**Chapter 4**

**Design**

This chapter outlines the design decisions for this Level 4 project. As it was previously described, WEAVE is based on the earlier Java standalone application IWE. This affects to a great extent these decisions. Following the well-established software engineering principle of reusability, guidance for the design of WEAVE was to reuse any good aspects of IWE’s design while improving its weaknesses.

**4.1. Storage of Data**

The scope of this Level 4 project does not include the creation of a web based author interface. This means that the design of WEAVE must ensure an easy and efficient way for importing and storing the existing worked examples. In addition, WEAVE must support uploading updates to existing examples. As described in **Chapter 2**, the examples are stored in the form of XML files. To remind yourself of the structure of these files, please refer to **Figure 2.2**.

During the exploration process of IWE, a weakness of storing the information about the examples in the XML files used by IWE was identified. Since these files are easy to access and modify by the authors of worked examples, and this is typically much easier than modifying the worked examples using the author interface, one may be tempted to make changes to the examples manually. However, references to some objects may be present in more than one of these XML files. For example, in the Documents.xml file, fragments are stored as individual elements identified by a fragment id. When the steps for the example are defined in the Processes.xml file, each fragment is referred to with its id and also, for an unexplained reason, the fragment text is included. A problem with storing information about the examples in files is that if an element is modified in one of the files, consistency about this element must be ensured across all copies, or else the system fails. It is not trivial having to find the same feature across multiple files and in the end consistency and validity are not guaranteed. This appears to be a case of poor software engineering, as there is no apparent reason for the fragment text to be copied between XML files, and not just the fragment id.

Django provides object relational model (ORM) database functionality. This is an efficient way to manage objects and their relationships and is a preferred method for managing the data for the worked examples because it guarantees consistency.

In the new system the worked examples are stored in a database where the relationships between different objects are expressed via foreign keys. So, for example, the same fragment is not copied but referred to by foreign key. In addition to ensuring consistency, using a database adds a level of reliability that the data is valid because internal integrity checks are made before saving an object to the database. Further advantage is that the Django administrator interface allows an easy means of exploring and modifying the examples in a consistent and safe manner.

**4.2. The Data Model of Worked Examples**

The way the worked examples are structured in the XML files generated by the author interface of IWE is shown in detail on Figure X. Below is a quick reminder.

* Each worked example consists of one or more panels (1), which show the text of a document for the worked example (2).
* The text of each document of a worked example is split into different fragments (3) (portions of the text for the example).
* Processes define the steps for each example (X).
* Each step of an example consists of a set of changes (4) and an explanation (5).
* A change can involve either showing/hiding/highlighting/unhighlighting a fragment (6) of a document (7) or asking a question (8) (open-ended or multiple choice)

The structure of the XML files determines to a great extent how the worked examples are stored in the database. **Figure 4.1** shows the UML Entity-Relationship diagram for the worked examples in WEAVE. The numbers associated with a particular relationship between the entities correspond to the numbers in the bullet points above to illustrate the similarities between the two methods of representation of the worked examples for the different systems. Please note the bullet point that has X instead of a number. This element in the XML files was found unnecessary when storing the worked examples in the database and it would have led to redundancy. Instead, each worked example (referred to as an application in both Figure X and Figure 4.1) is associated with a set of steps. The order of these steps is determined by the order attribute of the Step entity. When a worked example is loaded, all the steps for this example are retrieved and are shown in the order defined by their order attribute.

Please note that this is a simplified version of the objects storing the worked examples for illustration purposes. As with the XML files, there are objects for the different type and style of documents and fragments. Inserting them in the UML Entity-Relational diagram on Figure 4.1. would have made it unclear and difficult to read. The type and style of documents and fragments are defined using foreign key relationships with the respective objects. On Figure 4.1. these foreign key relationships are expressed by adding FK next to the name of the object for the particular type or style involved in this relationship.

**4.3. Design for the Translation of the Worked Examples from XML Elements into Database Objects**

The database is populated with the elements stored in the XML files produced by the author interface of IWE via a command line interface. A population script- population\_script.py file- is run on the command line. It takes one optional parameter to specify the path to the XML files. If this parameter in not provided, the script will look for the files in a default directory for storing the examples. More details on the implementation of this population script are provided in Section X. Instructions on how to run the script and where the example directory is are provided in a readme.txt file in the project as well.

