**Chapter 5**

**Implementation**

This chapter outlines the most important and interesting aspect of the implementation of WEAVE.

**Technology choices**

This section describes the considerations taken into account during the process of selecting the technologies for this Level 4 project. The final choices for the technologies used were selected on the grounds of these considerations, the constraints of this project and maximum coverage of the requirements by these technologies.

**Considerations taken into account**

Due to the size and the nature of this project, the technologies for it were selected carefully based on the following criteria:

* Ease of development. This criteria is important due to the constraints of the Level 4 project. It was preferred that the selected technologies allow quick and easy implementation and testing.
* Maximum coverage of the requirements. The selected technologies should allow the satisfaction of the requirements in an efficient way and they should allow flexibility for any possible changes.
* Abstraction of other components. Separating different components is a well-recognised software engineering practice and would improve the maintainability of the code.
* Sufficient documentation. Learning or improving skills in the chosen technologies is one important benefit of this project so the presence of clear and detailed documentation is desired.
* Prior experience. Having good knowledge in the technologies used will result in a smaller learning curve and maximum effort could be concentrated in development, testing and improvement of the product.

**Choices**

**The backend**

Based on the criteria listed above, the Python programming language in complement with the Django web framework were selected for implementing the backend component. Django provides a well-developed and easy to use database abstraction layer. Each entity in the database is represented as a Python object and there is no need for writing SQL. All the data needed by Weave can be stored in the form of objects and can be accessed, modified, and deleted easily both programmatically and via an administrator interface. This serves well for both debugging and implementing the required functionality of the application. Abstraction is achieved due to the fact that if a different database component was needed the only change would be to change the Django database backend. Sufficient prior experience coming from studying Python in the first year at the University of Glasgow and the Django framework in the Distributed Information Management 3 module enable better and faster development. Furthermore, both Python and Django are very well-documented and developing web based application using them has turned into an enjoyable activity thanks to the award-nominated beginner’s guide to web development with Python and Django “How to Tango with Django”(reference) available online.

**The web interface**

There are two groups of technologies for the web interface- for the server and the client side.

**Server Side**

The Django web framework has a sufficient support for serving clients’ requests. This is achieved via direct communication with the database. Django allows clear separation between presentation and business logic due to the usage of the model-view-template software design pattern which guarantees better maintainability and readability of the code. This pattern is described in a greater detail in **Section X**(expand+add figure)

**Client side**

An important role in the selection of technologies for the client side of the web interface plays the compatibility with the browsers which are expected to be used in schools, mainly Internet Explorer, Google Chrome and Mozilla Firefox. Typically, pupils in schools are constrained to use the browsers that are installed on the school machine and acceptable appearance and behaviour is crucial.

**HTML5.** This is the markup language chosen for the generation of the web pages. It is preferred to other markup languages due to multiple benefits it provides. Firstly, HTML5 provides an easy access to contents and elements which helps for design and debugging purposes. Secondly, it allows for writing of cleaner code where style and content are separated. Last but not least, HTML5 supports excellent cross-browser compatibility.

**CSS** Cascading Style Sheets (CSS) is used for defining the appearance of the web pages. It enforces separation of concerns between the HTML elements and their presentation. Furthermore, the use of CSS allows the control and flexibility over the appearance of different elements and results in a cleaner HTML code.

**JavaScript.** Being free, open-source and supported by the majority of browsers, this scripting language is used to deal with the interactions of the client and the webpage.

**JQuery** Having the same benefits as JavaScript, JQuery is an excellent solution for simplifying client-side scripting. It is used for defining the behaviour of different components on user interaction with the website.

**Ajax.** Asynchronous JavaScript and XML (AJAX) is used for sending asynchronous requests to the server side avoiding the need for reloading the web page. This reduces the network overhead and the behaviour of the application “feels” closer to a desktop one.

**Bootstrap.** This is the most popular framework for developing responsive design of web applications. WEAVE is intended for school computers and the screen sizes may vary across schools. Furthermore, it is not guaranteed that students will use full screen size at all times of interaction. Due to the nature of WEAVE, students may need to open a different window with more information needed for solving a problem. These needs lead to the responsiveness of the application being crucial.

