EIS GROUP™

Usage and Customization Guide
OpenL Tablets Web Services
Release 5.15

Document number: TP_OpenL_WebServices_UCG_2.1_LSh

Revised: 08-12-2015

Table of Contents

1 Preface 4

1.1 Audience 4
1.2 How This Book Is Organized 4
1.3 Related Information 4
1.4 Typographic Conventions 5

2 Introduction 6

3 OpenL Tablets Web Services Configuration 8

3.1 Configuration Points 9
Data Source Configuration 9
System Settings 14
Configure a Number of Threads to Rules Compilation 15
Logging Requests to Web Services and Their Responds 15
REST Services Settings 15
Aegis Databinding Configuration 15
Instantiation Strategy Configuration 16

4 OpenL Tablets Web Services Customization 17

Customization Points 17
Service Configurer 18
Dynamic Interface Support 20
Interface Customization through Annotations 21
JAR File Data Source 23
Data Source Listeners 23
Variations 23

5 Appendix A: Tips and Tricks 27

5.1 How to Use OpenL Tablets Webservices from Java Code 27
5.2 How to Use OpenL Tablets REST Services from Java Code 28

1 Preface

OpenL Tablets is a Business Rules Management System (BRMS) based on the tables presented in Excel and Word documents. Using unique concepts, OpenL Tablets facilitates treating business documents containing business logic specifications as executable source code.

OpenL Tablets provides a set of tools addressing BRMS related capabilities including *OpenL Tablets Web Services application* designed for integration of business rules into different customer's applications.

The goal of this document is to explain how to configure OpenL Tablets Web Services for different working environments and how to customize the services to meet particular customer requirements.

- The following topics are included:
- Audience
- How This Book Is Organized
- Related Information
- Typographic Conventions

1.1 Audience

This guide is targeted at rule developers who set up, configure and customize OpenL Tablets Web Services to facilitate the needs of customer rules management applications.

Basic knowledge of Java, Apache Tomcat, Ant, and Excel is required to use this guide effectively.

1.2 How This Book Is Organized

Information on how to use this guide	
Section	**Description**
Introduction	Provides overall information about OpenL Tablets Web Services application.
OpenL Tablets Web Services Configuration	Describes the configuration of OpenL Tablets Web Services for different environments.
OpenL Tablets Web Services Customization	Explains how to customize OpenL Web Services to meet customers' needs and requirements.
Appendix A: Tips and Tricks	Describes how to use OpenL Tablets web services from Java code.

1.3 Related Information

The following table lists sources of information related to contents of this guide:

Related information	
Title	**Description**
OpenL Tablets WebStudio User Guide	Describes OpenL Web Studio, a web application for managing OpenL Tablets projects through web browser.
OpenL Tablets Reference Guide	Provides overview of OpenL Tablets technology, as well as its basic concepts and principles.
OpenL Tablets Installation Guide	Describes how to install and set up OpenL Tablets software.
http://openl-tablets.sourceforge.net/	OpenL Tablets open source project website.

1.4 Typographic Conventions

The following styles and conventions are used in this guide:

Typographic styles and conventions	
Convention	**Description**
Bold	• Represents user interface items such as check boxes, command buttons, dialog boxes, drop-down list values, field names, menu commands, menus, option buttons, perspectives, tabs, tooltip labels, tree elements, views, and windows. • Represents keys, such as **F9** or **CTRL+A.** • Represents a term the first time it is defined.
`Courier`	Represents file and directory names, code, system messages, and command-line commands.
`Courier Bold`	Represents emphasized text in code.
Select **File > Save As**	Represents a command to perform, such as opening the **File** menu and selecting **Save As.**
Italic	• Represents any information to be entered in a field. • Represents documentation titles.
< >	Represents placeholder values to be substituted with user specific values.
Hyperlink	Represents a hyperlink. Clicking a hyperlink displays the information topic or external source.
[name of guide]	Reference to another guide that contains additional information on a specific feature.

2 Introduction

Many OpenL Tablets rule management solutions need to expose business rules as Web Services. Each solution usually has a unique structure of the rules and implies a unique structure of Web Services. To meet requirements of a variety of customer project implementations, OpenL Tablets Web Services application provides the ability to dynamically create web services for customer rules and also offers extensive configuration and customization capabilities.

Overall architecture of OpenL Tablets Web Services frontend shown in the *Figure 1* is expandable and customizable. All the functionality is divided into pieces; each of them is responsible for a small part of functionality and could be replaced by another implementation.

