COURSE MANAGEMENT WEB SYSTEM

by

Li Tan

a Report submitted in partial fulfillment
of the requirements for the SFU-ZU dual degree of
Bachelor of Science
in the School of Computing Science
Simon Fraser University
and
the College of Computer Science and Technology
Zhejiang University

SIMON FRASER UNIVERSITY AND ZHEJIANG UNIVERSITY
Spring 2010

APPROVAL

Name: Li Tan

Degree: Bachelor of Science

Title of Report: Course Management Web System

Examining Committee:

Dr. Qianping Gu, Supervisor

Greg Baker, Senior Lecturer, Supervisor

Dr. Ramesh Krishnamurti, Examiner

Date Approved: _____

Abstract

In this report, I present a web based course management system that we develop for School of Computing Science at SFU. This project aims at integrating grade-book, marking and submission functions together. It also introduces the grouping function that the current system does not have. This project can provide convenience to students and faculty members in managing courses, assignments, exams and grades. It is based on a modern and powerful web development framework called Django. I first give a complete picture of the system to give an idea of what the four modules (grades, marking, submissions and grouping) do and their relationships. Then I focus on the design and implementation of the marking module for which I am mainly responsible. Topics of the discussion include use cases analysis, data model description, web requests, web responses as well as other specific development issues.

This is a very valuable project because not only I have an opportunity to learn some modern technologies in developing web systems but also I can make contributions to the School of Computing Science. I hope this project is a good start to renew the current course management system.

Contents

Chapter 1

Introduction

In this project, we will develop a web based course management system for the School of Computing Science at SFU. The system mainly includes the following four modules: grades, marking, assignment submissions and student grouping.

The main purpose of this project is to use a modern website development tool to come up with an integrated system that have all its parts functioning together. The current system has the first three parts mentioned above, but they were not designed together in the first place. So in the design phase, we consider the system as a whole and the interactions between the four modules. Another reason is that many instructors are calling for a mechanism for them to assign grades to groups for a course project rather than to every individual student every time. They want a fast way to get work done. To support the grouping functionality, we feel it is necessary to design our system from scratch.

We choose Django [2] as the website development platform and Python as the programming language. That means the system will be far easier to maintain than the current one. The good thing about Django is that it supports many types of databases such as Mysql and PostgreSQL [4]. The server side scripting is also provided by the powerful query functionalities of Django [8]. As it is stated in [18], "Server-side scripting is a web server technology in which a user's request is fulfilled by running a script directly on the web server to generate responses. It is usually used to provide interactive web sites that interface to databases or other data stores." While we can choose one type of database during the development period, we can choose another one during production with minimum change in the configuration.

Now I will give a brief overview about the background and the most important features

of each module.

1.1 Background

SFU's websites have been an important part of daily campus life in each semester. As a student, you have concerns as to which course you will take and who the instructor and TA's are for the course. You may also want to get informed on what grades you get for assignments and exams. As a faculty member, you are concerned with the students in your courses, and how to compute and assign grades to a student quickly. You may also have concerns on who are in the same group for a course project and what comment you want to give each student or a group to justify the grades they deserve. In all this information, the following are the fundamental pieces of information in the system: the information about students and faculty members; the information about the courses offered at each semester; the memberships of each course offering and each member's relationship to this course offering (i.e., as an instructor, TA or student). This core data of the system serve as a foundation for grades, marking, submissions and grouping modules to work together.

1.2 Grades module

1.2.1 Main functionalities

For a student, the grades module is like the current grade-book used by the School of Computing Science at SFU[13]. A student can view the courses he/she is enrolled in and the activities of each course, the grades he/she get on them and the performance distribution of the class (e.g., by histograms).

For an instructor, he/she can manage the activities by adding and editing them. These changes are exposed to students as well.

An activity is a general term here. It could mean:

- A numeric activity, which is typically an assignment or exam and is given a numeric grade.
- A letter activity, which is mainly used for final performance evaluation and is given a letter grade.

- A calculation activity that contains a formula used in calculation. For example, by giving assignments, midterms and final exams different weightings, an instructor can specify a formula to calculate the final numeric grade for every student. Students do not need to submit any deliverables for a calculation activity because it is used by the instructor to quickly calculate grades for students. According to the distribution of the final numeric grades, the instructor can decide cut-off values to give the final letter grade for every student. An instructor can also set an activity to be group activity (not for calculation activity). For a group activity, students submit deliverables by groups and can be marked by groups.

1.2.2 Module interaction

Activities information is available to the marking, submissions and grouping module. The numeric marks will be assigned through the marking module.

