verificar.
   * @param parameterName o nome do parâmetro do serviço, para a composição da
   *        mensagem de erro.
   * @throws IllegalArgumentException caso a string seja nula.
   */
  public static void verificarStringNull(String str, String parameterName)
    throws IllegalArgumentException {
    if (str == null || str.trim().length() == 0)
      throw new IllegalArgumentException("Parameter " + parameterName
        + " is required.");
  }
```

Figura 19 – Método verificarStringNull

```
private ClienteListDocument parseCliente2XML( Cliente[] cliente ) throws Exception {

    ClienteListDocument pDoc = ClienteListDocument.Factory.newInstance();
      pDoc.addNewClienteList();

      for (int i=0; i < cliente.length; i++){
          parsePOJO2XML(cliente[i], pDoc.getClienteList().addNewCliente());
      }
      return pDoc;
    }
```

Figura 20 – Método parseCliente2XML

```
public static void parsePOJO2XML(Object pojo, Object xml) throws Exception {
  if (pojo == null) {
    throw new IllegalArgumentException("pojo == null");
  }
  if (xml == null) {
    throw new IllegalArgumentException("xml == null");
  }
  // Referências feitas fora do loop para otimizar chamadas.
  Class pojoClass = pojo.getClass();
  Class xmlClass = xml.getClass();

  Method[] methods = pojoClass.getMethods();
  for (Method m : methods) {
    String methodName = m.getName();
    // Trata apenas métodos get que pertencem as classes definidas no
    // namespace da uniriotec.
    if (methodName.startsWith("get")
      && m.getDeclaringClass().getName().contains("uniriotec")) {
      // Obtém o método get do atributo da classe "pojo".
      Method mget = pojoClass.getMethod(methodName, new Class[] {});
      // Obtém o método set do atributo respectivo na classe xml.
      methodName = "s" + methodName.substring(1);
      Method mset = xmlClass.getMethod(methodName, mget.getReturnType());
      // Obtém o atributo efetivamente retornado.
      Object getted = mget.invoke(pojo, new Object[] {});
      // Se o objeto não for nulo, atualiza o objeto xml.
      if (getted != null) {
        mset.invoke(xml, new Object[] { getted });
      }
    }
  }
}
```

Figura 21 – Método parsePOJO2XML

e. Importar Cliente e ClienteListDocument.

```
import br.uniriotec.beans.Cliente;
import beans.controls.cliente.ClienteListDocument;
```

i. O objeto ClienteListDocument encontra-se no jar criado a partir do XSD, por exemplo, ClienteTypes_xmlbeans_apache.jar.

2.3 Criação de DataSource para Postgresql

Caso deseje-se utilizar o SGBD PostgreSQL como banco de dados para os testes, os seguintes passos devem ser realizados.

1. Iniciar o servidor no domínio criado, por exemplo, GDIEP_domain (Figura 22).

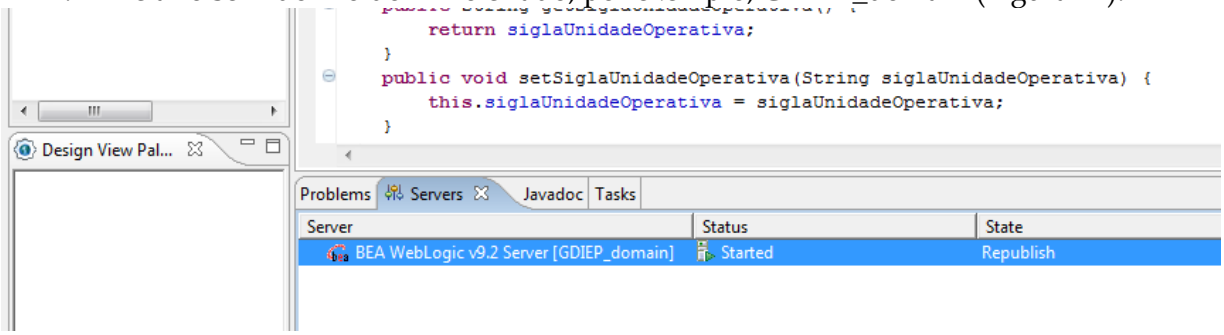

Figura 22 – Servidor GDIEP_domain iniciado

2. Executar Admin Server Console (Figura 23).

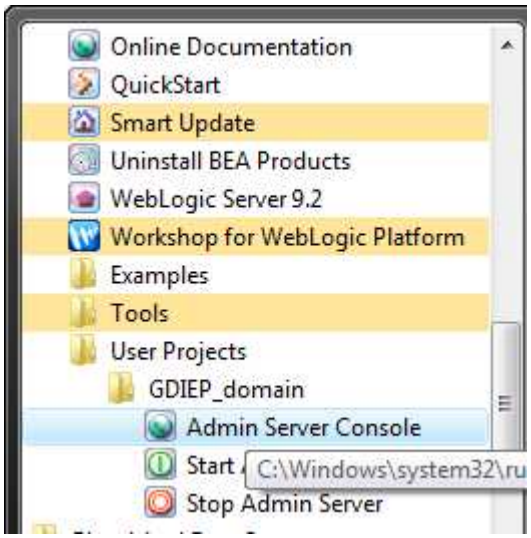

Figura 23 – Menu para execução do Admin Server Console

3. Após logar no servidor, ir para a página de configuração de DataSources (Figura 24).

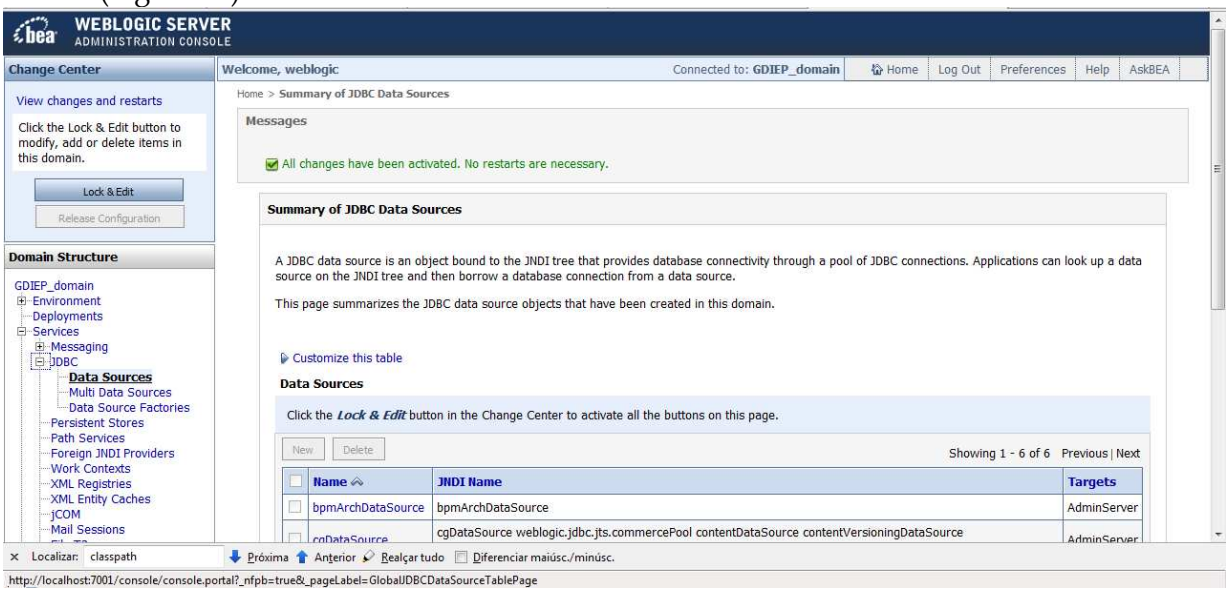

Figura 24 – Página de configuração de data sources

4. Clicar em "Lock & Edit" e então no botão "New" em "Data Sources"
5. Preencher as informações para configuração do Data Source, testar a conexão, escolher o servidor e concluir a configuração.
6. Importante:
    a. No caso do WebLogic Server 9.2, o driver do postgres é o da versão JDBC3 - postgresql-8.3-604.jdbc3.jar.
    b. Este arquivo deve estar configurado no Classpath do servidor.
        i. Em "C:\bea\user_projects\domains\Domain_Servicos\bin", abrir o arquivo startWebLogic.cmd no notepad.
        ii. Incluir o caminho para o diretório onde o driver foi salvo na linha onde é referenciada a variável CLASSPATH, por exemplo:

set
SAVE_CLASSPATH=%CLASSPATH%;C:\bea\weblogic92\server\lib\postgresql-8.3-604.jdbc3.jar

3 Testes do WebService implementado

Este capítulo descreve as possibilidades de testes de serviços.

3.1 Realização de testes utilizando o testClient do Workshop

1. No próprio Workshop, executar o serviço no servidor (Figura 25)

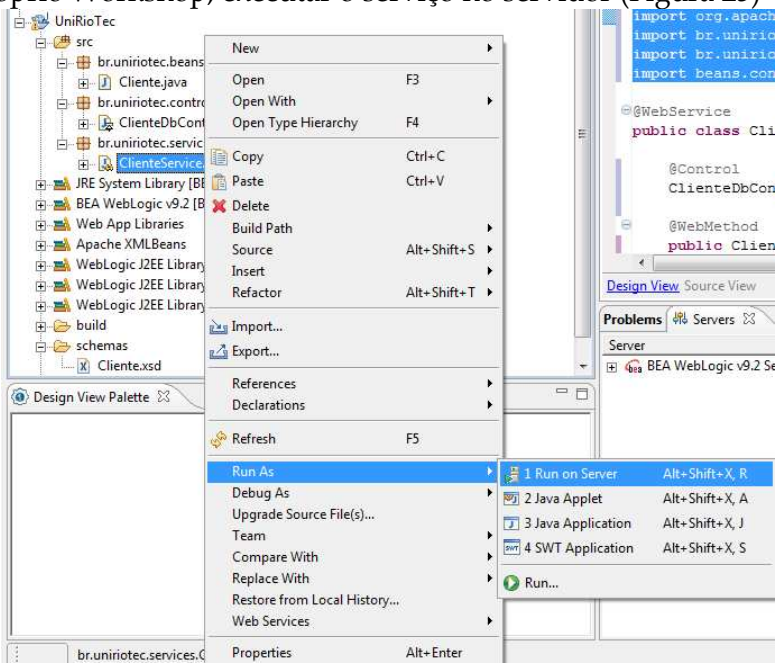

Figura 25 – Execução do serviço no servidor