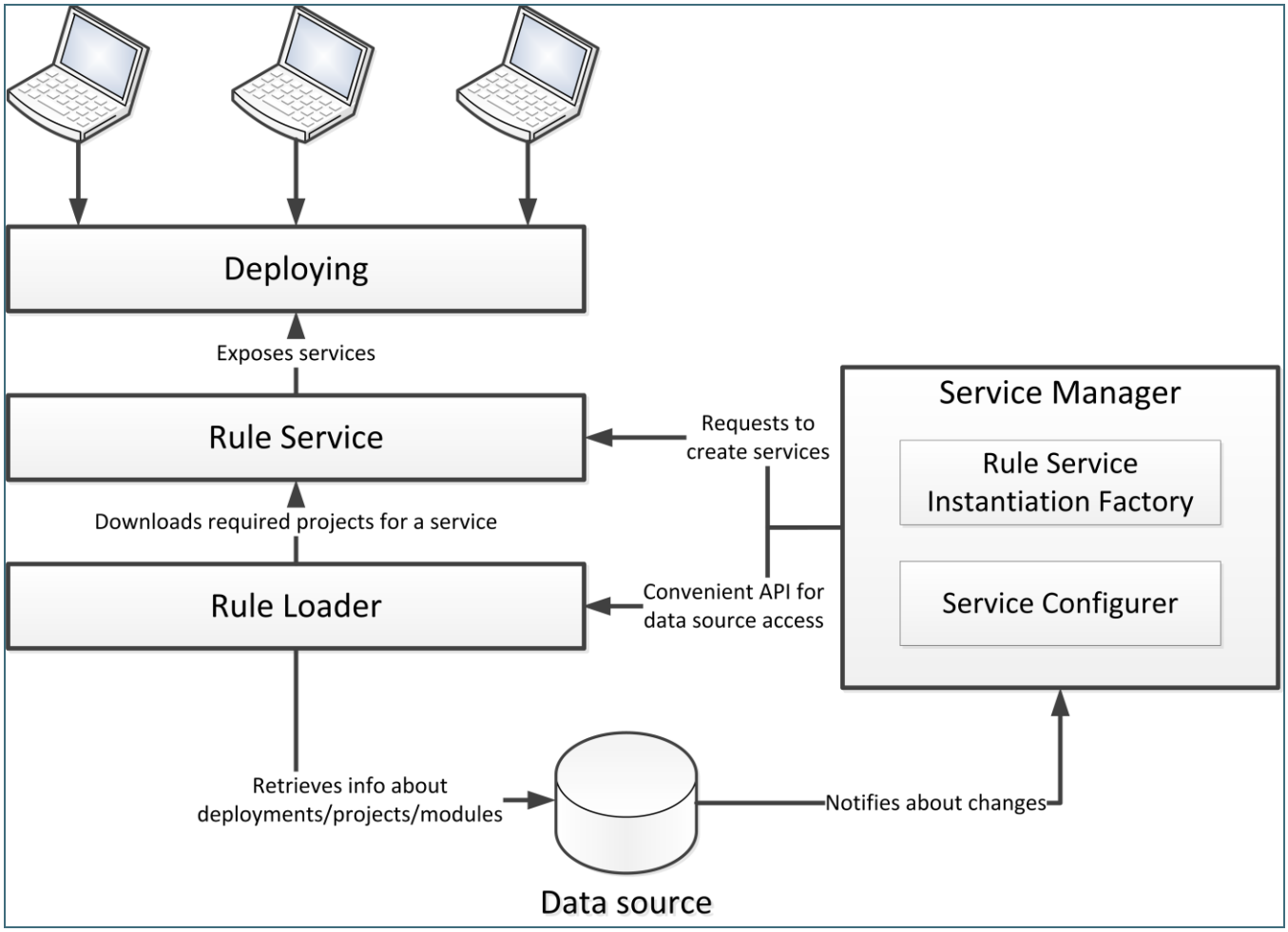

Figure 1: Overall OpenL Tablets Web Services architecture

OpenL Tablets Web Services application provides the following key features and benefits:

- Ability to easily integrate customer business rules into various applications running on different platforms.
- Ability to use different data sources such as a central OpenL Tablets Production Repository, file system of a proper structure, etc.

- Ability to expose multiple projects/modules as a single web service according to a project logical structure.

The subsequent chapters describe how to set up Data Source, Service Configurer, and Service Exposing Method. All the components to be configured and customized are shown in the *Figure 2*.

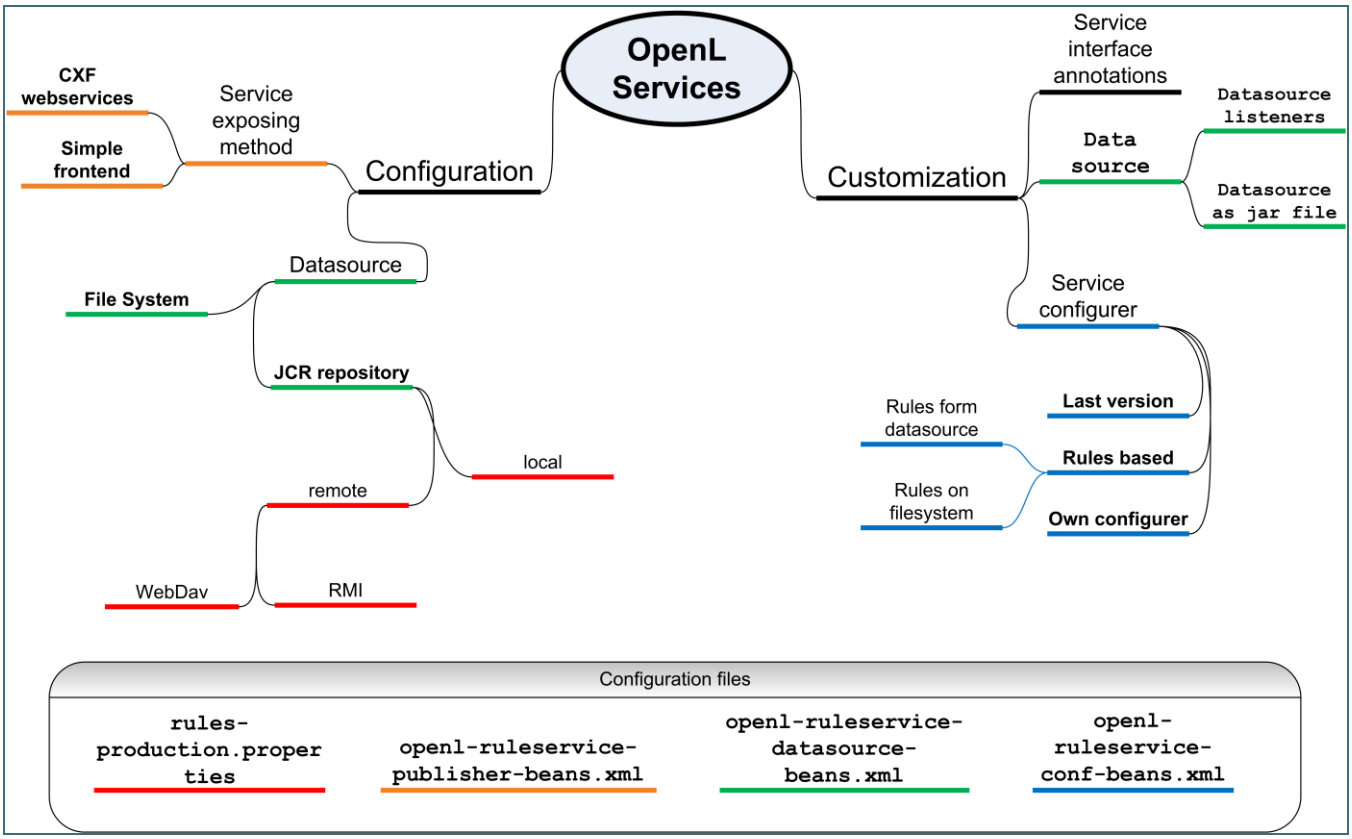

Figure 2: Configurable and customizable components of OpenL Tablets Web Services application

3 OpenL Tablets Web Services Configuration

OpenL Tablets Web Services application architecture provides the possibility to extend the mechanisms of the services loading and deployment according to the particular project requirements.

All OpenL Tablets Web Services configurations are specified in Spring configuration files with several `.properties` files. By default OpenL Tablets Web Services application is configured as described in the next paragraph.

Data source is configured as `FileSystemDataSource` located in the "/openl/datasource" folder. All services will be exposed using CXF framework inside the OpenL Web Services war file that you can download from http://openl-tablets.sourceforge.net/downloads. (All calls will be processed by CXF servlet.) `LastVersionProjectsServiceConfigurer` is used as a default service configurer. It takes the last version of each deployment and creates the service for each project using all modules contained in the project. All services will be exposed without services class and all methods of the service will be enhanced by runtime context.

If required, you can change the Web Services configuration by overriding the existing configuration files. All overridden beans should be located in the `openl-ruleservice-override-beans.xml` file. The list below provides default OpenL Tablets Web Services configuration files:

Default OpenL Tablets Web Services configuration files	
File	**Description**
`openl-ruleservice-beans.xml`	Main configuration file that includes all other configuration files. This file is searched by OpenL Tablets Web Services in the classpath root.
`openl-ruleservice-datasource-beans.xml`	Contains data source configuration.
`openl-ruleservice-loader-beans.xml`	Contains loader configuration.
`openl-ruleservice-publisher-beans.xml`	Contains common publisher configuration.
`openl-ruleservice-webservice-publisher-beans.xml`	Contains publisher configuration for webservices (SOAP).
`openl-ruleservice-jaxrs-publisher-beans.xml`	Contains publisher configuration for RESTful services.
`openl-ruleservice-conf-beans.xml`	Contains service configurer.
`openl-ruleservice.properties`	The main file containing properties for OpenL Tablets Web Services configuration.
`project-resolver-beans.xml`	Configuration for OpenL Tablets project resolving. The beans for reading rules from the data source specified in the loader.
`rules-production.properties`	Configuration file for JCR based data source. If a different type of data source is used, this file is ignored.