1.3 Marking module

1.3.1 Main functionalities

Marking module deals with how to give numeric marks to students in a course on each numeric activity. Mark can be assigned directly to an individual student or a group if the activity is a group activity. If one activity has several logically separated components, an instructor can give a mark for each component with feedback. To save time, we introduce *common problems* which are the mistakes commonly seen in students' submissions. An instructor or TA can configure common problems and use the description of this common problems as comments when marking. Also, for each mark given, an instructor or TA can provide additional information such as mark adjustment (due to various reasons), late penalty and a file attachment.

Marking will be initiated by clicking a link beside the submission entry of a student's work or a group's work. There will also be an indicator beside it saying whether it has been marked or not. We also keep records about the history of who gives whom what mark for which activity and when. These records can be investigated when there is an issue or confusion on a particular mark.

Another handy function is that an instructor will be able to copy the course setup

(i.e., the basic information of all activities together with their marking components and submission components) from one course in some semester to another in the subsequent semester.

1.3.2 Module interaction

Information about numeric activities comes from the grades module while the marks given in the marking module are stored in the grades module and can be viewed through grade-book. For marking groups, grouping membership information comes from the group module.

1.4 Submissions module

1.4.1 Main functionalities

Student can submit their assignments or deliverables in multiple configurable components. Each component should have types configured by the instructor or TA's. Like the current submission server, only zip/rar/tgz file formats are allowed and students are able to override their previous submissions with newer ones.

1.4.2 Module interaction

A submission can be tagged as not graded or graded. This requires information from the grades module. Submissions can be made by students for a group where they belong to, which needs information from the grouping module. This module also deals with the marking ownership of every submission to avoid instructors and TA's marking the same thing at the same time without being aware.

1.5 Grouping module

1.5.1 Main functionalities

There are two ways to form groups. One way is that the instructor assigns students to groups. Another way is that students create groups spontaneously and one group member can invite other students to join. While the membership assigned by instructor takes effect right away, a student who is invited by other students can decide whether to accept the

invitation. When the group is created, the group creator can specify which group activities to associate with this group (although groups in most courses do not change during the whole semester and apply to all group activities). A student can submit assignments on behalf of his/her group and can be marked through the group.

1.5.2 Module interaction

As mentioned in the above sections, it provides grouping information to all the other modules.

1.6 Overall goal of modules design

In summary, the goal is to decouple these modules so that they can be developed almost in parallel and with integration ongoing along the way as well. These practices accord with the idea of the *Scrum* development model. As a brief introduction (for this can be found in [19]),"*Scrum* is an agile process for software development. With *Scrum*, projects progress via a series of iterations called sprints." We normally add relatively small features to different modules steadily with minor worry about the integration.

The rest of the report is organized as follows. In Chapter 2, the important technical terms and concepts in our project are introduced. From Chapter 3 and on, the discussion is basically around the design and implementation of the marking module. Chapter 3 analyzes all the uses cases in the marking module. Design of data models in the marking module are described in Chapter 4. Handling web requests handling and generating web responses are the topics of Chapters 5 and 6 respectively. Chapter 7 presents a couple of specific issues including security and interactive user interface design. In Chapter 8, a concrete example of view is given for your reference. Finally, Chapter 9 concludes this report.

Chapter 2

Preliminaries

In this chapter, I am going to introduce some technical terms in our project. Basically, we group them into three layers [2]: database layer, functional layer and presentation layer. It is natural to do it in this way. First, data models (or table schemas) in the database can be separately defined and it should be well defined before the details of system logic can be worked out. The functional layer is about how to implement the use cases or the system logic. It includes how to process user web requests and how to interact with the database. The presentation layer is loosely connected with the other two. It focuses on how to display the information passed from the functional layer to users. It can change the way it displays things with little effects on the other two layers. However, the presentation layer is very important because it is the layer users directly interact with. The user interfaces should be clear and easy to use.

2.1 Database layer

In Django, each module in our system is called an *application*. In each *application*, we define all the relational table schemas in this module, which are called *models* [7]. We can retrieve from the database a single object representing one row in a particular table. We can also retrieve a series of objects satisfying certain conditions we specify. These queries are done by using *queryset* that initially contains the information about the query. It will hit the database and be evaluated only when we actually need the results of it [8]. While we could still use raw SQL to communicate with the database, *queryset* already provides a strong API that we can use in almost all situations. It offers a quick approach to make a query

that even involves complex relationships between tables (e.g. foreign key or many-to-many relationships).

2.2 Functional layer or view layer

The view layer is where the actual system logic takes place. View here may sound a little misleading. It not only handles what data to present to a user after the user makes an Http GET request, but also handles what data to save to the database after a user submits an Html form with an Http POST request (an Html form on a web page allows a user to input data that is sent to the server for processing). After handling a GET request, the view passes to the presentation layer the information to show to the user. The information is put into a `context` object before passed to the presentation layer. After handling a POST request, if the data can be validated, it saves updates to the database and then redirects to another page which shows a success message. If the data has errors, the view will pass the form with invalid fields and error messages to the presentation layer so that the user can try to correct the errors and submit again. Therefore, we can see the view layer is a data processing layer between the database and the presentation layer.

