cacore RESTFUL wrapper

Design DOCUMENT

Version 1.0

02/14/2013

**Version History**

[Provide information on how the development and distribution of the Design Specification will be controlled and tracked. Use the table below to provide the version number, the author implementing the version, the date of the version, the name of the person approving the version, the date that particular version was approved, and a brief description of the reason for creating the revised version.]

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Version Number** | **Implemented**  **By** | **Revision**  **Date** | **Approved**  **By** | **Approval**  **Date** | **Description of Change** |
| 0.1 | *Prasad Konka* | *02/14/13* |  |  | *Draft version* |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

***Notes to the Author***

[This document is a template of a Design Specification document for a project. The template includes instructions to the author, boilerplate text, and fields that should be replaced with the values specific to the project.

* Blue italicized text enclosed in square brackets ([text]) provides instructions to the document author, or describes the intent, assumptions and context for content included in this document.
* Blue italicized text enclosed in angle brackets (<text>) indicates a field that should be replaced with information specific to a particular project.
* Text and tables in black are provided as boilerplate examples of wording and formats that may be used or modified as appropriate to a specific project. These are offered only as suggestions to assist in developing project documents; they are not mandatory formats.

**When using this template, the following steps are recommended:**

1. Replace all text enclosed in angle brackets (e.g., <Project Name>) with the correct field document values. These angle brackets appear in both the body of the document and in headers and footers. To customize fields in Microsoft Word (which display a gray background when selected) select File->Properties->Summary and fill in the appropriate fields within the Summary and Custom tabs.

After clicking OK to close the dialog box, update all fields throughout the document selecting Edit>Select All (or Ctrl-A) and pressing F9. Or you can update each field individually by clicking on it and pressing F9.

These actions must be done separately for any fields contained with the document’s Header and Footer.

1. Modify boilerplate text as appropriate for the specific project.
2. To add any new sections to the document, ensure that the appropriate header and body text styles are maintained. Styles used for the Section Headings are Heading 1, Heading 2 and Heading 3. Style used for boilerplate text is Body Text.
3. To update the Table of Contents, right-click on it and select “Update field” and choose the option - “Update entire table”.
4. Before submission of the first draft of this document, delete this instruction section “Notes to the Author” and all instructions to the author throughout the entire document.

**Level of required documentation and management rigor**

* Project management is the application of knowledge, skills, tools, and techniques to project activities to meet project requirements. Project management is accomplished using project management knowledge, skills, tools, and techniques. These can be applied with varying degrees of rigor, depending on project demands. In order for projects to be successful, it is important for the project team to diligently apply these with a degree of rigor that best promotes achievement of the project's objectives. For example, the more risk, complexity, budget, etc a project has the more project documentation and management rigor will be required to successfully manage and document the intricacies of that project. To what level this rigor expands to beyond this basic template, and is applied to the management of the project, should be at the discretion of the project manager unless otherwise instructed. When in doubt as to what level of management and/or documentation rigor to apply to a project, in almost all cases, more is better than less.]

TABLE OF CONTENTS

[1 Introduction 5](#_Toc348594703)

[1.1 Purpose of The Design Specification Document 5](#_Toc348594704)

[2 General Overview and Design Guidelines/Approach 5](#_Toc348594705)

[2.1 Assumptions / Constraints / Standards 5](#_Toc348594706)

[2.2 Alignment with Federal Enterprise Architecture (FEA) 5](#_Toc348594707)

[3 Architecture Design 5](#_Toc348594708)

[3.1 Logical View 7](#_Toc348594709)

[3.1.1 Wrapper UI 7](#_Toc348594710)

[3.1.2 Code Generator: 9](#_Toc348594711)

[3.2 Hardware Architecture 10](#_Toc348594712)

[3.3 Software Architecture 11](#_Toc348594713)

[3.3.1 Wrapper UI: 11](#_Toc348594714)

[Working with WSDL: 14](#_Toc348594715)

[Working with EJB: 15](#_Toc348594716)

[3.3.2 CODE GENERATION: 16](#_Toc348594717)

[3.4 Security Architecture 20](#_Toc348594718)

[3.5 Communication Architecture 20](#_Toc348594719)

[3.6 Performance 20](#_Toc348594720)

[4 System Design 20](#_Toc348594721)

[4.1 Business Requirements 20](#_Toc348594722)

[4.2 Database Design 21](#_Toc348594723)

[4.3 Data Conversion 21](#_Toc348594724)

[4.4 Application Program Interfaces 21](#_Toc348594725)

[4.5 User Interface Design 22](#_Toc348594726)

[4.6 System Performance 30](#_Toc348594727)

[4.7 Section 508 Compliance 30](#_Toc348594728)

[Appendix A: Design Specification Approval 31](#_Toc348594729)

[APPENDIX B: REFERENCES 32](#_Toc348594730)

[APPENDIX C: KEY TERMS 32](#_Toc348594731)

[APPENDIX D: Mapping Schema 32](#_Toc348594732)

# Introduction

## Purpose of The Design Specification Document

The Design Specification document documents and tracks the necessary information required to effectively define architecture and system design in order to give the development team guidance on architecture of the system to be developed. Design documents are incrementally and iteratively produced during the system development life cycle, based on the particular circumstances of the IT project and the system development methodology used for developing the system. These documents are initially created during the Planning Phase of the project and updated as necessary throughout the design process. The preliminary versions of these documents are reviewed during the EPLC Stage Gate Preliminary Design Review; the final versions are input to the EPLC project Detailed Design Review. Its intended audience is the project manager, project team, and development team. Some portions of this document such as the user interface (UI) may on occasion be shared with the client/user, and other stakeholder whose input/approval into the UI is needed.

# General Overview and Design Guidelines/Approach

This section describes the principles and strategies to be used as guidelines when designing and implementing the system.

## Assumptions / Constraints / Standards

* This version of RESTful Wrapper would wrap SOAP webservice and EJB 2.x only. Wrapper Code generator uses CXF WSDL2JAVA tool to generate web service client interfaces. WSDL uploaded to the Wrapper tool should be compliant with WSDL v1.1 and the tool assumes that WSDL is valid to work with CXF WSDL2JAVA.
* In the case of EJB Wrapper, Code generator uses uploaded POJO or POJO JAR to marshall and unmarshall request and response. Uploaded POJO should be valid to use with corresponsing uploaded EJB.
* The Wrapper UI and generator let user wrap either EJB or SOAP web service at a time. Wrapping SOAP web service and EJB can not be combined into one RESTful resource.
* Wrapper generation starts with uploading a POJO class or POJO classes JAR. A service that uses simple types for input and output is not supported.

## Alignment with Federal Enterprise Architecture (FEA)

# Architecture Design

The caCORE RESTful Wrapper has two modules:

* **Wrapper UI**: A user interface to define mappings from RESTful resource to existing SOAP web service or an EJB. The UI lets user to start with a POJO class file or Jar of POJO class files to define RESTful resources. Uploaded POJO(s) should the input and/or output of corresponding EJB or SOAP web service operations. User can load definitions of existing SOAP web service (WSDL) or EJB 2.x JAR to show its operations in a tree view. The UI lets user to map operations from RESTful resource to SOAP Web service or EJB. This mapping can be saved into an XML and loaded into the UI at a later time for editing.

Version 1.0 of RESTful Wrapper supports WSDL 1.1 that can be parsed by CXF WSDL2JAVA tool. JBoss 5.x is supported to deploy RESTful Wrapper generated web application. This version of RESTful Wrapper is limited to support Session EJB of version 2.x

* **Code Generator**: Code generation takes Wrapper UI generated mapping file as its input and generate necessary RESTful resource artifacts and client interfaces to interact with SOAP web service or EJB. Generated RESTful resource would use generated client API to interact with SOAP web service or EJB. The output of code generator is to generate a deployable WAR file with configured RESTful resources from the mapping file.



**Mapping:** Mapping captures information about RESTful Resource to be created and the information about the data service that it is going to wrap to provide RESTful methods.

APPENDIX A below shows the schema for mapping XML. Use of Mapping XML decouples Wrapper UI and Wrapper code generator. Even though the code generation is initiated from the Wrapper UI, the code generation can be intiated from any client with Mapping XML as an input.

**Wrapper Resource:**The generated Wrapper Resource provides RESTful implementation for all mapped methods. When a client makes a request to generated RESTful resource method, it would internally call mapped EJB or SOAP Webservice through corresponding generated client code.

The following diagrams shows the working of RESTful Wrapper Resource with a SOAP web service.



The following diagrams shows the working of RESTful Wrapper Resource with EJB.



## Logical View

### Wrapper UI

The Wrapper UI provides convenient user interface to define RESTful resource and to map it with existing SOAP webservice or EJB. RESTful Wrapper UI is Java Swing based application with three panels and a menu bar. Left panel displays RESTful resource(s) and operations in a tree structure. Right panel displays uploaded WSDL or EJB JAR information and operations. Middle panel displays graph the animates relation between RESTFul resource method and operation on SOAP Web service or EJB service. Each POJO uploaded represents a RESTful resource and each resource presents CREATE, UPDATE, READ and DELETE methods to map with EJB or SOAP web service operations.

**File:** Following diagram shows use case view Wrapper UI File menu item. The Wrapper mapping starts with opening an existing POJO or a Jar of POJOs. Clicking on File menu item and Open POJO sub menu item opens a file dialog that lets user select a POJO class file. Clicking on File menu item and Open POJO JAR sub menu item opens a file dialog that lets user select a POJO JAR file. Clicking on Options sub menu item opens a window to enter values for Project Root directory, Project Output directory options. Clicking on Save sub menu item saves the mapping in xml format. Clicking on Open Mapping sub menu item opens a file dialog to select a mapping file. After successful validation, Wrapper UI represents the mapping in the UI. Clicking on Exit sub menu item closes theWrapper UI.



