OpenL Tablets Web Services Usage and Customization

OpenL Tablets 5.12

OpenL Tablets BRMS

Document number: OpenL_WS_Usage_5.x_1.0
Revised: 04-21-2014

Contents

Preface **4**

How This Book Is Organized 4
Audience 4
Related Information 4
Typographic Conventions 5

Chapter 1: Introduction **6**

Chapter 2: OpenL Tablets Web Services Configuration **8**

Configuration Points 9
Data Source Configuration 9
Service Exposing Method 11
System Settings 14
Configure a number of threads to rules compilation 14
Logging Requests to Web Services and their Responds 14
Instantiation strategy configuration 15

Chapter 3: OpenL Tablets Web Services Customization **16**

Customization Points 16
Service Configurer 17
Multimodule with Customized Dispatching 20
Dynamic Interface Support 21
Interface Customization through Annotations 22
JAR File Data Source 23
Data Source Listeners 23
Variations 23

Troubleshooting Notes **27**

Service Configurer for OpenL Tablets Tutorials 27

Preface

OpenL Tablets is a Business Rules Management System (BRMS) based on 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 document goal 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.

How This Book Is Organized

Chapter 1: Provides overall information about OpenL Tablets Web Services application.

Chapter 2: Describes the configuration of OpenL Tablets Web Services for different environments.

Chapter 3: Explains how to customize OpenL Web Services to meet customers' needs and requirements.

Audience

This guide is targeted at rule developers who will 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.

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.

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.

Chapter 1: 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.