2. Caso necessário, confirme o servidor que executará o serviço e clique em "Next" (Figura 26).

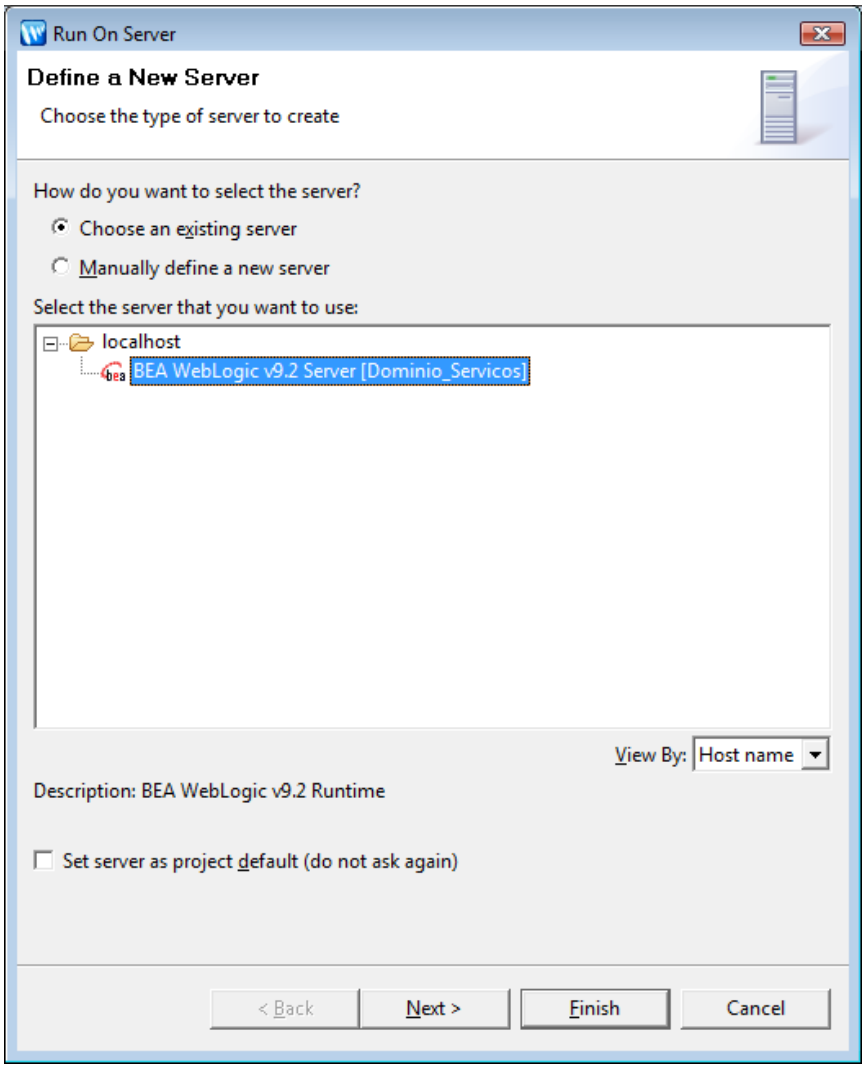

Figura 26 – Janela para escolha do servidor que executará o serviço

3. Adicione o projeto que será executado (Figura 27) e clique em "Finish".

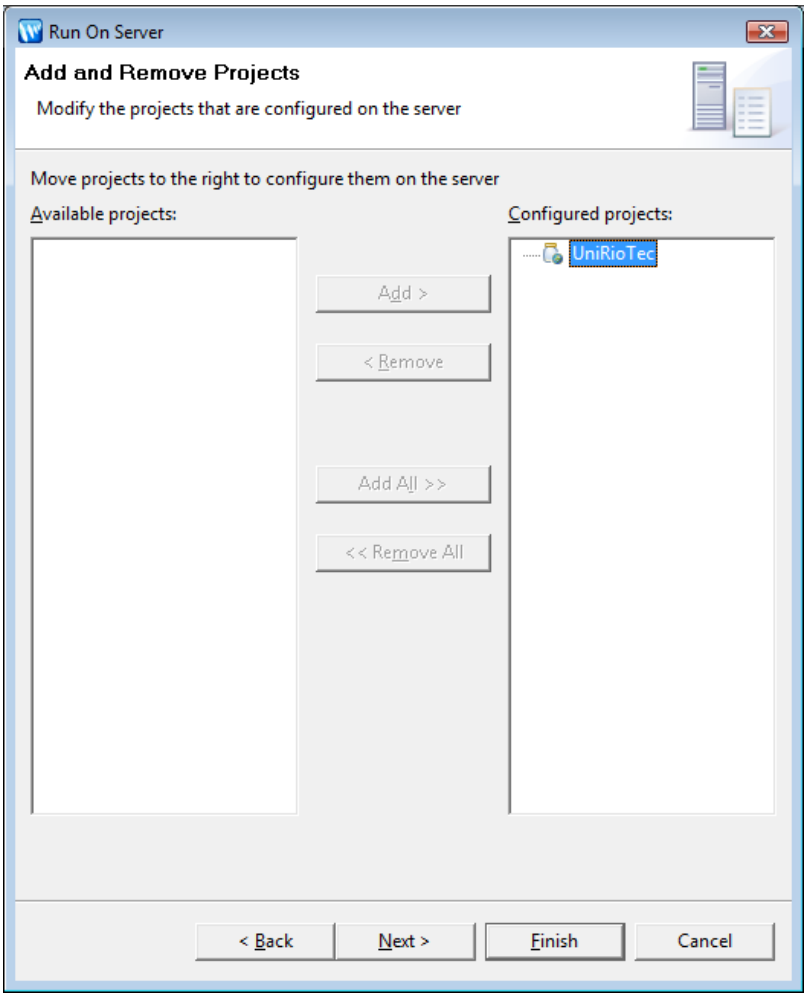

Figura 27 – Janela para adição do projeto a ser executado

4. Após a publicação do serviço no servidor, a tela do Test Client deverá ser exibida (Figura 28). Caso ocorra algum erro, verifique no console do servidor se a botão "Release Configuration" está habilitado, conforme a Figura 29. Se estiver, clique nele para liberar a configuração e retorne novamente ao primeiro passo.

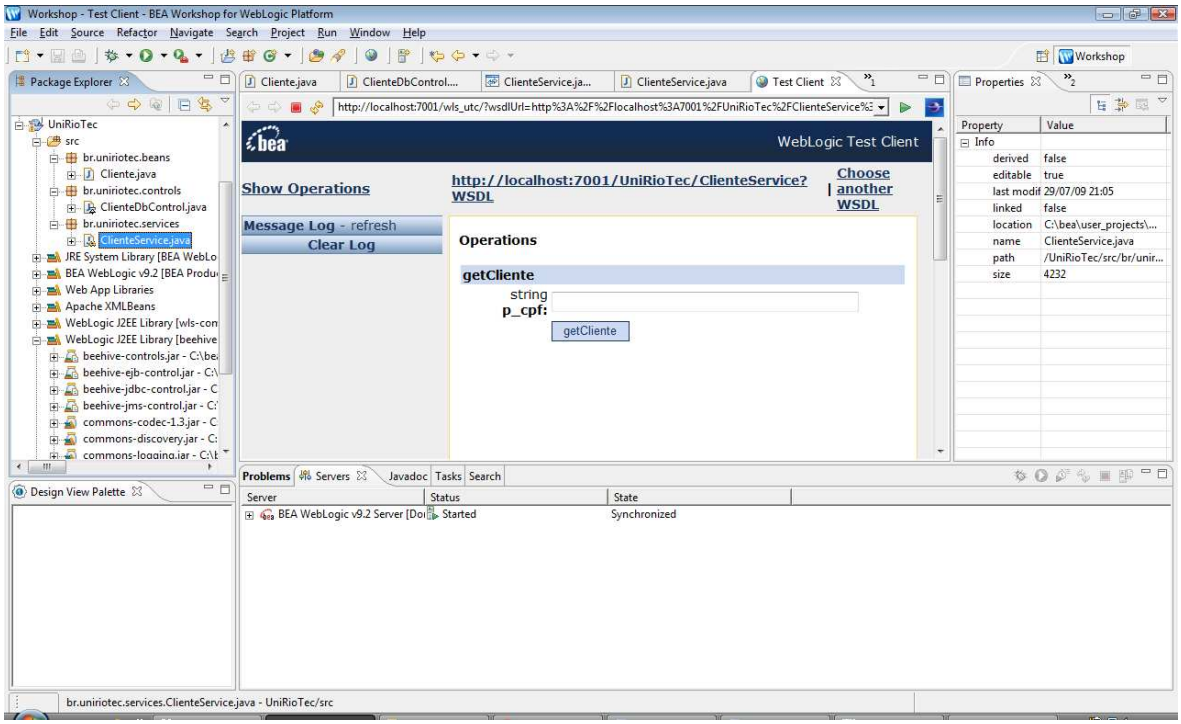

Figura 28 – Tela do Test Client, para o teste do serviço

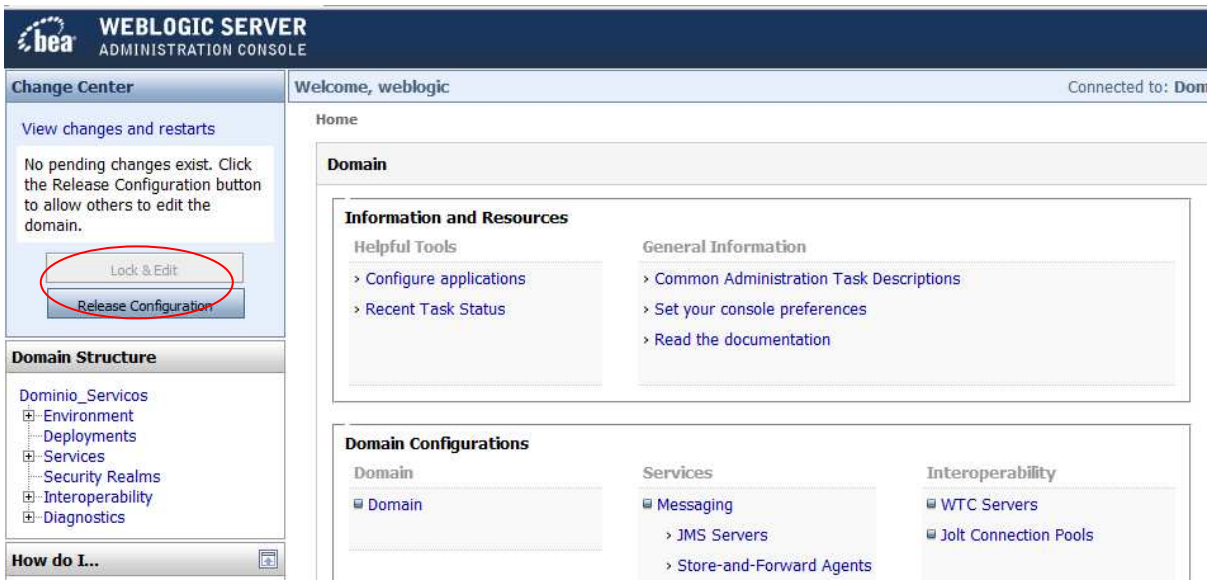

Figura 29 – Liberação da configuração necessária (Release Configuration)

