WeSAHMI SDK Description (D8)

October 31, 2007

Abstract.

WeSAHMI SDK Description is the WeSAHMI deliverable D8. It describes the application specific parts of the platform and thus assists developers in creating new interactive services for mobile devices. The document contains descriptions of application specific code in Wesahmi Server, Web Server, and Browser.

Contents

1 Introduction

The purpose of this document is to provide information about application specific parts of the Wesahmi framework described in [1]. It is aimed for developers who intend to develop new applications based on the system. The document lists all methods that contain sections that need to be modified in the `WesahmiServer` module, discusses servlet development, and application specific parts of the Browser.

2 Overview

This document is structured as follows: Section 3 identifies application specific segments inside the `WesahmiServer` module, Section 4 presents application specific functionality in servlets, and finally Section 5 discusses application specific elements of the Browser.

3 WesahmiServer

This Section presents all application specific elements in the `WesahmiServer` module. It deals with data identifiers in Section 3.1, application specific regions in classes that handle data input in Section 3.2, database encapsulation in Section 3.3, eventing service encapsulation in Section 3.4, bits of application specific code of `Controller` class in Section 3.5, and remaining application specific parts in 3.6.

3.1 Data Identifiers

Each data item that is passed through the `WesahmiServer` module has a specific data identifier attached to it. Each servlet residing on the Web Server has a set of these identifiers describing their dynamic content. The identifiers are used also as filters in the FUEGO eventing service subscriptions made for each servlet.

Furthermore, the identifiers are used as column names of tables in the module's database. The `Interpreter` also expects same identifiers to be the names of the XML elements in the messages but it can be modified to perform a transformation if necessary.

In `Controller`, identifiers of database keys are stored in String List `dbKeys_` and identifiers associated with messages that should not be stored in the database are stored in String List `dmIndicators_`.

3.2 Data Input

This Section represents the application specific implementations of methods presented by interfaces `ExtModelAPI` and `ClientAPI` They are both used to exchange data with external modules.

ExtModelAPI

The `Interpreter` class of the `WesahmiServer` module implements the `ExtModelAPI` Java Remote Method Invocation (RMI) interface. It transforms the XML messages to Arrays of application specific data containers and buffering messages from the `WesahmiServer` architecture to the `External Model`. Method `sendMessage` is presented in Table 1.

Table 1: sendMessage method of ExtModelAPI

Method Signature:	`void sendMessage(String message) throws RemoteException;`
Parameters:	`String message` XML formatted message
Return value:	`void`

Modifications:	Application specific part of this method is encapsulated in the private `processSpecialMessages` method of the same module. It is used to perform message specific tasks. The `sendMessage` uses String List `keyIdentifiers_` to identify message elements used as the database keys and String List `dmIdentifiers_` to identify elements of messages that must be sent directly to all associated Clients and not stored in the database.
	
	The general format of an input XML message is:
	
	`<ExtMessage>`
	  `<key1>value1</key1>`
	  `<key2>value2</key2>`
	  `...`
	`</ExtMessage>`

ClientAPI

The `Controller` class of the `WesahmiServer` module implements the `ClientAPI` RMI interface. It allows Servlets residing on the web server to provide input from the Client. Method `sendClientInput` is presented in Table 2.

Table 2: sendClientInput method of ClientAPI

Method Signature:	`boolean sendClientInput(Map input) throws RemoteException;`	
Parameters:	`Map input`	Client input data as a Map of String keys and Object values.
Return value:	`true` if successful, `false` otherwise	
Modifications:	The method uses String Array `dbKeys_` to identify input elements that are used as database keys. Additionally the method uses a String List `dmIndicators_` that contains key values associated with messages that must be sent directly to the `ExternalModel` using the `BufferAPI` and not be stored in the database.	

3.3 Data Storage

This Section discusses application specific implementations of methods presented by interface `Database`. In the `WesahmiServer` module they are located within the `WesahmiDB` class. Method `getData` is presented in Table 3 and method `putData` is presented in Table 4.

Table 3: getData method of Database interface

Method Signature:	`List getData(List<ModelElement> keys,`
	          `List<ModelElement> tags) throws Exception;`	

Parameters:	`keys`	A List containing database query keys as name-value pairs
	`tags`	A List containing queried database table columns, i.e. data identifiers, as names with null values

Return value:	List containing database query results as name-value pairs
Modifications:	The method is used to retrieve data from the database. The implementation of this method is thus highly dependent on the underlying database schema. It should be divided to suitable submethods, e.g. one for each distinct database operation.

