MOOC Personalization application : developer manual

LIRIS - SILEX

September 7, 2014

Contents

Chapter 1

Introduction

This manual delivers informations for the developers who would like to bring modifications and improvements to the MOOC personalization core application. Its general organization is described, as well as descriptions of its main mechanisms. Lots of informations about the way processes are run in the application can also be found directly in the code comments.

The application code is available on github `https://github.com/fclerc/PersoInterfaces`, and a demo version can be found at `http://elearning-dev.univ-lyon1.fr/persua2mooc/`.

Before reading this manual, we strongly recommend you to read the user manual, which describes the model used in order to realize personalization. It also provides an overview of the interfaces and their functionalities, very helpful to understand the code.

There are still some TODOs in the application code, which are things I didn't have enough time to do, but that could be treated to add more functionalities to the application or improve the user experience. We will also provide explanations about what could be the next steps in the development of the application all along this report.

Chapter 2

Technologies and installation

Data is stored in XML for all the elements of the PERSUA2$_{MOOC}$ model, and in json for the other elements required or built in the application. On the server side we only use PHP. But, as most of the interfaces require quick interactions with the user, we used JavaScript a lot, together with the jQuery framework. We also use D3.js (data-driven documents) in order to display statistics. CSS developments were sometimes accelerated with Bootstrap, but most of the time we realized specific stylesheets, corresponding to our specific needs on the interfaces.

To install this application and realize tests locally, you only need a PHP server and a web browser. To integrate the application to a MOOC platform, things are a bit more complicated and described in chapter 9.

Chapter 3

Organization of the application

The application is organized in several folders :

- The **root** folder contains the PHP files used to display the interfaces of the application; they will be detailed all along this manual.
- The **css**, **img**, **fonts** and **js** folders are self-understanding, and called by the various interfaces of the root folder. In this application we use jQuery, bootstrap and the D3.js (Data-Driven Documents) library. The other js and css files were created during development to answer our specific needs.
- The **data** folder contains all the data required to realize personalization and to test pedagogical strategies. In this folder :
  - **learners** contains one folder per learner (the name of the folder is the id of the learner), each one containing its profile and the html files containing the lists of activities that have already been generated for this learner during the MOOC.
  - **resources** contains an XML file describing the resources available in the MOOC.
  - **models** contains the schemas used for the learner profile model and contexts models, and the XML file describing the activities available on the MOOC platform (pedagogical properties).
  - **info** contains json files storing informations extracted from the models. These informations will be displayed to the user to provide him more information about what he has to do.
  - **teacher** contains all the elements defined by the pedagogical team of the moocs : strategies and sequence contexts to enable personalization, and live context and profiles to realize tests. It also contains a file associating each sequence of the MOOC with a strategy and a sequence context.
- The **PHPhelpers** folder contains several PHP files that are used in the interfaces, or that need to be called manually (in order to generate documentation for example). Some of them are pre-treatments realized in PHP in order to simplify the work in JavaScript. You can change their content, and load it in your browser or call it directly in command line to execute.

- The **resources** folder contains files used for the general working of the application. It currently contains only one file, used to generate the homepage.
- The **translation** folder will be described in section 8.1, and currently enables to translate in English or French the content of the pages.

Chapter 4

HomePage

The homepage enables the user to go on the different interfaces and select the files he wants to use. The forms used to lead him to the different interfaces are almost always the same : a title, a short explanation, a select input, 4 buttons... Thus, we used the same piece of code to generate these forms, and all the data concerning what has to be written is contained in one json file: resources/filePageData.json. This file also contains informations that are hidden to the user in the forms, like the name of the PHP file that has to be called for a specific section (more details are given in the code of index.php).

Chapter 5

XML data and modification interface

5.1 XML files organization and conventions

5.1.1 XML Schema

All the data used in this application for personalization are stored in XML : pedagogical properties, learner profiles, contexts, resources and rules. All elements but the rules are described in XML Schema. These schemas are enough documented to understand the role of each element they contain, and the user manual provides information about the role played by all these models. There are just two differences between the models used in PERSUA2$_{MOOC}$ and the way they are implemented. Indeed, for some indicators like this one concerning the result to a quiz, Quiz2 :

```
<knowledgeUnit id="LP46">
  <name id="LP47" fixed="true">Quiz2</name>
  <value id="LP48">70</value>
</knowledgeUnit>
```

all profiles will contain the same value for the 'name' tag, which is 'Quiz2', whereas the 'value' will change from one to another. For this reason, we add the attribute fixed = 'true' for the tags that will contain the same value in all profiles. The second difference is that we need to add IDs to all the tags in a XML file, for manipulation reasons in our code (it makes things much easier) and for modelization reasons (it enables to make references to elements of the profile for example). For this reason, we wrote a PHP class XMLIdAdder, that automatically adds ids to all tags in a XML file, respecting the form the user wishes (look at the code to see all the options available; you can make some tests with the launcher script idAdderRunner.php).