There is a mapping between URL's and views. Each view function is responsible for handing the requests targeted at one particular type of URL. For example, a view function can respond to a user request to `http://server_base/course_id/new_activity`.

Apparently, not every user has the right to access all the URL's, so we also need an authorization mechanism. We use the SFU authorization server as our authorization *middleware*, meaning login and logout actions happen there before a user could go into our system. On the other hand, within our system, some view functions would be run only when the user is authorized to access the corresponding URL. We put special functions called *decorators* [12] to do the job before running that view function.

To test our system, Django provides a unit testing environment based on the Python unit testing library [16] in which we can write unit tests for each module. We can test our design of each relational table in the database to verify if it works correctly. We can also emulate a user request and verify if the Html response is valid and its contents are as expected.

2.3 Presentation layer

The presentation layer is based on the customized Html format files called *templates*.

As mentioned earlier, that the view function passes the context object to the presentation layer. Usually the context object contains a bunch of variables including single objects and aggregation objects like lists, sets and dictionaries which are basic data types in Python [17]. The fields and methods of these variables can be referenced in the template.

Django templates API also gives built-in *tags* to make it more convenient to write Html code[10]. For instance, it provides an *if-else* clause and a *for* loop. These seem like dynamic features, but the flow behind the scene is that the template will be processed by Django (e.g., the *for* loops is expanded and all variable references are resolved) before the final Html file is generated and passed to the browser.

For presenting a model object, there is one thing very useful called *model form*. Suppose we have defined an activity model class with title and description fields in it. Then we can bind to an activity object a model form that will be rendered in Html with a textbox and text-area in it. The Html code for the form is generated automatically by *Django*. Based on that, we can also define our own *filters* to display our form with different pieces of Html code.

To design the layout, we use *CSS*(Cascading Style Sheets) file to change our web page's layout by assigning differentiating attributes (i.e., `id` and `class`) to Html elements. We use external jquery [15] libraries such as *Datatables* to render table data. *Datatables* has comprehensive built-in features like sorting, searching and paginating[1], which makes the web pages more interactive.

2.4 Division of responsibilities

As our project leader, SFU Computing Science faculty member Greg Baker is basically responsible for coordinating the four modules and the overall quality assurance. He also lays the database foundation, implements notification and news feed functionalities and deploys the system onto real servers. The four modules of this project are realized by different SFU Computing Science students. SFU Computing Science student Vincent Zhao is responsible

[1]Pagination here means consecutive numbering to indicate the proper order of the entries in a table to show

for the grades module. SFU Computing Science students Youyou Yang and Jiangfeng Hu take charge of the submissions module. SFU Computing Science students Yiran Zhou and Xiong Yi are responsible for the grouping module. Finally, I am responsible for designing and implementing the marking module.

Before I go into the details of the marking module starting from the next chapter, here is some clarifications worth of mentioning. In the following chapters, we will use course *staffs* to mean instructors or TA's that can do administrative works on marking. The *activities* in the following discussion will only refer to *numeric activities* because only numeric activities can be marked. I will use words *grade* and *mark* interchangeably to mean the same. The word *marking record* will also appear very often. A *marking record* contains not only information about the mark itself (e.g., the mark value, late penalty, file attachment), but also the context information including by whom and when the mark is given.

Chapter 3

Use Cases Analysis

Use cases should be analyzed before defining data models and designing user interfaces, so in this chapter, we are going to see the details of each use case in the marking module. I also selectively include some figures of my implementation results. Note that the data shown in these figures is not real. The main use cases in the marking module fall into two general categories: activity configuration and marking [1].

3.1 Activity configuration

As I have mentioned in the introduction, instructors usually want to divide an activity into components, giving each of them some points. I will call them marking components or simply components if there is no ambiguity. For each component, the instructor may want to define a bunch of common problems found in student's submissions so that the descriptions of common problems can be reused in the comments or feedbacks given to a student about how he/she does on that component.

3.1.1 Add, edit or delete marking components

By following a link from the activity information page, the staff can go to a page containing all the components of that activity, edit or delete the ones that already exist and add new ones by inputting its title, description and the max mark out of the total mark of the activity.

The staff can also change the relative order of the components (i.e., put the more

important ones in front). In other words, the positions of the components can be modified. The components will be presented in a table with each row representing one component. By clicking the arrows on the left of each row, a staff can swap any two adjacent components. (A similar feature is provided for reordering activities on a course information page in the grade-book as well.)