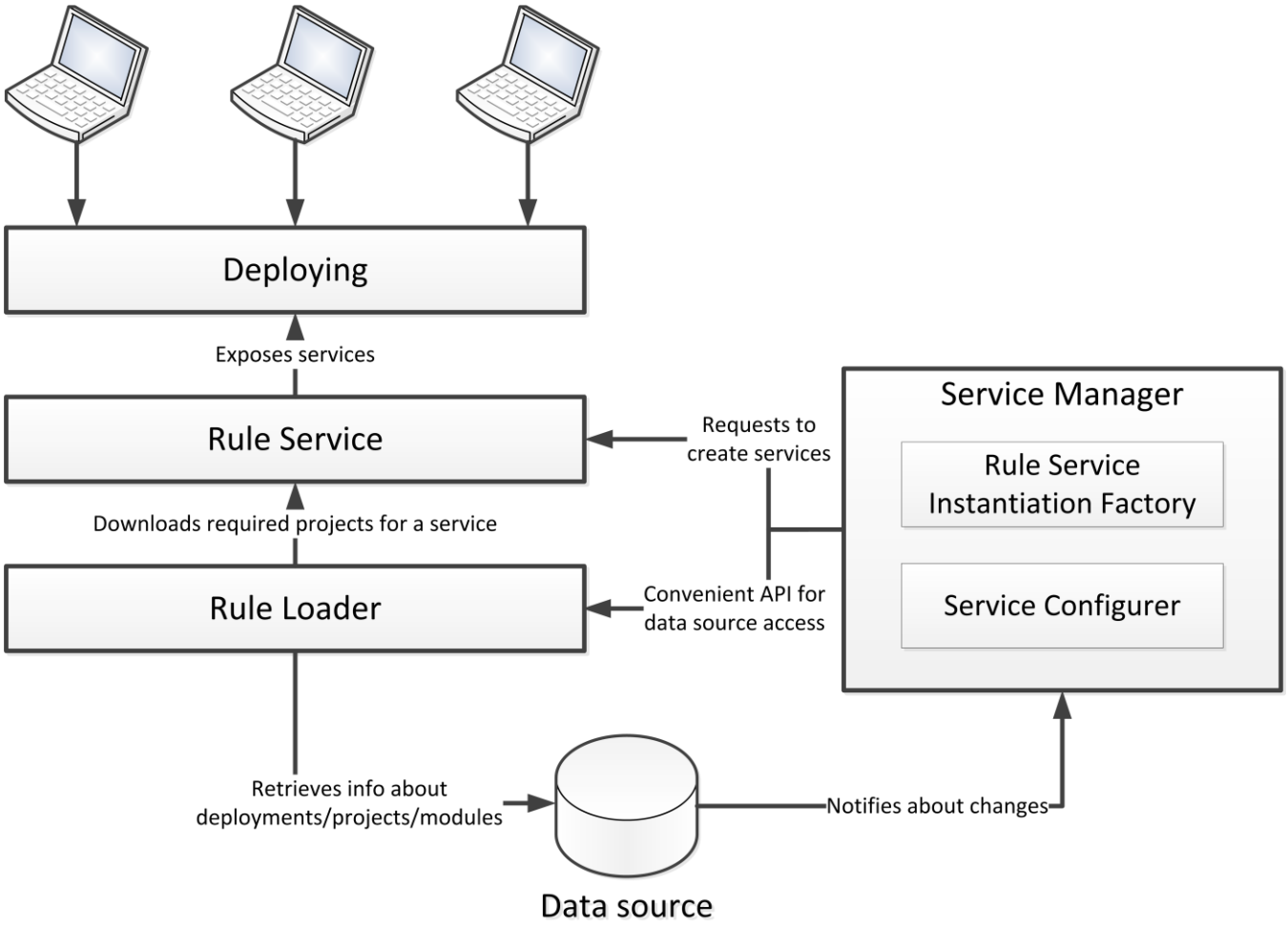

Figure 1: Overall OpenL Tablets Web Services architecture

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*.

