



# OpenL Tablets Web Services Usage and Customization

## **OpenL Tablets 5.9.1**

### **OpenL Tablets BRMS**

**Document number: OpenL\_WS\_Usage\_5.x\_1.0**  
**Revised: 04-06-2012**

---



OpenL Tablets Documentation is licensed under a [Creative Commons Attribution 3.0 United States License](https://creativecommons.org/licenses/by/3.0/us/).

# Contents

---

<b>Preface.....</b>	<b>4</b>
How This Book Is Organized .....	4
Audience .....	4
Related Information .....	4
Typographic Conventions.....	5
<b>Chapter 1: Introduction .....</b>	<b>6</b>
<b>Chapter 2: OpenL Tablets Web Services Configuration .....</b>	<b>8</b>
Configuration Points .....	9
Data Source Configuration .....	9
Service Exposing Method.....	11
<b>Chapter 3: OpenL Tablets Web Services Customization .....</b>	<b>14</b>
Customization points: .....	14
Service Configurer .....	14
Multimodule with Customized Dispatching .....	18
Interface Customization through Annotations.....	18
JAR File Data Source .....	19
Data Source Listeners .....	19

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

# Typographic Conventions

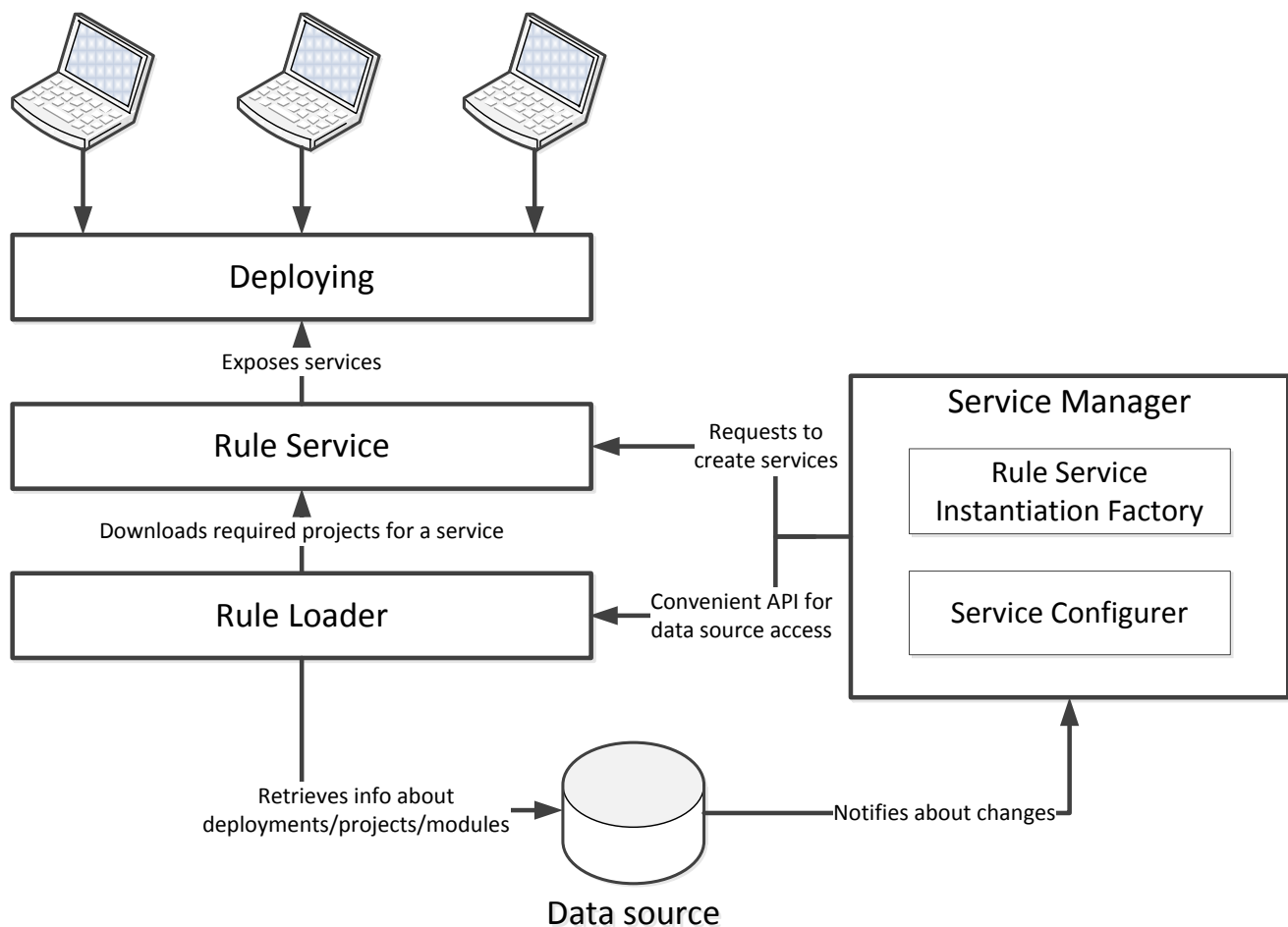
The following styles and conventions are used in this guide:

Typographic styles and conventions	
Convention	Description
<b>Bold</b>	<ul style="list-style-type: none"><li>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.</li><li>Represents keys, such as <b>F9</b> or <b>CTRL+A</b>.</li><li>Represents a term the first time it is defined.</li></ul>
<i>Courier</i>	Represents file and directory names, code, system messages, and command-line commands.
<b>Courier Bold</b>	Represents emphasized text in code.
Select <b>File</b> > <b>Save As</b>	Represents a command to perform, such as opening the <b>File</b> menu and selecting <b>Save As</b> .
<i>Italic</i>	<ul style="list-style-type: none"><li>Represents any information to be entered in a field.</li><li>Represents documentation titles.</li></ul>
< >	Represents placeholder values to be substituted with user specific values.
<a href="#">Hyperlink</a>	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.



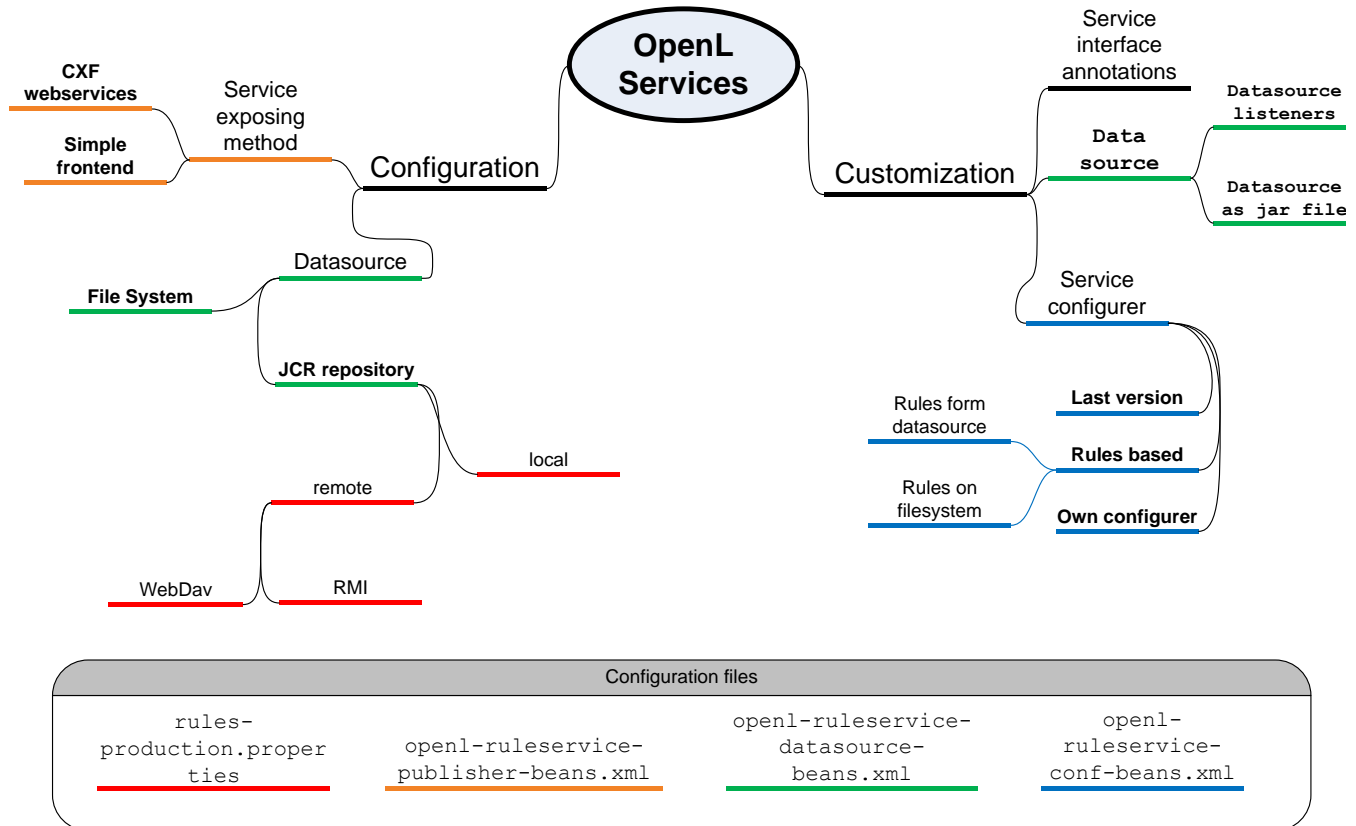
*Figure 1: Overall OpenL Tablets Web Services architecture*

