Domain Specific Mashup Platform Generation Framework

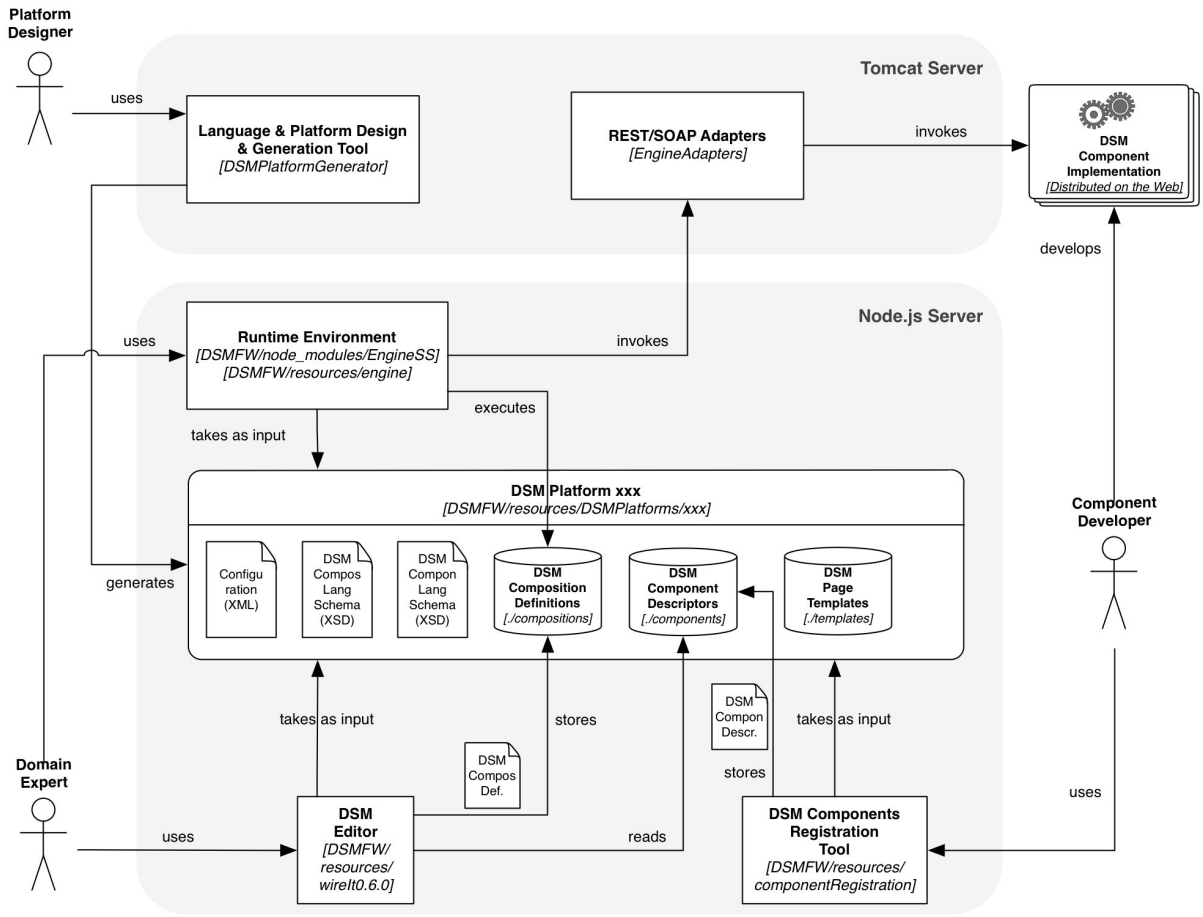
Project technical documentation

1. Premise

This document provide an overall description of the project. Most significant files of the sub-project composing it are introduced and a wider picture shows how these different parts relate to each other. Many details about the project and to its main algorithms are discussed in the paper ***Domain-Specific Mashup Platforms for End Users: A Conceptual Development Approach***; refer to it to find more details.

2. Introduction

The whole code base is distributed over several sub-projects. In particular, there are some Java web projects to be deployed on a Java-enabled application server (Tomcat in our case) and a node.js project. The following figure shows the high-level architecture of the system. For each architectural component it is provided (within square brackets) the reference to its location in the code base (i.e., the source folder contained in the project package), indicating which project/project folder contains the implementation of that specific component.



Next, we briefly discuss how the system works introducing the various architectural components, which will be detailed in the next sections.