The Service Manager is the main component of OpenL Tablets Services Frontend containing all major parts, such as a loader, a ruleservice, and a Service Configurer. For more information, see [OpenL Tablets Developer Guide***]***. It knows all currently running services and intelligently controls all operations for deploying, undeploying, and redeploying the services. These operations are only performed in two cases:

- the initial deployment at the start of OpenL Tablets Services Frontend

- processing after making changes in the data source .
  The Service Manager is always a data source listener as described below.
- Detailed information about all configuration files is provided in the following sections: Configuration Points

3.1 Configuration Points

Any part of OpenL Tablets Services Frontend can be replaced by the user's own implementation. For information about the system architecture, see ***[***OpenL Tablets Developer Guide***].*** If the common approach is used, the following components must be configured:

Configuration components	
Component	**Description**
Data Source	Informs the OpenL Tablets system where to retrieve user's rules.
Service Exposing Method	Defines the way services are exposed. It can be a web service, any framework, for example, a simple Java framework, etc.

The following sections provide detailed information on how to configure these components:

- Data Source Configuration
- System Settings
- Configure a Number of Threads to Rules Compilation
- Logging Requests to Web Services and Their Responds
- Instantiation Strategy Configuration

Data Source Configuration

The system supports the following data source implementations as described in the following sections:

- JCR Repository
- File System Data Source
- Service Exposing Method

JCR Repository

To use JCR repository as a data source, set the "`jcr`" value to `ruleservice.datasource.type` property in the `openl-ruleservice.properties` file which is by default located in the `<TOMCAT_HOME>\webapps\<web services file name>\WEB-INF\classes` directory, and all JCR repository settings must be specified in the `rules-production.properties` file located in the same directory.

The main property in JCR repository settings is `production-repository.factory` that defines the repository access type in one of the following ways:

- **Local repository**
  This repository is located on the user's local machine as a folder.
  The repository factory must be as follows:
  ```
  production-repository.factory =
  org.openl.rules.repository.factories.LocalJackrabbitProductionRepositoryFactory
  ```
  Additional property that defines location of the JCR repository:

```
production-repository.local.home = /<OPENL_HOME>/production-repository
```

Note: This is the default setting used in OpenL Tablets, so there is no need to edit the `rules-production.properties` file.

Note: Only one application can use a local repository. That is, OpenL Tablets Web Services and OpenL Tablets WebStudio cannot be used with a local repository at the same time. If multiple applications need to access a repository, remote access to the repository must be provided for all applications.

- **Remote repository**

  This repository is located on a remote server. It is recommended to download all sources related to JCR repository from `http://openl-tablets.sourceforge.net/downloads` page, link to the Repository ZIP file.Repository package is a ZIP file containing `repository server, config files and an empty JCR repository`. Then copy `openl-tablets-remote-repository-server-X.X.X.war` file from the repository package into the `\<TOMCAT_HOME>\webapps` folder.

  Note: Remember that the Jackrabbit `war` file must be run before the OpenL Tablets Web Services `war` file. Tomcat runs `war` files alphabetically.

  Remote repository can be accessed by the following protocols:

Protocols for accessing remote repository	
Protocol	**Description**
RMI	To set up access to the repository, copy the `jackrabbit` folder from the downloaded repository package into `\<TOMCAT_HOME>\bin\. The jackrabbit` folder includes two files. The `bootstrap.properties` file contains settings indicating where the repository is located, and the URL which must be used for remote access as follows: • `repository.home={the folder where user's production repository is located}` • `rmi.url={URL for remote access to the repository}`, for example, `//localhost:1099/production-repository` The repository factory must be as follows: production-repository.factory = org.openl.rules.repository.factories.RmiJackrabbitProductionRepositoryFactory Additional property that defines the remote repository location: `production-repository.remote.rmi.url = //localhost:1099/production-repository`

WebDav	The repository factory must be as follows:
	`production-repository.factory = org.openl.rules.repository.factories.WebDavJackrabbitProductionRepositoryFactory`
	Additional property that defines the remote repository location:
	`production-repository.remote.webdav.url = http://localhost:8080/production-repository`

Security: If a secured remote JCR repository is used, define login and password and secret key in the `rules-production.properties` file.

Attention! A problem can arise if one instance of Tomcat is used for both web archives: jackrabbit-webapp and the OpenL Tablets Web Services `war` file. Tomcat will hang during the startup because Web Services application tries to connect to the DataSource on startup. Trying to connect to the DataSource in the "JCR remote using WebDav" case means that there are connections by the datasource URL. But Tomcat applies such connections and waits for the end of deployment of all web applications. As a result this causes a deadlock since the Web Services application tries to connect to another application, and this one cannot respond before the Web Services application is not deployed.

To resolve the issue, use one of the possible solutions:

1. Use several Tomcat instances: one of them contains Jackrabbit-webapp, and the other contains the OpenL Tablets Web Services application.
2. Use another Application Server which supports access to web applications that have already been deployed before all the other web applications started. For example, WebSphere.

File System Data Source

Using a file system as a data source for a user's projects means that the projects are placed into a local folder. This folder represents a single deployment containing all the projects.

Note: This type of data source does not support deployments and versioning by default. If you want to enable deployments support, you have to set `ruleservice.datasource.filesystem.supportDeployments` property to true. If you want to enable versioning support for deployment, you have set `ruleservice.datasource.filesystem.supportVersion` property to true.

To configure a local file system as a data source, set the `ruleservice.datasource.type` property in the `openl-ruleservice.properties` file which is by default located in the `<TOMCAT_HOME>\webapps\<web services file name>\WEB-INF\classes` directory, to "`local`".

Users can also pack their rule projects to a `jar` file and use this file as a data source. For more information, see JAR File Data Source.

File sys

Note: By default, data source is configured to the file system data source.

Attention! For proper parsing of Java properties file, the path to the folder must be defined with a slash ('/') as the folders delimiter. Back slash "\" is not allowed.

Service Exposing Method

Service Exposing Method specifies the method to expose user's OpenL Tablets Services.

Common flow of service exposing is as follows:

1. Retrieve service descriptions that must be deployed from the service configurer.
2. Undeploy currently running services that are not in services defined by the service configurer.

   Some services can become unnecessary in the new version of the product.
3. Redeploy currently running services that are still in services defined by the service configure, i.e. service update.
4. Deploy new services that were not represented earlier.

