GARUDA

THE WAY BIOLOGY CONNECTS

Garuda SDK v1.0β Developer Document

Author: Nikos Tsorman (tsorman@sbi.jp)

Samik Ghosh (ghosh@sbi.jp)

The Systems Biology Institute

Contents

1. PURPOSE

This document provides the Java language bindings for the Garuda Protocol v1.0. The protocol itself is language agnostic and specified in the JSON format (refer Protocol document for details). The purpose of this document is to facilitate the development of Garuda-enabled gadgets in the Java programming language without the need to code for the raw protocol messages. This document outlines the use of the Java SDK to a developer who wishes to enable the Garuda Layer on their software.

What is the difference between software and a Garuda gadget?

Software that implements the Garuda Protocol (in any language like Java, Python etc.) becomes a Garuda Gadget. It can be deployed in the Garuda platform and can communicate with other Garuda gadgets. Thus, in summary, it can be said **- “*All gadgets are software, but all software are not gadgets*”**

Is Garuda Gadgets only available in Java?

Not really. As mentioned earlier, the Garuda Protocol is language agnostic and can be implemented in any language. Currently, the Java language binding is provided as an SDK as many developers are using Java as their language of choice. C++ and Python bindings for the Garuda protocol are also available.

2. INSTALLATION

In order for the backed to be used, the following four libraries are required to be added in the projects classpath. These jars are included in the bundle.

- **GarudaBackend.jar:** the Garuda backend SDK. It holds all the necessary classes to make an effortless connection to Garuda Core.
- **garuda-csr.jar:** This is the client library that handles all the connections with the Core. Is being used by the SDK.
- **jsonic-1.2.10.jar** a lightweight JSON library for JAVA.
- **log4j-1.2.16.jar** is being used to log the messages that are being sent from the Core. In order to enable this functionality, a log4j.xml file should be

added on the root of the src folder.

3. INITILIZATION OF THE BACKEND

The first step in using the Java backend for Garuda is to create an instance of the backend. For the creation of a new **`GarudaClientBackend`** instance, the following information should be passed. All of them are required. In the event that one of them is missing, an **IllegalArgumentException** will be thrown

- `gadgetUUID` : The Id under which this gadget is going to be registered to Garuda. This is a unique UUID and can be generated from here for the time being. http://www.famkruithof.net/uuid/uuidgen
- `gadgetName` : The name under which this gadget will be visible on the Garuda Platform.
- `iconPath` : The path to the icon that will be displayed on the Dashboard.
- `outFFs` : List of output File Formats. The types of files that this gadget can pass to another through Garuda Core. This list holds instances of `FileFormat` where they include the file extension e.g. **xml** and the file type e.g. **SBML**.
- `inFFs` : List of the input File Formats. The types of files this gadget can accept through the core. Similar as the outFFs list.
- `categories` : A list of all the categories that this gadget falls into.
- `providerName` : the name of the provider of this Gadget.
- `description` : a summary of the gadget and its functions.
- `screenshots` : a list of paths that point to the screenshots that will be displayed on the Dashboard.

What UUID should I use?

As mentioned above, for the current version of the Garuda Platform, developers can use a UUID generated by them using a public generator like the link provided above. However, for final release of the platform, each gadget will be provided a unique UUID by Garuda (generated on registration of a developer account) and should be in the code.

The code for initialization of the backend will look like this:

```
try {
backend = new GarudaClientBackend(GARUDA_ID, GARUDA_NAME,
ICON_PATH , outffs , inffs ,
                   CATEGORIES  , PROVIDER , DESCRIPTION ,
SCREENSHOTS ) ;
      backend.addGarudaChangeListener(this) ;
      backend.initialize() ;
      backend.registerGadgetToGaruda(LAUNCH_COMMAND);
} catch (GarudaConnectionNotInitializedException e) {
// Handle error
} catch (NetworkException e) {
// Handle error
}
catch (BackendNotInitializedException e) {
// Handle error
}
```

As you can see, after the initialization, there are 3 key calls.

1. `backend.addGarudaChangeListener(this):` The registration of the parent class of the backend to the backend as a changeListener (see next paragraph for a more extended explanation),

2. `backend.initialize():` The initialization of the backend in which the connection with the core is taking place. **NOTE:** The initialize activates the connection of the backend with the Garuda Core. Whenever the gadget wants to reconnect with the Garuda Core, it should make this call to the backend.