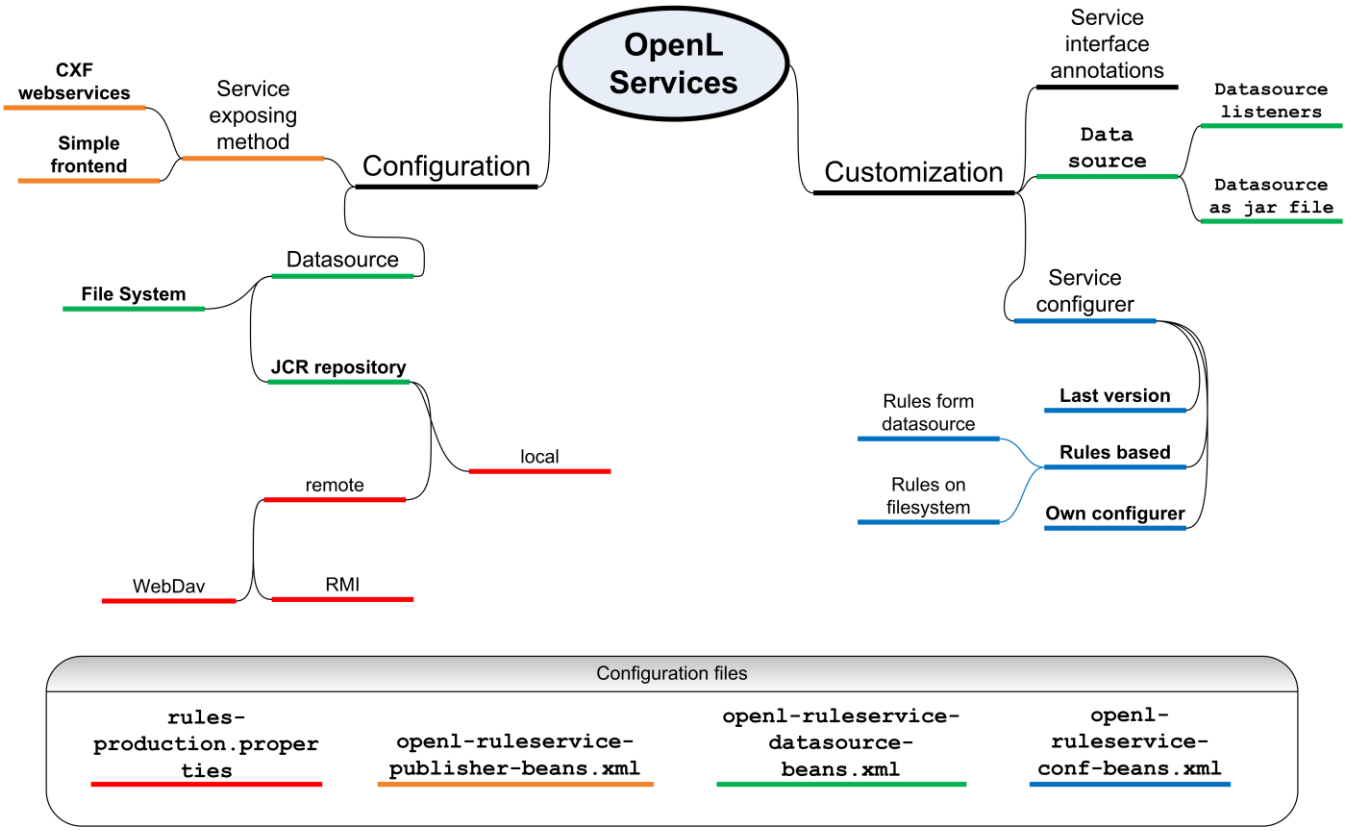

Figure 2: Configurable and customizable components of OpenL Tablets Web Services application

Chapter 2: 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:

- `openl-ruleservice-beans.xml` – It is the main configuration file that include all other configuration files. This file will be 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 publisher configuration.
- `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` – The configuration file for JCR based data source. If you use a different type of data source this file will be ignored.

The Service Manager is the main component of OpenL Tablets Services Frontend containing all major parts (a loader, a ruleservice, and a Service Configurer (*see OpenL Tablets Developer Guide for more information*)). It knows all currently running services and intelligently controls all operations for deploying, undeploying, and redepoying the services. These operations will only be performed in two cases: the initial deployment at the start of OpenL Tablets Services Frontend; and processing after making changes in the data source (The Service Manager is always a data source listener as described below).

You will find detailed information about all that configuration files further in this document.

Configuration Points

You can replace any part of OpenL Tablets Services Frontend by your own implementation. (Refer to *OpenL Tablets Developer Guide* for information about the system architecture.) If you use the common approach you should configure the following components:

- **Data Source**: Informs the OpenL Tablets system where to retrieve your rules.
- **Service Exposing Method**: Defines the way your services will be exposed. It can be a web service (any framework), simple java framework, etc.

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

Data Source Configuration

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

- Central OpenL Tablets Production Repository (JCR repository)
- File system with the proper directory structure.

JCR repository

If you use JCR repository as a data source, your "datasource" bean in the `openl-ruleservice-datasource-beans.xml` file (by default located in the `<TOMCAT_HOME>\webapps\<web services file name>\WEB-INF\classes` directory) should be: `org.openl.rules.ruleservice.loader.JcrDataSource`, and all JCR repository settings should be specified in the `rules-production.properties` file that is located in the same directory.

For that you should remove or mark as comment all properties and change class to `org.openl.rules.ruleservice.loader.JcrDataSource` in the bean definition. Here is an example of the JCR data source bean:

```
<bean id="datasource" class="org.openl.rules.ruleservice.loader.JcrDataSource" />
```

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

1. **Local repository**. Repository is located on your local machine as a folder.

   The repository factory should be:

   ```
   production-repository.factory =
   org.openl.rules.repository.factories.LocalJackrabbitProductionRepositoryFactory
   ```

   Additional property that defines location of the JCR repository:

   ```
   production-repository.local.home = /openl/common-repository
   ```

   NOTE: This is the default setting used in OpenL Tablets, so you don't need to edit the rules-production.properties file.

   NOTE: Only one application can use a local repository. That is, you cannot use OpenL Tablets Web Services and OpenL Tablets WebStudio with a local repository at the same time. In case multiple applications need to access a repository, remote access to the repository should be provided for all the applications.
2. **Remote repository**. Repository is located on a remote server. We recommend you to download all sources related to JCR repository (Repository package is a zip file containing `repository server`,

config files and empty JCR repository) from http://openl-tablets.sourceforge.net/downloads page, **Repository** (zip file) link. 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 jackrabbit war file should be run earlier than your OpenL Tablets Web Services war file (Tomcat runs war files alphabetically).