**4.2. Authentication and Privacy**

A significant discussion point was how to authenticate teachers and their students due to the privacy and ethical issues discussed in **Section 3.3.2**.

As a reminder, the main issue around privacy/ethics is to avoid storing information in the system that could identify individual schools, teachers or pupils. If this can be achieved in the design, then data gathering for worked example authors and system designers, essential to improve both worked examples and the system itself, can be carried out with no requirement for informed consent for data usage by the pupils or teachers.

The requirement here, then, is that names for teachers, classes and pupils do not contain identifying information.

One possibility was for students to create their own accounts and give their usernames to the teacher, so that he/she could monitor their progress. However, this approach could potentially result in various complications.

* Students would need to be explicitly instructed that their accounts should not reveal their true identity, since this information would be stored in the system database, and visible to systems administrators.
* Students would not have any benefit of having their own accounts as only the teacher will be the one who would use their usernames for something meaningful, i.e. to check their progress.
* This approach solves identification at an individual level but each student needs to belong to a group as well, to assist the teacher in monitoring the progress of a particular class, so this becomes an additional management overhead for the teacher.

A second option was considered, in which the teacher would create an account for each of their classes. For this account, they would need to create usernames for their students. Keeping in mind that a teacher would often have more than one class and that each class consists of twenty to thirty students, the following problems arise:

* The teacher would need to create a lot of accounts and this could be a trivial and time consuming task.
* The teacher would need to ensure that they will be able to match each of their students to their id since they must not use any names due to the privacy issues mentioned above.
* Account names should not be guessable by other pupils, to avoid any one of them pulling a prank on another pupil by using their username. Hence the teacher can't use an easy sequence of anonymous usernames such as p1, p2, p3 etc. On the other hand, long, anonymous usernames should be avoided since these are tedious to store and transmit and are likely to lead to errors in use.

To get around the privacy issues and the problems with the options described above, a third approach based on the general idea of the second one was adopted. The main points of this approach are:

1. Teachers create their own accounts, with a username that does not identify them or their school.
2. Once logged in, they will be able to create groups, one for each of their classes, with a name that does not identify the school.
3. On creation of the group, the teacher needs to specify the number of the students in this class. WEAVE will then generate random ids for these students. An id consists of two random letters followed by a single digit. While being short enough to be easily remembered, such an id ensures that the privacy of students is protected due to its random nature. Furthermore, the number of possible combinations for all student ids is large enough so that it will be highly unlikely that students will be able to “guess” one of their classmate’s student id and work with the examples on their behalf.

Talking to a lead teacher, Mr Peter Donaldson, who is part of the PLAN C project(reference), a potential inconvenience of this approach was identified. Using WEAVE for longer periods than one academic year could lead to a significant increase of the groups. In addition to the growth of the number of groups, some teachers might prefer to use the same name for their classes across years. To resolve these issues, a further classification of groups by the academic year the group belongs to was adopted.

Mr. Donaldson was generally happy with the idea that teachers select the number of students for each group at the creation of this group. However, he pointed out that it is possible for a student to arrive in a class at a later stage than the beginning of the academic year. Using the selected approach would have required that the teacher creates a new group just to add one student only. This could be very problematic, because data about the same students would be spread across two different groups and most of the students will be given two student ids which could become really confusing for both teachers and students. To avoid these problems, the option for teachers to update a group was added to the design decisions for the authentication part. In addition, groups can also be deleted in case of creation of unneeded groups.

**4.3. Presentation of Data to the Teacher**

The main goal of the teacher interface is to present to the teacher data associated with a particular group or student. Three different types of data are recorded from the student interface:

* Time at each step.
* The direction of the transition to each step i.e. is the student going backwards or forwards to a step.
* Answers to questions.

Careful consideration was needed to reach a solution that would visualise this data in a way which would enable the teacher to comprehend it easily and encourage further analysis. In addition, the ability for the teacher to be able to view data both at a class and at an individual level further influenced the design decisions.