The OKEP model (pedagogical properties) of the platform is realized thanks to the meta-model AKEPI (also expressed in XMLSchema).

5.1.2 Pedagogical strategy

The strategy is the only element not described in XML Schema currently, but its structure is very simple and directly understandable through examples. Furthermore, developers are free to change the way rules are stored and handled in the application (in fact the model PERSUA2$_{MOOC}$ doesn't care about the way rules are stored in an implementation).

In this implementation, the general structure of a strategy is the following :

```
<strategyRules>
  <exploitedProfile>
    data/teacher/profiles/profile1.xml
  </exploitedProfile>
  <exploitedContext>
    data/teacher/liveContexts/liveContext1.xml
  </exploitedContext>
  <pedagogicalProperties>
    data/models/foveaPedagogicalProperties.xml
  </pedagogicalProperties>
  <rule id="R1">...</rule>
  <rule id="R2">...</rule>
</strategyRules>
```

Here we decided to store the path to a few files used for the definition of the strategy on the interface. This is not necessary, as these paths could be stored somewhere else in the code. The most important part is the list of rules. Here is the structure of a rule :

```
<rule id="R1">
  <priority>3</priority>
  <if>...</if>
  <then>...</then>
  <else>...</else>
</rule>
```

The 'if' part is constituted of conditions, each condition containing one or more constraints, possibly combined together with 'and' and 'or' :

```
<constraint>
  <indicator>LP25</indicator>
  <operator>=</operator>
  <referencevalue>absent</referencevalue>
</constraint>
```

The value contained in 'indicator' is the id given to the corresponding element in the profiles or in the live contexts.

Complex conditions can be defined by combining several constraints with 'or' and 'and' tags :

```
<or>
  <and>
    <constraint>...</constraint>
    <constraint>...</constraint>
  </and>
  <constraint>...</constraint>
</or>
```

In our implementation, <and> and <or> tags can only contain two conditions (you can't have a 'and' containing 3 constraints for example).

The <then> and <else> contain the same structure of elements, describing the activities available on the platform :

```
<then>
  <activities>
    <activity>
      <typeofactivity>A002</typeofactivity>
      <parameters>
        <parameter>
          <id>P005</id>
          <value>Twitter</value>
        </parameter>
      </parameters>
    </activity>
  </activities>
</then>
```

typeofactivity and id (in parameter) are references to the ids given in the pedagogicalProperties file. Of course, several parameters and activities can be defined at the same time.

5.2 Documentation generation

In order to provide information to the user about the models he is using in the application, some documentation is generated thanks to the XMLSchemas, the resources file and the pedagogical properties file. This generation is made in PHP (except for pedagogical properties), as a pretreatment before displaying these informations in JavaScript in the interfaces. The PHP scripts used to generate these informations are:

- generateJSONFromXMLSchema.php : extracts as much documentation as possible from a XML Schema file : type of data allowed, and annotations given to explain the meaning of the elements. It is used for learner profile and contexts. The results are stored in json files, in data/infos
- getDataFromResourceStructure.php : extracts all the values given to the different parameters in the resources file. It creates a json file, providing the list of resources names, the list of categories used for the resources,...

On the interfaces, these informations are displayed thanks to functions contained in the file scaleDisplayers.js

5.3 Files manipulation

In order to store all the files manipulated in this application, two PHPhelpers are used :

- saveXMLDocument.php, which is called through XMLHttpRequests in javascript, enables to save files very simply

- fileHandler.php, also called through XMLHttpRequests in javascript, enables to perform more 'complex' tasks on files : removing, renaming, creating, duplicating.

When the user wants to create a new file, for example a new profile, he has two choices. The first one is to duplicate an already existing file. The other choice is to create a new 'empty' file, already containing the basic structure required to enable the interface to manipulate it and allow modifications. Thus you will need to have an empty profile somewhere in the application, which is duplicated each time the user wants to create a new file. To ease the creation of this empty file, we wrote a script createEmptyFromXML.html (this is made in javascript). If you already have a learner profile in your application, you can just run this script once, and it will create a new XML, empty.xml, where all leaf values have been removed (but the exactly same structure still exists). It just doesn't remove the values of elements having the attribute 'fixed' = true.