3. `backend.registerGadgetToGaruda(LAUNCH_COMMAND):` Registration to the core which involves passing of all the information about the gadget to the core. As the `LAUNCH_COMMAND`, it is what someone would type on the console in order to launch the application. The usage of absolute paths to the gadget is recommended.

USAGE OF THE LAUNCH_COMMAND

The launch command is a key part in building and registering a gadget on Garuda. If the developer plans to make his/her gadget available on multiple OS, it is their responsibility to detect the OS in code and provide the appropriate launch command to the register message. If the developer prefers to provide separate binaries of their gadgets for each OS, then can build separate gadgets (with separate UUIDs).

This register message `backend.registerGadgetToGaruda(LAUNCH_COMMAND)` can be deprecated in the future where the gadgets are downloaded and deployed from within the dashboard (though the Garuda Gateway). However, even in that case, the LAUNCH_COMMAND has to be provided by the gadget developer. Specific examples and details will be provided when the Garuda Protocol (and Java SDK) is updated in later versions.

The second step to do in order to make the application accept the notification from the backend is to make the class that will handle these notifications to be a PropertyChangeListener. In order to do this, you have to implement PropertyChangeListener on the class you are working in.

```
    MyGarudaClass implements PropertyChangeListener
```

After that, you will have to create a new method called:

```
@Override
    public void propertyChange(PropertyChangeEvent evt)
    {
   if ( evt instanceof GarudaBackendPropertyChangeEvent )
      {
      GarudaBackendPropertyChangeEvent garudaPropertyEvt =
  ( GarudaBackendPropertyChangeEvent ) evt ;
      . . .
      }
      else
      {
      . . .
      }
}
```

As you have noticed, the property chance event is cast to type `GarudaBackendPropertyChangeEvent`. This is done to distinguish the Garuda thrown events from any other `PropertyChange` events that this class may accept. It also helps the developer to retrieve more conveniently the objects that are being passed with the events.

4. INSIDE THE PROPERTYCHANGE METHOD

Once the backend has been initialized, property chance events will be sent from the backend to the parent class for every interaction with the core. In this section, examples will be presented for the various events and their corresponding handlers.

These examples are the code that will be inserted inside the `propertyChange` method in order for the events to be handled.

1. **Handle error messages that will be send from the Core or the backend**

```
if( garudaPropertyEvt.getPropertyName().equals(GarudaClie
ntBackend.GOT_ERRORS_PROPERTY_CHANGE_ID))
   {
         // code to handle the error message.
            String message =
   garudaPropertyEvt.getFirstProperty().toString() ;
   }
```

2. **Handle the notification for the connection of the backend to the Core.**

```
   if
   ( garudaPropertyEvt.getPropertyName().equals(GarudaCli
   entBackend.GADGET_CONNECTION_ESTABLISHED_ID))
   {
         // code to handle connection message.
   }
```

3. **Handle the notification for the disconnection from the Core.**

```
if( garudaPropertyEvt.getPropertyName().equals(GarudaC
lientBackend.CONNECTION_TERMINATED_ID))
{
        // code to handle disconnection message.
}
```

4. **Handle the case that there is no proper initialization of the Backend**

```
if( garudaPropertyEvt.getPropertyName().equals(GarudaC
lientBackend.CONNECTION_NOT_INITIALIZED_ID))
{
        //Code to handle non-initilization of the backend.
}
```

5. **Handle the case that there is an error with the registration.**

```
if( garudaPropertyEvt.getPropertyName().equals(GarudaC
lientBackend.GADGET_REGISTRATION_ERROR_ID))
{
        //Code to handle error on registration.
}
```

6. **Handle the case of a successful registration to Garuda.**

```
if( garudaPropertyEvt.getPropertyName().equals(GarudaC
lientBackend.GADGET_REGISTRERED_ID))
{
        //Code to handle the successful registration
message.
}
```

7. **Handle the response that deals with a sent data request successfully.**

```
if( garudaPropertyEvt.getPropertyName().equals(GarudaC
lientBackend.SENT_DATA_RECEIVED_RESPONSE))
{
       //Code to handle the successful registration
message.
       String message =
garudaPropertyEvt.getFirstProperty().toString() ;
}
```

8. **Handle the response that deals with a sent data request unsuccessfully.**

```
if( garudaPropertyEvt.getPropertyName().equals(GarudaClie
ntBackend.SENT_DATA_RECEIVED_RESPONSE_ERROR))
{
       //Code to handle the successful registration message.
String message
=garudaPropertyEvt.getFirstProperty().toString()
}
```