3.1.2 Add, edit or delete common problems

By following some link from the activity information page, the staff can also go to a page showing all the common problems of the activity. He can edit or delete the ones that already exist. He can also add a new common problem by selecting a component of this activity to associate with, the title, description and the associated penalty. The student who has this problem may or may not be penalized to the extent suggested. These penalty values are not enforced but merely suggestive.

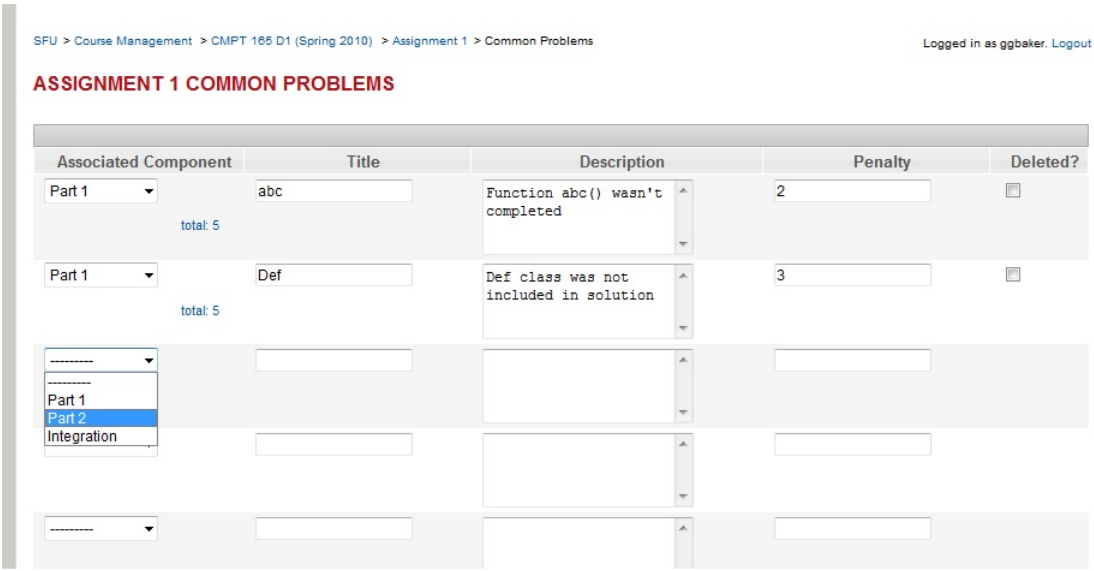

Figure 3.1: Common problems configuration

As shown in Figure 3.1, for each common problem, whenever you select the associated component, it tells you the max mark of the component (at the corner below the drop-down box) and you should specify a penalty value no higher than this value. The user interface for configuring the marking components is similar to this one.

These configurations are preparations for marking to be conducted, because we will see

that typically staff can mark an assignment component by component and specify which common problems a student (or a group) has in the submission.

3.1.3 Copy course setup

An instructor can copy the setup of one course to another course. Normally, this is done for the same courses in different semesters. Setup here is a general term, which includes all the activities (numeric or letter) in a course. For each numeric activity, all its marking components, common problems and submission components will also be copied. This functionality saves a lot of time for instructors. We can call the course that the instructor is going to setup the `target`, and the one whose setup will be copied the `source`.

There is an uncommon but important case here: if some activity in the `target` already has the same name (or short name) as that of any activity in the `source`, we will warn the user that he/she should rename that activity in the `target` in order to avoid it being overwritten by the one in the `source`.