5.4 Modification and display of XML files

On some interfaces, like the profile modification, the user needs to modify the values of a XML file. On other interfaces, like the strategy definition on the left part, the user just needs to visualize the file, and click on the elements of this file to define his rules. Thus, a generic interface has been created in order to answer these needs : xmlManipulator.js. It contains a function manipulateXML() that can be called by other pages to display the file (with simple arguments like the path to the xml file and the id of the that will contain the result), by simply using lists and indentation to correspond to the tree structure of a XML file. The treated XML can be of any depth, as the algorithm to display it proceeds recursively. Two modes are available with this displayed tree :

- 'modify' to edit the values in the file (but not the tag structure)
- 'select' : when the user clicks on a tag name, a JavaScript event 'leafValueReading' is triggered, containing some informations about the indicator that was clicked on and the html from where the file is displayed (this is useful when several files are displayed on the same page, and we need to know from which div of the page events are triggered). This is why an argument 'reader' must be passed to the manipulateXML() function : it is the object (indicated by its id) which will trigger, receive and treat these events, and thus manage all the interactions with the user. Modification of values are not possible in this mode.

The modify mode is used in fileValuesModification.php interface. The select one is used in the rules_definition.php.

Concerning the resources modification (resourcesModification.php and resourcesManipulator.js), the same principle is used in order to display the tree of resources defined. But, as this file is pretty big and complicated, we decided not to use exactly the same interface, which would not be very understandable for the user. Thus we took the same code structure, but parameters are modified thanks to a form.

Chapter 6

Strategies and association interfaces

6.1 Strategies

This interface is the most complex one, and managed by the file rules_definition.php. We will describe the main organization of the code to display the data and interact with user, more details are directly available in the code.

First the profile and live context XML files are displayed in different containers (html divs), in 'select' mode. Then, on the right part, the activities are displayed together with their parameters, using the pedagogical properties file. When clicking on an activity or a parameter name, an event is triggered, with the same name as the event triggered by XMLManipulator : 'leafValueReading'. These events are triggered and received by the same 'reader' as above (in our case it is the #Rules div). While the script displays the activities and parameters, it also stores (temporarily) informations about these elements, to then be able to build the rules and display informations to the user. Once everything is displayed, the interactions with the user can begin to have him create a rule. When creating a new rule, 14 operations and states are considered :

- 0: Creation of a new rule
- 1: Selection of indicator
- 2: Selection of operator
- 3: referenceValue is given
- 4: Selection of activity for then
- 5: Selection of parameter for then
- 6: value of parameter is given for then
- 7: Selection of activity for else
- 8: Selection of parameter for else
- 9: value of parameter is given for else

- 10: Priority for the rule is given
- 11: Rule is ready to be edited
- 12: Rule is saved
- 13: chosing 'AND' or 'OR' (should be another number, but I only added it in the end)

If the user follows the instructions given in the rule, the whole process is almost completely realized in his normal way, going from operation i to i + 1. But lots of buttons are available in the interface, enabling him to go from any operation i to almost any other operation j. Everytime such a button present in the rule is clicked on, the same event 'leafValueReading' is triggered, and received by the same reader. All these 'leafValueReading' events are then treated in the code to react to the user interactions. The line enabling to receive these events is :

```
$("#Rules").on("leafValueReading", function(event, value, id, container)
```

Then, depending on the current state of the rule being defined, the action wanted by the user is realized (or just ignored if this action is nonsense, for example if he clicks on an activity whereas an indicator is expected).

These main parts of the code enable to display all the elements required by the user, and to treat his actions to define rules.

On this interface, the script closeEditorWarning.js is also called : it creates an alert when the user wants to leave the page, inciting him to save his data if he hasn't already made so. This file is also called in resourcesModification.php.

6.2 Association of strategies and sequence contexts to MOOC sequences

The PHP script enabling this association is sequence_association.php. It is composed of two main parts : the first one displays a big form with lots of selects in order to enable the user to realize the associations. This form's action page is also sequence_association.php, thus the second part of this file treats the data sent by the form to store it in a json file : data/teacher/sequence_association.json.

Chapter 7

Activities generation

A class has been created in order to generate the lists of activites for the learners : ActivitiesGenerator. When creating an object of this class, the only argument is the path to the strategy file used to create the lists. Then, the generate() function is called by indicating all the other required elements : profile and contexts. Thus, the same object can be used to generate the lists of activities for several learners.