5. Preencher o parâmetro do método, por exemplo, p_cpf, clicar no botão referente ao método e analisar o resultado.

    a. A Mensagem de requisição de execução do método getCliente gerada pelo Test Client (Service Request) e a mensagem de resposta (Service response) são apresentadas na Figura 30.

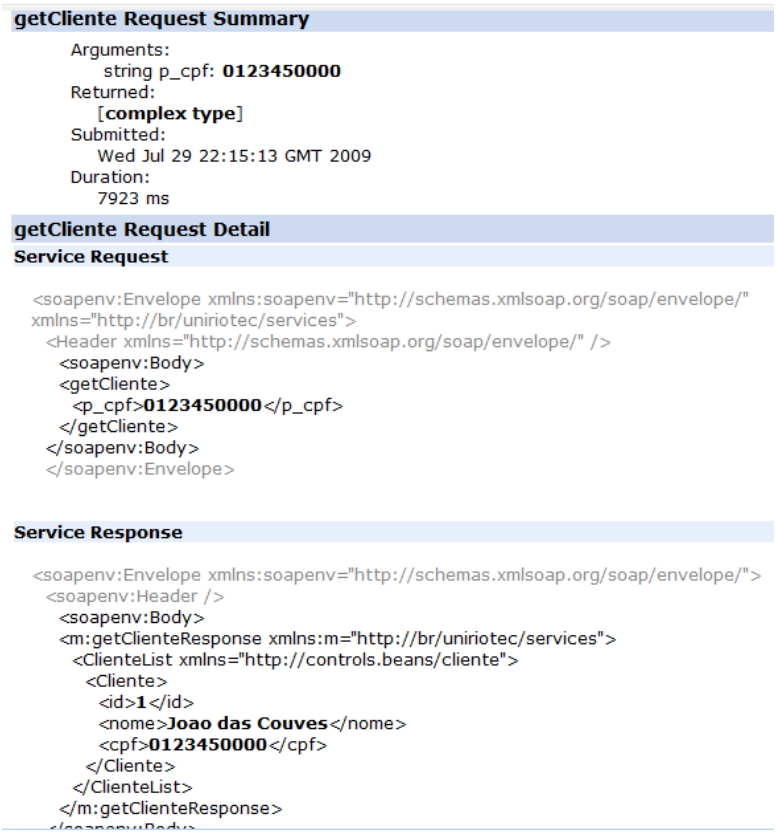

Figura 30 – Resultado do teste do método getCliente

6. Caso não seja preenchido o parâmetro do método para realizar a consulta, é retornada uma mensagem de erro (Figura 31 e Figura 32).

Figura 31 – Mensagem de requisição sem cpf preenchido

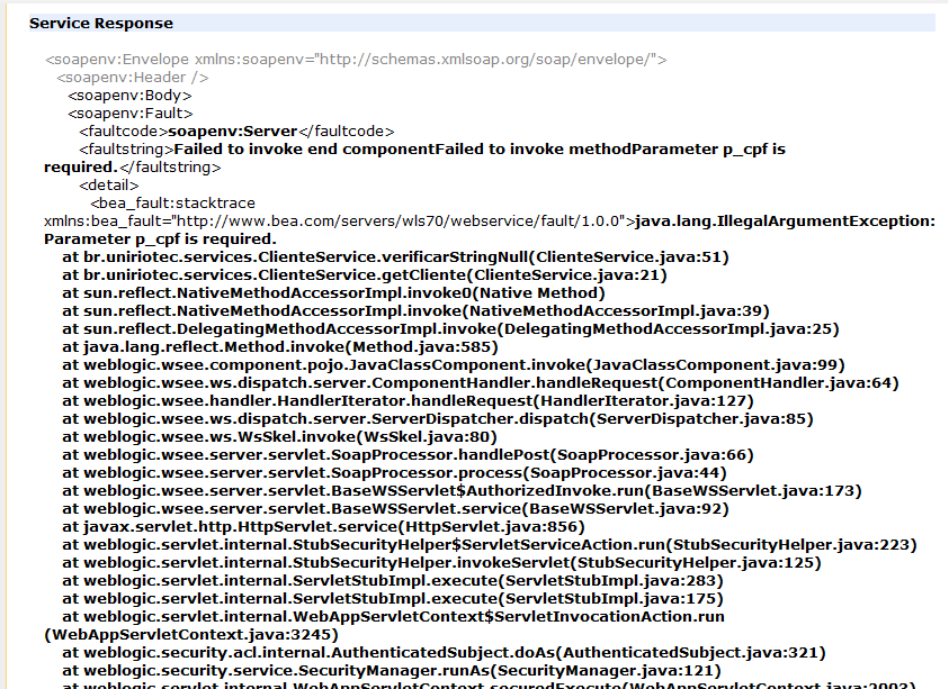

Figura 32 – Erro retornado na mensagem de resposta

3.2 Realização de testes utilizando o SOAPUI

Outra forma de realizar o teste do serviço é utilizando a ferramenta SOAPUI[6]. O SOAPUI é uma ferramenta free amplamente utilizada para testes de web services. A seguir, são descritos os passos necessários para testar o serviço ClienteService usando o SOAPUI.

1. Primeiro é necessário gerar o arquivo WSDL do WebService implementado.
    a. Clicar com o botão direito no WebService ClienteService ➔ WebServices ➔ Generate WSDL (Figura 33).

[6] http://www.soapui.org/

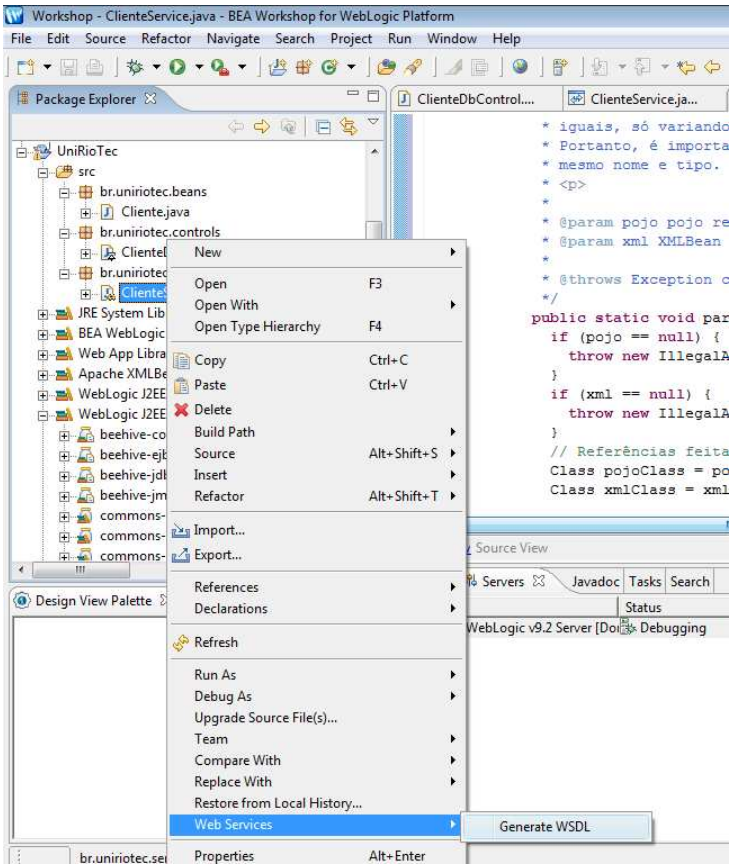

Figura 33 – Geração do arquivo WSDL do WebSevice implementado

2. Executar SOAPUI.
3. Criar projeto a partir do WSDL do serviço (Figura 34)

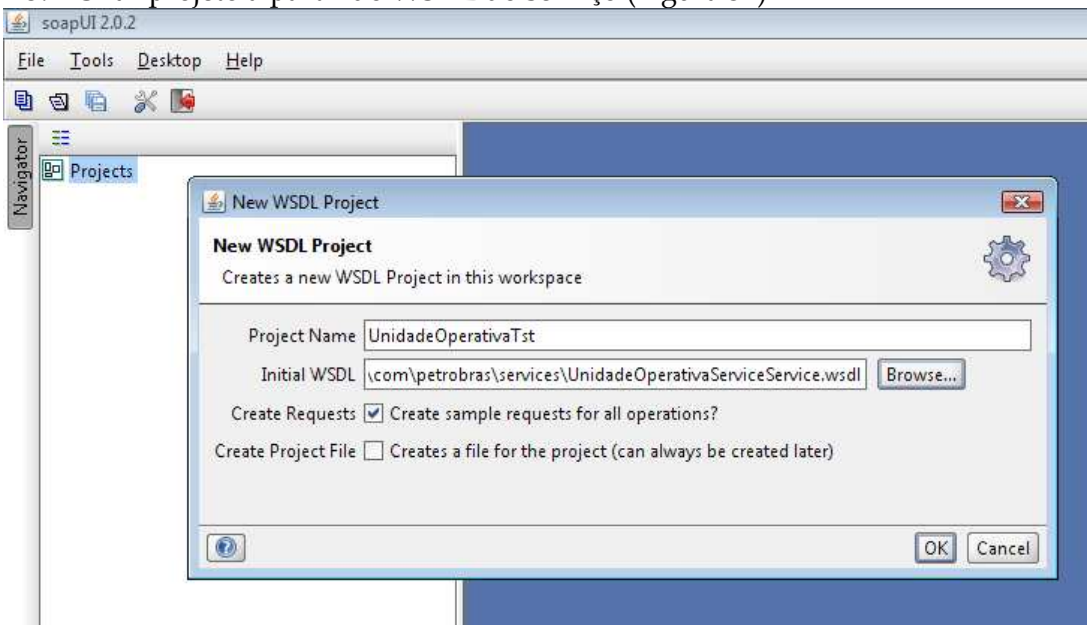

Figura 34 – Criação do projeto a partir do WSDL do serviço