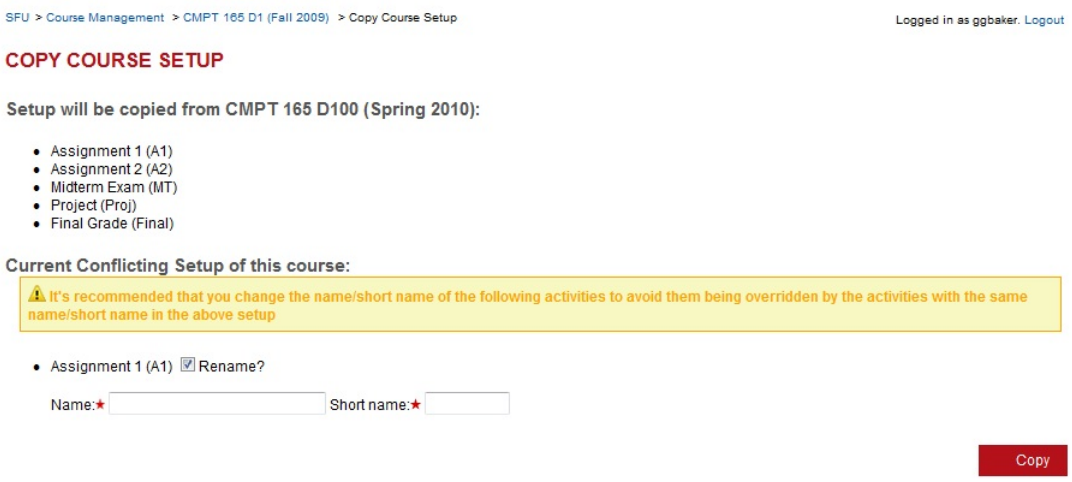

Figure 3.2: Copy course setup

In Figure 3.2, we see that the instructor `ggbaker` wants to copy the setup from CMPT 165 D100 (Fall 2009) to CMPT 165 D1 (Spring 2011). There are five activities in the source setup. However, the target setup already contains `Assignment 1` which conflits with `Assignment 1` in the soruce setup. The warning text recommends the instructor to rename this activity in the target setup to resolve the naming conflict.

3.2 Marking

3.2.1 Mark for one student or one group

On the activity information page in the grade-book, the staff will be presented a table showing all the students with their marks if they are graded or otherwise with `no grade` signs. They can pick up one student who has not been graded yet and click an icon on this row to enter the marking page. On this page, all the components are shown side by side with their title and an array of common problems if any. The input fields are the comment and the mark value. There are fields where staff can input additional information including late penalty (in percentage), score adjustment with the reasons, overall comment and file attachment. Except for the mark values, all other fields are optional (i.e., can be left blank).

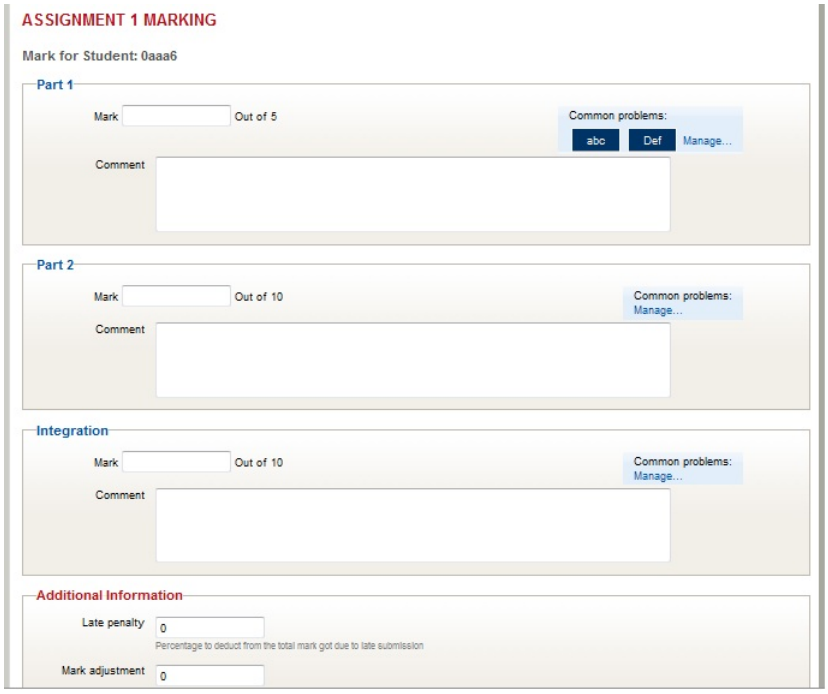

Figure 3.3: Marking component by component

Figure 3.3 shows the user interface for marking `Assignment 1`. There are three components called `Part 1`, `Part 2` and `Integration`. Each component will be given a mark and comment. At the bottom, the instructor can input additional inforamtion into those fields (they are not completely shown here).

If the activity is a group activity, all these can be done for a group, too. The members

in a group share the mark information. Each of them sees the marks given to their group when logging on to grade-book. The interface for showing group grades and for picking a group to mark has a two level list style. The first level contains the list of groups and below each group there is a second level containing its group members. The staff can click an icon to show or hide the second level.

3.2.2 Give marks to all students

The above is the usual way a staff gives a mark to a particular student or group where they need to give marks to components of the activity. However, instead of giving marks to components, sometimes a staff wants to bypass this and give a total mark to a student or group directly to save time. He can do this by clicking a `Mark All` button and go to a table containing rows for all the students or groups with a text box on each row. He can enter a mark into a text box or leave it blank (if he decides not to mark the student or group on that row), then submit the whole form.

3.2.3 Import/export all students' marks

In the previous use case, if a staff has at hand a CSV (comma-separated values) file containing the students' grades, the data can be imported. The contents of the file should abide by a special format such that each row starts with a student's *user-id* or *student number* followed by a decimal grade with a comma in between. If the content of the file is error free, the result will be presented to the staff for reviewing before submitting. On the other hand, a staff can also export all students' grades for a particular activity into a CSV file as well.