To generate the list of activities, this class uses two other classes. On each rule, the first to be used is ConditionChecker, which enables to check any condition contained in the 'if' part of the rule. The second is 'ConsequenceGenerator', its goal is, starting from the 'then' or 'else' part of a rule, to generate the recommendations corresponding to what is described in XML (using for example filters in order to chose among all the available resources which ones have to be displayed to the user).

Once all rules have been treated and a first list of activities is obtained (for each activity to be realized by the learner several informations are given, like the text to display to him, or the duration of this activity), a final function, displayActivities(), is called. It has 2 roles : treating the sequence constraint in order to limit the number of activities and the total time it would theoretically take the learner to do it; and display the activities to the learner in the form of a list.

Chapter 8

Other functionalities

8.1 Translation of the application

Here is the process to follow for the translation of the application using JavaScript. Python is also required to be available in command line locally to convert some files. We use the tools from `localeplanet.com`. Like many other translation mechanisms, it is based on files in the form 'key' => 'value'. All files are in the 'translation' folder. The js files in this folder directly come from the website.

- Identify the strings that you need to translate. Initially, these strings are in english (when these are just a few words, like for the indicators names), or in a special form used to have unique keys. For example, on the homepage (which I also call 'fileChoice' page), the title in the html file is 'fileChoice.h1', and the paragraph under is 'fileChoice.instructions'.
- Use the file 'translate.pot' and indicate the strings (keys) you want to be translated (anywhere you want in this file). This string must be in 'msgid' and 'msgstr' must be void here[1]. If you use keys like 'fileChoice.h1', try to respect the conventions used, for example using 'fileChoice' as the beginning of your key for each string contained in this page.
- Use the software poedit (`http://poedit.net/`) to open the .po file of your choice, fr.po or en.po.
- With poedit, update the .po with translate.pot
- Translate the newly displayed keys with poedit and save.
- Open a command line in the /tools folder, and execute : **python po2json.py ../fr.po** (or en.po...). It converts the .po file into a json file, which is then used by javascript to realize the translation.

[1] This step can be quite long...to go a little bit faster, you can open a new empty file in your favorite text editor, simply make a list (1 string = 1 line) of all the strings that have to be translated, and use the following regexp (on Windows) to have it displayed in the right 'msgid...msgstr...' form : search (.+)\r and replace with msgid "$1"\rmsgstr ""\r\r

- Finally, to make your string be translated, you must convert it with the translation object '_' : do *text = _(text)* in js. To do so, 2 solutions : 1. if your text is directly displayed in javascript : simply display '_(text)' 2. if your string is in your html page (possibly generated in PHP) : use the class 'toTranslate' for the element containing this text, there is a script on each page that selects all the 'toTranslate' elements, and replaces their content with translation.

Two languages are currently available, French and English. On homepage, 2 flags are present for the user to chose the lang he wants. By default it is french, and the variable containing the lang used by user is contained in SESSION variables. The file langFinder.php enables to get this variable, or take into account a language change from the user.

Of course, the en.po file doesn't translate most of the strings used in the application, as these are already in English.

8.2 Statistics generation

The process to generate the statistics is simple, and is a mix between PHP (to realize a pre-treatment on the data), and js (with D3.js to display the charts) :

- We go through all the learner profiles (in the data/learner folders) to see, for each indicator, all the values taken.
- Still in PHP, for each indicator, we aggregate the data (for example for the 'sex' indicator, count the number of 'F' and 'M')
- In js, we use this data and functions adapted from D3.js to display the charts.

Chapter 9

Integration to a MOOC platform

What we described here is the core application used in order to realize personalization. However, in order to indeed perform personalization, a certain amount of work has to be realized to integrate it to a MOOC platform. Here is a list showing the main things to realize in order to have this system interact with your MOOC platform and indeed provide personalization to the learner :

- Adapt all the models to your platform, especially for the pedagogical properties file, which describes all the functionalities available on the platform.
- Set algorithms in order to compute the values for the indicators in profiles for each learner. These values have to be inserted directly in the XML files describing the students.
- Set a mechanism able to compute the live context and provide it to the list activities generator (this is not necessary, you might decide to have your rules only based on the learners profiles...but using this notion of live context could really enable you to provide the best personalizations to the learners).
- Set an interface in order to enable the generated lists of activities to be displayed to the learners directly when they're connected to the platform.

Before applying personnalization to a particular MOOC on the platform, some other things will have to be realized, like adapting the learner profile to the MOOC in particular. The list of all resources availble in the MOOC will also have to be given to the system (it would be possible to define it all on the resources interface, but it would be too long, a pre-treatment to write the XML directly is necessary).