After the platform designer has analyzed the target domain and he/she has a clear idea of the features required by the domain specific mashup (DSM) platform (we are assuming this phase as done and we do not discuss it), the first step he/she has to go through is the design and generation of the DSM platform. The platform designer will use for this purpose the *Language & Platform Design & Generation Tool* (see Section DSMPlatformGeneration). Through the tool's interface the designer can select the set of features he/she wants to be supported by the target DSM platform. The generation process will then create a *DSM Platform xxx* (i.e., a folder with a set of documents and sub-folders), where xxx will be replaced by an ID identifying the just generated DSM platform (see Section DSMFW). Clearly, it will be generated one folder for each DSM platform. The content of this folder is then used by the *Runtime Environment* and the *DSM Editor* to adapt their aspect and functionalities so that to be able to support the DSM features selected by the platform designer during the DSM platform design phase (these components take as input the URL of the DSM Platform specific folder they need to access). The domain expert uses the *DSM Editor* to create new mashups and the *Runtime Environment* to execute the mashups stored in the *DSM Composition Definition* repository of the specific *DSM Platform* he/she is using. Also the *DSM Component*

Registration Tool takes the folder URL as input to register the DSM component descriptors (developed by a component developer) in the platform where they have to be used. While the descriptors must be registered and stored within the system, their implementation can be deployed/run wherever on the Web.
During the execution of a mashup the Runtime Environment manages all the process control flow and data flow, including the invocation of the components constituting the mashup. The actual DSM component invocation goes through a set of adapters (*REST/SOAP Adapters* - see Section EngineAdapters), which act as mediators between the RT environement and the *DSM Components Implementations* (e.g., managing protocol and format specific issues).

Two more pieces of the system are provided in the project package and described below. One is a web application project including some REST/SOAP services needed by the example DSM platform already installed in the DSMFW. This example DSM platform target the research evaluation domain (see Section ResEvalServices).
The other one is a set of folders including node.js and some node modules needed to correctly run the system (see Section node & node_modules).

Next section provides the very fast and zero-effort way to deploy the whole project starting from the files provided in the `deployable` folder in the project package. The the following sections will describe each sub-project providing a description of their purpose, their internal structure and how to deploy and/or configure them, so that it is possible to extend/modify the project which is open source. Following the indications provided in these sections it is possible to easily deploy the whole system starting from the code given in the `sources` folder of the project package.
Note: all the folders contained in the `sources` folder are Eclipse projects, therefore, they can be directly imported both in Eclipse or NetBeans.

3. Zero-effort project setup/deployment

We assume that a Tomcat installation is present on the computer (version 6.0.32 has been tested, but all the following versions should work as well).
NOTE: the Tomcat port must be set to 8070 (instead of the default 8080). These port can be changed to any number (is has been changed from 8080 cause of conflict with node.js). In case one wants to change it i must be updated in the `EngineSS.js` file, plus in the components and composition descriptors of the example platform for research evaluation, in case one wants to use it - see ResEvalServices).

The project package we refer next is the folder (named DSM FW Pack) containing all the files related to the project.

Deploying the system on a Mac OS or Unix-based OS in 2 steps

a. Copy the WAR files present in *DSM FW Pack/deployable/TomcatWARs* in the *webapp* folder of your Tomcat installation

b. Copy the `DSMFW`, `node` and `node_modules` folders under *DSM FW Pack/deployable/NodeFolders* in the same filesystem folder, i.e., the root folder ("/").

Deploying the system on any other operating system

Deployment on other systems may require just an additional operation. The path where the `DSMFW` folder is located is used by the class `GenerateDSMLanguagesAndPlatform.java` of the DSMPlatformGneration project. In case the `DSMFW` folder is not located at the path "/" its path must be updated in this Java class so that the system can work (e.g., in Windows this path could be changed to "C:"). See the "Deployment" section of DSMFW for more details.

Once modified this class the DSMPlatformGneration project must be exported again as WAR and deployed on Tomcat. The other deployement steps remain exactly the same

4. DSMPlatformGeneration

Deployment: Apache Tomcat

Purpose: Provides platform designer with an interface to design their target DSM platform (i.e., to select which features they want the platform to support). Once the selection is done it generates the DSM Platform folder (along with language definitions etc.) for the new platform.