9. **Handle the arrival of incoming data from another gadget through the Core.**

```
if( garudaPropertyEvt.getPropertyName().equals(GarudaC
lientBackend.LOAD_DATA_PROPERTY_CHANGE_ID) )
{
       Gadget theOriginGadget =
(Gadget)garudaPropertyEvt.getFirstProperty() ;
       String theFilePath =
(String)garudaPropertyEvt.getSecondProperty()  ;
       // rest of the code to handle the incoming file.
}
```

10. **Handle the request for the loading of a gadget inside this gadget (plugin loading)**

```
if( garudaPropertyEvt.getPropertyName().equals(GarudaC
lientBackend.LOAD_GADGET_PROPERTY_CHANGE_ID) )
{
       Gadget loadableGadget =
(Gadget)garudaPropertyEvt.getFirstProperty() ;
       String launchPathofGadget =
(String)garudaPropertyEvt.getSecondProperty() ;
       // rest of the code to handle the loading of the
Gadget.
}
```

11. **Handle the incoming list of compatible gadgets for a previously sent request (see next section for more details)**

```
if( garudaPropertyEvt.getPropertyName().equals(Garu
daClientBackend.GOT_GADGETS_PROPERTY_CHANGE_ID ) )
{
       List<Gadget> listOfCompatibleGadgets =
backend.getCompatibleGadgetList();
       // rest of the code to handle the list of
compatible Gadgets.
}
```

5. CALLS TO CORE THROUGH THE BACKEND

Finally we will examine the calls that someone can make from the backend to the Core itself and their effects. As you have noticed during the initialization of the backend, there are 3 calls being made. These are the only ones that are being made in succession and they are always required to have a successful connection to the Core. Failure to do so, may lead to restricted functionality, or a non-connectable gadget. In more detail, these calls are:

1. `backend.addGarudaChangeListener(this) ;` This call makes the

parent class of the backend to listen to it. If absent there will be no notifications from the backend whatsoever.

2. `backend.initialize() ;` This call makes the initial connection to the core and a link is established. After this call is made, all the requests and the responses from the core are enabled. If absent there is no connectivity whatsoever.
3. `backend.registerGadgetToGaruda(LAUNCH_COMMAND);` The final call is the one that registers, even though before the registration the ability to send requests is possible and responses may be received, the true Garuda functionality is enabled only after a successful registration.

After these 3 calls, the main functionality of Garuda is enabled, so the rest of the backend calls are enabled for use. These calls and their functionality are as follows:

4. `requestForLoadableGadgets ( final String fileType , final File file )` requests from the Core the list of compatible Gadgets given the filetype (e.g. xml , sbml etc…) and the file that is to be sent to one of the compatible gadget. The file extension, is being retrieved automatically from the file itself.

5. `requestForLoadableGadgets ( final String fileType , final String fileExtension )` same as above with the difference the second argument is the file extension of the file that needs to be sent. This method can be used to determine the loadable gadget list without having a `File` object at hand.

6. `sentDataToGadget (final Gadget gadget , final File file)` sents a file to another Gadget trough the Core. The target gadget can be determined from the list of loadable gadgets retrieved from the Core.

7. `sentLoadDataResponse ( boolean result , Gadget target )` sends a response regarding the result of the loaded data request received from another gadget. The response can be either true if all went well or false in the case that the file was incompatible with this gadget. The parameter `target`, is the gadget that the core will relay the message to.

8. `sentLoadGadgetResponse ( boolean result )` sends a response back

to the Core for the result on the loading of a gadget as requested. The result should be set as true for success and false for failure of the launch.

`List<Gadget> getCompatibleGadgetList ()` gets the list of the compatible gadget that has been received from the core. Initially this list is null and is populated after a `requestForLoadableGadgets` call has been made. The backend will fire an event with id *`GOT_GADGETS_PROPERTY_CHANGE_ID`* when the list has been received from the Core (see chapter 4.11). This function always return the compatible gadget list from the last call made to `requestForLoadableGadgets`.

6. OTHER CALLS

Finally there are other calls that have to do with the management of the backend itself. The developer may use them to check on the status of the backend or to utilize the information retrieved by the core.

1. `isInitialized ()` checks if the backend is properly initialized (i.e. connected to the Core and running.
2. `stopBackend ()` force stops the backend. The connection with the Core is terminated, all the running threads are stopped and the compatible gadget list is set to null.

7. CONTACT

For any suggestions, questions or bug reports regarding the GarudaCoreClientBackend, please contact tsorman@sbi.jp, ghosh@sbi.jp