**Upload:** Clicking on Upload menu item and WSDL sub menu items opens file dialog to select a WSDL file. Clicking on Upload menu item and EJB JAR sub menu items opens file dialog to select a EJB JAR file. Wrapper UI parses uploaded file and present its operations in a tree view.



**Generate:** Clicking on generate menu item and RESTful resource sub menu item intiates code generation process. The process progress is displayed in a window including info messages, warnings and errors.



**Help:** Clicking on Help menu item opens a window displaying version and build of RESTful Wrapper. It also displays RESTful Wrapper user documentation link to NCI wiki.



### Code Generator:

Code generator component is responsible to generate all necessary artifacts resulting a deployable RESTful resource wrapping EJB or SOAP Webservice. Mapping information generated by Wrapper UI is the input for code generator component. Code generator component reads mapping information and after successful validation, it proceed to generate following artifacts:

1. Client code: If RESTful service is wrapping a SOAP Webservice, code generator uses Apache CXF WSDL2Java to generate client stub and its supporting classes. If RESTful service is wrapping a EJB JAR, code generator generates client stub and its supporting classes to wrap calls to EJB.
2. Configuration files: Code generator generates web application configuration files web.xml and spring configuration file (cxf-beans.xml).
3. RESTful resource: Code generator generates RESTful resources based on the mapping configuration. Multiple RESTful resources are generated if mapping configuration has multiple mappings.
4. Compile Generated code: Code generator compiles generatd code and report any errors. On error, the generation process will be aborted.
5. Generate deployable .war file: After successful code generation and compilation, code generator created deployable .war file in given output folder.



## Hardware Architecture

[Insert any related hardware architecture documents or provide a reference to where they are stored.]

## Software Architecture

## 

### Wrapper UI:

caCORE RESTful Wrapper UI is a swing based user interface to create RESTful resource mapped with existing EJB or SOAP web service. UI architectures consists of following major components:

1. MainFrame: This component is responsible to render Wrapper UI frames and menu bar. Following is the class model.



1. Tree Component: Tree component is responsible to render tree structure for RESTful resources and EJB or SOAP Web service operations.



1. Graph component: Graph component is responsible to represent mapping between RESTful resource method and EJB or SOAP web service operation. This component supports removing the mapping and setting attribute value on a mapping. Following is the class model:



Working with Wrapper UI mapping starts with opening a POJO or POJO JAR. Wrapper UI follows following rules opening a POJO or POJO JAR.

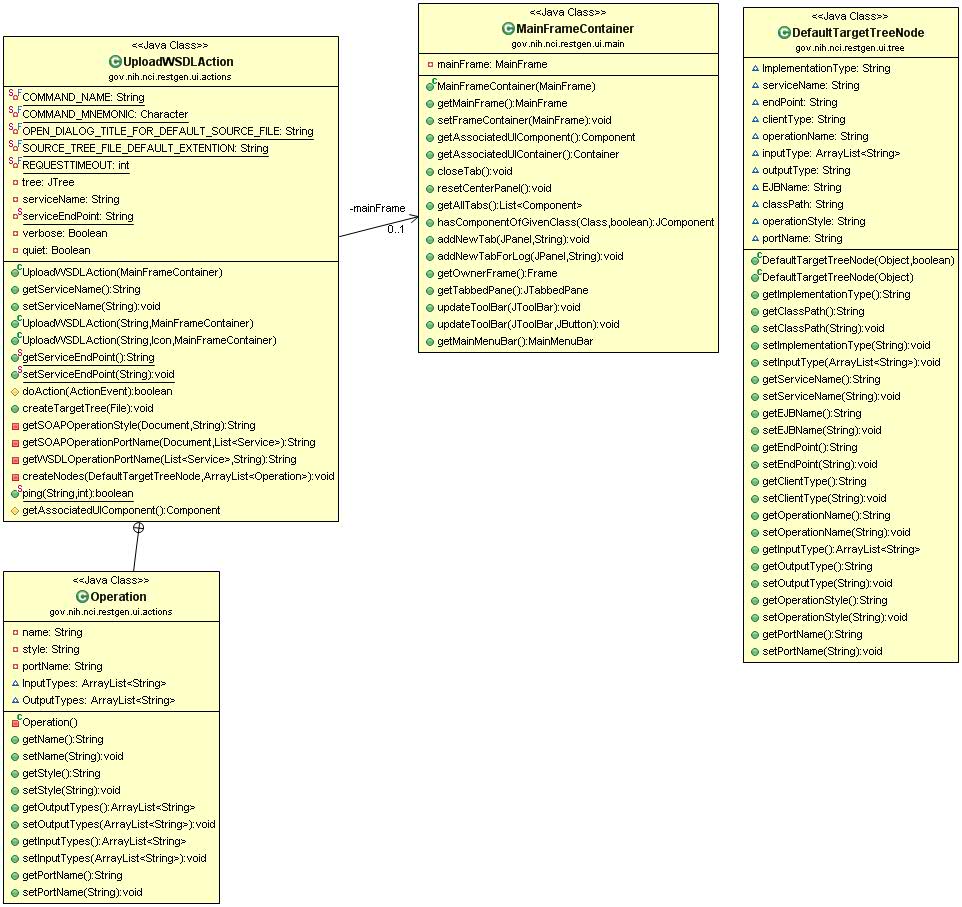
1. Loading Abstract POJO is not valid.
2. POJO should be a valid class and it should be serializable, has a no-argument constructor, and allows access to properties using getter and setter methods that follow a simple naming convention.
3. Uploaded POJO is used to marshall and unmarshall input and output data using generated RESTful resource.

### Working with WSDL:

The WSDL component downloads and parses the components of the SOAP based webservice from a .wsdl file available at the public URL.The downloaded wsdl file is validated with the cxf validator and below SOAP artifacts are parsed from the file:

1. SOAP Service Name
2. SOAP Service Operations.
3. SOAP Service End Point.
4. SOAP Service Port.

The parsed SOAP service operations are displayed as target tree nodes in the right panel of the GUI which can be mapped to the POJO resource methods displayed on the left panel.Following is the class model.

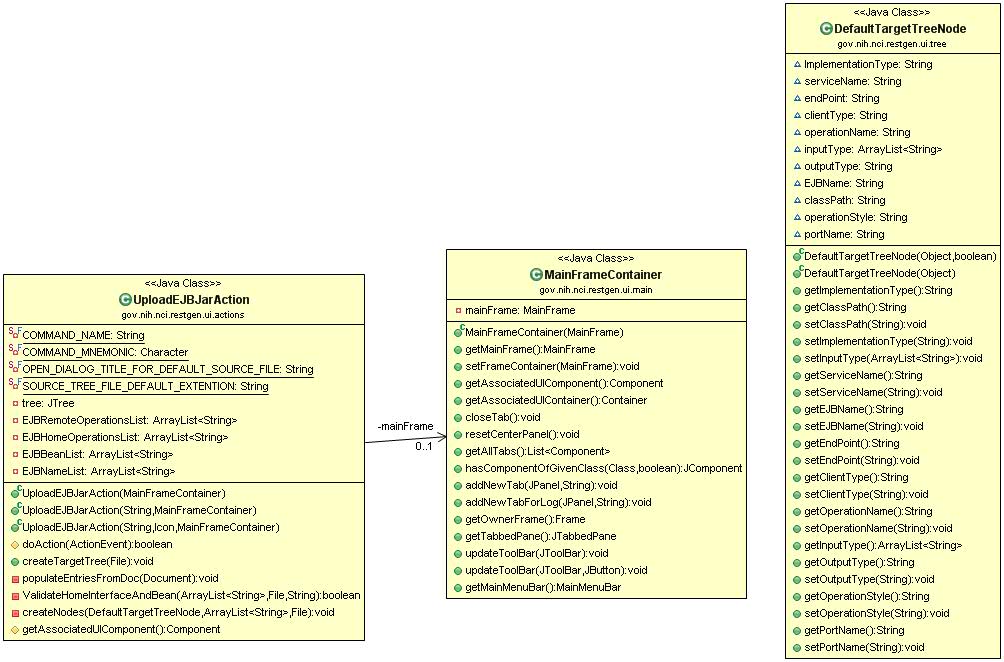


### Working with EJB:

The EJB component validates the contents of the uploaded EJB jar file and parses the EJB deployment descriptor file (ejb-jar.xml) to obtain the below artifacts:

1. List of SessionBeans.
2. Session Bean Name.
3. Session Bean Class.
4. Session Bean Remote and Home Interface .
5. Remote Interface Operations

The parsed EJB operations are displayed as target tree nodes in the right panel of the GUI which can be mapped to the POJO resource methods displayed on the left panel. Following is the class model.



### CODE GENERATION:

Code Generation is initated by calling RESTfulWrapperGenerator with GeneratorContext. RESTfulWrapperGenerator will in turn call all necessary generators based on given GeneratorContext. Following sequence diagram shows the details:



If GeneratorContext is initialized with mapping XML path, RESTfulWrapperGenerator reads mapping XML and use it to generate mapping object model to process. Following diagram shows the steps involved with Generator generate process.



RESTfulWrapperGenerator runProcess() includes calls to individual following generators in a sequential order.

1. RESTfulClientGenerator: This is responsible to generate client interfaces for EJB or SOAP Web services.
2. RESTfulResourceGenerator: This is responsible to generate RESTful resources including its methods making calls to EJB or SOAP web service through generated client API.
3. RESTfulWebResourceGenerator: This is responsible to generate Spring configuration file including generated RESTful resource information and CXF servlet through web.xml.