Remote repository can be accessed by the following protocols:

a. **RMI**

To set up access to the repository, you should copy 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 should be used for remote access as follows:

- repository.home={the folder where your production repository is located}
- rmi.url={URL for remote access to the repository}, for example, //localhost:1099/production-repository

The repository factory should be:

```
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
```

b. **WebDav:**

The repository factory should be:

```
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
```

Attention! A problem can arise if you use one instance of Tomcat for both web archives: jackrabbit-webapp and OpenL Tablets Web Services war file. Tomcat will hang on 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 will be connections by the datasource URL. But Tomcat applies such connections and waits for the end of deployment of all web applications. So we have 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, you can use one of the possible solutions:

1. Use several Tomcat instances: one of them will contain jackrabbit-webapp, and the other will contain 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 your projects means that the projects are placed into local folder; this folder will represent a single deployment containing all the projects.

NOTE: This type of data source does not support versioning.

To configure a local file system as a data source you should override the following beans as shown below:

```
<bean id="datasource" class="org.openl.rules.ruleservice.loader.FileSystemDataSource">
       <constructor-arg name="loadDeploymentsFromDirectory" value="… path to the folder
       containing the projects"/>
       <property name="localWorkspaceFileFilter" ref="localWorkspaceFileFilter"/>
       <property name="localWorkspaceFolderFilter" ref="localWorkspaceFolderFilter"/>
</bean>
```

You can also pack your rule projects to a jar file and use this jar file as a data source. See the JAR File Data Source section further in this document.

NOTE: By default, your configuration of data source looks as follows:

```
<bean id="datasource" class="org.openl.rules.ruleservice.loader.FileSystemDataSource">
       <constructor-arg name="loadDeploymentsFromDirectory"
       value="${ruleservice.datasource.dir}"/>
       <property name="localWorkspaceFileFilter" ref="localWorkspaceFileFilter"/>
       <property name="localWorkspaceFolderFilter" ref="localWorkspaceFolderFilter"/>
</bean>
```

Where `${ruleservice.datasource.dir}` means the property from the `openl-ruleservice.properties` configuration file. **Attention!** For proper parsing of java properties file, the path to the folder should be defined with slash (‘/’) as the folders delimiter. (Back slash ‘\’ is **not allowed**!)

Service Exposing Method

Service Exposing Method specifies the method you will use to expose your OpenL Tablets Services.

Common flow of service exposing is as follows:

1. Retrieve service descriptions that should be deployed from service configurer.
2. Undeploy currently running services that aren’t in services defined by service configurer(Some services can become unnecessary in new version of product)
3. Redeploy currently running services that are still in services defined by service configure(service update)
4. Deploy new services that were not represented earlier.

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

1. **CXF Web Services:** Publisher that exposes your services as Web Services using CXF framework. Configuration should be as follows:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:util="http://www.springframework.org/schema/util"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/util
http://www.springframework.org/schema/util/spring-util.xsd">

       <import resource="classpath:META-INF/cxf/cxf.xml" />

       <!-- Bean helps to read configuration and apply it to override types -->
       <bean id="rootClassNamesBindingFactoryBean"

       class="org.openl.rules.ruleservice.databinding.RootClassNamesBindingFactoryBean">
              <property name="rootClassNames" value="${ruleservice.binding.rootClassNames}" />
       </bean>

       <bean id="serviceDescriptionConfigurationRootClassNamesBindingFactoryBean"

       class="org.openl.rules.ruleservice.databinding.ServiceDescriptionConfigurationRootClas
sNamesBindingFactoryBean"
              scope="prototype" >
              <property name="additionalRootClassNames"
ref="rootClassNamesBindingFactoryBean"/>
       </bean>

       <bean id="serviceDescriptionConfigurationSupportVariationsFactoryBean"

       class="org.openl.rules.ruleservice.databinding.ServiceDescriptionConfigurationSupportV
ariationsFactoryBean"
              scope="prototype" >
              <property name="supportVariations" value="${ruleservice.isSupportVariations}"/>
       </bean>