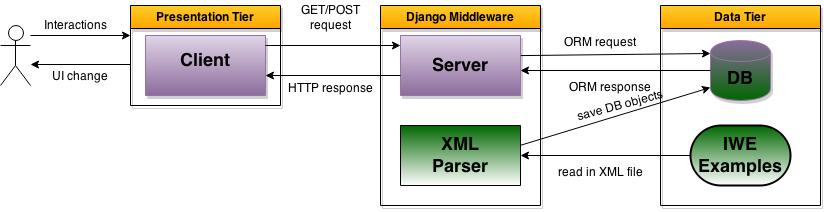
In a discussion with Mr. Donaldson- as a teacher who would use such data- it has been decided that presenting the data in the form of graphs would be beneficial to teachers as they would be able to identify patterns and any exceptional events for a particular worked example. These graphs should reveal information about the performance of the whole class as well as of individual students at each step of the example keeping in mind that some steps have questions. For this purpose, five different types of graphs were decided upon:

* Average Time. This graph would show the average performance of the students in the whole class on a particular example. Different steps will be represented on the x-axis of the graph by their step number. The average time spent on each step will be shown by the y-axis. Since the x-axis consists of the step number, this is not very informative to the teacher because they would need to look at the actual example to remind themselves about the context of the step. To avoid the need for that, hovering over the point representing the step will show the beginning of the explanation. Clicking on that point will open a window showing the whole text of the explanation, the average time spent on that step and how many students made a backwards transition to the step. As mentioned above, some steps contain a question, rather than an explanation. Such steps will be identified by a question mark in front of the step number on the axis label. Instead of showing an explanation on mouse hovering, a message encouraging teachers to click in order to see students’ answers is shown. Clicking on the point for that step will show the question, and a bar chart with all the possible answers and how many and which pupils selected each option.
* Student Time. This graph is conceptually the same as the Average Time graph. Instead of showing information about the whole class, however, it shows the total amount of time spent at each step by a selected student.
* Student Answers. This bar chart shows the options for a selected question and the number of students who chose each option. Hovering with the mouse over each bar shows the list of students who selected the answer represented by this bar.
* Class Steps. This graph shows information about the time spent at a chosen step of an example by the students of a class. It is in the form of a bar chart where each bar represents a student’s attempt. This means that there might be more than one bar for each student if they have attempted the selected step more than once- each bar revealing information about the time spent by that student at a particular attempt of the step.
* Class Summary. This is a table showing summary information about the total time spent by each student at a particular example, how many times they returned to previous steps and the last step they reached. This would show the teacher how much effort did the student put in each example, how many problematic or unclear steps they encountered as well as whether they completed the example or which step they gave up on. This is ideal as a quick overview of a class's progress, highlighting problematic students.

**4.4. Architecture**

The architecture of Weave consists of three distinct tiers as visualised on **Figure 4.1**.

* Presentation tier. This is the top level of the overall architecture also known as the client side web interface. It defines the appearance of the website by rendering HTML and CSS and provides means for users to interact with the application. The clients are in the form of web browsers. On every interaction, they send requests to the server in the form of HHTP GET or POST requests.
* Django Middleware. This tier consists of two distinct components.
  + The first component of this tier serves as a communication point between the client and the database. In this tier the requests from the client are parsed and translated into ORM requests- a language understandable by the database. These requests are passed forward to get or store the information in the request from/in the database. After the backend generates an ORM response, the middleware is responsible for translating it into an HTTP response and passing it back to the presentation tier.
  + The second component is the connection point between IWE and WEAVE. This is where the translation of the XML elements takes place, so that the examples can be stored in the form of ORM objects. This translation is done with an XML parser. The relationships between these elements are established and the database is populated with the objects defined by the parser.
* Data layer. This tier represents the database in which all the information used or generated by WEAVE is stored in the form of database objects (typically known as rows in a relational database). On GET requests the backend tier responds with an object meeting the criteria specified in the ORM request. On POST requests, the database creates a new object with the features specified in the request and stores it in the database.

Figure 4.1.