Structure:

a. `src`

i. `GenerateDSMLanguagesAndPlatform.java`: This is a Java servlet invoked by the `featureSelectionUI.html` interface (see below). It generates the DSM Platform folder for the platform being developed. Then it generates a suitable composition language schema definition (an XSD) and a component description language schema definition (an XSD); these languages support all and only the features selected during the platform design done through the above mentioned interface. In addition, one sub-folder for storing the future composition definitions and one for storing the component descriptors specific of this DSM platform are created. Finally, it is created a platform configuration document (an XML) containing the list of selected/supported features, the URLs of the language schema XSDs and the URLs of the composition and component descriptors repositories (folders) just created. The DSM Platform folder is then deployed into the `DSMFW/resources/DSMPlatforms` folder and named with a unique platform ID.

ii. `interfaceGenerationMain.java`: this Java main class simply allows one to generate the `featureSelectionUI.html` automatically taking as input the whole features list encoded in the `features.xml` file contained in the `schema` folder (see below).

b. **WebContent**

i. `featureSelectionUI.html`: the HTML interface showing the list of all the features supported by the system that the platform designer can select from to design his/her target DSM platform.

ii. `tooltip`: JS library used to show features description as tooltip on the mouseover in the `featureSelectionUI.html` page.

iii. schemas

1. `CompositionSchema.xsd`: complete XSD schema equivalent to the Unified Mashup Model. It is used as basis to build the DSM composition languages starting from a list of selected features (see the paper).
2. `ComponentSchema.xsd`: complete XSD schema equivalent to the Unified Mashup Model. It is used as basis to build the DSM composition languages starting from a list of selected features (see the paper).
3. `features.xml`: complete list of the descriptions of the supported features (see the paper for details about features description format)

5. EngineAdapters

Deployment: Apache Tomcat

Purpose: Provides the REST and SOAP adapters used by the runtime engine to interact with the components implemented through these technologies. Adapters are provided as REST services and must be invoked through a POST message. The adapters are are modules that can be implemented with any technology and deployed anywhere. As said now they are implemented in Java and must be deployed in a Web server like Tomcat. They could also be implemented directly in the node.js server (DSMFW, see below).

Structure:

a. **src**

i. `engine.adapters`

1. `REST.java`: Used to connect to mashup components

implemented as REST services. Supports only GET and POST operations. Check the code and code-comments for more details.

2. `SOAP.java`: Used to connect to mashup components implemented as SOAP services. Asynchronous interactions are not suitably supported yet. Check the code and code-comments for more details.

ii. `engine.utils`

1. `Conversion.java`: Utility for stream-to-string conversion and XML-to-String and vv. conversions.

6. ResEvalServices

Deployment: Apache Tomcat

Purpose: Test services for the example DSM platform for the research evaluation domain. This example platform is pre-deployed in the node.js server (DSMFW/resources/DSMPlatforms/531084) where are included both components descriptors and compositions definitions.

Structure:

a. src

i. `reseval.ResEvalServices`: Includes a set of REST services implementing the logic for the following mashup components: DISI Researchers, Researcher By Name, Scholar, Publications Per Year. All services produce and consume JSON data. These “service components” (each implemented by one of the rest services in this class) are described by according components descriptors already included in the folder DSMFW/resources/DSMPlatforms/531084/components.

7. DSMFW

Deployment: Just copy this folder in any location on the file system. This location must be defined in the `GenerateDSMLanguages.java` class (`DSMFWFolderURI` field - `DSMPlatformGenerator` project). Currently it is set to the root folder (“/”), therefore, everything will work correctly on a Mac OS system if the folder is copied in the filesystem root. On different operating systems or in case the folder is copied to another location, the `DSMFWFolderURI` field of the `GenerateDSMLanguages.java` class must be updated accordingly.

Purpose: This web app to be run in Node.js represents the core of the system, allowing to use the generated platforms to create and execute DSM mashups. It contains the generated DSM platforms, the runtime environment (spit into server-side and client-side engines), the mashup editor and also a mashup component registration tool. In addition, beeing this a Node.js web app, it also includes the server definition managing all the request for the various part of the application (defined in the `dispatchingServer.js` file).