       <!-- Data binding type in WebServices(set it to the "serverPrototype" bean). -->
       <bean id="aegisDatabindingFactoryBean"
              class="org.openl.rules.ruleservice.databinding.AegisDatabindingFactoryBean"
              scope="prototype">
              <property name="writeXsiTypes" value="true" />
              <property name="overrideTypes"
ref="serviceDescriptionConfigurationRootClassNamesBindingFactoryBean"/>
              <property name="supportVariations"
ref="serviceDescriptionConfigurationSupportVariationsFactoryBean" />
       </bean>

       <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.frontend.ServerFactoryBean"
            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 local="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 local="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.WebServicesRuleServicePublisher">
            <property name="serverFactory" ref="webServicesServerPrototypeFactory" />
            <property name="baseAddress" value="${ruleservice.baseAddress}" />
      </bean>

</beans>
```

NOTE: If you use Openl Tablets Web Services war application then the base address should be relational. The full web service address will be:
`webserver_context_path/ws_app_war_name/address_specified_by_you.`

2. Simple Java Frontend

Customization shall be 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 may 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 system tries to compile too many rules at once, it fails with out of memory exception.

The setting that limits 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 you with the 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 you should set `ruleservice.logging.enabled = true`.

Instantiation strategy configuration

The system provides you 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 future for memory save)

To disable the lazy initialization strategy you should set `ruleservice.instantiation.strategy.lazy = false`. (All modules compile on application launch.)

Chapter 3: OpenL Tablets Web Services Customization

If a project has specific requirements, developer should create a new maven project that extends OpenL Tablets Web Services and add or change required points of configuration. You should add the following dependency to the pom.xml file with the version used in your 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>
```

You can use the following maven plugin to control the Web Application building with your own 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

- **Service Configurer**: defines all services to be exposed and modules contained in each service.
- **Multimodule with Customized Dispatching**: gives you a possibility to handle dispatching between modules.
- **Dynamic Interface Support**: generate interface for services at runtime.
- **Interface customization through annotations**
- **JAR File Data Source**: pack your rule projects into a jar file and place the archive in the classpath.
- **Data Source Listeners**

- **Variations**: additional calculation of the same rule with a slight modification in its arguments.

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 of different deployments. 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 also can be only one version of some 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.

You can implement different module gathering strategies according to your needs. You can choose deployments and projects with concrete values of some property (For example, service for some LOB, service containing modules with expiration date before some date, etc.) or by using versions of deployments (or use both these approaches).

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

Service Description

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

1. Service name: unique service identifier.
2. Service URL: URL path for the service (absolute for console start and relative to context root for ws.war case).
3. Service class: interface of the service that will be used at the server and client side.
4. Rules: a module, or a set of modules that will be combined together as a single rules module.
5. “Provide runtime context” flag: indicates whether the runtime context should be added to all rule methods or not. If it is “`true`” then `IRulesRuntimeContext` argument should be added to each method in the service class.
6. “Support variations” flag (optional): indicates whether the current service supports variations or not (see the *Variations* section further in this Guide).

You can create your own implementation of Service configurer interface - `org.openl.rules.ruleservice.conf.ServiceConfigurer` - and register it as a Spring bean with the “`serviceConfigurer`” name, or you 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` – you can define all your modules as OpenL Tablets rules using the `RulesBasedConfigurerTemplate.xlsx` template located in the `org.openl.rules.ruleservice` project. If you have downloaded OpenL Tablets Web Services as a war file, you will find the project in the `WEB-INF\lib` folder. Otherwise, the project artifact is located within the Central Maven repository. The template is described in the *RulesBasedConfigurerTemplate Description* section further in this document. You can use rules representing your Service Configurer, from different locations:

    a. **From file system:** Your rules are located in some folder. Configuration should be as follows:

```
<bean id="serviceConfigurer"
class="org.openl.rules.ruleservice.conf.FileSystemRulesBasedServiceConfigurerFactoryBe
an"><property name="folderLocationPath" value="./test-resources/"/>

<property name="moduleName" value="RulesBasedConfigurer"/>

</bean>
```

    b. **From Data Source:** Your rules are located in a DataSource. In this case you can redeploy your rules for service configurer in the DataSource. Configuration should be as follows:

```
<!-- Determines the services that should be exposed using RulesLoader.  -->
<bean id="serviceConfigurer"
class="org.openl.rules.ruleservice.conf.FileSystemRulesBasedServiceConfigurerFactoryBe
an">
       <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>
```