Here is the class model for Wrapper Generator. All RESTful Generator classes extends Generator and implement its operations. Generator.generate() operation is called on each of these generators. Generate operation run init(), preprocess(), runProcess(), postProcess() in sequence. All these operations throws GeneratorException. Each of these operations uses Logger from GeneratorContext to log info, warning and error messages. This logger can be used by Wrapaper client application to display generator status and progress.



Following is the class model for mapping.



## Security Architecture

[Insert any related security architecture documents, including integrity controls, or provide a reference to where they are stored.]

## Communication Architecture

[Insert any related communication architecture documents or provide a reference to where they are stored.]

## Performance

[Insert any performance documents or provide a reference to where they are stored.]

# System Design

## Business Requirements

The caCORE SDK generated runtime system's infrastructure exhibits an n-tiered architecture with client interfaces, server components, backend objects, data sources, and additional backend systems. SDK RESTful interface is built to use existing architecture and based on JAX-RS (Java API for RESTful Web Services) standard to serve client requests over HTTP and return formatted results in XML format. SDK RESTful implementation supports typical CRUD operations:

1. GET: Read access of the resource without side-effects.
2. PUT: create a new resource.
3. DELETE: remove a resource.
4. POST: update an existing resource.

The major advantages of SDK RESTful implementation are:

1. They are highly reusable across platforms (Java, .NET, and PHP) since they rely on basic HTTP protocol
2. They use basic XML instead of the complex SOAP XML and are easily consumable
3. Based on MDA and easy to build fully deployable web application
4. Built-in security with CSM

SDK RESTful implementation requires users to start with a UML model to generate the application. This step is a potential barrier for many of users in NCIP community to generate a data service and to share data. The community has built data services based on different technologies and development of the data exchange between the applications using these services is not an easy task. RESTful implementation provides a standardized interface and protocol to service client application across platforms. SDK RESTful Wrapper is aimed to lower the barrier to create RESTful services based on existing data sources. The wrapper will provide interface in RESTful manner while it internally maps with existing data source to query data. SDK RESTful Wrapper will provide User Interface where developers can create RESTful resources wrapping existing data services.

## Database Design

N/A

## Data Conversion

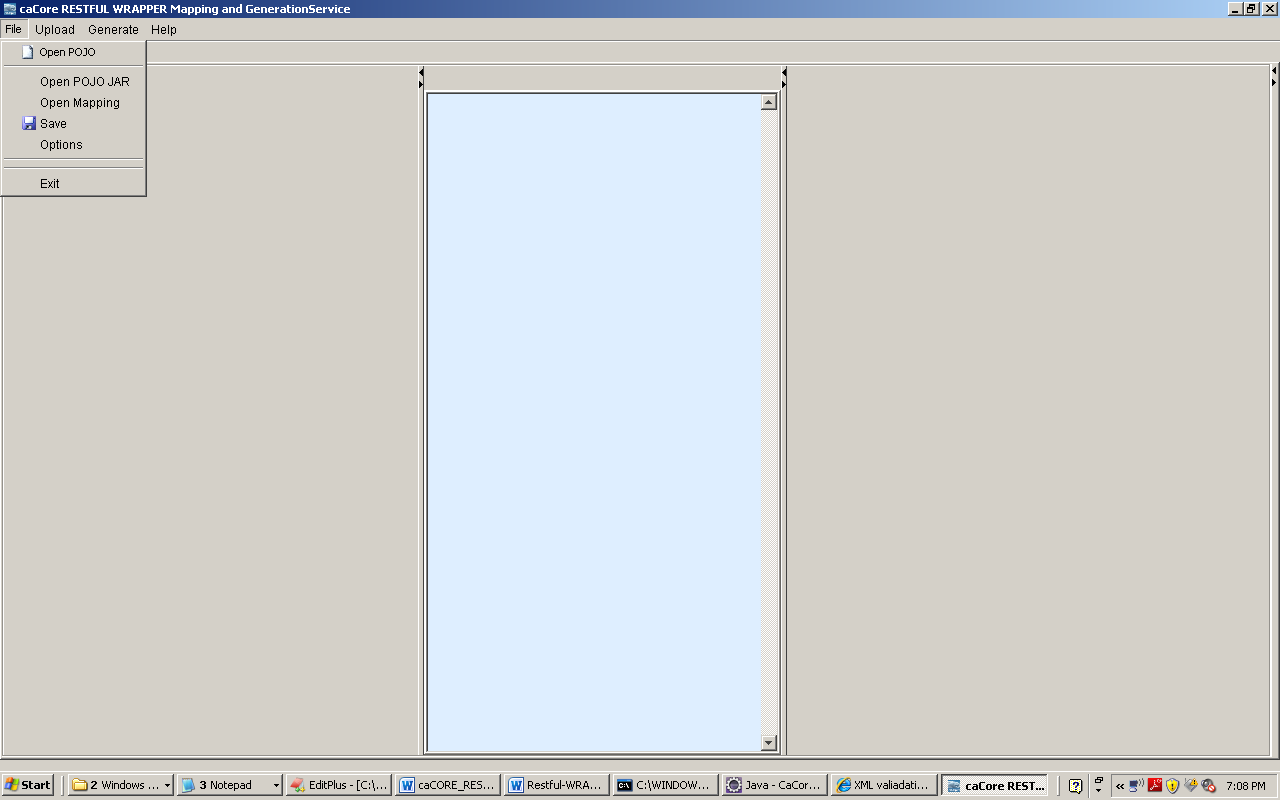
N/A

## Application Program Interfaces

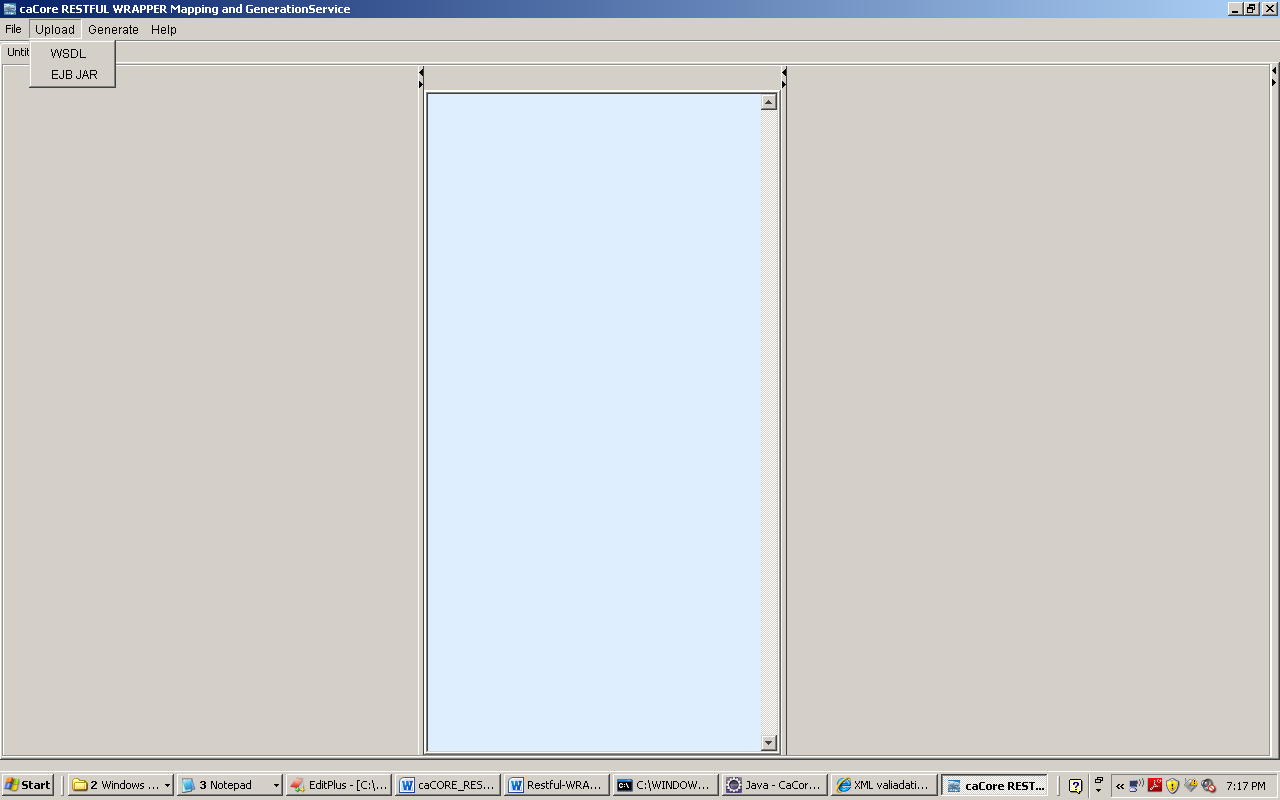
[Insert any application program interface documents or provide a reference to where they are stored.]