**4.5. User interface**

As already mentioned, WEAVE is intended for three distinct groups of users, each with their different needs. However, for the purpose of this Level 4 project, only two of these groups will influence the user interface. Addressing the needs of authors is beyond the scope of this project. This is an intermediate step between moving from using entirely IWE to using entirely WEAVE. For this purpose, no author graphical user interface is provided. To create new examples, authors need to use the author interface of IWE. These examples are then added to WEAVE via a command line interface. . Adding these worked examples to WEAVE is done via a command line interface for the time being. and contact the administrators of WEAVE with a request to add the newly created example to the application. This is why the section on the user interface is split into two subsections only, which describe the user interfaces for students and for teachers.

**4.5.1. Student User Interface**

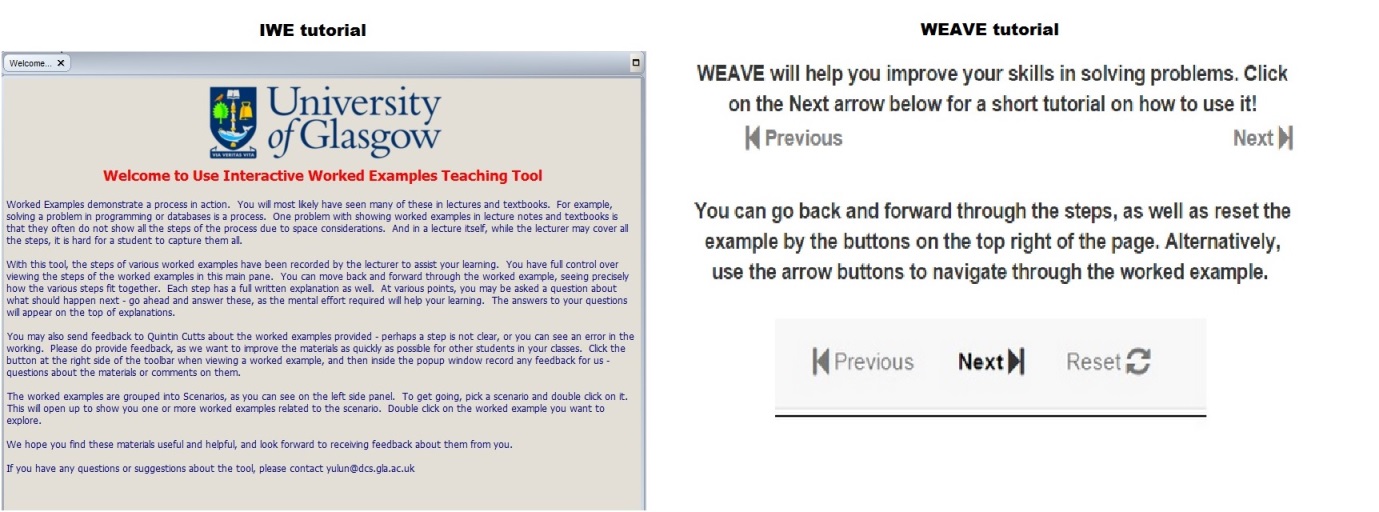
A core purpose of this Level 4 project is to translate the student part of IWE into a more easily deployable online version. Careful consideration about the layout of the student interface of IWE is evident. The evaluation of IWE proved that the current interface is well accepted by students. A screenshot of this interface is presented on **Figure 2.3**. Generally, it has been decided to take advantage of Dr. Song’s findings and to reuse a very similar interface.

**4.5.1.1. Home page**

The need for some additional features of this interface arises due to the fact that teachers need to be able to identify their students in order to monitor their interaction with the examples. To ensure that students are using the system in the intended way and to encourage them to use the details provided by their teacher, the examples will be hidden to them until they log in – either by entering some authentication information or by identifying themselves as anonymous users.

Once the student has logged in, the area prompting the user for details is exchanged for the list of worked examples to encourage them to choose one to work on. Due to the fact that the system is required to accommodate examples created by many teachers across the UK, a filtering by the name of the worked examples functionality has been provided.