4. Abrir o Request Editor correspondente ao método que se deseja testar (Figura 35).

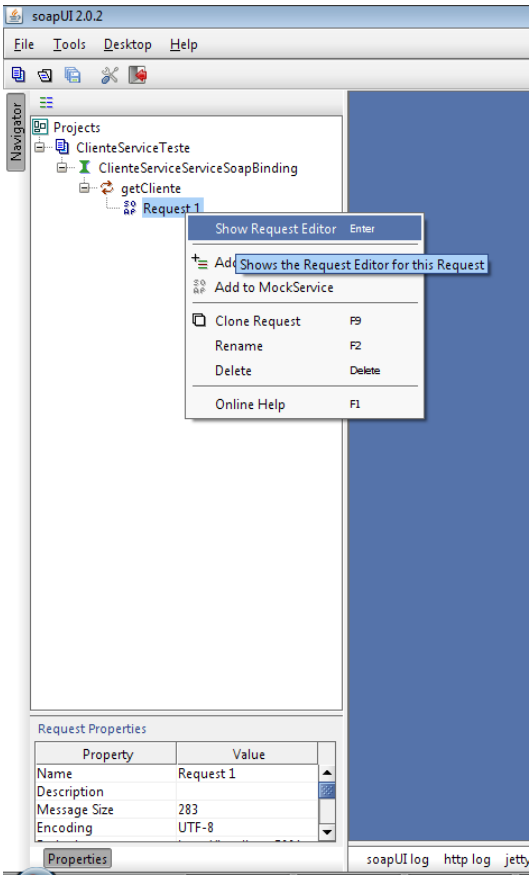

Figura 35 – Abrindo do request editor

5. Preencher o parâmetro do método, por exemplo, substituir "?" por "0123450000" (Figura 36).

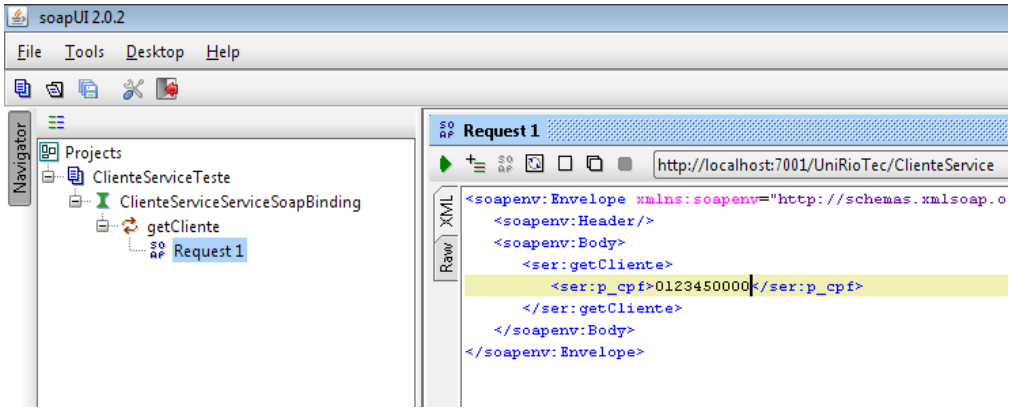

Figura 36 – Ajuste do parâmetro *cpf* do método do serviço

6. Clicar no botão de execução (Figura 37).

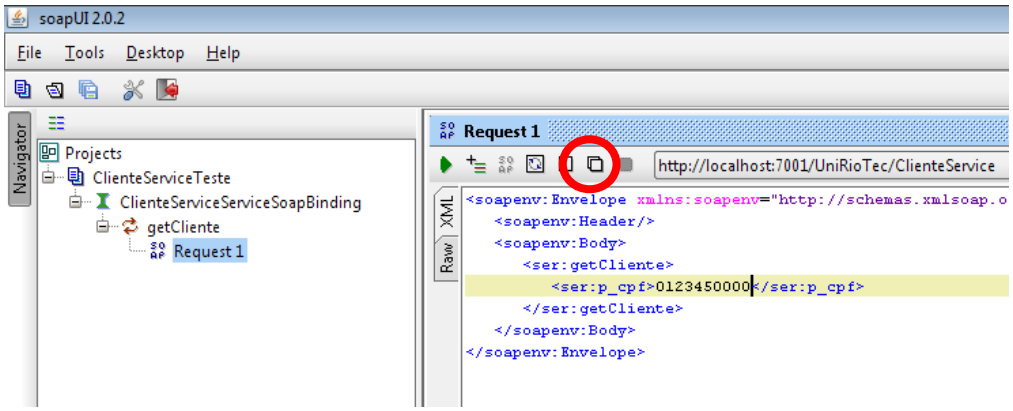

Figura 37 – Botão para execução do serviço

7. O XML de resposta deve ser semelhante ao apresentado a seguir (Figura 38).

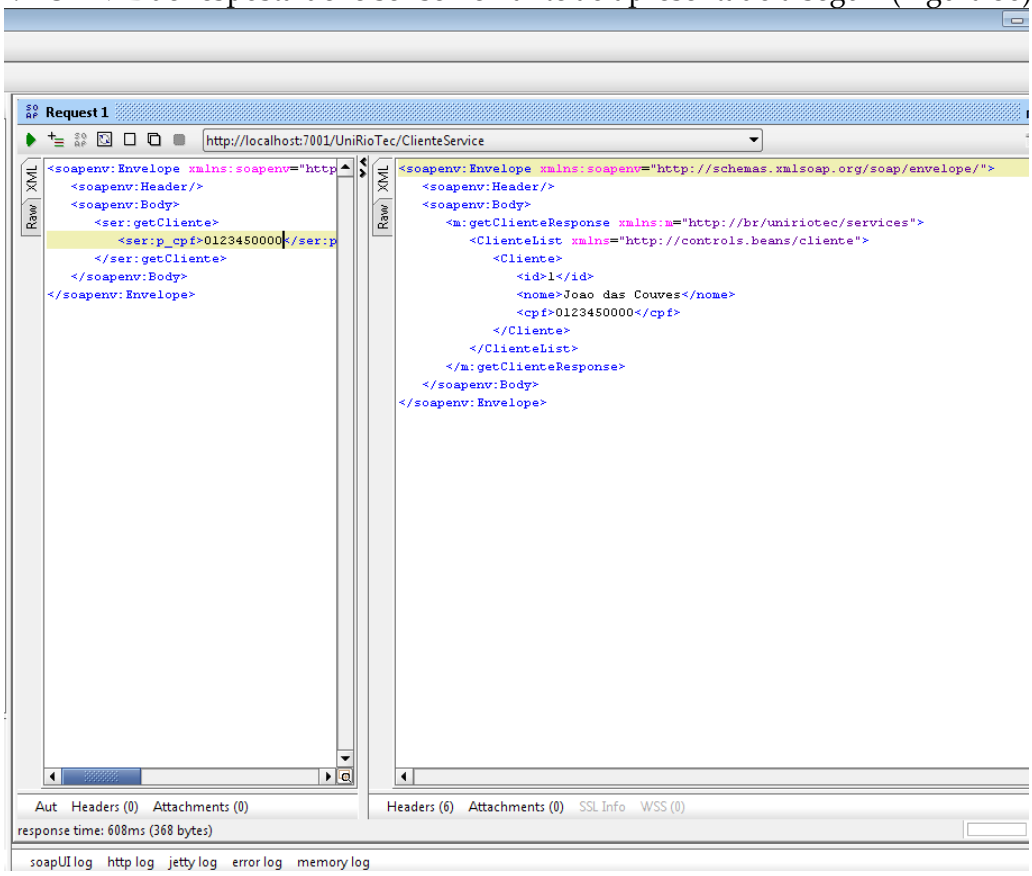

Figura 38 – XML de resposta

3.3 Realização de testes implementando um cliente Java para o serviço

O serviço também pode ser testado através da implementação de um cliente Java para o mesmo, como descrito no passo-a-passo a seguir.

1. Criar projeto Java (Figura 39).

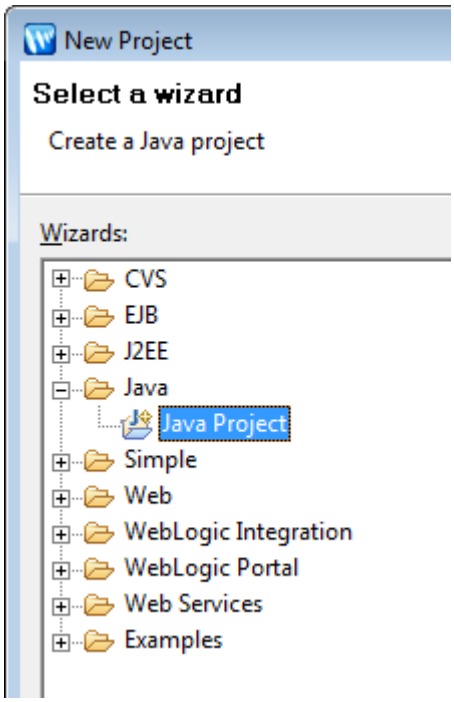

Figura 39 – Criação do projeto Java

2. O próximo passo corresponde à geração do Stub para o webservice utilizando uma task do Ant. Para isto é necessário incluir referência no projeto para weblogic.jar, webserviceclient.jar e para axis.jar.
    a. Para adicionar referência para weblogic.jar, configurar o buildpath, estendendo a variável WEBLOGIC_HOME, de acordo com as figuras a seguir (Figura 40, Figura 41, Figura 42, Figura 43).