To set the method of exposing services, specify a Spring bean with the `ruleServicePublisher` name in the `openl-ruleservice-publisher-beans.xml`, by default, configuration file. Users can implement their own publisher using a framework of their choice; or users can use one of the following predefined implementations of `RuleServicePublisher`:

1. **CXF Web Services:** Publisher that exposes user's services as Web Services using CXF framework. Configuration is as follows:

```
<bean id="webServicesDataBinding"
class="org.springframework.beans.factory.config.MethodInvokingFactoryBean" scope="prototype">
		<property name="targetObject" ref="aegisDatabindingFactoryBean"/>
		<property name="targetMethod" value="createAegisDatabinding"/>
		<property name="singleton" value="false"/>
	</bean>

	<!-- <bean id="dataBinding" class="org.apache.cxf.jaxb.JAXBDataBinding"
		scope="prototype"> <property name="contextProperties"> <map> <entry> <key>
		<util:constant static-
field="com.sun.xml.bind.api.JAXBRIContext.ANNOTATION_READER"
		/> </key> <bean class="org.jvnet.annox.xml.bind.AnnoxAnnotationReader" />
		</entry> </map> </property> </bean> -->


	<!-- Main description for the one WebService -->
	<!-- All configurations for server (like a data binding type and interceptors)
		are represented there. ServerFactoryBean configuration is similar to a CXF
		simple frontend configuration(see http://cxf.apache.org/docs/simple-frontend-
configuration.html)
		but without namespace "simple". -->

	<bean id="webServicesLoggingFeature"
class="org.openl.rules.ruleservice.logging.LoggingFeature">
		<property name="loggingEnabled" value="${ruleservice.logging.enabled}" />
	</bean>

	<bean id="webServicesServerPrototype"
class="org.apache.cxf.jaxws.JaxWsServerFactoryBean"
		scope="prototype">
		<property name="dataBinding" ref="webServicesDataBinding" />
		<property name="features">
			<list>
				<!-- Comment/Uncomment following block for use/unuse logging
feature.
					It can increase performance if logging isn't used. -->
				<ref bean="webServicesLoggingFeature" />
			</list>
		</property>
	</bean>

	<!-- Prototypes factory. It will create new server prototype for each new
		WebService. -->
	<bean id="webServicesServerPrototypeFactory"

	class="org.springframework.beans.factory.config.ObjectFactoryCreatingFactoryBean">
		<property name="targetBeanName">
			<idref bean="webServicesServerPrototype" />
		</property>
	</bean>

	<!-- Initializes OpenL Engine instances according to web services configuration
		description and calls DeploymentAdmin to expose corresponding web service -->
	<!-- Exposes web services. -->
	<bean id="webServiceRuleServicePublisher"
		class="org.openl.rules.ruleservice.publish.JAXWSRuleServicePublisher">
		<property name="serverFactory" ref="webServicesServerPrototypeFactory" />
		<property name="baseAddress" value="${ruleservice.baseAddress}" />
	</bean>
```

Note: When using Openl Tablets Web Services war application, the base address must be relational. The full web service address is as follows: `webserver_context_path/ws_app_war_name/address_specified_by_you.`

Publisher that exposes user's services as Restful Services using CXF framework. Configuration is as follows:

```
	<bean id="JAXRSJacksonDatabindingFactoryBean"
		class="org.openl.rules.ruleservice.databinding.JacksonObjectMapperFactoryBean"
		scope="prototype">
		<property name="enableDefaultTyping"
ref="JAXRSserviceDescriptionConfigurationEnableDefaultTypingFactoryBean" />
		<property name="overrideTypes"
ref="serviceDescriptionConfigurationRootClassNamesBindingFactoryBean"/>
		<property name="supportVariations"
ref="serviceDescriptionConfigurationSupportVariationsFactoryBean" />
	</bean>

	<bean id="JAXRSJacksonServicesDataBinding"
class="org.springframework.beans.factory.config.MethodInvokingFactoryBean" scope="prototype">
		<property name="targetObject" ref="JAXRSJacksonDatabindingFactoryBean"/>
		<property name="targetMethod" value="createJacksonDatabinding"/>
		<property name="singleton" value="false"/>
	</bean>

	<bean id="JAXRSJSONProvider"
class="com.fasterxml.jackson.jaxrs.json.JacksonJsonProvider" scope="prototype">
		<constructor-arg ref="JAXRSJacksonServicesDataBinding"/>
	</bean>

	<bean id="JAXRSAegisElementProvider"
class="org.apache.cxf.jaxrs.provider.aegis.AegisElementProvider" scope="prototype"/>

	<bean id="JAXRSWebApplicationExceptionMapper"
class="org.apache.cxf.jaxrs.impl.WebApplicationExceptionMapper">
		<property name="addMessageToResponse" value="true"/>
	</bean>

	<bean id="JAXRS200StatusOutInterceptor"
class="org.openl.rules.ruleservice.publish.jaxrs.JAXRS200StatusOutInterceptor">
		<property name="enabled" value="${ruleservice.jaxrs.responseStatusAlwaysOK}"/>
	</bean>

	<bean id="JAXRSServicesServerPrototype"
class="org.apache.cxf.jaxrs.JAXRSServerFactoryBean"
		scope="prototype">
		<property name="dataBinding" ref="JAXRSServicesDataBinding" />
		<property name="features">
			<list>
				<ref bean="JAXRSServicesLoggingFeature" />
			</list>
		</property>
		<property name="outFaultInterceptors">
			<list>
				<ref bean="JAXRS200StatusOutInterceptor"/>
			</list>
		</property>
		<property name="outInterceptors">
			<list>
				<ref bean="JAXRS200StatusOutInterceptor"/>
			</list>
		</property>
		<property name="providers">
			<list>
				<ref bean="JAXRSJSONProvider"/>
```

```
                        <ref bean="JAXRSAegisElementProvider"/>
                        <ref bean="JAXRSWebApplicationExceptionMapper"/>
                    </list>
                </property>
        </bean>

        <!-- Prototypes factory. It will create new server prototype for each new
                WebService. -->
        <bean id="JAXRSServicesServerPrototypeFactory"

        class="org.springframework.beans.factory.config.ObjectFactoryCreatingFactoryBean">
                <property name="targetBeanName">
                        <idref bean="JAXRSServicesServerPrototype" />
                </property>
        </bean>

        <!-- Initializes OpenL Engine instances according to web services configuration
                description and calls RuleServicePublisher to expose corresponding web service -
->
        <!-- Exposes web services. -->
        <bean id="JAXRSServicesRuleServicePublisher"
                class="org.openl.rules.ruleservice.publish.JAXRSRuleServicePublisher">
                <property name="serverFactory" ref="JAXRSServicesServerPrototypeFactory" />
                <property name="baseAddress" value="${ruleservice.baseAddress}" />
        </bean>
```

2. **Simple Java Frontend**

   Customization is as follows:

```
<!-- Simple front end to access all services. -->
<bean id="frontend" class="org.openl.rules.ruleservice.simple.RulesFrontendImpl"/>

<!-- Initializes OpenL Engine instances according to web services configuration
        description and calls DeploymentAdmin to expose corresponding web service. -->
<bean id="ruleServicePublisher"
class="org.openl.rules.ruleservice.simple.JavaClassRuleServicePublisher">
        <property name="frontend" ref="frontend"/>
</bean>
```

System Settings

There are several options extending rules behavior in OpenL Tablets:

- Custom Spreadsheet Type
- Rules Dispatching Mode

These settings can be defined as JVM options for Tomcat launch. Such JVM options affect all web applications inside the same Tomcat instance that can cause a problem when different applications work with different settings. To avoid these problems, a special configuration file was introduced where all that settings can be defined. This configuration file has a higher priority than JVM options, so in case a particular setting is set both as a JVM option and in configuration file, the value defined in the configuration file will be used for rules compilation.