Description of RulesBasedConfigurerTemplate

The RulesBasedServiceConfigurer is a simple configurer where all modules selection logic is defined in OpenL Tablets rules.

The configurer also supports changing the service exposing logic: if in time according to the project's needs number of services and their content varies. In such case service configurer should be located in the Data source, and changes in the given deployment representing the service configurer will cause re-exposing the services with a new logic.

The *RulesBasedConfigurerTemplate* is an excel workbook that represents rules for selecting modules for your services. It includes five worksheets:

1. **Services** – The main worksheet describing all your services

2. **Modules** – The template worksheet for selecting modules for a particular service; you can easily copy it for a new service.
3. **Examples** – several examples of rules that define modules for some service.
4. **Util** – Utility method(s) that help you to select required modules.
5. **Env** – Environment table.

Services Worksheet

In the main worksheet, there is the **Datatype Service** table that describes all your services. Fields in this datatype table are similar to common service description (see the *Service Description* section above in this document) except for the **modulesGetter** field. This field points the rule method that should be invoked to determine whether the particular module belongs to some service or not.

All services are defined in the **Data Service services** table.

In the Services worksheet, there are also two helpful methods to retrieve Rules loader and all deployments.

Modules Worksheet

This sheet contains table(s) that provides selection of modules for services defined on the **Services** worksheet. A number of tables should be equal to the number of services in the **Services** worksheet. The tables can be of the following types: Decision Table (DT), TBasic Table, or Method Table.

As you can see from the signature in the example template, it takes a deployment, a project from this deployment, and module from the project. The rule should return a boolean value indicating whether the module belongs to some service or not. The name of the DT is specified in Data table “services” as a **modulesGetter** for a particular service. RulesBasedServiceConfigurer at the Java side iterates all deployments, projects, and modules and checks the suitability of each module by calling methodGetter.

Your should define a set of conditions for selecting deployments, projects and modules by their names, versions, and properties to process deployments, projects, and methods.

Deployments properties:

1. **deploymentName** of type String (Attention! Not **name** property)
2. **commonVersion** of type org.openl.rules.common.CommonVersion (Attention! Not **version** property)
3. **effectiveDate** of type java.util.Date
4. **expirationDate** of type java.util.Date
5. **lineOfBusiness** of type String

Project properties:

1. **name** of type String
2. **version** of type org.openl.rules.common.CommonVersion
3. **effectiveDate** of type java.util.Date
4. **expirationDate** of type java.util.Date
5. **lineOfBusiness** of type String

Module properties:

1. **name** of type String
2. **type** of type org.openl.rules.project.model.ModuleType
3. **classname** of type String

Util Worksheet

In the **Util** worksheet, there is a helpful method for the selection of modules - `getLastVersion(deplName)` which returns the last version of the deployment specified by name.

Env Worksheet

This worksheet contains environment settings for imports in the configurer.

NOTE: Please do not make any changes in the imports provided in the template. You can only add your own imports if required.

Multimodule with Customized Dispatching

There is additional mode for multimodule which gives you a possibility to handle dispatching between modules by your own logic. That means OpenL Tablets will pass the control of selection of the needed module to your class. The following steps indicate how to adjust multimodule with your own dispatching:

1. Create java interface representing your rules.
2. For each method from the interface determine the dispatching:

    a. For methods that represents **Data tables** you should provide implementation of *org.openl.rules.ruleservice.publish.cache.dispatcher.ModuleDispatcherForData* and mark that method by the *org.openl.rules.ruleservice.publish.cache.dispatcher.DispatchedData* annotation.

    b. For methods that represent **Rules** you should provide implementation of *org.openl.rules.ruleservice.publish.cache.dispatcher.ModuleDispatcherForMethods* and mark that method by the *org.openl.rules.ruleservice.publish.cache.dispatcher.DispatchedMethod* annotation.
3. Create your implementation of *org.openl.rules.ruleservice.publish.RuleServiceInstantiationStrategyFactory* that will return *DispatchedMultiModuleInstantiationStrategy* instead of lazy multimodule (by default) and register it in `openl-ruleservice-override-beans.xml`.