Figure 3.4 is a picture taken when the instructor `ggbaker` is marking all students on `Assignment 1` and he has already successfully imported a file containing eight students' marks. He now is reviewing these marks to make sure they are correct. Note that even though the first student `0aaa0` has already been marked, the mark still can be overwritten if needed.

3.2.4 View marking summary

By clicking a `Show Details` icon beside the student's grade in the grade-book, a staff can go to a page showing the detailed summary of that marking record. It contains the

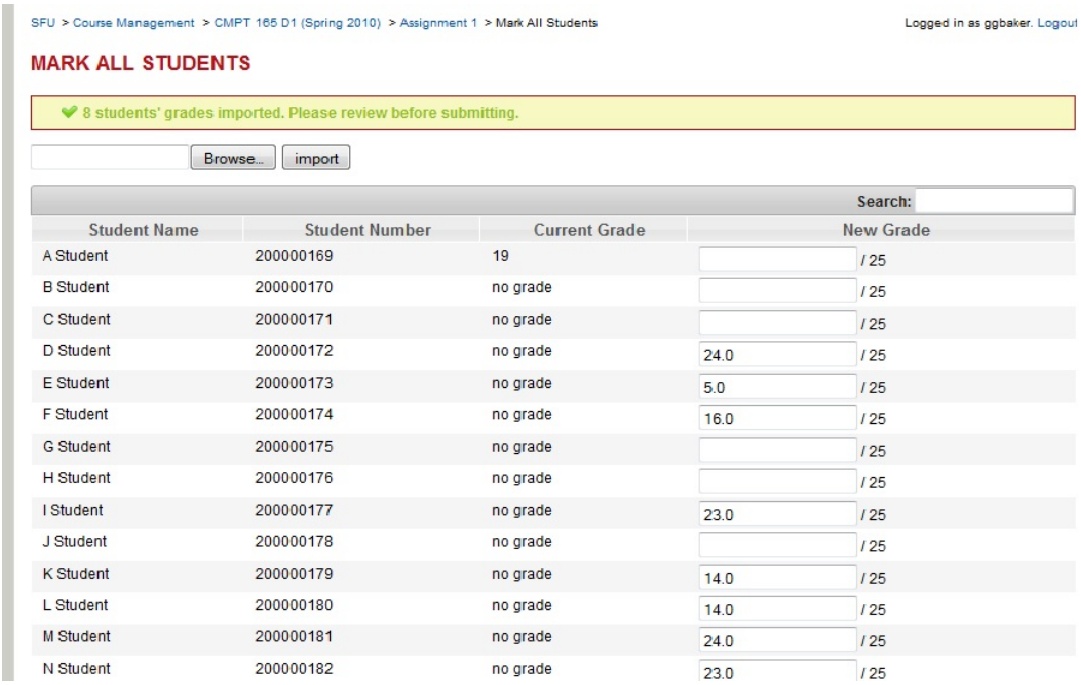

Figure 3.4: Marking all students with imported data

following three pieces of information:

- Basic information including who created the mark at what time, how this mark is given to this student either individually or via the group this student is in (if this is for a group activity).
- Additional information as described in Section 3.2.1. A staff can download the attachment to review.
- The marking details on components, namely the comment and mark given to each component of the activity (this information may or may not be available depending on whether the mark was assigned component by component or only a total mark was given).

Figure 3.5 shows that students' grades are presented in the activity information page in grade-book. For those students who have not been graded yet, a `no grade` sign diplayed in the `Grade status` column and you can mark them by clicking the `paper and pencil` icon. For those who have been graded, their marks and links to their submission are shown in the