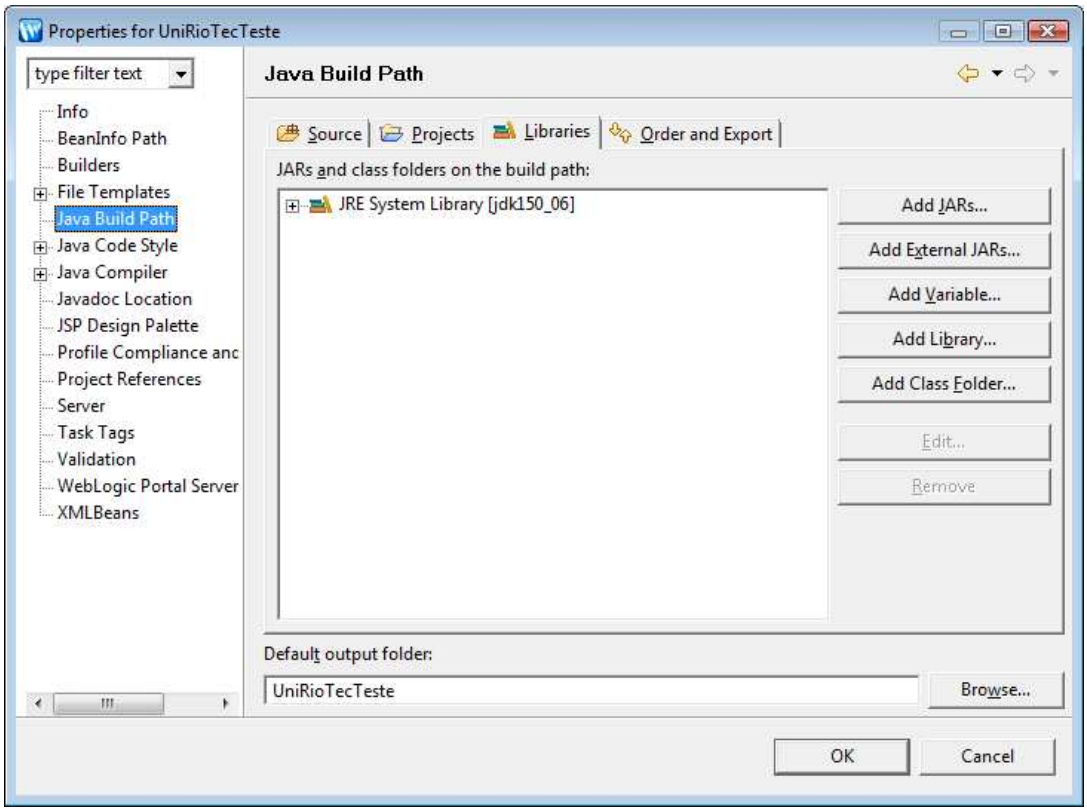

Figura 40 – Janela Libraries de Java Build Path para extensão da variável WEBLOGIC_HOME

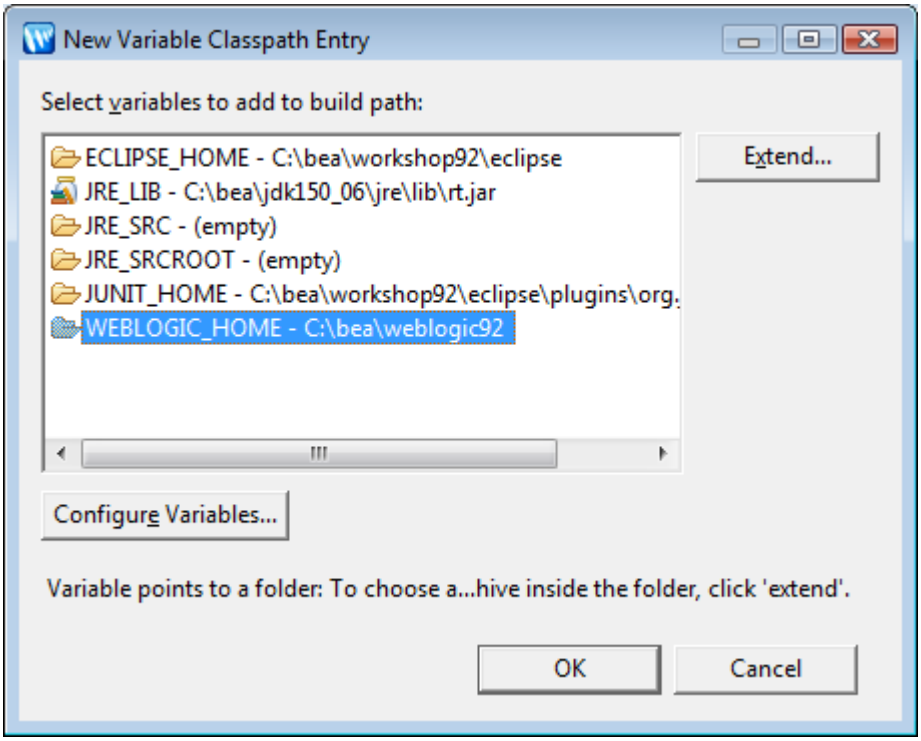

Figura 41 – Seleção do WEBLOGIC_HOME para extensão de variáveis

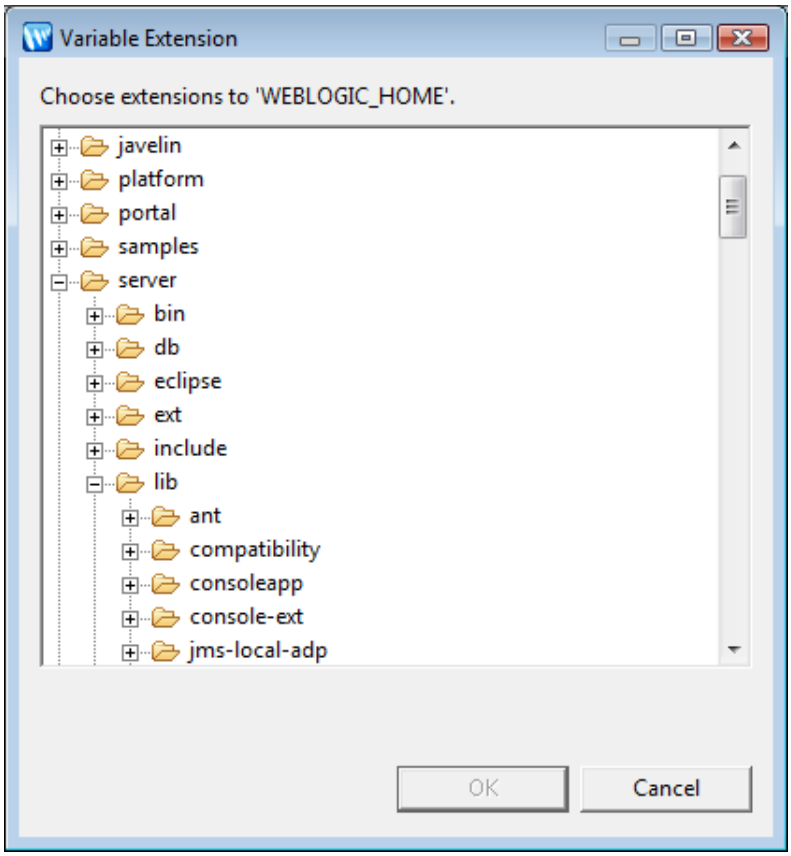

Figura 42 – Janela de escolha de variáveis para extensão WEBLOGIC_HOME (server\lib)

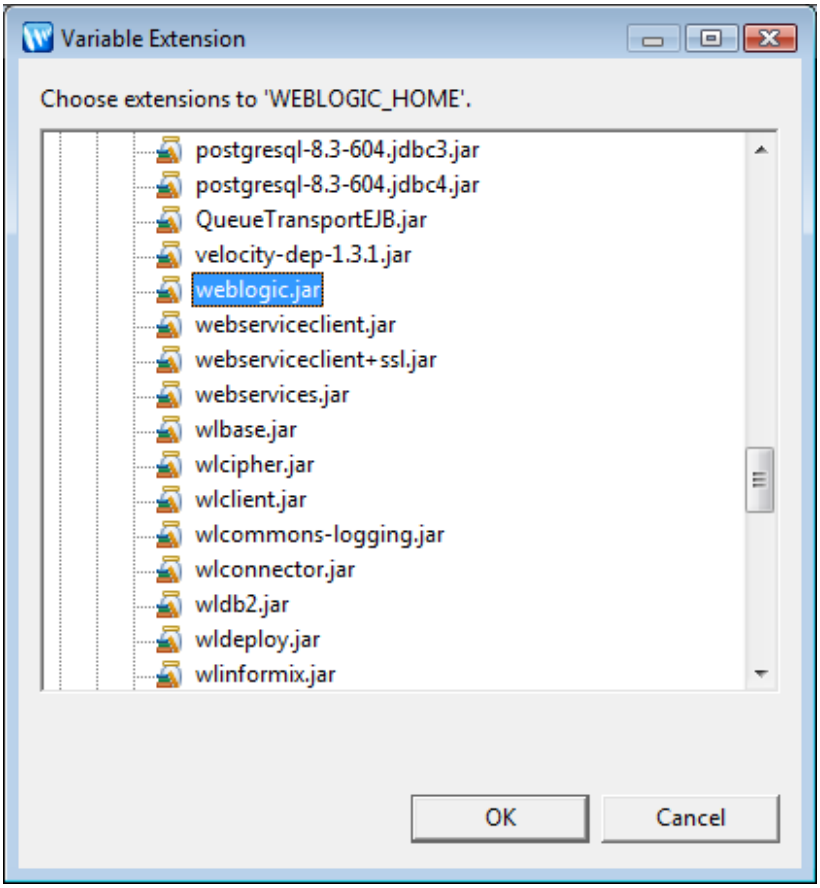

Figura 43 – Seleção da variável de extensão weblogic.jar

b. Realizar as mesmas etapas para o arquivo webserviceclient.jar.

c. Para adicionar referência para axis.jar, estender a variável ECLIPSE_HOME (Figura 44, Figura 45, Figura 46).