Table 4: putData method of Database interface

Method Signature:	`boolean putData(List<ModelElement> keys, List<ModelElement> elements) throws Exception`
Parameters:	`keys` A List containing database query keys as name-value pairs `elements` A List containing target database table columns, i.e. data identifiers, as names with new values
Return value:	`true` if database operation was successful, `false` otherwise
Modifications:	The method is used to update (put) data in the database. The implementation of this method is thus highly dependent on the underlying database schema. It should be divided to suitable submethods, e.g. one for each distinct database operation.

3.4 Eventing Service

This Section discusses application specific implementation of a notifyIncoming! method presented by interface `fuegocore.notify.core.Notifiable`. In the `WesahmiServer` module it is implemented in `Subscriber` class. The method is presented in 5.

Table 5: notifyIncoming method of Fuego's Notifiable interface

Method Signature:	`void notifyIncoming(Notification n)`
Parameters:	`n` An instance of Notification class representing an incoming notification
Return value:	`void`
Modifications:	The method is called by the `EventingService` module to deliver notifications. It contains an application specific region of code that filters out database keys (`flightNumber`, `sdt`, and `clientID`) and a Fuego specific `type` elements from the notification before passing it on. This should be modified to resemble the implementation's requirements.

3.5 Controller

The `Controller` class of the `WesahmiServer` module has several implementation specific parts. They are listed in Table 6.

Table 6: The methods containing application specific code in Controller class

Method	**Table**
`run`	7
`processSubNew`	8
`processSubRefresh`	9
`processClient`	10
`getView`	11
`storeView`	12
`processFlightData`	13

Table 7: run method of Controller class

Method Signature:	`void run()`
Parameters:	None
Return value:	`void`

Modifications:	The method contains the main loop of the `Controller`. It has references to data identifiers used as keys for the database.
	`clientIDStr = (String) contentsTable.get("clientID");`
	`flightNumberStr = (String) contentsTable.get("flightNumber");`
	`sdtStr = (String) contentsTable.get("sdt");`
	Additionally it has references to methods that use these identifiers:
	`List<Flight> flights = new ArrayList();`
	`flights = processFlightData(contentsTable);`
	`processSubNew(sub,clientIDStr, flights);`
	`processSubRefresh(sub, contentsTable);`

Table 8: processSubNew method of Controller class

Method Signature:	`void processSubNew(fi.wesahmi.notification_service.Subscription sub,`
	`                   String clientID, List<Flight> flights)`

Parameters:	`sub` The SIP subscription to be processed. `clientID` A data identifier for the client, used as a db key `flights` A list of data storage helper classes.
Return value:	`void`

Modifications:	The method is used to process a new SIP subscription. It holds references to application specific data identifiers (`clientID`, `flightNumber`, and `sdt`) and uses them to construct an initial URL to be sent for the `Browser` as a response.

Table 9: processSubRefresh method of Controller class

Method Signature:	`void processSubRefresh(fi.wesahmi.notification_service.`
	`Subscription sub, Map contentsTable)`

Parameters:	`sub` The SIP subscription to be processed. `contentsTable` The contents received in the body of the SIP subscription
Return value:	`void`

Modifications:	The method is used to process SIP subscription refreshes. It holds references to application specific data identifiers (`clientID` and `flightNumber`) and calls application specific methods:
	`flights = processFlightData(contentsTable);`
	`client = processClient(sub, clientID, flights);`
	`storeView(newView, client);`

Table 10: processClient method of Controller class

Method Signature:	`Client processClient(fi.wesahmi.notification_service.`
	`Subscription sub, int clientID, List<Flight> flights)`

Parameters:	`sub` The SIP subscription to be processed. `clientID` A data identifier for the client, used as a db key `flights` A list of data storage helper classes.
Return value:	An instance of `Client` class that corresponds to the identified old client or a new client.
Modifications:	The method checks whether a client is old or new and returns an `Client` instance. It holds references to data identifier `clientID` and `Flight` helper classes.

Table 11: getView method of Controller class

Method Signature:	`View getView(long viewID, List<Flight> flights)`

Parameters:	`viewID` identifier for the View instance `flights` list of `Flight` instances associated with the `View`
Return value:	Instance of `View` that matches the parameters.
Modifications:	The method retrieves a `View` instance that matches the given parameters. It holds references to `Flight` helper class.

Table 12: storeView method of Controller class

Method Signature:	`boolean storeView(View view, Client client)`
Parameters:	`view` Instance of `View` to store. `client` Instance of `Client` to associate with the view.
Return value:	`void`
Modifications:	The method stores view if it is a new one and associates the given client with it. It holds references to application specific `Flight` class and calls application specific `getView` method. `View curView = getView(view.getID(),(List<Flight>)view.getFlights());`

Table 13: processFlightData method of Controller class

Method Signature:	`List processFlightData(Map contentsTable)`
Parameters:	`contentsTable` Contents of the SIP subscription body.
Return value:	Returns a list of `Flight` instances containing flight specific data found in the contentsTable.
Modifications:	The method retrieves flight specific data from the SIP subscription contents. It is fully application specific method.