**Font Awesome.** This is an open source library used to simplify the user interface through the use of familiar icons for visualising possible means of interactions with the webpage.

**HighCharts.** This free library is chosen because it allows the easy creation and control of interactive charts which are needed for the teacher interface of WEAVE.

**Connecting the Different Tiers**

As described in **Section 4.4.**, the architecture of WEAVE involves three tiers- the presentation tier, the Django middleware and the Data tier. The Django middleware is split into two distinct sub-tiers- one to serve the communication between the client and the database, and one to deal with the imports of worked examples created by the author interface of the old system. For the purpose of this chapter, this architecture model will be split into two parts. The first part will represent the components in the purple area shown on Figure 4.X. These components realise the communication between the client and the server. The second part will represent the components responsible for the transformation of the worked examples created by IWE into database objects. These components are coloured in green on the same Figure.

**Client-Server Communication**

The different components responsible for the Client-Server communication are glued together via the Django variation of the *Model-View-Controller* design pattern- *Model-Template-View* (*MTV*).

**Model**

The model represents the data stored in the system. There are two types of data objects- one for the worked examples and one for the usage data. Figure 5.1. shows how these objects were defined in a UML Entity-Relationship diagram.

Part A of the figure is strongly influenced by the structure of the XML files which define the examples. Here, however, the relationships between objects are imposed by the use of foreign keys. For example, in the Documents.xml file, fragments are defined by their id and by the text of the fragment. Processes.xml file defines the steps for an example again by both the id and the text of fragments. In this implementation, instead of the Document entity and the Step entity to be associated with a fragment via CharFields for the fragment id and the text of a fragment, fragments are defined as separate entities and documents and processes refer to them via foreign keys. This contributes to a much easier and reliable modification of worked examples due to the fact that if a change needs to be made to a fragment, this change will need to be made in one place only- the fragment object.

**Template**

Templates describe how the data is presented on screen. They are equivalent to the view in the well-known MVC design pattern. Each template is an HTML file defining the different elements to be rendered on screen. The style of these elements is defined via CSS. Depending on the user interaction with each of the interfaces, elements can be destroyed, hidden, created or modified accordingly using JQuery. Such calls are predominant in the teacher interface. For example, if the teacher wants to see the answers for a question in a particular example, a dropdown with the relevant questions appears upon selection of the worked example. If they wanted to see the total time a pupil worked on a particular example, the dropdown list of questions would be exchanged with a dropdown list of pupils belonging to the selected group.

The templates define alternative elements to be rendered depending on whether the user is logged in and if they are- based on their previous interactions with the system. There are different elements that can be rendered depending on previous interactions with the system. For example, the home page of the teacher interface shows register/log in sections when the teacher has not logged in, and if they have- areas to register/update/delete a group, view pupil list for a group and view statistics options. This is done via a template language condition:

{% if user.is\_authenticated %}

show the elements for areas to register/update/delete a group, view pupil list for a group or view statistics options

{% else %}

show text boxes to register/login

{% endif %}

Similar template language is used to iterate over the list of examples which were passed to the context when the main page of the student interface is rendered. More details on how the list of students is passed are provided in the description of the Http requests below.

There are elements occurring in all templates of the interfaces. For example, the navigation bar on the top of each page. To improve maintainability and to avoid too much repetition of the same code over and over again, the reoccurring content is extracted into a base template from which all other templates inherit.

**View**

In this case view plays the role of the Python callback function for a particular URL. Views are the equivalent of the Controller in the MVC design pattern. Different parameters may be passed via the request made by the client. There are two types of requests:

* Http requests to render a page. Each request has its context, containing information such as the client’s machine for example. Many pages require some information to be passed upon rendering. For example, when a pupil authenticates themselves, the list of the existing worked examples must be rendered on screen. The view responsible for serving this URL knows that this page requires this list, so it will send an ORM request to the database for all examples. It will add these examples to the context dictionary and respond to the client’s request by rendering the template for the requested URL, passing the examples via the context dictionary.
* AJAX requests. These requests do not render another page but are used to get information from/sent information to the database. There are two types of AJAX requests served by the views:
  + GET requests. In these requests, the client asks for information from the database. Such requests are used mostly in the teacher interface when the teacher views the usage data of their pupils. The requested information is passed in the form of a dictionary and the responsible view gets the required variables using the keys of this dictionary.
  + POST requests. These are requests to store some information in the database. Logging of usage data for the worked examples, such as the time spent on a particular step, is done via these requests. The response typically indicates whether the data was stored in the database successfully. More details how logging of data is implemented follows in Section X.