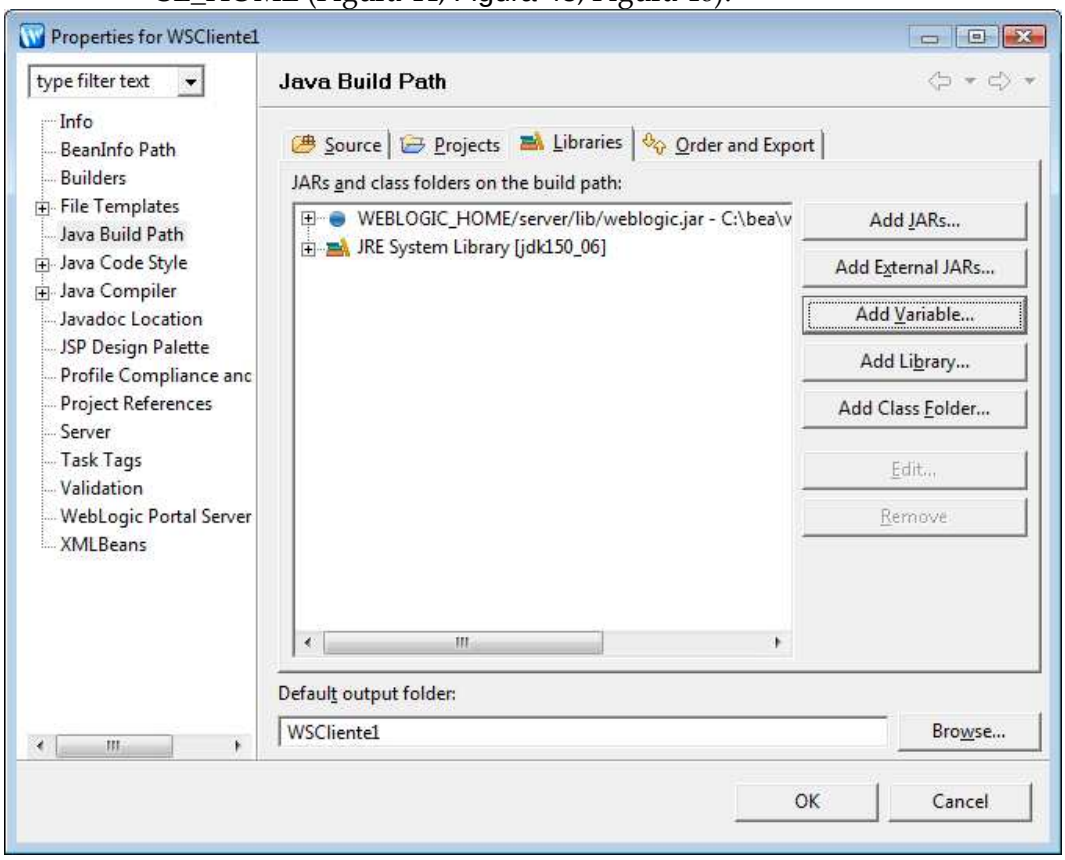

Figura 44 – Janela de extensão de arquivos para a variável ECLIPSE_HOME

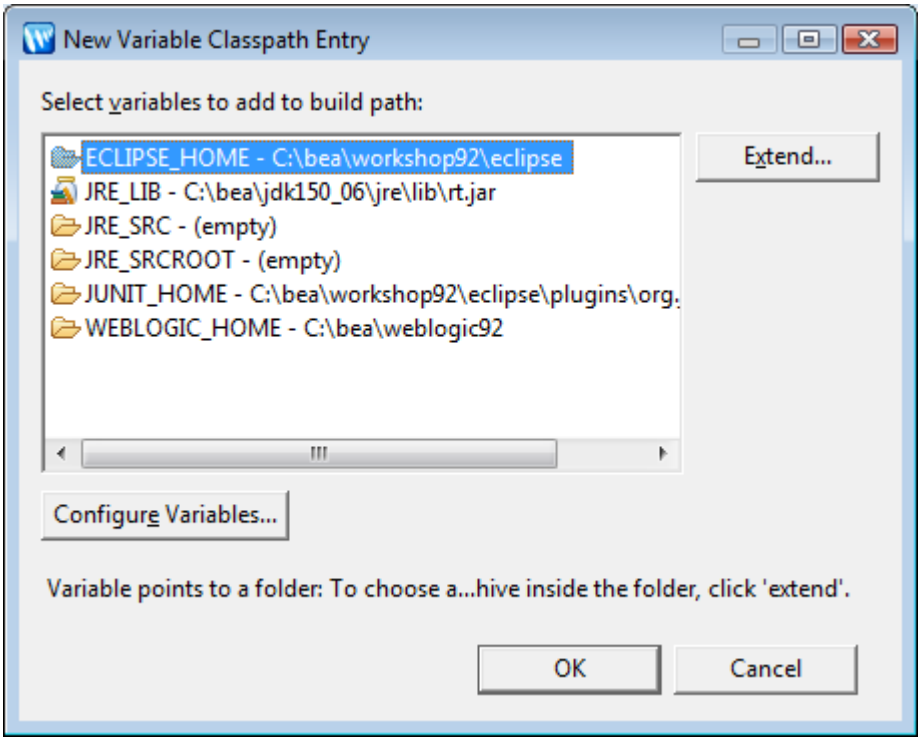

Figura 45 - Seleção do ECLIPSE_HOME para extensão de variáveis

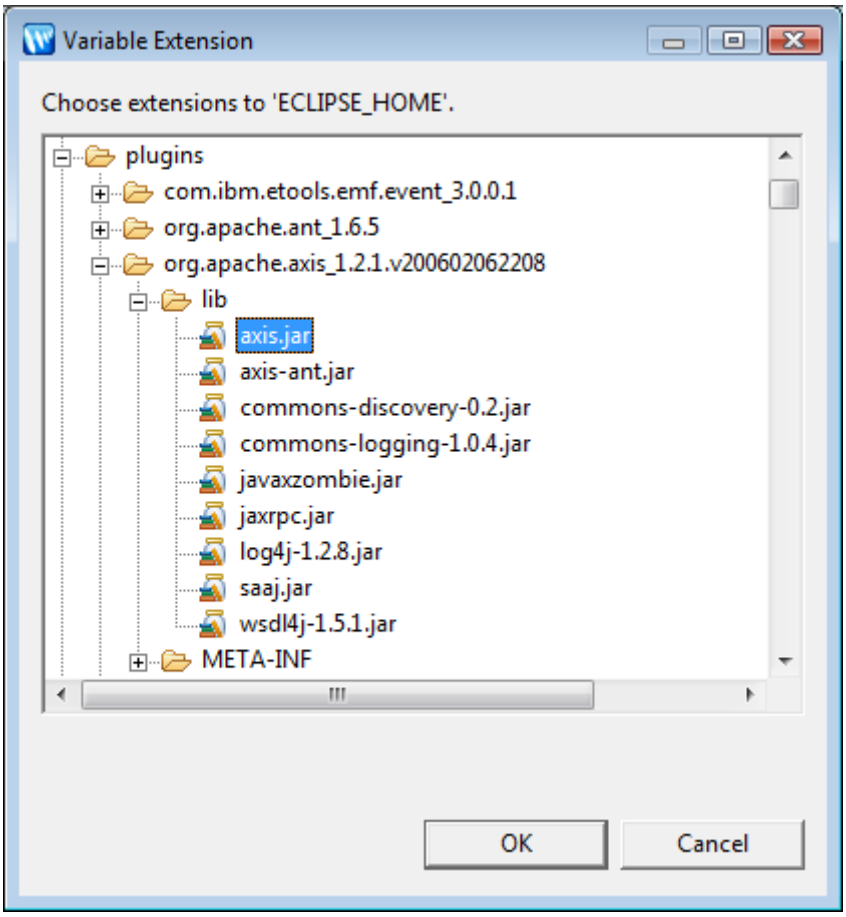

Figura 46 - Seleção da variável de extensão axis.jar

3. Criar arquivo com o nome "build.xml" (Figura 47).

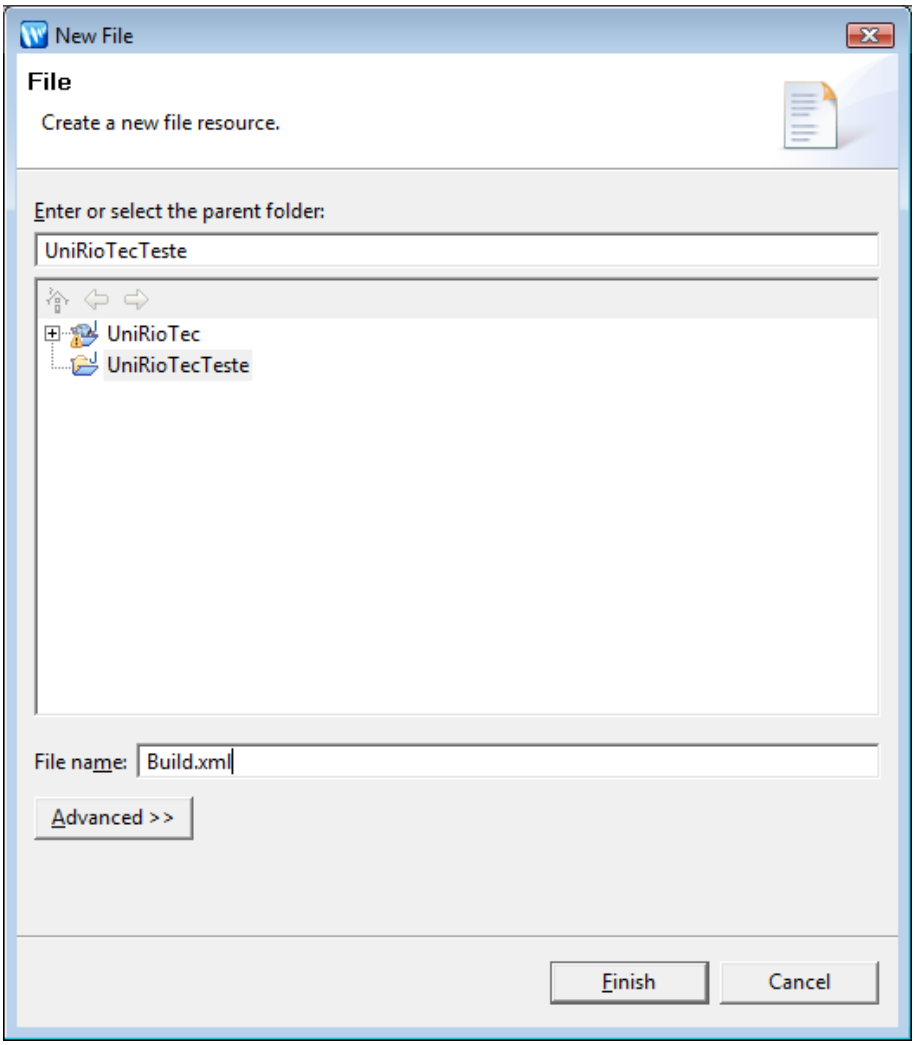

Figura 47 – Criação de arquivo para o código da task Ant.