Notes:

- *ModuleDispatcherForData* and *ModuleDispatcherForMethods* must have public constructor without parameters. The aim of these classes is to select the needed Module according to Runtime context and the executed method (that means the rule name and arguments for the method representing **Rule,** and **Data table** for method representing data).
- If you use dispatched multimodule then the interface with annotated methods is obligatory, otherwise you will get an exception.
- If you have simultaneously getter and setter for some Data you can annotate only one of them.
- You can provide different dispatching logic for different methods.
- See example in *org.openl.rules.ruleservice.multimodule.DispaspatchedMultiModuleTest*.

Dynamic Interface Support

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

This feature is enabled by default. If you want to use dynamic interface, you should not define static interface for a service.

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

You can apply this configuration in projects in the `rules.xml` file. See the following example:

```
<project>
    <!--  Project identifier. String value which defines project uniquely. -->
    <id>project-id</id>
    <!-- Project name. -->
    <name>projecy-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 you need to 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 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, 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 should 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

Interceptors for service methods

You can easily specify interceptors for service methods using the following annotations:

1. `@ServiceCallBeforeInterceptor` – method annotation to define before interceptors, array of interceptors should be registered in annotation parameter. All interceptors should implement org.openl.rules.ruleservice.core.interceptors.ServiceMethodBeforeAdvice interface.
2. `@ServiceCallAfterInterceptor` - method annotation to define after interceptors, array of interceptors should be registered in annotation parameter. There are two types of after interceptors:
    a. After Returning interceptor: intercepts result of successfully calculated method with possibility of post processing of return result (even result conversion to another type, (!be careful) then this type should be specified as the return type for method in service class). After Returning interceptors should inherit *org.openl.rules.ruleservice.core.interceptors.AbstractServiceMethodAfterReturningAdvice*
    b. After Throwing interceptor: intercepts method that has thrown Exception with possibility of post processing of error and throwing another type of exception. After Returning interceptors should inherit *org.openl.rules.ruleservice.core.interceptors.AbstractServiceMethodAfterThrowingAdvice*

Annotation customization for dynamic interfaces

You can use annotation customization for dynamically generated interfaces. This feature is only supported for a project which contains `rules-deploy.xml` file.

You need to add 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>
```

You should define a template interface with annotated methods with 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, you have the following methods in the generated dynamic interface:

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

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

If you need to add annotation to the first method, just use this signature in the template interface.

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

In cases `MyType` is generated in runtime class, it is required to use a type that is assignable from `MyType` class. For example:

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

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

`@AnyType` annotation value is a java regular expression of canonical class name. You should use this annotation, if you need more details to define template method.

JAR File Data Source

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

Then you should add the following bean to unpack the projects to the specified folder - it should 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 production repository modification: new deployment added). Service manager is always data source listener, because it should handle all modifications in data source. You can add your own listener (implementing *org.openl.rules.ruleservice.loader.DataSourceListener*) for additional controlling of data source modifications with needed 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 your 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 you need 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.

How it Works

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

- Original method with a corresponding rule signature and
- Method with injected variations.

The method enhanced with variations has a signature similar to original method. You will also add the argument of type `org.openl.rules..variation.VariationsPack` as the last argument; the return type should be generic `org.openl.rules..variation.VariationsResult<T>`, where `T` is the return type of the original method. (**Warning**: if you use your own service class instead of 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 some variation. We provide 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 should 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 predefined '`Original calculation`' ID.
- `ArgumentReplacementVariation` – Variation that replaces the 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 special path. JXPath is used to analyze paths and set values to corresponding fields, so you should use JXPath-consistent path expressions. This variation takes for constructor: the index of the argument that needs to be modified, path to field that should be modified in JXPath notation, value to be set instead of original field value, and ID.