## User Interface Design

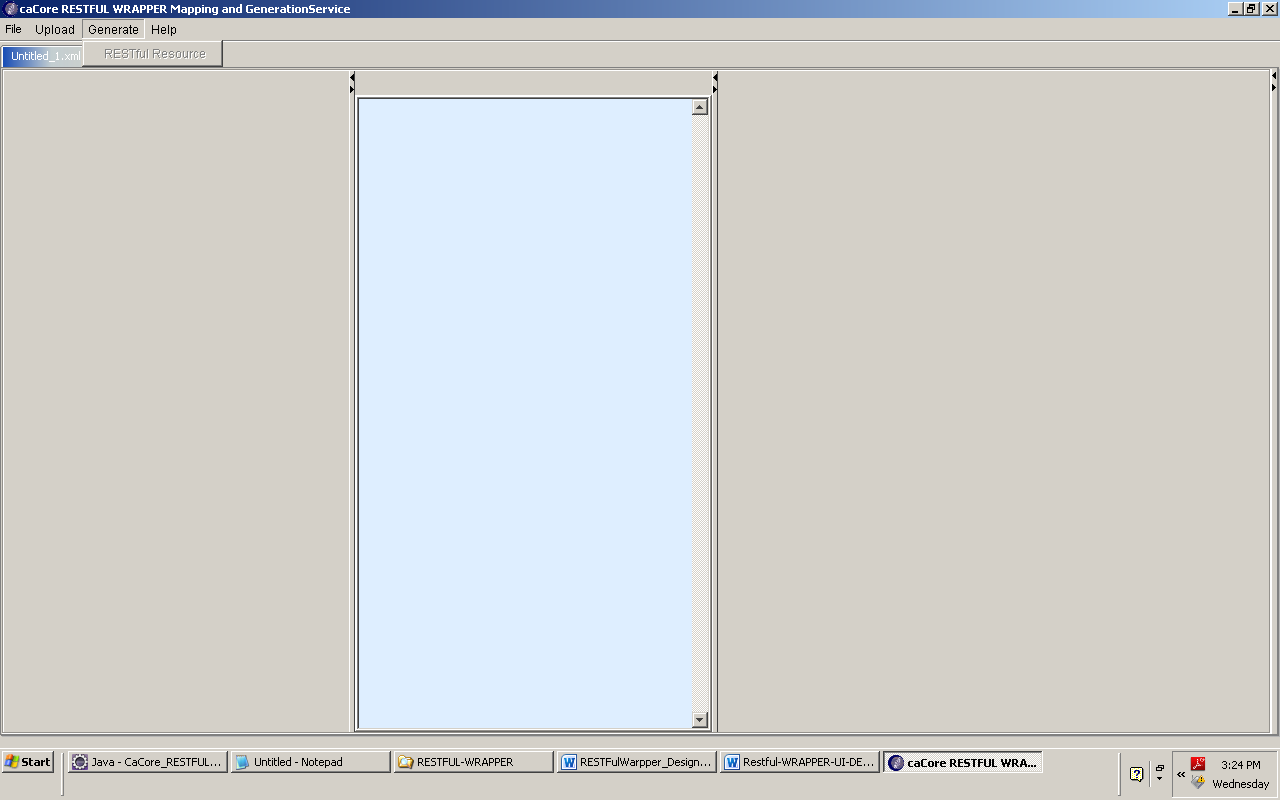
i) Following diagram represents Wrapper UI with File menu clicked.



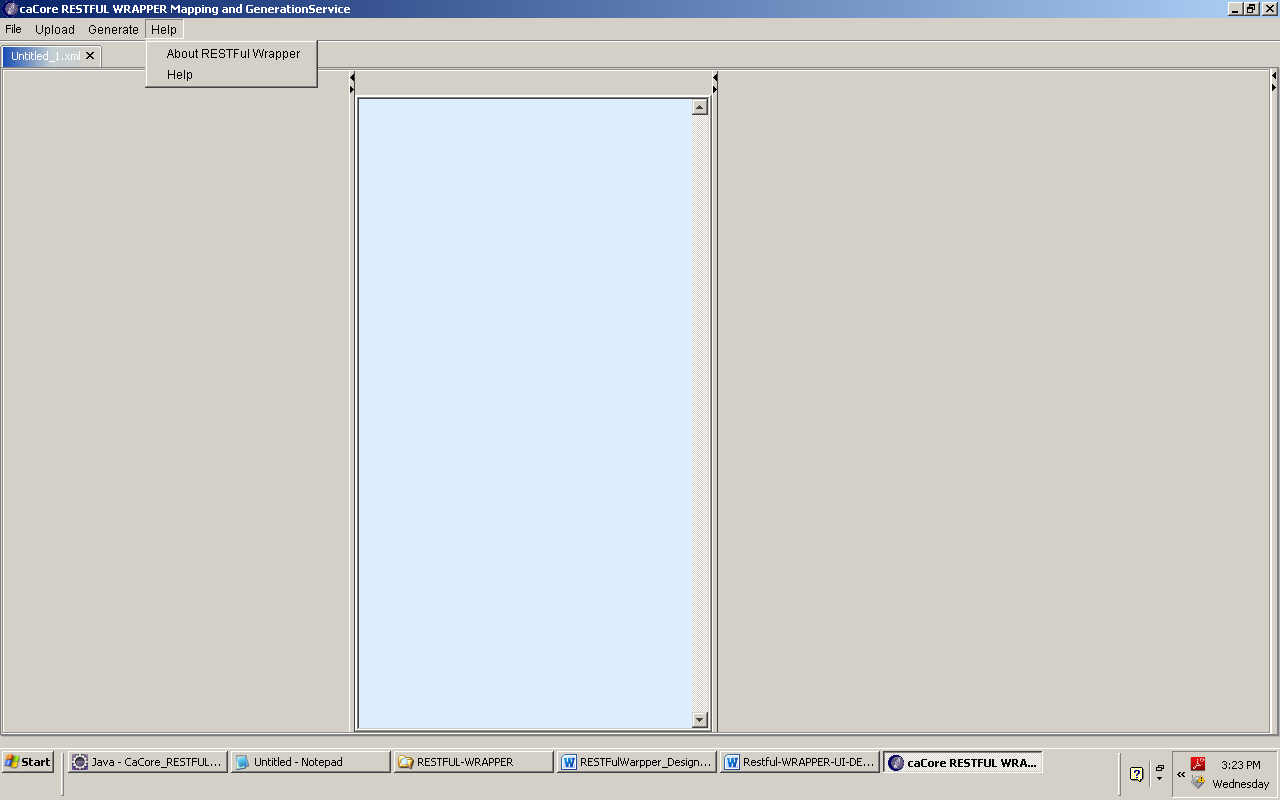
ii) The below diagram represents Wrapper UI when the Upload Menu is clicked



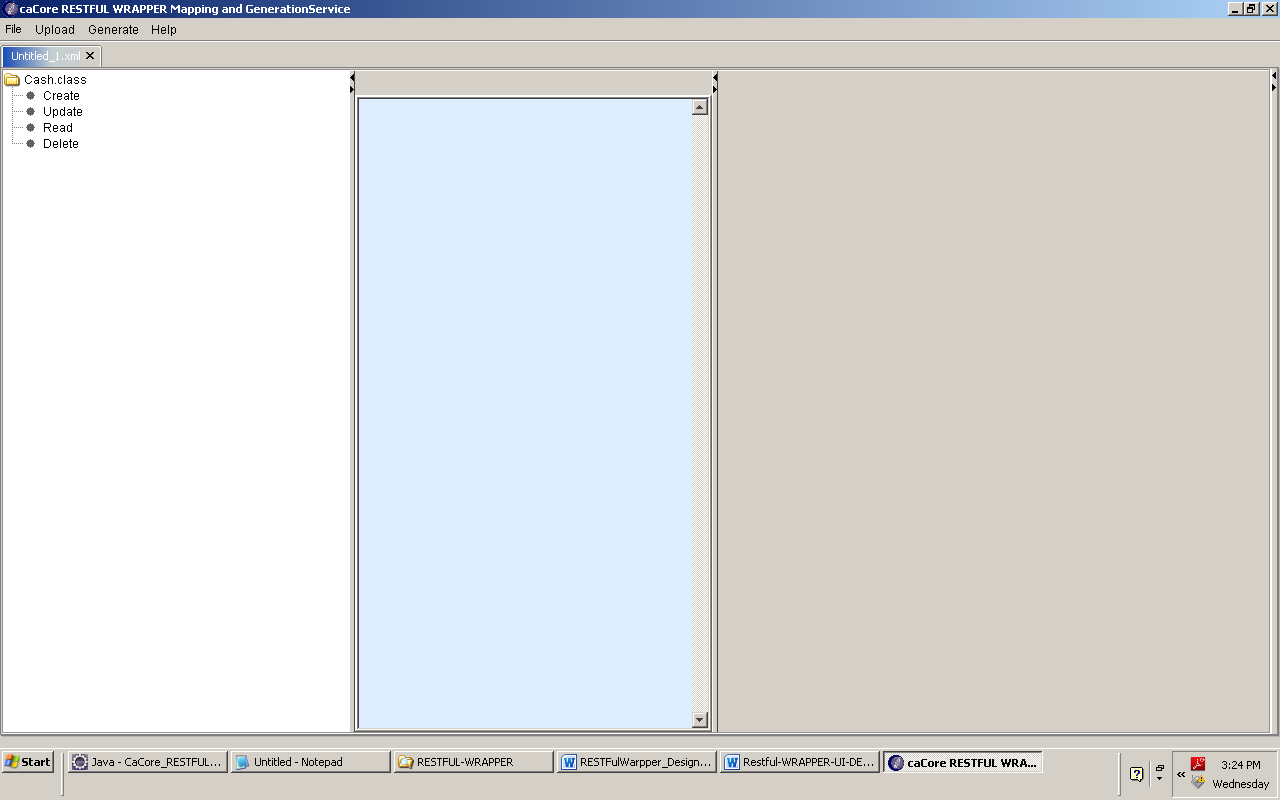
iii) The below digram represents Wrapper UI when Generate Menu is clicked