In order to be able to exploit the worked examples viewer in an optimal way and to familiarise students with it, a tutorial appears on the main page. The idea for having a tutorial was borrowed from IWE. However, the way the tutorial was constructed there was identified as potentially ineffective at communicating all the information the pupil needs to know before working on examples due to the fact that it contains a lot of text which may discourage some of the pupils to read it. Furthermore, even if they read the tutorial, they may not understand what is referred to in the text because they may have not seen the worked examples viewer and its features in advance. A different approach was chosen for the tutorial of WEAVE. It is split into different steps describing an individual feature using minimal text and a screenshot of the feature. Screenshots of the tutorials in the two systems are shown on Figure 4.2. for comparison.



**4.5.1.2. Page for viewing an example**

The page for viewing a worked example is very similar to the one used for the IWE student interface. However, due to constraints imposed by the size of the screens in schools, the design needed to be adjusted accordingly. The area for selecting an example (referred to as **Element 1** in **Figure 2.3**) is placed on the navigation bar with all the examples appearing in a drop down menu on request. This saves a significant portion of the screen which can be used for the problem specification instead. Another space consuming element is the bar showing the current step (**Element 3** in **Figure 2.3)**. In WEAVE this information is shown as a part of the explanation instead.

**4.5.2. Teacher Interface**

**4.5.2.1. Home page**

**4.5.2.1.1. Non-logged in teachers**

The purpose of the teacher interface requires the teacher to be logged in. Therefore, on the first visit of the page the teachers are presented with registration and log in areas only.

**4.5.2.1.2. Logged in Teachers**

The discussion in the design decisions on the authentication to WEAVE and the way usage data is presented to the teachers identifies the main sections of the interface for the logged in teacher. Options for all the activities a teacher can undertake via the teacher interface are present on their home page to avoid the need for transitions between different pages and to simplify navigation of the website. The metaphor here is of a control dashboard. The main page is split into three areas.

* Area for registering, updating and deleting a group. These three options are provided in the same area on the screen. When the teacher selects the desired option, the elements for this area change accordingly. For example, when the user wants to create a group, they need to enter the group name and the number of students for that group. On update or deletion of a group, on the other hand, they select the group name from a dropdown list. The list of existing groups is shown to remind which group names are unavailable to this teacher. The textbox for entering the number of students accepts integer input only for error prevention purposes. On the submission of the request to create/update/delete a group, a message confirming the status of the action is show.
* View Group area enabling the teacher to select a group for which to view the student ids. Again, for simplicity and error prevention, the teacher selects the group via a dropdown list rather than typing its name.
* View Progress icon which navigates the teacher to the statistics page.

**4.5.2.2. View Group Page**

Teachers are able to see the random student ids for a class in the View Group page. Because managing login details with school pupils is known to the project supervisor to be a problematic issue, this page was designed in close consultation with teachers to ensure the simplest way for them to distribute pupils ids. Furthermore, it is designed to ensure that the anonymity of data gathered by the system is complete.

The student ids are provided in a table with empty columns for the student name and two identical columns with the student ids. There are two expected modes of use:

1. Teachers print this group sheet and fill in the names of their students by hand. This avoids any potential problems with storing identification information in the system. Teachers can then cut one of the columns for the student ids, further cutting the column into the individual student ids and hand them privately to each student. The teacher keeps a paper record of which student has which id.
2. The system is designed to enable teachers to copy and paste the student id column into a spreadsheet containing the student names. This can be printed as before, or projected for students to read off.

**4.5.2.3. View Progress Page**

Most of the design decisions for the different types of graphs are explained in **Section 4.3.** above. The teacher needs to select the particular group and the type of data they are interested in. If there is no data for that selection or the selection is invalid, an appropriate message appears on the screen. Otherwise, a graph is shown. This graph is downloadable to enable saving the data for progress at different points in time and could be used for comparison by the teacher.

The following chapter will describe how the design decisions for WEAVE were implemented.

**Chapter 5**

**Implementation**

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

**5.1. Technology choices**

This section describes the considerations taken into account during the process of selecting the technologies for this Level 4 project. It then follows with the list of technologies chosen based on these considerations and the constraints of this project.

**5.1.1. 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:

* Scalability. WEAVE is intended to be used in around 400 schools in Scotland by thousands of pupils- and if successful, then internationally. This is why the system being scalable is one of the most important considerations taken into account.
* Ease of development. This criterion 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. They should also 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 and 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.