4. Criar task Ant, segundo o código a seguir (Figura 48), colar o código no arquivo "build.xml" criado no passo anterior.

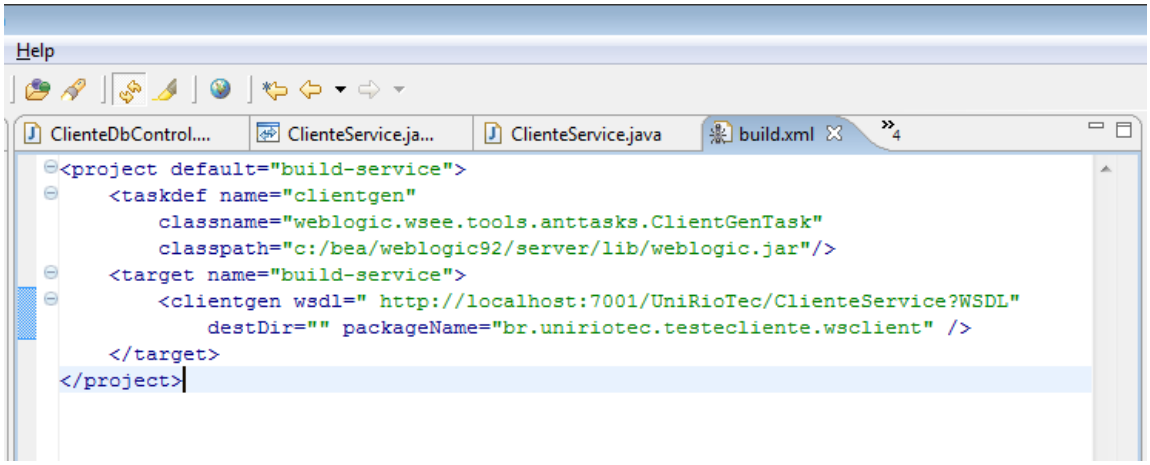

Figura 48 – Ant task para criação do cliente Java

Observe que foi necessário incluir no classpath o caminho para o arquivo weblogic.jar: "classpath="c:/bea/weblogic92/server/lib/weblogic.jar"""

5. Executar o arquivo (Run As ➔ Ant Build) (Figura 49).

6. Após a execução, atualize o projeto (pressionar F5).

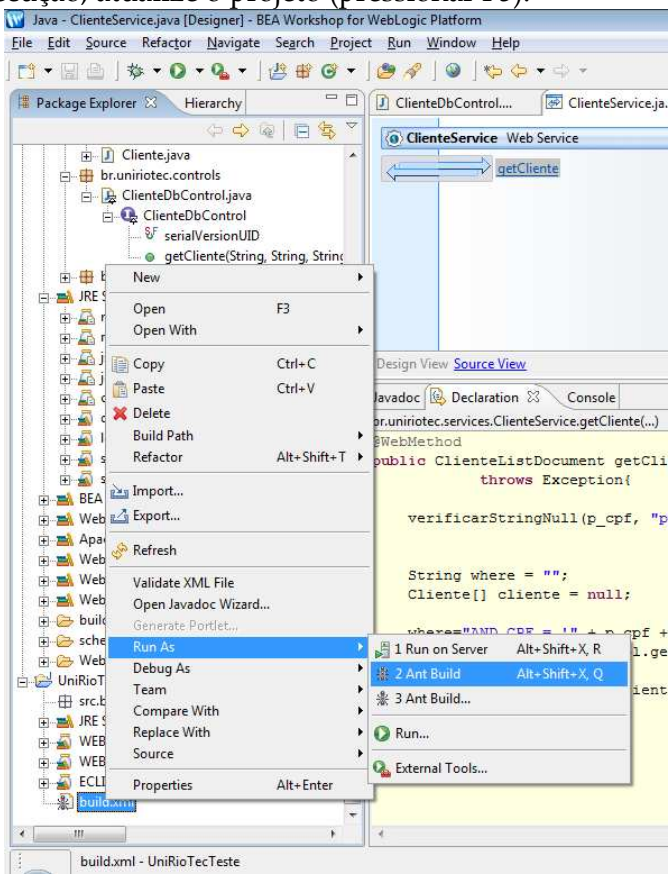

Figura 49 – Execução do arquivo build.xml

7. O resultado de execução será algo semelhante ao apresentado na Figura 50. A classe Cliente_Stub é criada. Esta classe é responsável por realizar o encapsulamento da chamada do web service. Desta forma, o cliente invoca o método da classe stub, que por sua vez invoca o método do serviço. O serviço retorna a resposta para o stub que repassa para o cliente (Figura 51).

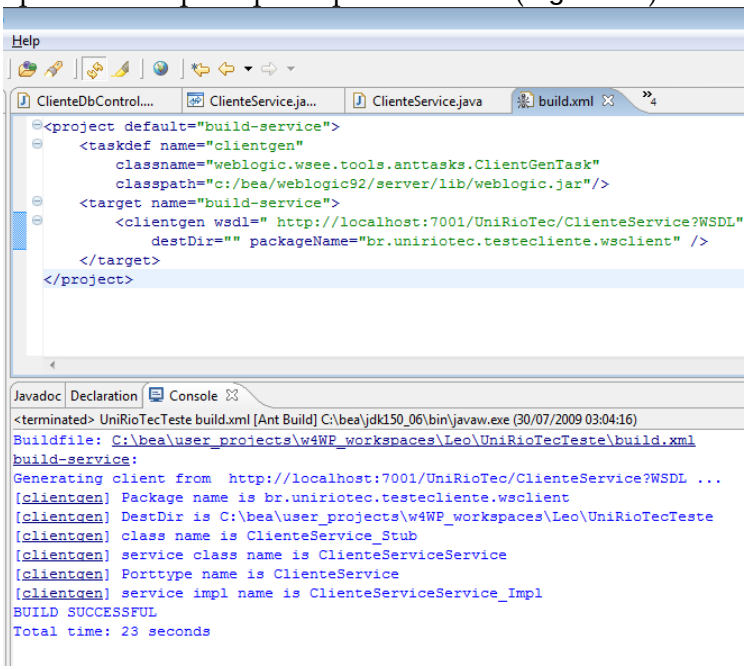

Figura 50 – Resultado da execução do build.xml

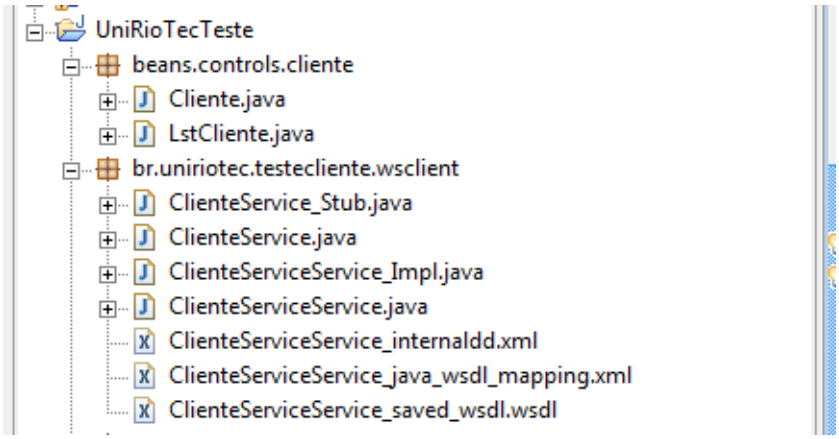

Figura 51 – Classes criada após a execução da task Ant

8. Implementar classe cliente Java de acordo com o código a seguir (Figura 52):

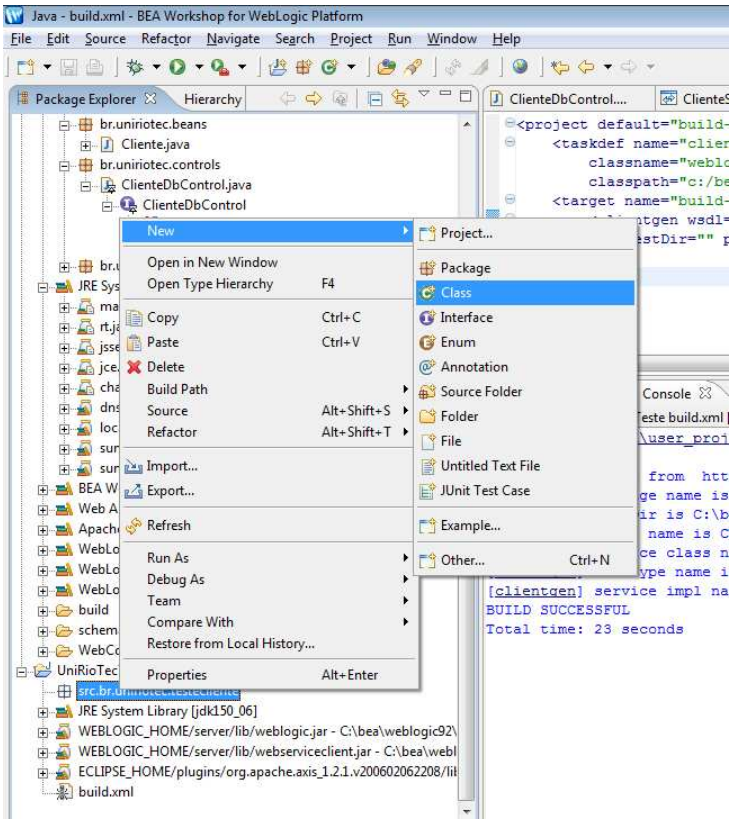

Figura 52 – Criação de classe Java

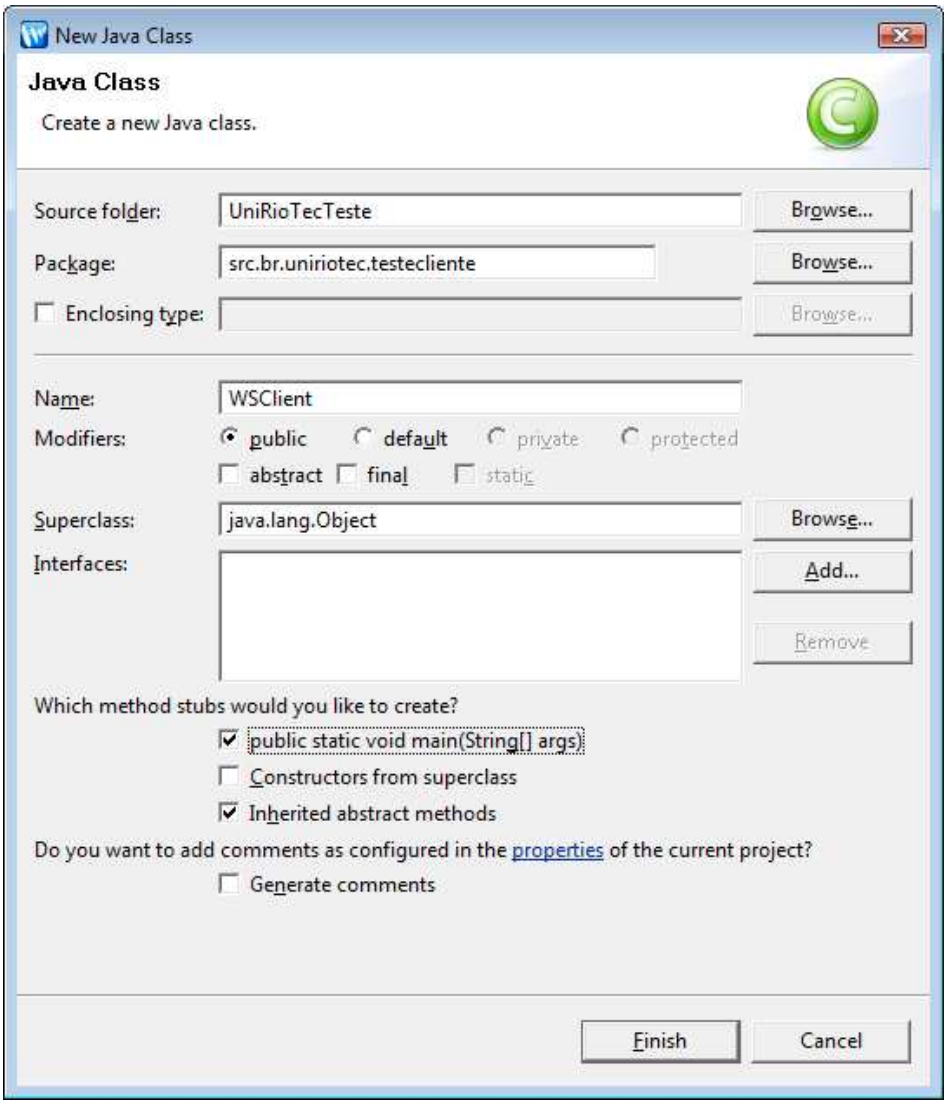

Figura 53 – Especificação da classe Java WSClient

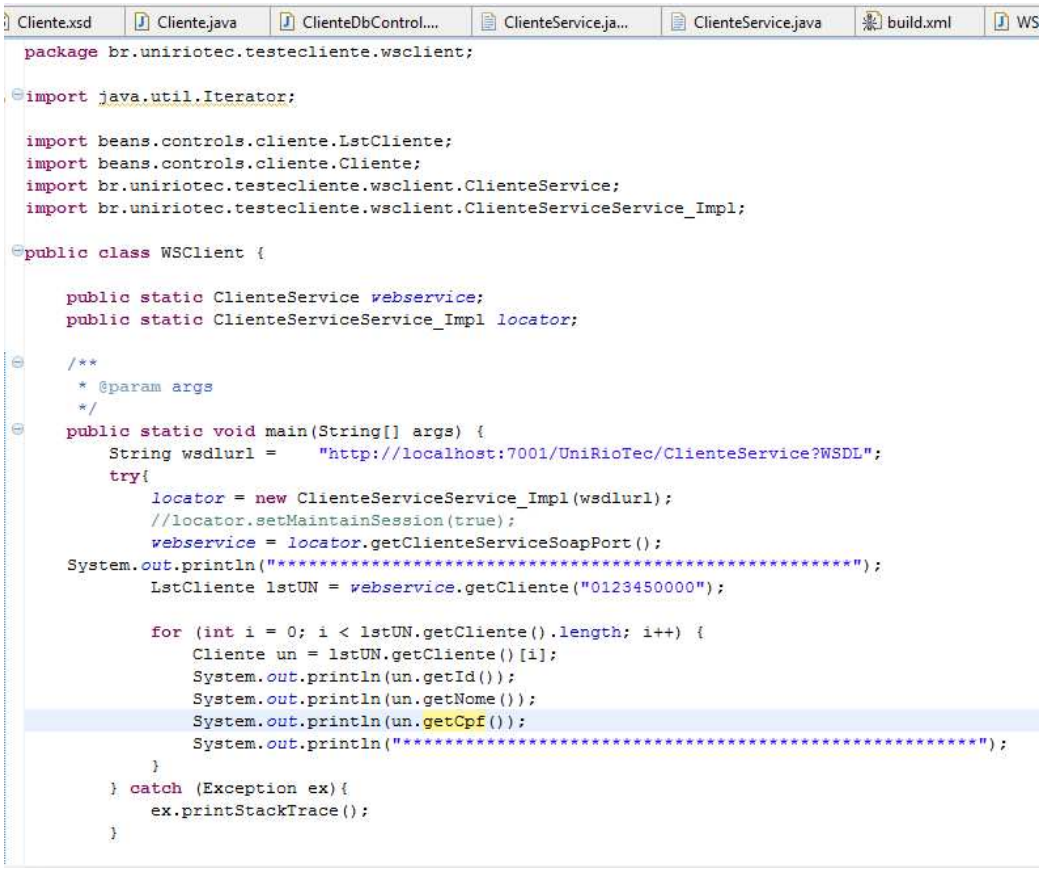

Figura 54 – Código da classe WSClient