iv) The below diagram represents Wrapper UI when Help menu is clicked



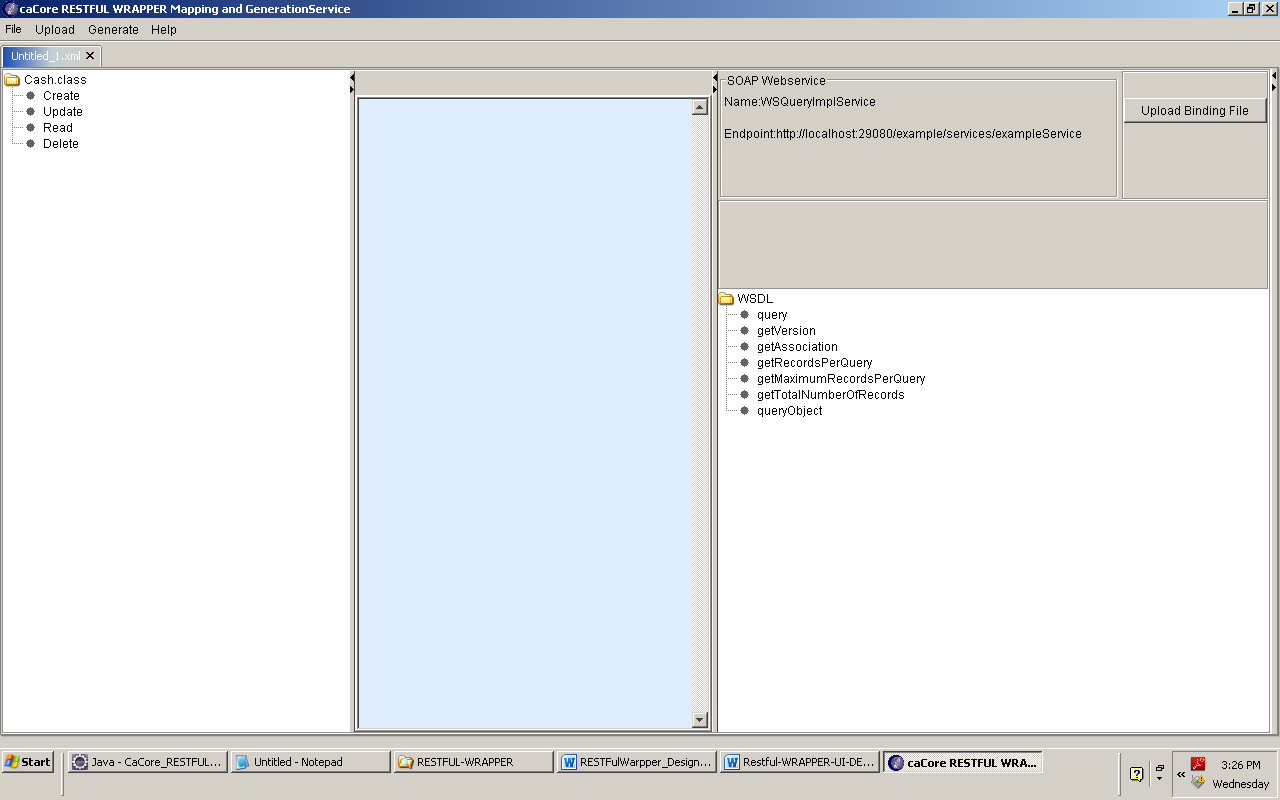
v) The below diagram represents the Wrapper UI when a POJO class has been uploaded and is displayed as a tree in the left panel of the UI.



vi) The Below diagram represents the Wrapper UI when a wsdl file is uploaded from an URL,

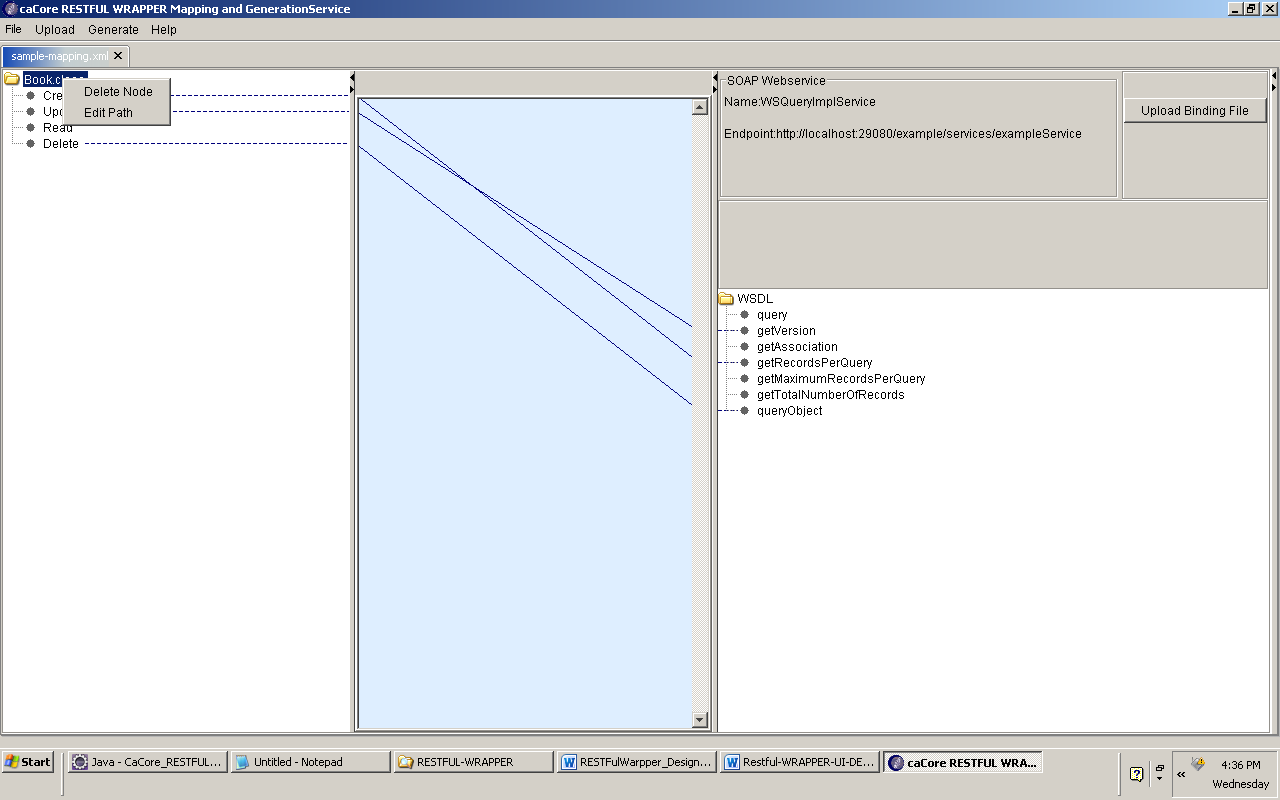
The right hand panel displays the below items:

1. SOAP Service Name.
2. SOAP Service EndPoint.
3. Binding File Path.
4. Upload Binding File button.
5. Tree with nodes displaying the SOAP operations

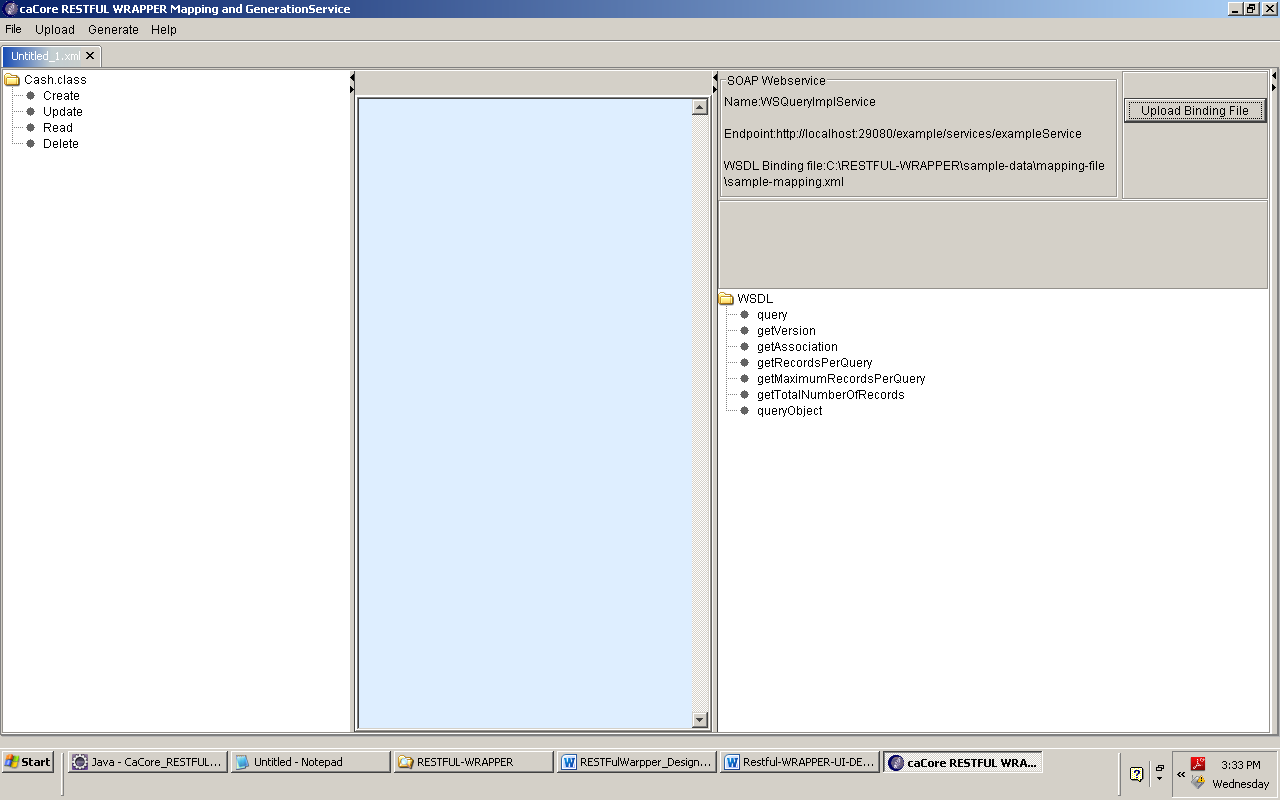


In the above diagram the “Upload Binding File” button allows the user to enter the path for the WSDL binding file