System settings are defined in the `system.properties` configuration file which is registered in ruleServiceInstantiationFactory from `openl-ruleservice-core-beans.xml`:

```
                <property name="externalParameters">
                        <bean
class="org.springframework.beans.factory.config.PropertiesFactoryBean">
```

```
                <property name="location" value="classpath:system.properties"/>
            </bean>
        </property>
```

Configure a Number of Threads to Rules Compilation

The system supports parallel rules compilation. A rules compilation consumes a large amount of memory. So, if the system tries to compile too many rules at once, it fails with an out of memory exception.

The setting that limits the amount of threads to compile rules is defined in the `openl-ruleservice.properties` file located in the `<TOMCAT_HOME>\webapps\<web services file name>\WEB-INF\classes` directory.

By default, only three threads can compile rules in parallel, i.e. `ruleservice.instantiation.strategy.maxthreadsforcompile = 3`.

For example, if you want to permit only one thread to compile rules, set value to one, i.e. `ruleservice.instantiation.strategy.maxthreadsforcompile = 1`.

Logging Requests to Web Services and Their Responds

The system provides an ability to log all requests to OpenL Web Services and their responds.

The setting is defined in the `openl-ruleservice.properties` file located in the `<TOMCAT_HOME>\webapps\<web services file name>\WEB-INF\classes` directory.

By default, the logging is disabled, i.e. `ruleservice.logging.enabled = false`.

To enable the logging, set `ruleservice.logging.enabled = true`.

REST Services Settings

The system provides an ability to use HTTP 200 Status for all RESTful services requests.

The setting is defined in the `openl-ruleservice.properties` file located in the `<TOMCAT_HOME>\webapps\<web services file name>\WEB-INF\classes` directory.

By default, this feature is , i.e. `ruleservice.jaxrs.responseStatusAlwaysOK = false`.

To enable this feature, set `ruleservice.jaxrs.responseStatusAlwaysOK = true`.

Aegis Databinding Configuration

The system provides an ability to configure aegis databinding settings.

Please, refer to CXF Aegis Databinding documentation for additional information about Aegis Databinding.

The setting is defined in the `openl-ruleservice.properties` file located in the `<TOMCAT_HOME>\webapps\<web services file name>\WEB-INF\classes` directory.

```
The default settings are:
ruleservice.aegisbinding.readXsiTypes = true
ruleservice.aegisbinding.writeXsiTypes = true
ruleservice.aegisbinding.ignoreNamespaces = false
```

Instantiation Strategy Configuration

The system provides an ability to choose an instantiation strategy.

The setting is defined in the `openl-ruleservice.properties` file located in the `<TOMCAT_HOME>\webapps\<web services file name>\WEB-INF\classes` directory.

By default, the lazy initialization strategy is enabled, i.e. `ruleservice.instantiation.strategy.lazy = true`. Modules compile on the first request and can be unloaded in the future for memory save.

To disable the lazy initialization strategy, set `ruleservice.instantiation.strategy.lazy = false`. All modules compile on the application launch.

4 OpenL Tablets Web Services Customization

If a project has specific requirements, developers must create a new maven project that extends OpenL Tablets Web Services and add or change required points of configuration. Add the following dependency to the `pom.xml` file with the version used in the project specified:

```
<dependency>
	<groupId>org.openl.rules</groupId>
	<artifactId>org.openl.rules.ruleservice.ws</artifactId>
	<version>5.10.0</version>
	<type>war</type>
	<scope>runtime</scope>
</dependency>
```

Use the following maven plugin to control the Web Application building with user's custom configurations and classes:

```
<plugin>
	<groupId>org.apache.maven.plugins</groupId>
	<artifactId>maven-war-plugin</artifactId>
	<configuration>
		<warSourceDirectory>webapps/ws</warSourceDirectory>
		<!—Define war name here-->
		<warName>${war.name}-${project.version}</warName>
		<packagingExcludes>
		<!—Exclude unnecessary libraries from parent project here-->
			WEB-INF/lib/org.openl.rules.ruleservice.ws.lib-*.jar
		</packagingExcludes>
		<!—Define paths for resources. Developer has to create a file with the same name
to overload existing file in the parent project-->
		<webResources>
			<resource>
				<directory>src/main/resources</directory>
			</resource>
			<resource>
				<directory>war-specific-conf</directory>
			</resource>
		</webResources>
	</configuration>
</plugin>
```

Customization Points

Customization points are described in the following table:

Customization points	
Point	**Description**
Service Configurer	Defines all services to be exposed and modules contained in each service.
Multimodule with Customized Dispatching	Provides a possibility to handle dispatching between modules.
Dynamic Interface Support	Generates an interface for services at runtime.
Interface customization through annotations	
JAR File Data Source:	Pack user's rule projects into a `jar` file and places the archive in the

Customization points	
Point	**Description**
	classpath.
Data Source Listeners	
Variations:	Additional calculation of the same rule with a slight modification in its arguments.

The following topics are included:

- Service Configurer
- Multimodule with Customized Dispatching
- Dynamic Interface Support
- Interface Customization through Annotations
- JAR File Data Source
- Data Source Listeners
- Variations

Service Configurer

The Service Configurer component defines all services to be exposed: modules contained in each service, the service interface, provide or not runtime context, and so on.

Modules for a service can be retrieved for different projects. Each deployment contained in a data source has a set of properties and can be represented in several versions. Deployment consists of projects that also have properties and contains some modules. There can also be only one version of a specific project in the deployment.

So each module for service can be identified by the deployment name, deployment version, the project name inside the deployment, and the module name inside the module.

Users can implement different module gathering strategies according to their needs. Users can choose deployments and projects with concrete values of a specific property, such as service for some LOB, service containing modules with an expiration date before a specific date, etc., or by using versions of deployments, or use both of these approaches.

OpenL Tablets users often want to create web services containing several rule projects or modules. Users have a possibility to unite multiple modules in one service using a simple service description. Service description contains all information about the desirable service: the service name, URL, all modules that form the service, the service class, and can be expanded to contain new configurations. To instantiate several modules, users can relay on the OpenL MultiModule mechanism that combines a group of modules as a single rules engine instance.