**5.1.2. Choices**

**5.1.2.1. The backend**

The choice for a web-application development framework was of crucial importance for this level 4 project. To facilitate the most suitable choice of framework, the Questions-Options-Criteria design model was used (Maclean, A.; Young, RM.; Bellotti, VME.; Moran, T. (1996), "Questions, Options, and Criteria: Elements of Design Space Analysis", in Moran, T.; Carroll, J., *Design Rationale Concepts, Techniques, and Use*, Lawrence Erlbaum Associates, pp. 53-106) . Figure 5.1. shows the question, the different options and the criteria considered for this project. Based on the maximum satisfaction of the criteria in Figure 5.1., the Django web framework , written in Python, was selected for implementing the backend component. Firstly Django will allow scalability due to the fact that it supports many database engines and switching between them only requires changing the settings of the project. This project will use the default database engine- SQLite because there will be no need for a more powerful one for the time being. However, as WEAVE evolves and becomes more widely deployed, it will be trivial to change to a more powerful engine. Secondly, Django allows rapid application development (RAD). It has support for many libraries which ensures that all the requirements for this project can be met. Django imposes separation of concerns which is discussed in more detail later in this chapter. 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”( http://www.tangowithdjango.com/) available online. Furthermore, I have a 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. Finally, Django is open-source so there was no need to pay for the software used to develop this project.

**5.1.2.2. The web interface**

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

**5.1.2.2. 1. 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 5.2.1**.

**5.1.2.2. 2. 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 machines 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 the multiple benefits it has. 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. HTML5 was preferred over XHTML(http://xhtml.com/), which was the other option under consideration, because HTML5 offers more flexibility.

**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. An alternative considered while making this choice was Less (<http://lesscss.org/>). It adds features to the CSS language and makes it more maintainable. However, it would have involved some initial learning overhead and due to the time constraints of this project it was decided to focus the predominance of the time on other parts of the implementation instead.

**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. Alternatives for this choice of technology were not considered due to the simplicity and powerfulness of this language for its intended purpose.

**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. It allows easy manipulation of HTML elements and removes cross-browser issues. Furthermore, it has a very large community and it is easy to understand, as opposed to the alternatives considered, which include Kendo UI (<http://www.telerik.com/kendo-ui>) and Wijmo (http://wijmo.com/).

**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. The alternative to using AJAX would have been to load pages synchronously. However, this would make the interaction cumbersome and slower.

**Bootstrap.** This is the most popular framework for developing responsive design of web applications. This is a combination of HTML, CSS, and Javascript code that is designed to facilitate building of user interface components. 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 to enable them to solve a problem. These needs lead to the responsiveness of the application being crucial. An alternative considered was PureCSS(purecss.io) which provides a set of small, responsive CSS modules that would load faster. However, Bootstrap was preferred because of the larger community and the greater number of features it offers.

**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.An example icon used across all the pages is the Home icon which users of websites are familiar with as a means to navigate to the home page. There are many open source libraries which could have been chosen for this purpose. These include Entypo (<http://www.entypo.com/>) and Typicons (typicons.com). Font Awesome was chosen on the grounds of providing better looking icons.

**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. It has been chosen mainly because it provides many pre-built graphs, such as the line graphs and column charts needed for viewing the progress of pupils. Another benefit is that the library is compatible with older browsers like Internet Explorer 6. This is needed because in schools across the UK older browsers are used predominantly. A library which would bring more powerful features would be D3(http://d3js.org/). However, it was not chosen because it does not work well with older browsers.

**5.2. 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 4.1**. 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.

**5.2.1. 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*).

**5.2.1.1. 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.

The objects storing the worked examples are strongly influenced by the structure of the XML files generated by the IWE author interface for holding these 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 specifying 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.

The objects storing the usage data for the worked examples are explained in detail in **Section 5.3**.

**5.2.1.2. 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 are 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.

**5.2.1.3. View**

Each 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. An example of such request would be when the pupil clicks on a worked example to work on- the client will send a request for the page for this worked example. Each request has its context, containing information such as the client’s machine, etc. 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 and 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 5.3**.