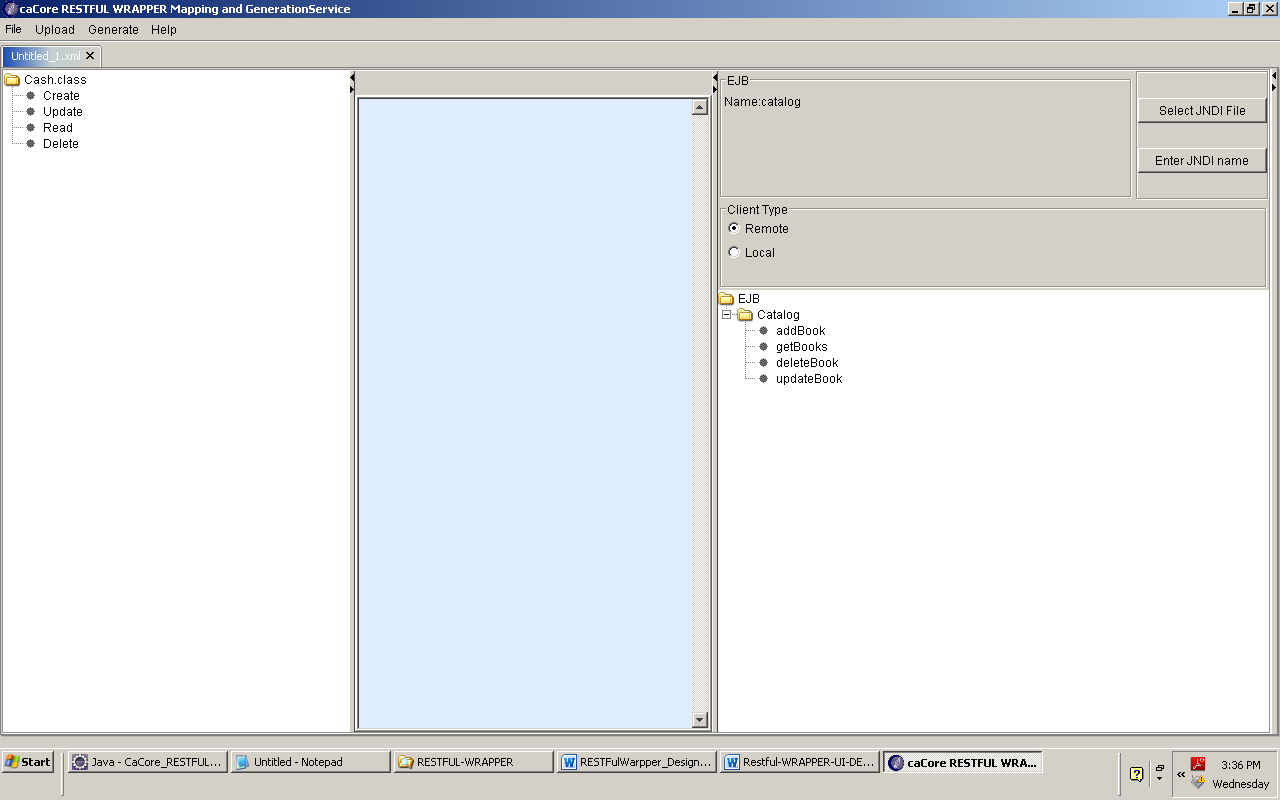
Below Diagram Represents Wrapper UI when Resource Path is to be entered for the POJO in the left panel



vii) Below Diagram Represents Wrapper UI when the Binding File Path has been entered



viii) Below Diagram represents the Wrapper UI when a EJB jar is uploaded to the target panel

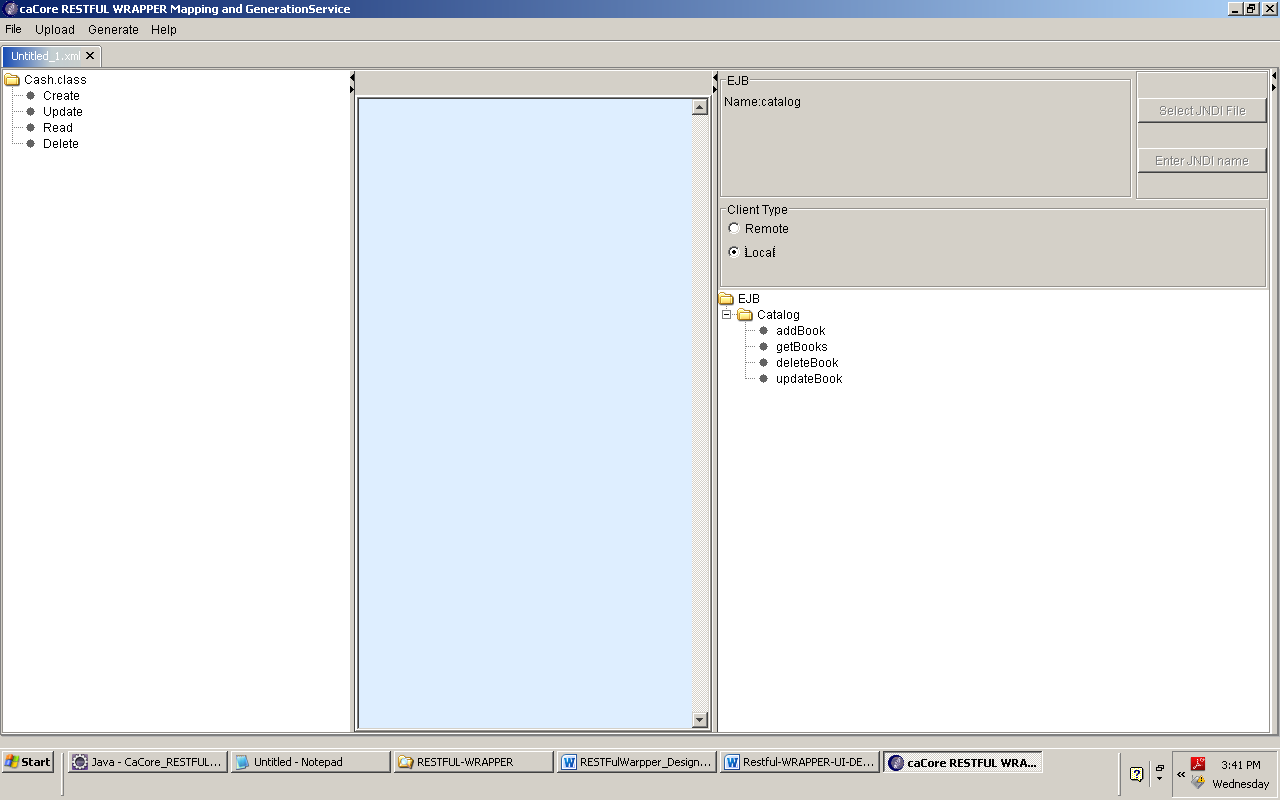


The right hand panel in the above diagram displays the below items:

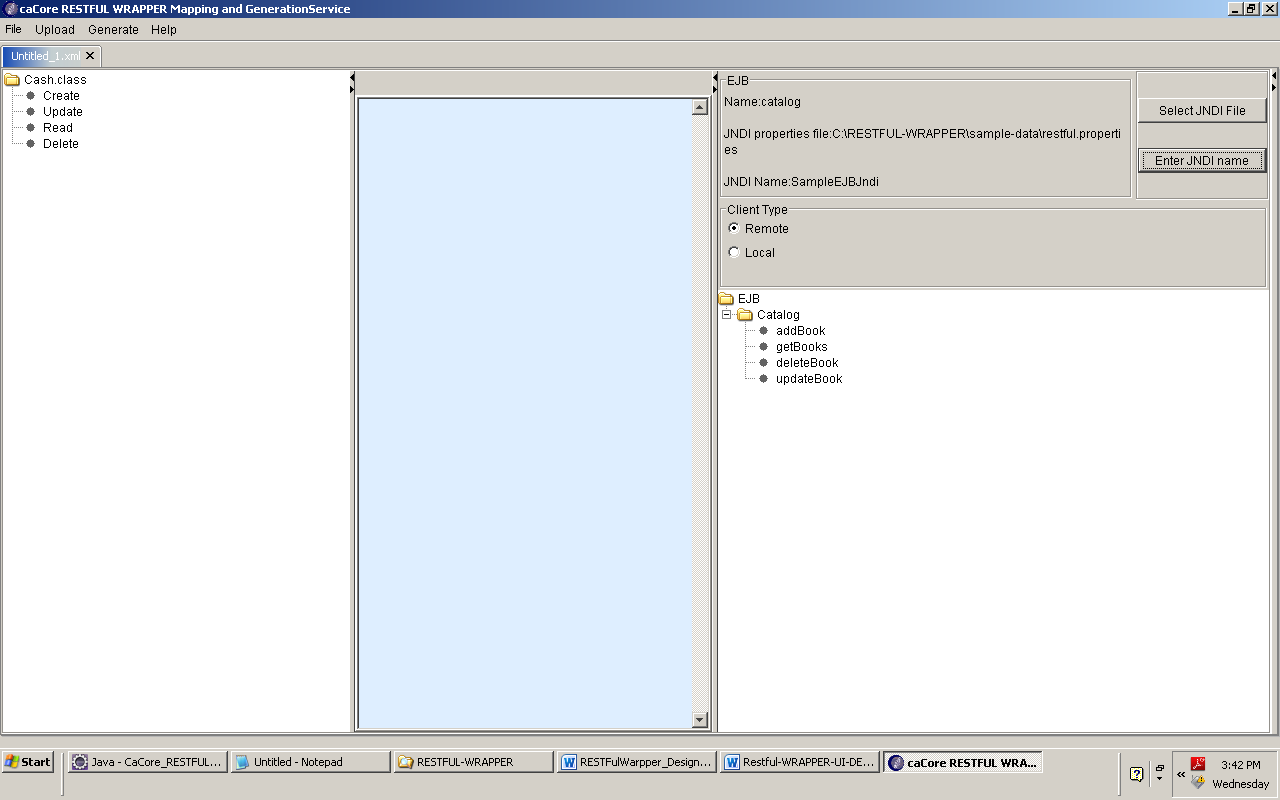
1. EJB Name.
2. Select JNDI File button to select the JNDI file.
3. Enter JNDI Name button to enter JNDI name.
4. Client Type Radio Button group to select Client Type either as “Remote” or “Local”
5. Tree with nodes displaying the EJB operations

The above two buttons(2,3) are disabled if the Client Type is “Local” but are enabled when the ClientType is “Remote”.

