JOnix Documentation

jOnix Platform Overview

The jOnix platform represents a cloud application which can be used in order to send, store, distribute and check the validity of ONIX data. There are multiple functions that are performed by jOnix platform:

- ONIX message validity checking
- ONIX message receival and storage
- ONIX message redistribution
- Unified interface
- Global entry point
- Security
- Addressability by organization
- Forwarding to remote third party services
- Receival of ONIX messages from third parties

This functionality is the basis of the platform. It shares some similar properties with an enterprise service bus, but it is simpler, easier to use, and operates only on ONIX messages.

jOnix Platform Architecture

jOnix comprises two major parts – the jOnix service and jOnix client. Jonix service is the central part of the jOnix platform as long as it provides the means for receiving, storing and dispatching the messages to remote parties.
The jOnix client is used to control the operation of jOnix service in a way that is acceptable for an average user. It allows to produce a security key, without which the jOnix service cannot be used. It has the names to specify the organization name for which the messages are processed and the destination addresses for the specified organization name. The jOnix client allows to send raw ONIX messages to the local platform, and it also allows to browse the ONIX and product data.

JOnix Service

Classes are listed by packages.

fi.metropolia.ereading:

RESTService – this class is the Application class of jOnix service and stores the data about resources and singletons of the Jersey JAX-RS service.

fi.metropolia.ereading.resources:

ONIXReader – this class contains all the web HTTP methods to control the access to the jOnix service. It has the following methods:

```
Response submitONIX(ONIXMessage message, @QuerysParam("key") String key);
```

This method receives a POST-request payload in order to store it in the database, to redistribute it to other endpoints and to filter the incoming messages according to their validity. The key represents a private key which is unique for a platform-using staff. The products are given a version and stored over the previous products if their reference numbers are the same.

```
Response getONIXMessageByAddressee(@PathParam("name") String addressee);
```

This method allows to retrieve all the messages for a particular addressee name with the HTTP GET request. Internally, the method refers to the database in order to filter the stored messages by the destination name, and then forms a Response containing the payload.

```
Response getProductByReference(@PathParam("reference") String reference);
```

This method retrieves all the data about a particular product from the database. The product is identified by its reference number. Only the newest version of the product is returned.

```
Response getProductByReferenceWithHistory(@PathParam("reference") String ref);
```

This method retrieves the history of changes for a particular product. It returns all the versions existing for a product reference, starting from the latest version to the oldest one. The record reference parameter helps to identify which product to retrieve.

```
Unmarshaller getUnmarshaller() throws Exception;
Marshaller getMarshaller() throws Exception;
```

These two methods create the marshaller an unmarshaller in order to use internally by the class. The instance of the marshaller returned is the MOXy marshaller, and, thus, supports JSON parsing according to the specification. Whenever the conversion from JSON to Object or visa versa is needed, one of these methods is called.

MessageFilter – this class is responsible for checking the message validity. It is a utility class used from the ONIXReader class submitONIX method. It only method is:

```
public static void filter(ONIXMessage message) throws Exception;
```

This method checks the body of the ONIX message tag by tag. If some problem is detected, an Exception with the cause is thrown. This exception must be caught in the calling method and the Response must be issued, that a particular request contained errors in the message payload.

fi.metropolia.ereading.background:

MongoResource – this class is a singleton to retrieve a mongo DB connection globally. It has one method – **getMongo()** - which returns the **MongoClient** instance which can be worked on in the rest of the application.

BusRegistrator – this class is a background listener. It discovers the information about new buses registered in the local jOnix space and adds them in order to redirect the messages. Implements ServletContextListener, specifically:

```
public void contextInitialized(ServletContextEvent arg0);
```

This method starts a repeated task which is executed every minute since the application starts. The inner class called **NewBusRegistrator** implements the Runnable interface. Inside the run method it connects to the mongoDB users` collection, and retrieves the user data row by row. When it detects that a new "outputBusAddress" has been added, the task automatically creates an instance of **Outlet** type, and registers it with the **BusFactory** instance, injected previously into class. Thus, the **BusRegistrator** maintains the list of all destinations to which the messages are to be redirected.