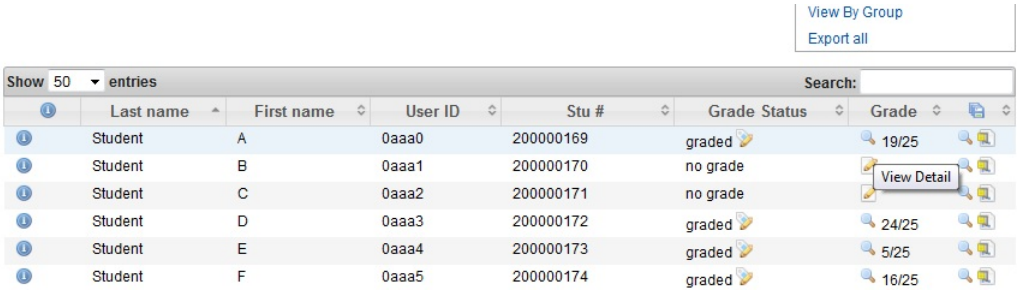

Figure 3.5: Student grades displayed in grade-book

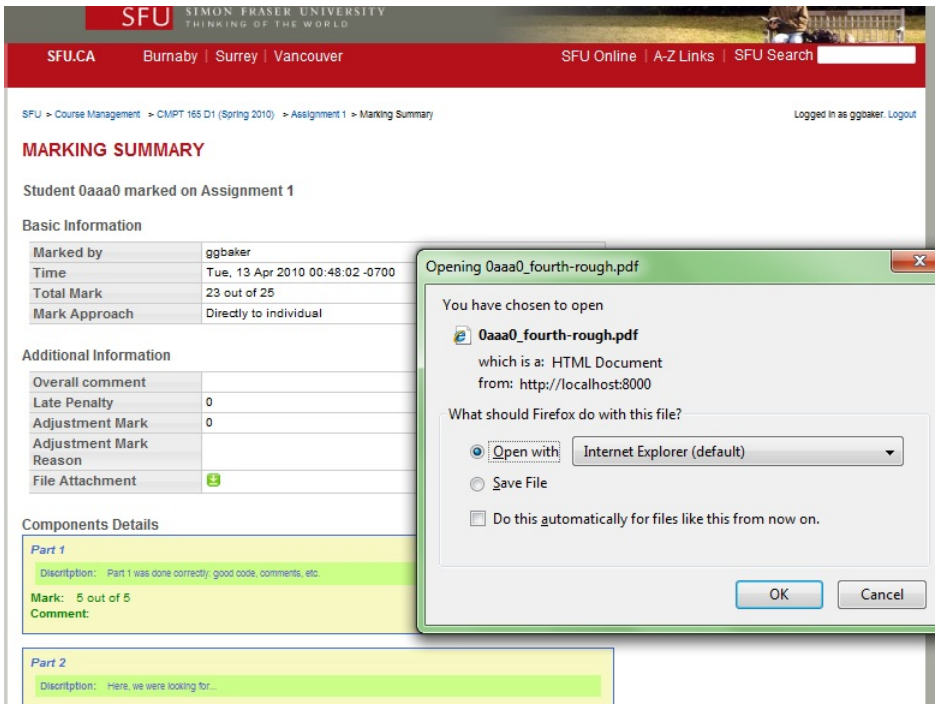

Figure 3.6: Marking summary

last two columns. By clicking a `reading glass` icon, you can go to the summary page as shown in Figure 3.6 below (in this case, it shows the marking summary of student `0aaa0` on `Assignment 1`). We can see the basic information section and the additional information section. The attachment has been downloaded and the user is going to either open it or save it. Lower at the page are the details of the marking information for each component in `Assignment 1`. For example, the student got 3 out of 5 for `Part 1`.

Provided that the grades on the activity have been released, a student can also see the same summary by clicking a `Show Details` icon besides the grade entry on his/her corresponding grade-book page.

3.2.5 Marking based on previous marks

From the marking summary page, a staff can give a mark to the same student or group again based on this marking record. The fields in the marking page will be filled with the same information as the original marking record and staff can begin modifying it. This functionality is convenient. It may happen that the staff gives mark to same student or group multiple times and different staffs give a student different marks at different times. All this information sometimes may be of interest. So another use case follows.

3.2.6 View marking history of one student or one group

The system keeps the records for each mark given in the past. All the marking records will be shown saying who gave the mark and when. All the marking records associating with a particular student on one activity will be displayed in chronological order. For a group activity, a staff can view all the marking records for one group as well. A student may be graded individually even for a group activity (e.g., the instructor gives him bonus mark for his prominent performance). In this case, for each marking record we also show by which method that mark is given, namely directly to the individual or via his/her group. Similarly as described in Section 3.2.4, a staff can view the details of each record and just follow a link besides it and even assign newer marks based on this.

In Figure 3.7, we see that student `0aaa0` has been marked on `Assignment 1` three times individually, twice by `ggbaker` and once by `0grad`. The group of this student has also been marked on this activity once by `ggbaker`. Only the mark shown at the top (given by `0grad`) is the valid one because it is the latest. Although this example seems to be uncommon in