7. Executar o cliente Java. (Run As ➔ Java Application) (Figura 55) Caso ocorra algum erro, verifique no console do servidor se a botão "Release Configuration" está habilitado, conforme a Figura 29. Se estiver, clique nele para liberar a configuração e execute novamente o servidor.

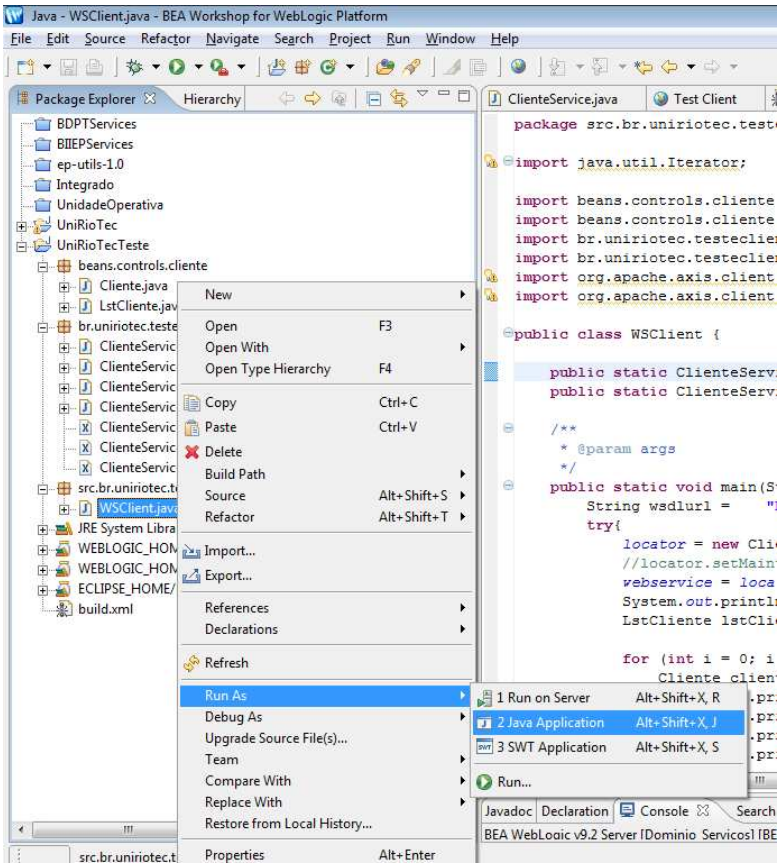

Figura 55 – Execução do cliente java como aplicação java

9. O resultado da execução deve ser algo semelhante à:

```
**********************************************************
1
Joao das Couves
0123450000
**********************************************************
```

4 Conclusão

O presente relatório teve como objetivo ressaltar os principais aspectos do desenvolvimento de serviços, em uma metodologia BOTTOM-UP. Dessa forma, o serviço é gerado a partir de uma consulta na base de dados, a qual foi solicitada por uma demanda de acesso a dados.

Foram apresentados detalhes da implementação de serviços utilizando as ferramentas da BEA para acessar um banco de dados armazenado no PostgreSQL. Um classe POJO foi definida para armazenar o resultado da consulta e rotinas de transformação são responsáveis por transformar objetos POJO em estrutura XML de acordo com arquivo XSD definido qual é retornada para o consumidor do serviço.

5 Referências

ERL, T., 2005, **Service-Oriented Architecture: concepts, technology, and Design**, Prentice Hall.

PAPAZOGLOU, MIKE P.; HEUVEL, WILLEM-JAN, 2007, **Service oriented architectures: approaches, technologies and research issues**, VLDB Journal, Springer-Verlag.