BusFactory – this class is a singleton injected into other classes on request. Its purpose is to maintain a list of destinations for the messages to be redirected to. Internally, it has the some methods. These methods perform methods on the internal field, which is a map:

```
            public void setNewBus(Outlet bus);

It sets a new Outlet to the table registry of Outlets.

            public void clean();

Clears all destinations for messages.

            public Outlet getBusIfExists(String organization);
```

Retrieves the destination from the storage if there is a destination for a provided organization name.

Outlet – this class is responsible for storing the data for one destination of the message. Internally it stores the organization for which the http-address to send messages to is specified, the http-address itself, and the http-method to send the message to the bus. These fields have their respective mutators and accessors.

The other two methods specified in the Outlet are:

```
            public void sendOnixMessage(ONIXMessage message);
```

This method send an ONIX message from this instance of jOnix service to other external services.

Internally, uses Apache HTTP library to resend the messages. In case codes 200, 202 or 204 are returned as a response, the operation is considered complete and successful.

```
public void deregister();
```

This method removes the destination from the list of valid destinations, if it is not accessible (the last sending operation failed). It sets the output address as invalid in the mongo DB database. During the next scan by **BusRegistrator**, this destination will be ignored.

JOnix Service Client

Classes are listed by packages.

fi.metropolia.ereading:

MongoResource – this class is responsible for supplying mongoDB connections through its getClient method. Returns the instance of type MongoClient. Accessible from every point of the jOnix client.

Registration – class responsible for logging the user in and registering him. This class is a backing bean used by JSF index.xhtml facelet. Has standard getters and setters for login and password, and three utility methods – register (registers the user), login (logs the user in), and proceed, which redirects the user to the personal space if the login operation has been successful.

UserSession – session scoped managed bean used for the duration of the connection of the user. When the user connects to the application, he is automatically given an instance of UserSession, which contains the current page (where the user is in JSF application), and his login (id in jOnix). Has setters and getters, and also a logout method which invalidates the session in case the user decides to log out.

AccountSettings – managed bean which is request scoped and used from the settings.xhtml facelet. It has a field storing the addressee, UserSession instance, and fields for the passwords. Apart from getters and setters, it has multiple utility methods:

```
	public String getAddresseeShow();

This method is used to supply the set addressee name for the settings.xhtml hint.

	public void registerContact(ActionEvent event);

The method saves the organization name supplied by the user into the database and
notifies the view of JSF about it.

	public String getKey();

		and

	public void generateKey(ActionEvent event);

These two methods are used to generate a private key for a user of the
application. When the key has been generated with the key generation in
settings.xhtml, it is passed over to the key field (with the getter method) of the
facelet where it is displayed.
The class also has the methods to set and change the password for the personal
account.

OutputBusBean – backing bean used by the output.xhtml. The bean has fields for the
http-address, to which the messages, incoming to the platform should be
redirected, and http-method, by which the messages should be sent. It has a number
of methods:
```

```
public String getShowMethod();
```

and

```
public String getShowAddress();
```

These methods are used by the JSF facelet to display the hint containing the set address of the destination.

```
public void addAddress(ActionEvent event);
```

This method stores the newly added http-address into the mongoDB, thus, making it available for the rest of the application. In case of failure, specifies that the address cannot be used.

SendMessageBean – backing bean responsible for sending the messages to the local jOnix service. It works in conjunction with send.xhtml facelet, and provides a way to enter a raw xml ONIX message and to send it with http into the local jOnix service with the following method:

```
public void sendMessage(ActionEvent event);
```

This method opens a connection for the address http://localhost:8080/jonix/send, connection and using Apache http-library sends it to the local platform, where it is stored in the database.

TableBean – backing bean used by browse.xhtml. This bean provides the table of all the messages sent for the addressee mentioned for the user. It has multiple fields.

```
public List<Message> getHeaders();
```

This method uses the database to retrieve the list of headers of messages for the table built in the browse.xhtml.

```
public List<Record> getRecords();
```

This method retrieves all the products related to a particular message (header).

Message – bean which stores the information about the ONIX Message header. The fields it contains are the sender, email, note, time, and a reference to the full ONIXMessage. It also has all the required getters and setters for these fields.

Record – class representing the information about the product, which is to be displayed in the browse.xhtml. Has fields reference which is the ONIX product reference, notification type which his the ONIX notificationType, and list of product identifiers.