By default, OpenL Webservices uses `LastVersionProjectsServiceConfigurer` which deploys last version projects from deployments. This implementation uses service configuration `rules-deploy.xml` file from the project root folder. This file can be created via OpenL Tablets Webstudio or created manually. See the following example of `rules-deploy.xml` file:

```
<rules-deploy>
	<isProvideRuntimeContext>true</isProvideRuntimeContext>
	<isProvideVariations>false</isProvideVariations>
	<serviceName>myService</serviceName>
```

```
        <serviceClass>com.example.MyService </serviceClass>
        <url>com.example.MyService</url>
        <publishers>
                <publisher>RESTFUL</publisher>
                <publisher>WEBSERVICE</publisher>
        </publishers>
        <configuration>
                <entry>
                        <string>someString</string>
                        <string>someString</string>
                </entry>
        </configuration>
</rules-deploy>
```

Project configuration		
Tag	**Description**	**Required**
isProvideRuntimeContext	Set to `true` if a project provides a runtime context, otherwise `false`. Default value is defined in the `openl-ruleservice.properties` file.	No
isProvideVariations	Set to `true` if a project provides variations, otherwise `false`. Default value is defined in the `openl-ruleservice.properties` file.	No
serviceName	Defines a service name.	No
serviceClass	Defines a service class. If not defined, generated class is used.	No
url	Defines URL for service.	No
publishers	Define a list of publishers for project.	No
configuration	Extension point for custom service configure.	No

The following topics are included:

- Service Description
- Description of the RulesBasedConfigurerTemplate

Service Description

Commonly, each service is represented by rules and the service interface, and consists of:

Service description		
#	**Service**	**Description**
1	Service name	Unique service identifier.
2	Service URL	URL path for the service. It is (absolute for the console start and relative to the context root for the `ws.war` case.
3	Service class	Interface of the service to be used at the server and the client side.
4	Rules	A module, or a set of modules to be combined together as a single rules module.
5	**Provide runtime context** flag	Indicates whether the runtime context must be added to all rule methods or not. If it is "`true`", `the IRulesRuntimeContext` argument must be added to each method in the service class.
6	**Support variations** flag (optional)	Indicates whether the current service supports variations or not. For more

Service description		
#	Service	Description
		information, see Variations.

Users can create their own implementation of Service configurer interface - `org.openl.rules.ruleservice.conf.ServiceConfigurer` - and register it as a Spring bean with the "`serviceConfigurer`" name, or users can use one of the following implementations provided by OpenL Tablets Web Services:

1. `org.openl.rules.ruleservice.conf.SimpleServiceConfigurer` – designed for usage with a data source having one deployment. Exposes deployment; creates service for one predefined project in this deployment.
2. `org.openl.rules.ruleservice.conf.LastVersionProjectsServiceConfigurer` – exposes deployments based on the last version; creates one service for each project in the deployment. Reads configuration for service deploy from `rules-deploy.xml` in project.
3. `org.openl.rules.ruleservice.conf.RulesBasedServiceConfigurer` – users can define all their modules as OpenL Tablets rules using the `RulesBasedConfigurerTemplate.xlsx` template located in the `org.openl.rules.ruleservice` project. If OpenL Tablets Web Services are downloaded as a `war` file, the project is located in the `WEB-INF\lib` folder. Otherwise, the project artifact is located within the Central Maven repository. The template is described in Description of the RulesBasedConfigurerTemplate. Users can use rules representing their Service Configurer, from different locations:
   - **From the file system**

     User's rules are located in a specific folder.

   Configuration is as follows:

```
<bean id="serviceConfigurer"
class="org.openl.rules.ruleservice.conf.FileSystemRulesBasedServiceConfigurerFactoryBean"><pr
operty name="folderLocationPath" value="./test-resources/"/>
<property name="moduleName" value="RulesBasedConfigurer"/>
</bean>
```

   - **From the Data Source**

     User's rules are located in a DataSource. In this case redeploy the rules for the service configurer in the DataSource. Configuration is as follows:

```
<!-- Determines the services that should be exposed using RulesLoader.  -->
<bean id="serviceConfigurer"
class="org.openl.rules.ruleservice.conf.FileSystemRulesBasedServiceConfigurerFactoryBean">
            <property name="ruleServiceLoader" value="ruleServiceLoader"/>
            <property name="deploymentName" value="your deployment name"/>
            <property name="projectName" value="your project name"/>
            <property name="moduleName" value="RulesBasedConfigurer"/>
      </bean>
```

Dynamic Interface Support

OpenL Tablets Web Services application supports interface generation for services at runtime. This feature calls Dynamic Interface Support. If static interface is not defined for a service, the system generates it. The system uses an algorithm that generates an interface with all methods defined in the module or, in case of multimodule, in the list of modules.

This feature is enabled by default. To use a dynamic interface, do not define a static interface for a service.

It is not a best practice to use all methods from the module in a generated interface, because one needs to be sure that all return types and method arguments in all methods can be transferred through network. Also, interface for web services must not contain the method designed for internal usage. For this purpose, the system provides a mechanism for filtering methods in modules. Methods can be configured by including or excluding them from the dynamic interface.

This configuration can be applied in projects in the `rules.xml` file. See the following example:

```
<project>

	<!-- Project name. -->
	<name>project-name</name>
	<!-- OpenL project includes one or more modules.  -->
	<modules>
		<module>
			<name>module-name</name>
			<type>API</type> -->
		<!--
			Rules root document. Usually excel file on file system.
		-->
			<rules-root path="rules/Calculation.xlsx"/>
			<method-filter>
				<includes>
					<value>.*determinePolicyPremium.*</value>
					<value>.*vehiclePremiumCalculation.*</value>
				</includes>
			</method-filter>
		</module>-->
	</modules>

	<!-- Project's classpath. -->
	<classpath>
		<entry path="."/>
		<entry path="target/classes"/>
		<entry path="lib"/>
	</classpath>
</project>
```

For filtering methods, define the `method-filter` tag in the `rules.xml` file. This tag contains `includes` and `excludes` tags.

If the `method-filter` tag is not defined in the `rules.xml`, the system generates a dynamic interface with all methods provided in the module or modules for multimodule. If the `includes` tag is defined for method filtering, the system uses the methods which names match a regular expression of defined patterns. If the `includes` tag is not defined, the system includes all methods. If the `excludes` tag is defined for method filtering, the system uses methods which method names do not match a regular expression for defined patterns. If the `excludes` tag is not defined, the system does not exclude the methods.

If OpenL Tablets Dynamic Interface feature is used, a client interface must also be generated dynamically at runtime. Apache CXF supports the dynamic client feature. For additional information, refer to http://cxf.apache.org/docs/dynamic-clients.html.

Interface Customization through Annotations

The following topics are included:

- Interceptors for Service Methods
- Annotation Customization for Dynamic Interfaces

Interceptors for Service Methods

Interceptors for service methods can be specified using the following annotations:

1. `@ServiceCallBeforeInterceptor`

   Method annotation to define before interceptors, array of interceptors must be registered in the annotation parameter. All interceptors must implement the `org.openl.rules.ruleservice.core.interceptors.ServiceMethodBeforeAdvice` interface.

2. `@ServiceCallAfterInterceptor`

   Method annotation to define after interceptors, array of interceptors must be registered in the annotation parameter.

   There are two types of after interceptors:

   - `After Returning` interceptor

     Intercepts the result of a successfully calculated method with a possibility of post processing of the return result, even result conversion to another type, then this type must be specified as the return type for the method in service class. `After Returning` interceptors must inherit `org.openl.rules.ruleservice.core.interceptors.AbstractServiceMethodAfterReturningAdvice`.

   - `After Throwing` interceptor

     Intercepts method that has thrown Exception with a possibility of post processing of an error and throwing another type of exception. `After Returning` interceptors must inherit `org.openl.rules.ruleservice.core.interceptors.AbstractServiceMethodAfterThrowingAdvice`

Annotation Customization for Dynamic Interfaces

Annotation customization can be used for dynamically generated interfaces. This feature is only supported for a project which contains the `rules-deploy.xml` file.