Structure:

a. **dispatchingServer.js**: this defines how the server manages any incoming request. All the static resources must be places in the resources folder. All the files in this folder are accessible with a GET request to their path (i.e., DSMFW/resources/{path}/{resourceFileName}) For all the request to non-static contents it must be defined an according handler (check already implemented handlers in the code). The main operations this class does at loading is preparing the server to listen to request and instantiating a server-side engine object (defined as a node.js module in DSMFW/node_modules/EngineSS.js).

b. **node_modules**

   i. `EngineSS.js`: This is the server-side engine counterpart manages all the execution of mashup compositions. All the data passing, components invocations (thorugh the adapters), etc. It also keeps the communications with the client-side engine through web sockets.

   ii. `JSAdapter.js`: This is used to manage the responses coming from the possible JavaScript components running on the client side. These communications are managed by the `EngineSS.js` through the client-server communications web sockets.

   iii. `CMPUtils.js`: simple utility class used throughout the `EngineSS.js` and `JSAdapter.js` for converting XML-2-string and vv. and JSON-2-string and vv.

c. **resources**

   i. `engine`

      1. `engineCS.js`: This is the client-side engine counter part. Its main purpose is instantiate the possible JavaScript mashup components and manage the communications with the server-side engine. In addition, this JavaScript class is included in the starting pages that are created when a mashup composition is saved. This starting page is used to start the execution of the related mashup composition and, if present, includes the JS

components of the composition itself.

2. `adapters`
    a. `JSAdapter.js`: This is the adapter used by client-side engine to interact with the possible JS components running in the starting page. The engine does not communicate directly with the components but uses adapters so that it is simple to manage in the future new technologies. E.g., an adapter for W3C Widgets should be added to manage such kind of UI components.

ii. `DSMPlatforms`
    1. `{platformId}`
        a. `configuration.xml`: This file contains the configuration of the generated platform identified by the id `platformId`. It contains the list of the features selected during the DSM platform design, the URLs of the DSM composition and components descriptor languages XSDs, the URLs of the components, composition and templates repositories for this platform (i.e., the `component`, `compositions` and `templates` folders contained in the `{platformId}` folder) and the domain syntax definition file URL. All this information are use both by the engine and the editor to adapt their functions, aspect and behaviour to the given DSM platform.
        b. `DSMCompositionLanguage.xsd`: This is the XSD schema definition for the composition language for this specific platform (i.e., ruling over the XML definition of the mashups).
        c. `DSMComponentDescriptionLanguage.xsd`: This is the XSD schema definition for the component description language for this specific platform (i.e., ruling over the XML descriptions of the mashups components; e.g., REST service, JS components, etc.).
        d. `domainSyntax.xml`: This file may be empty or contain the definition of the graphical syntax for some graphical constructs of the editor, e.g., the split or join constructs. Specific mashup components have their own graphical syntax, which is defined directly in their XML descriptors, through the `domainSyntax` attribute.
        e. `components`: This contains all the mashup components descriptors XMLs. In addition this can be used also to host

other components-related files if needed (e.g., the J files implementing JS components).

f. `compositions`: This contains all the mashup compositions definitions XMLs created with this platform. In addition, for each mashup composition XML there is a related HTML constituting the starting page (as discussed above in 6.c.i.1).

g. `templates`: This contains the templates that can be used in this platform to create the starting pages for the compositions. Each template is an HTML page containing whatever plus a set of viewports (i.e., placeholders) to be used at runtime to host the composition's UI components (if any). The viewport can be simple DIV elements idenfied by an ID starting with "_viewport_" and followed by any string.

iii. `wireIt0.6.0`: this is the folder containing all the mashup editor. The editor is an extend and personalized version of the editor provided with the open source WireIt library.

iv. `componentRegistration`: This folder contains some simple HTML page where a component developer can register a new component into a given DSM platform. Simply, the developer uploads the component descriptor file and indicates the URL identifying the target DSM platform. The page will send this file to a service that will store the file in the correct `components` folder for the specific target platform.

8. node & node_modules

Deployment: Copy the node and node_modules folders in the filesystem under the same path. The easiest option is to install them at the same path as the DSMFW folder (e.g., both node, node_modules and DSFM as folders in the root "/"). The important point is that the node and node_modules folders MUST be either in the same folder or ancestors in the file system with respect to the DSMFW folder.

Purpose: They constitute the core of node.js. Must be there so that any node.js server can run.

Structure:

a. **node:** This folder contains core files of node.js

b. **node_modules:** This folder contains core modules used by node.js. Moreover, it

contains some additional node.js modules that will be shared by all the node.js web apps deployed on the machine. Our DSMFW web app uses some of these shared modules (e.g., jquery, ws, httpdispatcher), therefore it is essential to use node_modules folder provided within the project package.