**Translation of the XML Elements into Database Objects**

The translation of the XML elements into database objects is done via a population script which uses an xml element tree parser. Each element in this tree is represented by its tag and a dictionary of its attributes. There are two possibilities for the attributes of the element. They can either be attributes of the database object for the respective database object, or references to other database objects which will be used to create a foreign key relationship between these objects. This parser uses the depth first search algorithm to traverse the elements in the XML files. The pseudocode for translating the Processes.xml file into the respective database objects is as follows:

open the xml file

initiate element tree parser on this file- **parser**

get the root of the tree from **parser**- **processes**

for **process** in **processes**:

get the name of the example this process is for from **process attributes**- **app\_name**

get the application object with name **app\_name** from the database- **application**

for each **step** in **process**:

get the **order** of this step from **step attributes**

create a **step** object with foreign key **application** and attribute **order**

for each **child** of **step**

if **child** has tag **change**:

find the **fragment** with the fragment id specified in the **change** **attributes**

find the **document** with the document name specified in the **change attributes**

get the **operation** to the fragment for this **change** from the **change** **attributes**

create the **change** object with foreign keys **step, fragment, document** and attribute **operation**

else if **child** has tag **explanation**:

get the **explanation text** from the **explanation attributes**

create **explanation** object with foreign key **step** and the **explanation text** as attribute

Please note that for simplicity reasons this preudocode describes the process for adding changes which involve showing/hiding/highlighting/unhighlighting of fragments and the presence of an explanation. It does not include the creation of changes which involve questions. However, these changes, as well as the rest of the XML files, are translated following the same logic.

From this pseudocode, it becomes clear that only relevant elements are stored in the database. For example, no objects were created for a process. Processes become redundant in this database organisation and they can be unambiguously represented via the **Step** objects instead.

Creating and updating objects are done via the Django call get\_or\_create and passing the primary key for this object. This call retrieves either a newly created object or an already existing object with the specified value for the primary key. When this object is retrieved from the database, its attributes are set to those coming from the attributes dictionary of the XML element. This ensures the correct creation and modification of the object and is the reason for the destructive update model of WEAVE.

**Logging of Usage Data**

Usage data is logged on every step transition. The data for each attempt of an example step is represented by a database object. There are two types of such objects to accommodate the two different types of steps- steps which involve showing/hiding/highlighting/unhighlighting of fragments of a documents and and showing an explanation, and steps which involve a question. The respective objects are UsageRecord and QuestionRecord. The UML class? diagram of these objects is shown on Figure X.

On change of the step, an AJAX post request is done invoking the relevant method in the views.py file depending on the type of the step. This POST request passes a dictionary with the necessary information for the data record:

* the example name, the step number, the time spent on this step and the direction for the transition for the step. This is the data sent for steps which involve changes of fragments and an explanation.
* the example name, the step number, the time spent on this step, the answer for the question and an indication whether this was a multiple choice question. This is the information sent for steps which involve a question.

To handle this request, the relevant view extracts all the usage data using the relevant keys of the dictionary passed by the AJAX call. However, this dictionary does not pass any information about the user this data is coming from. This information is accessed via session variables instead. When the pupils specify their details, these are stored as session variables. For example, to store the pupil ID as a session variable when the pupil specifies their details, the view serving the AJAX call will include:

request.session['pupil'] = pupil\_id

When the view for handling logging of data is invoked, it checks the session variables for the teacher id, group is and pupil id .This is done via a simple call such as:

pupil\_id=request.session.get(‘pupil’, None)

None is the default value to return if no session variable exists for a pupil. After checking the session variables for the teacher id, group id and pupil id, the method adds the ones that were present to the data record and saves the record to the database.