3.6 Other Application Specific Parts

This Section discusses other application specific regions of the `WesahmiServer` implementation. `View` class has one method, `initNotification`, that contains application specific code. It is presented in Table 14. Furthermore, the `WesahmiServer` module contains classes `RexGenerator` and `Flight` that are almost entirely application specific.

Table 14: initNotification method of View class

Method Signature:	`void initNotification(Client client)`
Parameters:	`client` The `Client` instance to be notified of the View's initial contents.
Return value:	`void`
Modifications:	The method sends initial notification to the target client. It holds references to `Flight` class.

4 Web Server

Web server is completely application specific component in the Wesahmi system, because every application has its own Web pages. Since it is completely separate component, it can be basically implemented on any technology. In the Wesahmi project, we used Apache Tomcat Servlet container. In addition to ordinary Web server, the Wesahmi Web server must add DRML elements into the Web pages and send client input via Java RMI to Wesahmi server.

4.1 Denoting updateble data

The DRML elements identify what data on a Web page can be updated by Wesahmi system. A browser automatically orders data updates from the Wesahmi server according to the DRML declarations. The DRML documents are added into *head* section of an XHTML document. Below is a simplified XHTML document example, which is created on Web server. In lines 3-7 resides a DRML document. It specifies on lines 4-6 that elements which have IDs *edt*, *gate*, and *boardingTime* can be updated by Wesahmi server. The elements are located in lines 13, 15, and 16, respectively. The corresponding data on Wesahmi server are labeled with the same IDs as in the DRML document.

```
1: <html xmlns="http://www.w3.org/1999/xhtml">
2:   <head>
3:     <dref xmlns="http://www.x-smiles.org/ns/drml">
4:       <item>edt</item>
5:       <item>gate</item>
6:       <item>boardingTime</item>
7:     </dref>
8:   </head>
9:   <body>
10:     <h1>Boarding pass</h1>
11:     <div>
12:       <table>
13:         <tr><td>Flight: London - Helsinki<span id="edt">16:15</span></td></tr>
14:         <tr><td>Passenger:</td><td>Joe Carmichael</td></tr>
15:         <tr><td>Gate:</td><td><span id="gate">23</span></td></tr>
16:         <tr><td>Boarding:</td><td><span id="boardingTime">15:45</span></td></tr>
17:       </table>
18:     </div>
19:   </body>
20: </html>
```

4.2 Sending client input

User submission of a form is received as a set of name-value pairs on the Web server. These pairs can be sent further to the Wesahmi server if required by the application. They are submitted via ClientAPI. The ClientAPI is built on top of JAVA RMI technology. The ClientAPI defines one method:

```
boolean sendClientInput(Map input) throws RemoteException;
```

The name-value pairs are stored in the *Map* object, which is then sent to the Wesahmi server. An example of sending user input via ClientAPI from a Servlet is listed below. That must be done before browser's page request is returned. In the example, the number of traveler's baggages and a seat she has selected is stored in the *Map*. The *Map* is submitted to the Wesahmi server which has IP address 10.1.0.107 in this example.

```
Map hash = new HashMap();
hash.put("baggages", baggage);
hash.put("clientSeat", seat);

try {
    ClientAPI capi = (ClientAPI)Naming.lookup("rmi://10.1.0.107/ClientAPI");
    capi.sendClientInput(hash);
}
catch (Exception e) {
    e.printStackTrace();
}
```

5 Browser

The browser does not need any application specific components. It only interpreters DRML documents and REX events. In addition, it is integrated into SIP client stack. These all are system specific features, so the X-Smiles browser can be used as such for any application implemented on the Wesahmi system.

X-Smiles must be configured before used for the Wesahmi system. It is done in X-Smiles configuration:

1. Run X-Smiles.
2. Open *Edit* → *Configuration*
3. From *Main* page see *Wesahmi system*
4. Select *Enable the service* and set *SIP Address*
5. Restart X-Smiles

When you have finished the configuration start first the client daemon and then X-Smiles.

In the Wesahmi demo system, we integrated an electronic flight ticket into the browser. It was used to hold the identity information (i.e., clientID, flight number, and departure time) for the server. Also this data could be set via X-Smiles configuration as explained above. This is naturally an application specific component on the browser, but, it is noteworthy, that identification could be done for instance by log in to the Web server to avoid any specific components on client side. In other words, that is not compulsory configuration setting for the browser.

References

[1] WeSAHMI Software Architecture and Interface Description. Technical report, WeSAHMI Project, 2007.