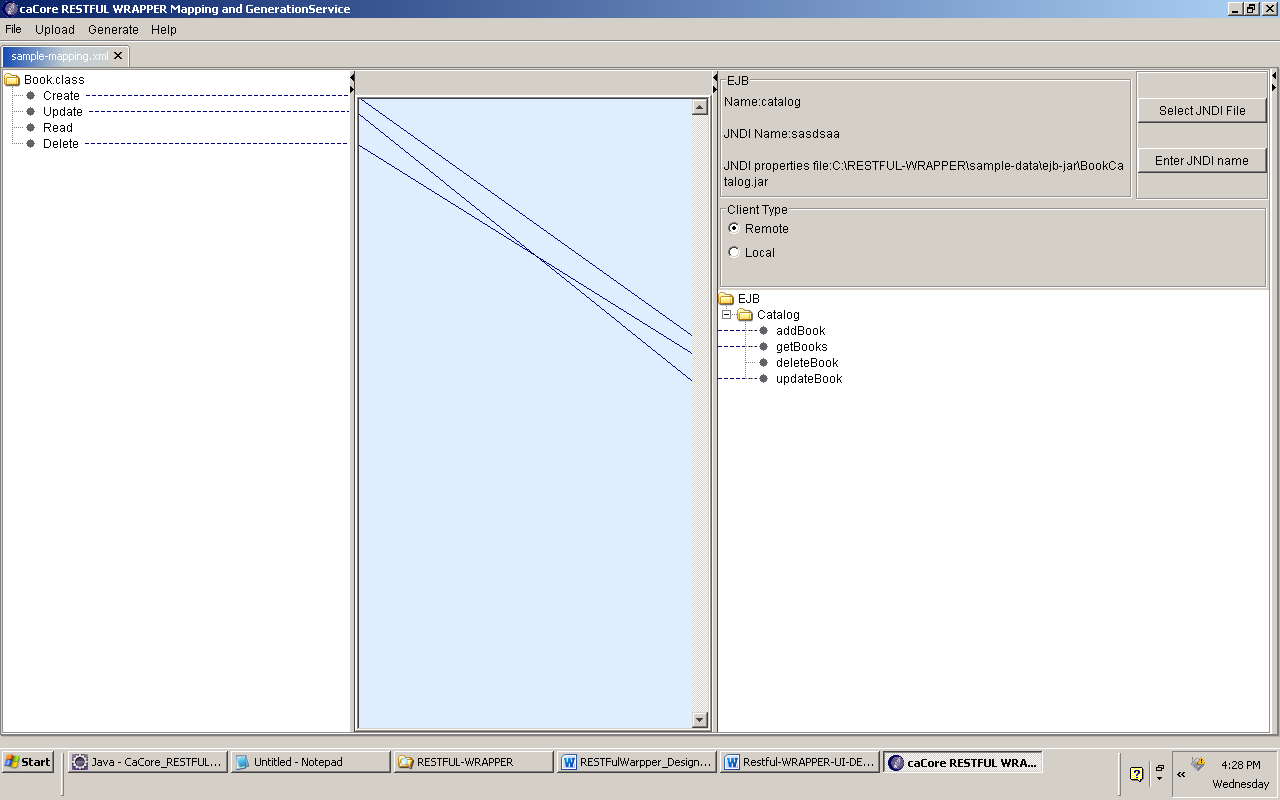
xi) Below diagram represents the Wrapper UI when the Client Type for the EJB has been selected as “Local”



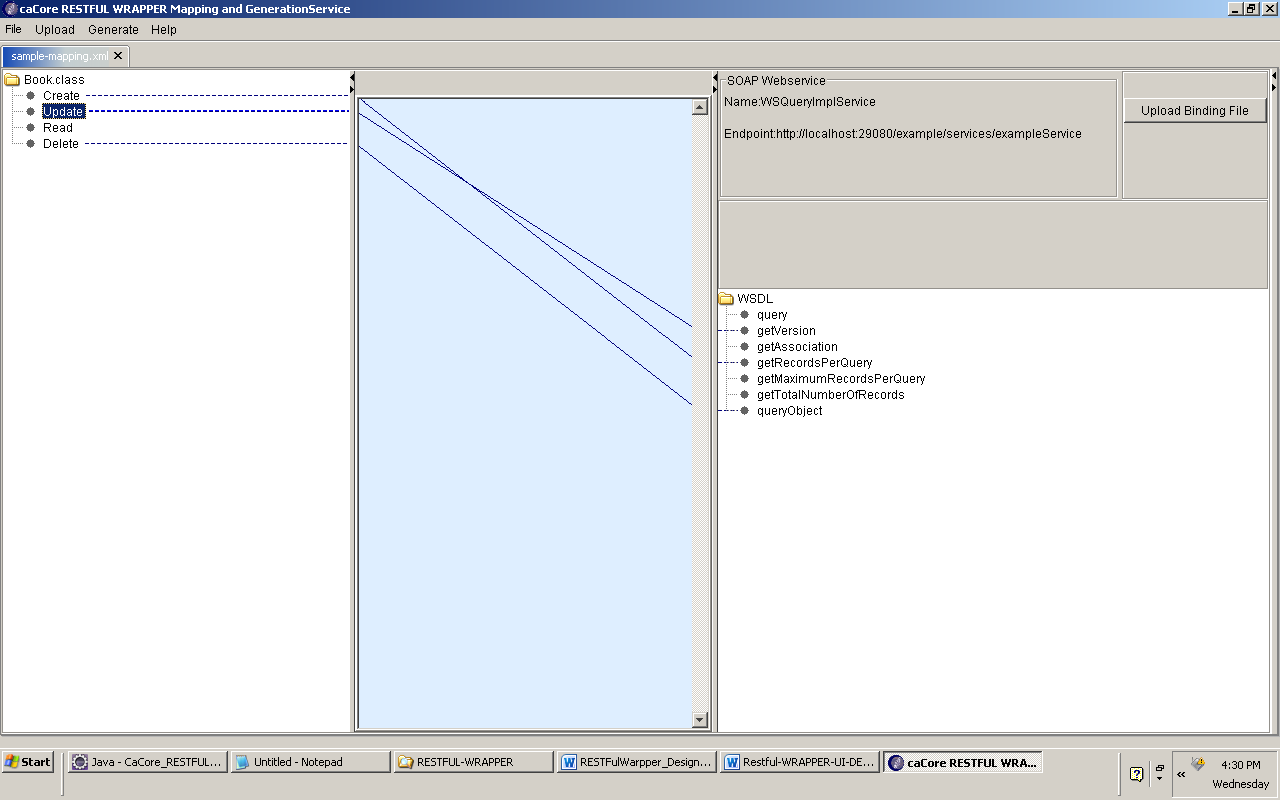
x) Below diagram represents the Wrapper UI when the “JNDI properties file” and “JNDI Name” are entered



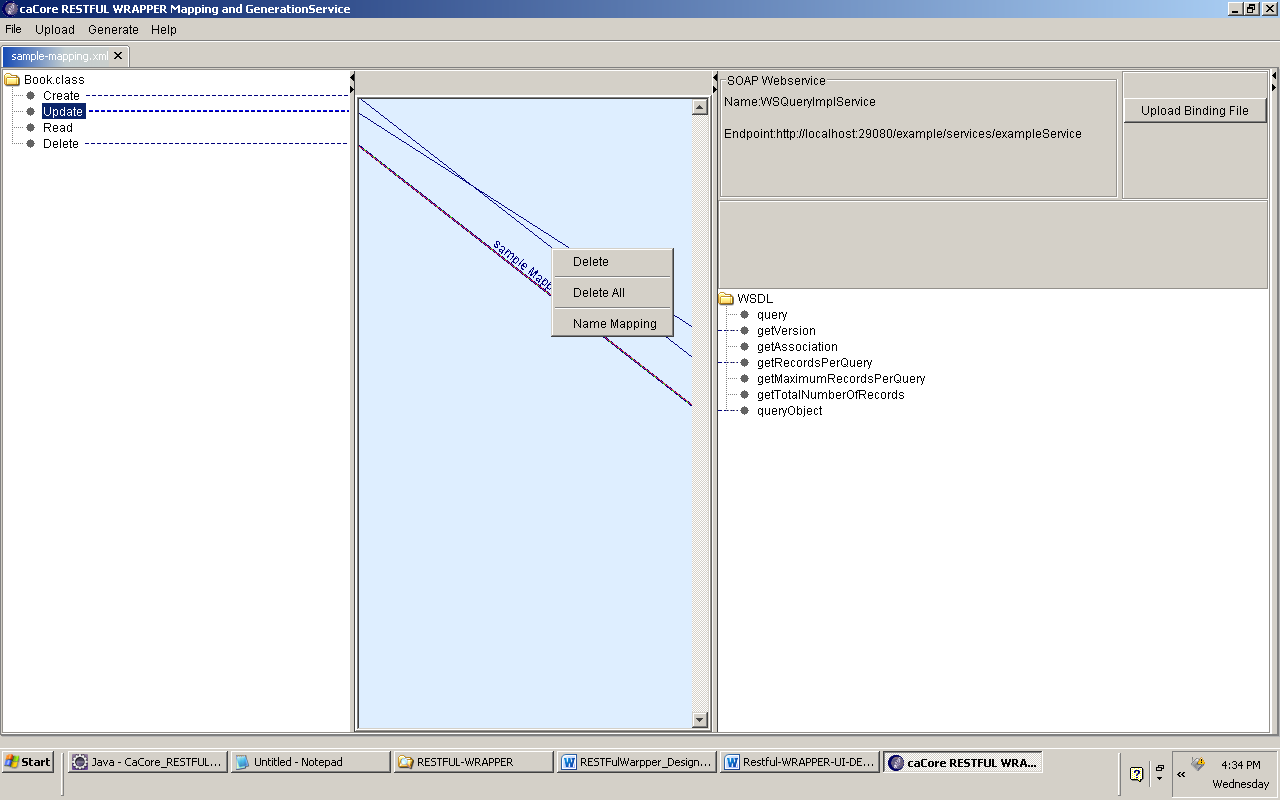
xi) Below diagram represents the Wrapper when source POJO and target EJB nodes are mapped to each other