Add a tag `interceptingTemplateClassName` to `rules-deploy`. For example:

```
<rules-deploy>
       <isProvideRuntimeContext>true</isProvideRuntimeContext>
       <isProvideVariations>false</isProvideVariations>
       <serviceName>dynamic-interface-test3</serviceName>
       <interceptingTemplateClassName>org.openl.ruleservice.dynamicinterface.test.MyTemplateC
lass</interceptingTemplateClassName>
       <url></url>
  </rules-deploy>
```

Define a template interface with the annotated methods with a signature that is the same as in the generated dynamic interface. Approach supports replacing argument types in the method signature with types assignable from the generated interface.

For example, for the following methods in the generated dynamic interface:

```
void someMethod(IRulesRuntimeContext context, MyType myType);
void someMethod(IRulesRuntimeContext context, OtherType otherType);
```

Add annotation to the first method, just use this signature in the template interface.

```
@ServiceCallAfterInterceptor(value = { MyAfterAdvice.class })
```

```
void someMethod(IRulesRuntimeContext context, MyType myType);
```

If `MyType` is generated in the runtime class, use a type that is assignable from the `MyType` class. For example:

```
@ServiceCallAfterInterceptor(value = { MyAfterAdvice.class })
void someMethod(IRulesRuntimeContext context, @AnyType(".*MyType") Object myType);
```

Note that this example uses the `@AnyType` annotation. If this annotation is skipped, this template method is applied for both methods, because Object is assignable from both types `MyType` and `OtherType`.

`@AnyType` annotation value is a Java regular expression of a canonical class name. Use this annotation, if more details are needed to define a template method.

JAR File Data Source

The system enables to pack rule projects and the `rules.xml` project descriptor into a JAR file and place the archive in the classpath. Put the JAR file with the project to the `\<TOMCAT_HOME>\webapps\<web services file name>\WEB-INF\lib`.

Add the following bean to unpack the projects to the specified folder - it must be the folder used in FileSystemDataSource:

```
<bean id="unpackClasspathJarToDirectoryBean"
    class="org.openl.rules.ruleservice.loader.UnpackClasspathJarToDirectoryBean">
          <property name="destinationDirectory" value=".. path to the folder to unpack
    projects…"/>
</bean>
```

Data Source Listeners

Data source registers datasource listeners do not notify some components of OpenL Services Frontend about the modifications. There is only one event type for the production repository modification: new deployment added. Service manager is always a data source listener because it must handle all modifications in the data source. Users can add their own listener, implementing `org.openl.rules.ruleservice.loader.DataSourceListener`, for additional controlling of the data source modifications with the required behavior and register it in datasource.

Variations

In highly loaded applications performance of execution is a crucial point in development. There are many approaches to speed up the application. One of them is to calculate rules with variations. A variation means “additional calculation of the same rule with a slight modification in its arguments”. Variations are very useful when it is required to calculate a rule several times with similar arguments. The idea of this approach is to once calculate rules for particular arguments and then recalculate only the rules or steps that depend on the modified, by variation, fields in those arguments.

The following topics are included:

- How It Works
- Predefined Variations
- Variations Factory

- Variations from Rules
- Example

How It Works

For a rule that can be calculated with variations there must be two methods in a service class:

- original method with a corresponding rule signature
- method with injected variations

The method enhanced with variations has a signature similar to the original method. Add the argument of type `org.openl.rules..variation.VariationsPack` as the last argument; the return type must be generic `org.openl.rules..variation.VariationsResult<T>`, where `T` is the return type of the original method.
Warning: when using user's own service class instead of the one generated by default, the original method must be defined for each method with variations.

The `VariationsPack` class contains all desired variations to be calculated. The `VariationsResult<T>` class contains results of the original calculation, without any modifications of arguments, and all calculated variations that can be retrieved by variation ID. There can be errors during calculation of a specific variation. There are two methods to get result of a particular variation:

- `getResultForVariation(String variationID)` – returns the result of a successfully calculated variation
- `getFailureErrorForVariation(String variationID)` – returns the corresponding error message

Note: The result of original calculation can be retrieved in the same manner as for all variations by the special '`Original calculation`' ID that can be used in code as `org.openl.rules.project.instantiation.variation.NoVariation.ORIGINAL_CALCULATION`.

Predefined Variations

In common, a variation contains a unique ID and is responsible for the modification of arguments and restoring original values. The ID is a '`String`' value used to retrieve the result of the calculation with this variation. By default, the variation's abstract class `org.openl.rules.project.instantiation.variation.Variation` has two methods: `applyModification` and `revertModifications`. The first method modifies arguments; the second rolls back the changes. For this purpose a special instance of Stack is passed to both these methods: in the `applyModification` method, the previous values must be stored; in `revertModifications`, the previous values can be retrieved from the Stack and saved into arguments.

There are several types of predefined variations in the `org.openl.rules..variation` package:

- `NoVariation`
  An empty variation without any modifications. Used for the original calculation. Has a predefined '`Original calculation`' ID.
- `ArgumentReplacementVariation`
  Variation that replaces an entire argument. It was introduced because `JXPathVariation` cannot replace a value of root object, argument. Argument index, value to be set instead of the argument and ID are necessary to construct this variation.
- `JXPathVariation`
  Variation that modifies an object field or replaces an element in array defined by the special path. JXPath is used to analyze paths and set values to corresponding fields, therefore use JXPath-consistent path

expressions. This variation takes for constructor: the index of the argument that needs to be modified, path to the field that must be modified in the JXPath notation, value to be set instead of the original field value, and ID.

- `ComplexVariation`
  Combines multiple variations as a single variation; suitable when it is required to modify different fields or arguments.
- `DeepCloningVariation`
  Can be used when unable or unwilling to revert changes of a specific variation that will be delegated to the `DeepCloningVariation`. This variation clones user's arguments so there is no need to care about any problems caused by the changes in arguments. Also, this variation is not recommended because of the performance drawbacks: the arguments cloning takes time, so the variations usage can be useless.

If these predefined implementations do not satisfy the required requirements, it is easy to implement user's own type of variation that inherits the `org.openl.rules..variation.Variation` class. Custom implementations can be faster than the predefined variations in case they use direct access to fields instead of a reflection as in *JXPathVariation*.

Note: Data binding for custom implementations of variation must be provided to pass the variations through SOAP in WebServices.

Variations Factory

The `org.openl.rules.project..VariationsFactory` class is a utility class for simple creation of predefined variations. It uses the following arguments:

- `vairationId` – Unique ID for a variation.
- `argumentIndex` – Index (0-based) of an argument to be modified.
- `path` – Path to the field to be modified (or just a dot ("`.`") to modify root object (that means the argument)).
- `valueToSet` – Value to set by path.
- `cloneArguments` – Indicates whether cloning should be used.

Usually `VariaitonsFactory` creates the `JXPathVariation` variation which covers most cases of variations usage; but when a dot ("`.`") is specified as the path, then `ArgumentReplacementVariation` is constructed. The `cloneArguments` option just says to `VariaitonsFactory` to wrap created variation by `DeepCloninigVariation`.