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 the 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 redeploying 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 sections below: a central OpenL Tablets Production Repository (JCR repository), or file system with proper 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 settings 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 use the

`jackrabbit-webapp-2.2.8.war` file (you can download it from [http://repo1.maven.org/maven2/org/apache/jackrabbit/jackrabbit-webapp/2.2.8/jackrabbit-webapp-](http://repo1.maven.org/maven2/org/apache/jackrabbit/jackrabbit-webapp/2.2.8/jackrabbit-webapp-2.2.8.war)

[2.2.8.war](#)) to make your repository remotely accessible. Please save the downloaded war file to 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 download the `jackrabbit.zip` file from the OpenL Tablets web site (Currently the file is not accessible from this site. Please refer to OpenL Tablets team for the file). Unpack the file to the `<TOMCAT_HOME>\bin\jackrabbit` folder (create the `jackrabbit` folder if it does not exist). The `jackrabbit.zip` archive 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:1099/2TR`

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.datasources.dir}"/>
  <property name="localWorkspaceFileFilter" ref="localWorkspaceFileFilter"/>
  <property name="localWorkspaceFolderFilter" ref="localWorkspaceFolderFilter"/>
</bean>
```

Where `${ruleservice.datasources.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:** Deployment admin that exposes your services as Web Services using CXF framework. Configuration should be as follows:

```

<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>

<!-- Data binding type in WebServices(set it to the "serverPrototype" bean) -->
<bean id="dataBinding"
      class="org.openl.rules.ruleservice.databinding.AegisDataBindingConfigurableFactoryBean">
    <property name="writeXsiTypes" value="true" />
    <property name="overrideTypes" ref="rootClassNamesBindingFactoryBean" />
</bean>

<!-- Main description for the one WebService -->
<bean id="loggingFeature" class="org.openl.rules.ruleservice.logging.LoggingFeature">
    <property name="loggingEnabled" value="${ruleservice.logging.enabled}" />
</bean>

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

<!-- Prototypes factory. It will create new server prototype for each new
    WebService. -->
<bean id="serverPrototypeFactory"
      class="org.springframework.beans.factory.config.ObjectFactoryCreatingFactoryBean">
    <property name="targetBeanName">
        <idref local="serverPrototype" />
    </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="ruleServicePublisher"
      class="org.openl.rules.ruleservice.publish.WebServicesRuleServicePublisher">
  <property name="serverFactory" ref="serverPrototypeFactory" />
  <property name="baseAddress" value="{ruleservice.baseAddress}" />
</bean>
```

**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>
```

## 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>5.9.0</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.
- **Interface customization through annotations**

### 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 will often want to create web services containing several rule projects/modules. Users will have 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 rely 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.

You can create your own implementation of Service configurer interface -

`org.openl.rules.ruleservice.management.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.management.LastVersionProjectsServiceConfigurer` – exposes deployments based on the last version; creates one service for each project in the deployment.
3. `org.openl.rules.ruleservice.management.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.FileSystemRulesBasedServiceConfigurerFactoryBean">
  <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 a 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.

You 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.DispatchedMultiModuleTest*.

## 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 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*

## 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 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.