- `ComplexVariation` – Combines multiple variations as a single variation; suitable when you need to modify different fields/different arguments.
- `DeepCloningVariation`- Can be used when you cannot (or don't want to) revert changes of some variation that will be delegated to the `DeepCloningVariation`. This variation clones your arguments so you don't care about any problems caused by the changes in arguments. Also this variation is not recommended because of performance drawbacks: the arguments cloning takes time, so the variations usage can be useless.

If these predefined implementations don't satisfy your requirements it is easy to implement your 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 like in *JXPathVariation*.

Note: Data binding for custom implementations of variation should 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 you specify dot ("`.`") as the path then `ArgumentReplacementVariation` will be constructed. The `cloneArguments` option just says to `VariaitonsFactory` to wrap created variation by `DeepCloninigVariation`.

An alternative way is to use 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 some kind of the decision making process and this process can be written in the form of rules. There is 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 will return a set of variation descriptions that will be used to create variations.

How to write variations in rules:

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

   All `VariationDescriptions` returned from the rule will be passed to `VariationsFactory` (see the previous section) to construct variations and add them to initial `VariationsPack`.

2. Add `import` of `org.openl.rules..variation` package into the Environment Table (to make `VariationDescription` available for rules).
3. Mark methods with variations by special annotation
   `@VariationsFromRules(ruleName = "<name of the rule from the first step>")`
   Warning: The method for retrieving variations should be defined in a service class.
4. Enable variations in Service configurer and databinding.
   By default there is `ruleservice.isSupportVariations` option in `openl-ruleservice.properties` that you should 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 is called, `VariationsPack` can be `null`. In this case only variations from rules will be used. Otherwise, if non-null `VariationsPack` is provided in the arguments all variations will be calculated: the variations from rules and from `VariationsPack` in arguments).

Example

For example, suppose we have rules that calculate premium for a Policy:

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

Also we have a special Method for variations from rules:

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

In case you want to calculate premium with variations from rules your service class should 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
```

Troubleshooting Notes

Service Configurer for OpenL Tablets Tutorials

OpenL Tablets provides a set of tutorials (See the OpenL Tablets Tutorials page for details.) that you can use in OpenL Tablets Web Services.

Original tutorials cannot be used in web services with the default configurer – `LastVersionProjectsServiceConfigurer`, because it does not provide interface for web services and uses default interface from a particular rules project (the *generated interface*).

In this case you will get the following error message: “There is no implementation in rules for interface method <method description>”. The reason is the execution mode (see OpenL Tablets Reference Guide) used for compilation of rules in RuleService. When execution mode is enabled **Test tables** will be ignored during rules parsing. But at the same time, special methods that refer to the Test tables are represented in the generated interface, and as the result the exception occurs.

How to solve this problem?

1. Use specific **Service configurer** that defines non-null interface for your services.

   In this case the interface defined by **Service configurer** will be used for rules compilation instead of the default generated interface.
2. Regenerate interface of the tutorial with Test tables ignoring.

For this purpose you should open the `<tutorial root folder>/build/GenerateJavaInterface.build.xml` file for the given tutorial and add the `ignoreTestMethods="true"` option for openlgen Ant call.

As the result you will have something like that:

```
<target name="generate">

      <echo message="Generating org.openl.tablets.tutorial? Interface"/>

       <openlgen openlName="org.openl.xls" userHome="."

            srcFile="path to file"

          targetClass="org.openl.tablets.tutorial?.Tutorial?_RulesInterface"

            displayName="Tutorial- Title" ignoreTestMethods="true"

            targetSrcDir="gen"

      />

</target>
```

After that run **Generate org.openl.tablets.tutorial Interface.launch** script and make sure that the tutorial interface doesn’t contain methods representing **Test tables** (the methods with the

`org.openl.rules.testmethod.TestUnitsResults <Test table name in rules>TestAll()` signature). After that the tutorial can be used in OpenL Tablets RuleService.