**5.2.2. 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 element, 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 (in IWE, each worked example is known as a 'process') into the respective database objects is shown below. Please note that in this pseudocode **application** stands for worked example. This is the terminology of IWE.

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**:

get the **fragment** with the fragment id specified in the **change** **attributes** from the database

get the **document** with the document name specified in the **change attributes** from the database

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 an 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 get\_or\_create method 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 or modification of the object and is the reason for the destructive update model of WEAVE.

**5.3. 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 document and showing an explanation, and steps which involve a question. The respective objects are UsageRecord and QuestionRecord.

On change of the step, an AJAX post request is done invoking the relevant method in the views 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 and is stored in a UsageRecord object.

Or:

* 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 in a QuestionRecord object.

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 the line:

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 id 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 such session variable exists. 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 in the database.

**5.4. Resizing of the Worked Examples Panels and the Explanation Area**

A major implementation challenge for this Level 4 project was to satisfy the requirement to make the panels for the text of the worked examples, as well as the explanation panel, resizable. Difficulties arose due to two requirements for the system:

* The system must support worked examples which contain any number of panels for worked example text (not only one or two).
* Both horizontal and vertical resizing were required.

The JQuery *resizable* plugin provides the option to specify that an element is resizable. When specifying this, it can also be specified that when the resizable element is resized, other selected elements are resized as well. This can be done by using the alsoResize option supported for the resizable elements. The code for making an element with id *panel1* resizable and to specify that an element with id panel2 needs to be resized as well on resizing of panel1 would be as follows:

<script>

$(function() {

$("#panel1").resizable({alsoResize: "#panel2"});

});

</script>

However, this would mean that when making one panel wider, the other panel also becomes wider. By small addition to the plugin code, a similar option to resize the second element in the opposite direction could be added- alsoResizeReverse. To implement this option, it would have been required to change the sign in calculations done by the respective function. The above code would then look like this:

<script>

$(function() {

$("#panel").resizable({alsoResizeReverse: "#panel2"});

});

</script>

The following subsections will describe the problems using this approach.

**5.4.1. Vertical Resizing of Worked Examples Panels**

This solution above could work for resizing the panels horizontally if there were two panels for the examples only. Using it for more than two panels would behave in an undesired and unexpected way due to too much alsoResizeReverse dependency between the panels.

Another JQuery plugin which was found to behave better in this situation is the *colResizable* plugin. It treats the area of the worked examples as an HTML table where the panels are different columns which can be resized. This plugin was used for the vertical resizing.

**5.4.2. Horizontal Resizing of Worked Examples Panel**

The resizable JQuery plugin was ideal for the horizontal resizing of the panels. However, it was incompatible with the colResizable plugin used for the vertical resizing of the panels. This is why, the height of the panels and the explanation are recalculated and set manually on each resize event. This is the code:

$("#explanation\_container").bind('mouseup', function(e) {

$("#panel\_container").height($("#interface").height() - $("#explanation\_container").height());

});

What this code does is when the user is finished with the resizing of the explanation panel, is sets the height of the worked examples panels to be equal to the difference between the total height of the interface and the height of the explanation panel.

**5.5. Showing the Text of the Worked Examples in the Panels**

As described in Chapter 2 and Chapter 4, the documents containing the text of a worked example are split into many fragments. These documents are shown in separate panels on the viewer. These panels are in the form of HTML div elements. It is often the case that more than one fragment need to appear on one line in these div elements. This is why the fragments needed to be represented by inline elements rather than block elements in HTML. Inline elements allow multiple elements to be placed next to each other, while block elements are placed below each other. Span is the inline element chosen for representing each fragment. Please note that any inline element could have been chosen without observing any differences in the behaviour.

Due to the constraint that some fragments need to be inline with other fragments, a slightly undesired behaviour of the fragments arose when they do not fit in the width of the panel for a particular document. Words of these fragments can be split between two lines in unexpected parts. However, this is a limitation due to the use of HTML elements and their behaviour.

This chapter described the implementation of the major components of WEAVE. In the next chapter, the reader’s attention will be drawn to the evaluation of the system