An alternative way is to use a special `VariationDescription` bean that contains all fields described above in this section. It is useful to transmit a variation in WebServices and define variations in rules.

Variations from Rules

The process of determining the variations is also a specific kind of decision making process and this process can be written in the form of rules. There is a possibility to write rules defining variations according to the input arguments and use such a rule as the variations provider during the execution. That means the OpenL engine passes an argument to a particular rule, that have the same signature as the rule to be calculated with variations, that return a set of variation descriptions that will be used to create variations.

To write variations in rules, proceed as follows:

1. Define a special rule that returns `VariationDescription[]` and takes arguments similar to the method that must be calculated with variations.

All `VariationDescriptions` returned from the rule are passed to `VariationsFactory`, as described in Variations Factory, to construct variations and add them to initial `VariationsPack`.

2. Add the `import` of `org.openl.rules..variation` package into the Environment Table to make `VariationDescription` available for rules.
3. Mark methods with variations by the special annotation

   `@VariationsFromRules(ruleName = "<name of the rule from the first step>")`

   Warning: The method for retrieving variations must be defined in a service class.
4. Enable variations in Service configurer and databinding.

   By default, there is the `ruleservice.isSupportVariations` option in `openl-ruleservice.properties` that must be set to 'true'. It is passed to Service configurer to create services with variations support and to `AegisDatabindingConfigurableFactoryBean` that registers bindings for all predefined variation classes.

Note: When methods with `@VariationsFromRules` annotation are called, `VariationsPack` can be `null`. In this case only variations from rules are used. Otherwise, if a non-null `VariationsPack` is provided in the arguments, all variations are calculated: the variations from rules and from `VariationsPack` in arguments.

Example

For example, rules that calculate premium for a Policy:

```
Spreadsheet SpreadsheetResult processPolicy(Policy policy)
```

Also there is a special Method for variations from rules:

```
Method VariationDescription[] processPolicyVariations(Policy policy)
```

To calculate premium with variations from rules, the service class must contain the following methods:

```
SpreadsheetResult processPolicy(Policy policy);//original method
@VariationsFromRules(ruleName = "processPolicyVariations")

VariationsResult<SpreadsheetResult> processPolicy(Policy policy, VariationsPack
variations);//method enhanced with variations

VariationDescription[] processPolicyVariations(Policy policy);//method for retrieving the variations
from rules.
```

5 Appendix A: Tips and Tricks

5.1 How to Use OpenL Tablets Webservices from Java Code

This section illustrates how to write a client code that invokes OpenL Tablets Webservice projects. Another way can be used to invoke services but it is recommended to use Apache CXF framework to prevent additional effort for data binding.

A project in OpenL Tablets Webservice can be exposed via a static interface or dynamic interface, generated in runtime. A client code is different in each case. If the project uses a static interface, use the `ClientFactoryBean` class from CXF. Please, refer to CXF documentation for additional information.

The following example illustrates creation of the client code for static interface `MyClass`:

```
ClientProxyFactoryBean clientProxyFactoryBean = new ClientProxyFactoryBean();
clientProxyFactoryBean.setServiceClass(MyClass.class);
clientProxyFactoryBean.setWsdlLocation(getAddress() + "?wsdl");
//OpenL databinding factory
AegisDatabindingFactoryBean aegisDatabindingFactoryBean = new
    AegisDatabindingFactoryBean();
//Set variations support. Recommend to use the same value as a project in server. Can't be
    false, if service uses variations feature.
aegisDatabindingFactoryBean.setSupportVariations(true);
aegisDatabindingFactoryBean.setWriteXsiTypes(true);
//In case you need custom binding classes.
Set<String> overideTypes = new HashSet<String>();
overideTypes.add(<Some class>.class.getCanonicalName());
aegisDatabindingFactoryBean.setOverrideTypes(overideTypes);
clientProxyFactoryBean.setDataBinding(aegisDatabindingFactoryBean.createAegisDatabinding()
    );

MyClass myClass =(MyClass) clientProxyFactoryBean.create();
```

A dynamic client can be used for both static interface and dynamic interface, generated in runtime,configuration. Dynamic client is a feature of CXF framework. For dynamic interface, use `JaxWsDynamicClientFactory` factory. Please, refer to CXF documentation for additional information. The following example illustrates a creating dynamic client:

```
JaxWsDynamicClientFactory dynamicClientFactory = JaxWsDynamicClientFactory.newInstance();
ClassLoader oldClassLoader = Thread.currentThread().getContextClassLoader();
List<String> bindingFiles = new ArrayList<String>() {
      private static final long serialVersionUID = 1L;
          {
              add("binding.xml");
          }
      };
Client client = dynamicClientFactory.createClient(<Service WSDL URL>, bindingFiles);
```

`binding.xml` file content:

```
<jaxb:bindings version="2.2" xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
    xmlns:xjc="http://java.sun.com/xml/ns/jaxb/xjc"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <jaxb:globalBindings generateElementProperty="false" collectionType="indexed"/>
</jaxb:bindings>
```

5.2 How to Use OpenL Tablets REST Services from Java Code

This section illustrates how to write a client code that invokes OpenL Tablets REST services projects. Another way can be used to invoke services but it is recommended to use Apache CXF framework to prevent additional effort for data binding.

The following example illustrates creation of the client code for XML content type:

```
AegisDatabindingFactoryBean aegisDatabindingFactoryBean = new
    AegisDatabindingFactoryBean();
aegisDatabindingFactoryBean.setReadXsiTypes(true);
aegisDatabindingFactoryBean.setWriteXsiTypes(true);

Set<String> overrideTypes = new HashSet<String>();
overrideTypes.add(SomeClass.class.getName());
aegisDatabindingFactoryBean.setOverrideTypes(overrideTypes);

final AegisContextResolver aegisContextResolver = new
    AegisContextResolver(aegisDatabindingFactoryBean.createAegisDatabinding());
WebClient webClient = WebClient.create(#REST service url#,
            new ArrayList<Object>() {
                private static final long serialVersionUID = 5636807402394548461L;
                {
                    add(aegisContextResolver);
                }
});

webClient.type(MediaType.APPLICATION_XML);
Response response = webClient.get();
```

The following example illustrates creation of the client code for JSON content type:

```
JacksonObjectMapperFactoryBean jacksonObjectMapperFactoryBean = new
    JacksonObjectMapperFactoryBean();
jacksonObjectMapperFactoryBean.setEnableDefaultTyping(true);
Set<String> overrideTypes = new HashSet<String>();
overrideTypes.add(SomeClass.class.getName());

jacksonObjectMapperFactoryBean.setOverrideTypes(overrideTypes);
ObjectMapper mapper = jacksonObjectMapperFactoryBean.createJacksonDatabinding();

final JacksonJsonProvider jsonProvider = new JacksonJsonProvider();

WebClient webClient = WebClient.create#REST service url#,
            new ArrayList<Object>() {
                private static final long serialVersionUID = 5636807402394548461L;
                {
                    add(jsonProvider);
                }
            });

webClient.type(MediaType.APPLICATION_JSON);

Response response = webClient.get();
```

Note: If you use POST request more than one argument, you have to create a DTO, that contains field with method argument names and send this DTO object via `webClient.post()` method.