xii) Below Diagram represents the Wrapper UI when source POJO and target SOAP Operation nodes are mapped to each other



xiii) Below diagram represents the Wrapper UI to edit the mapping name for any mapping link



## System Performance

N/A

## Section 508 Compliance

[Insert any section 508 compliance related documents or provide a reference to where they are stored.]

Appendix A: Design Specification Approval

The undersigned acknowledge that they have reviewed the caCORE RESTful Wrapper **Design Specification** and agree with the information presented within this document. Changes to this **Design Specification** will be coordinated with, and approved by, the undersigned, or their designated representatives.

|  |  |  |  |
| --- | --- | --- | --- |
| Signature: |  | Date: |  |
| Print Name: | Jonathan Lin |  |  |
| Title: | Associate Director of Engineering |  |  |
| Role: | Government Sponsor |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Signature: |  | Date: |  |
| Print Name: | Zhengwu Lu |  |  |
| Title: | Technical Project Manager |  |  |
| Role: | Technical Project Manager |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Signature: |  | Date: | 2/14/13 |
| Print Name: | Prasad Konka |  |  |
| Title: | Architect |  |  |
| Role: | Architect |  |  |

APPENDIX B: REFERENCES

The following table summarizes the documents referenced in this document.

|  |  |  |
| --- | --- | --- |
| **Document Name** | **Description** | **Location** |
| *caCORE\_RESTfulWrapper\_Scope* | *Scope document* | [*https://ncisvn.nci.nih.gov/svn/cacoresdk/trunk/RestGen/doc/caCORE\_RESTfulWrapper\_Scope.docx*](https://ncisvn.nci.nih.gov/svn/cacoresdk/trunk/RestGen/doc/caCORE_RESTfulWrapper_Scope.docx) |
| *caCORE\_RESTfulWrapper\_Requirements* | *Requirement document* | [*https://ncisvn.nci.nih.gov/svn/cacoresdk/trunk/RestGen/doc/caCORE\_RESTfulWrapper\_Requirements.doc*](https://ncisvn.nci.nih.gov/svn/cacoresdk/trunk/RestGen/doc/caCORE_RESTfulWrapper_Requirements.doc) |
| *RESTfulWrapper\_System\_Use\_Case* | *System use cases* | [*https://ncisvn.nci.nih.gov/svn/cacoresdk/trunk/RestGen/doc/RESTfulWrapper\_System\_Use\_Case.doc*](https://ncisvn.nci.nih.gov/svn/cacoresdk/trunk/RestGen/doc/RESTfulWrapper_System_Use_Case.doc) |

APPENDIX C: KEY TERMS

The following table provides definitions and explanations for terms and acronyms relevant to the content presented within this document.

|  |  |
| --- | --- |
| **Term** | **Definition** |
| JAXB | Java Architecture for XML Binding |
| EJB | Enterprise Java Beans |
| POJO | Plain Old Java Object |
| WSDL | Web Service Definition Language |
| SOAP | Simple Object Access Protocol |
|  |  |
|  |  |
|  |  |

APPENDIX D: Mapping Schema

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8" standalone="yes"?>  <xs:schema version="1.0" xmlns:xs="http://www.w3.org/2001/XMLSchema">  <xs:element name="Component" type="Component"/>  <xs:element name="Implementation" type="Implementation"/>  <xs:element name="Input" type="Input"/>  <xs:element name="Link" type="Link"/>  <xs:element name="Mapping" type="Mapping"/>  <xs:element name="Method" type="Method"/>  <xs:element name="Operation" type="Operation"/>  <xs:element name="Options" type="Options"/>  <xs:element name="Output" type="Output"/>  <xs:element name="Resource" type="Resource"/>  <xs:element name="Source" type="Source"/>  <xs:element name="Target" type="Target"/>  <xs:complexType name="Mapping">  <xs:sequence>  <xs:element name="components" minOccurs="0">  <xs:complexType>  <xs:sequence>  <xs:element ref="Component" minOccurs="0" maxOccurs="unbounded"/>  </xs:sequence>  </xs:complexType>  </xs:element>  <xs:element name="links" minOccurs="0">  <xs:complexType>  <xs:sequence>  <xs:element ref="Link" minOccurs="0" maxOccurs="unbounded"/>  </xs:sequence>  </xs:complexType>  </xs:element>  <xs:element ref="Options"/>  <xs:element name="resources" minOccurs="0">  <xs:complexType>  <xs:sequence>  <xs:element ref="Resource" minOccurs="0" maxOccurs="unbounded"/>  </xs:sequence>  </xs:complexType>  </xs:element>  </xs:sequence>  <xs:attribute name="name" type="xs:string"/>  <xs:attribute name="version" type="xs:string"/>  <xs:attribute name="description" type="xs:string"/>  <xs:attribute name="createdOn" type="xs:dateTime"/>  <xs:attribute name="lastUpdatedOn" type="xs:dateTime"/>  </xs:complexType>  <xs:complexType name="Component">  <xs:sequence/>  <xs:attribute name="id" type="xs:string"/>  <xs:attribute name="location" type="xs:string"/>  <xs:attribute name="type" type="xs:string"/>  <xs:attribute name="kind" type="xs:string"/>  </xs:complexType>  <xs:complexType name="Link">  <xs:sequence>  <xs:element ref="Source"/>  <xs:element ref="Target"/>  </xs:sequence>  </xs:complexType>  <xs:complexType name="Source">  <xs:sequence/>  <xs:attribute name="id" type="xs:string"/>  <xs:attribute name="componentId" type="xs:string"/>  </xs:complexType>  <xs:complexType name="Target">  <xs:sequence/>  <xs:attribute name="id" type="xs:string"/>  <xs:attribute name="componentId" type="xs:string"/>  </xs:complexType>  <xs:complexType name="Options">  <xs:sequence/>  <xs:attribute name="outputPath" type="xs:string"/>  <xs:attribute name="rootPath" type="xs:string"/>  <xs:attribute name="wrapperType" type="xs:string"/>  <xs:attribute name="wsdlLocation" type="xs:string"/>  <xs:attribute name="ejbLocation" type="xs:string"/>  <xs:attribute name="wsdlBindingFile" type="xs:string"/>  </xs:complexType>  <xs:complexType name="Resource">  <xs:sequence>  <xs:element name="methods" minOccurs="0">  <xs:complexType>  <xs:sequence>  <xs:element ref="Method" minOccurs="0" maxOccurs="unbounded"/>  </xs:sequence>  </xs:complexType>  </xs:element>  </xs:sequence>  <xs:attribute name="name" type="xs:string"/>  <xs:attribute name="path" type="xs:string"/>  <xs:attribute name="pojoLocation" type="xs:string"/>  </xs:complexType>  <xs:complexType name="Method">  <xs:sequence>  <xs:element ref="Implementation"/>  </xs:sequence>  <xs:attribute name="name" type="xs:string"/>  <xs:attribute name="pathName" type="xs:string"/>  </xs:complexType>  <xs:complexType name="Implementation">  <xs:sequence>  <xs:element ref="Operation"/>  </xs:sequence>  <xs:attribute name="type" type="xs:string"/>  <xs:attribute name="name" type="xs:string"/>  <xs:attribute name="clientType" type="xs:string"/>  <xs:attribute name="path" type="xs:string"/>  <xs:attribute name="portName" type="xs:string"/>  <xs:attribute name="jndiProperties" type="xs:string"/>  <xs:attribute name="jndiName" type="xs:string"/>  </xs:complexType>  <xs:complexType name="Operation">  <xs:sequence>  <xs:element name="inputs" minOccurs="0">  <xs:complexType>  <xs:sequence>  <xs:element name="inputs" type="Input" nillable="true" minOccurs="0" maxOccurs="unbounded"/>  </xs:sequence>  </xs:complexType>  </xs:element>  <xs:element ref="Output"/>  </xs:sequence>  <xs:attribute name="name" type="xs:string"/>  <xs:attribute name="style" type="xs:string"/>  </xs:complexType>  <xs:complexType name="Input">  <xs:sequence/>  <xs:attribute name="name" type="xs:string"/>  <xs:attribute name="type" type="xs:string"/>  </xs:complexType>  <xs:complexType name="Output">  <xs:sequence/>  <xs:attribute name="name" type="xs:string"/>  <xs:attribute name="type" type="xs:string"/>  </xs:complexType>  </xs:schema> |