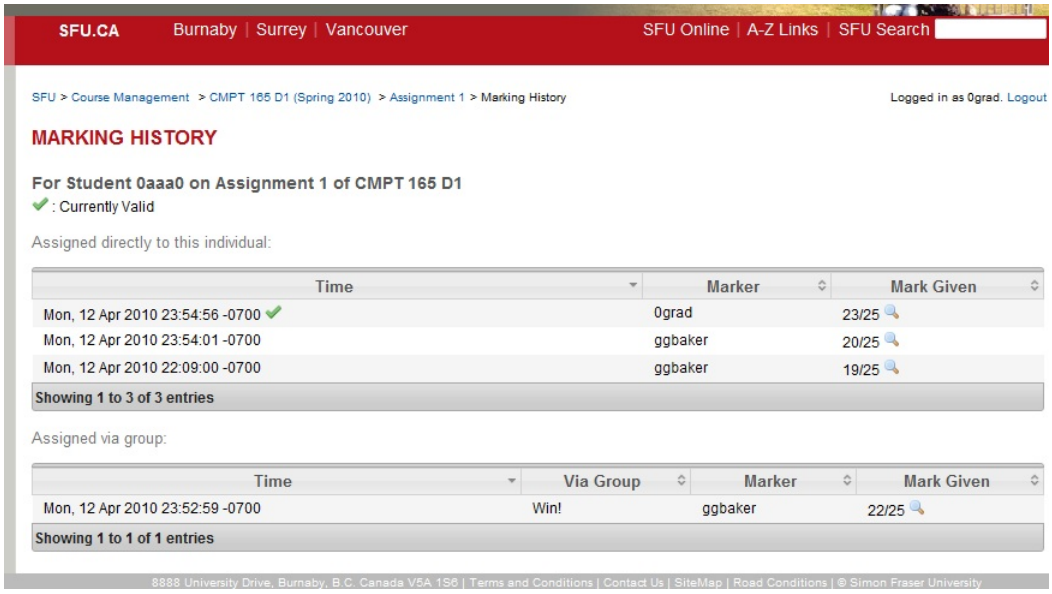

Figure 3.7: Marking records history

real life, it illustrates that the database keeps all the marking records for each student and each group. And for a group activity, a student can still be marked separately from other members in his/her group.

3.2.7 Change grade status

In very rare situations, a staff may set the status of a grade to `Academic Dishonest` due to plagiarism or `Excused` for reasons like student's sick leave. The staff can change this status after the mark has been given, and the student will get a severe notice when he/she logs on to the grade-book. Chapter 8 will take this use case as an example to demenstrate some concepts in implementation.

These use cases provide a basis for designing our data models in the marking module. Now we move on to the design of these data models to see how they support all these use cases.

Chapter 4

Data Model Description

In Django, we can define a model class to represent a type of object. One model essentially represents a relational table schema in the database with each row being an object of that class.

4.1 What the module needs

To summarize functionalities in the use cases we have seen, as far as the marking module is concerned, it should know:

(1) How to associate components to an activity.

(2) How to associate common problems to an activity component.

(3) How to give a mark for a student on one activity (and also its components).

(4) How to give the additional information of a marking record (e.g. the late penalty, overall comment, etc.).

(5) How to give a mark to a group on one activity (and also its components).

Note that, the marking module uses some data models defined in other modules. Now let us have a look at how the marking module uses these models in realizing the above six functionalities.

4.2 What we have from other modules

First, two data models *CourseOffering* and *Member* are used in all modules. The membership refers to the role of the person enrolled in the course offering (it could be instructor, TA or student). In grades module, there are two important data models connected with the marking module. One is *NumericActivity* (a subclass of *Activity*) which contains information of a numeric activity such as its name, due date, percent toward final grade and its max grade. The other one is called *NumericGrade*, which stores the grade one student gets on one activity. There can be only one such object for each student and activity in a certain course. It contains the information of the student membership to one course (a foreign key field to *Member*), the activity (a foreign key field to *NumericActivity*), a mark value and a grade status flag such as `No Graded`, tt Graded, `Excused`, etc. as menstioned in Section 3.2.7. The group module has classes *Group* and *GroupMember*. *GroupMember* contains a foreign key to *Activity* telling us that for which activity the student joins the group. We need to handle two tasks with regard to these two data models. First, given an activity, we want to find all the groups to mark (use case in Section 3.2.2). Second, given a student and an activity, we need to find which group he/she joins so that we can know if any mark has been given to him/her via the group on this activity (use case in Section 3.2.6).

4.3 Data model definitions and relationships

To deal with (1) and (2) listed in Section 4.1, the marking module needs to define two types of classes. The class *ActivityComponent* should have a foreign key to the *NumericActivity* and the class *CommonProblem* should in turn have a foreign key to the *ActivityComponent*. To implement (4), intuitively, we want a class *ActivityComponentMark* with mark value, comment and a foreign key to *ActivityComponent*.

Although in (3) the marking records are for an individual student while in (5) they are created for a group, they both will have the additional information mentioned in (4). Based on this rationale, we define three classes using an inheritance structure: the super class *ActivityMark* that only contains additional information and two sub classes *StudentActivityMark* and *GroupActivityMark* for individual student marking and group marking respectively. Therefore, a *ActivityMark* object represents a marking record and its specific type is either *StudentActivityMark* or *GroupActivityMark*. In *StudentActivityMark*, the only

field is a foreign key to *NumericGrade*. To give mark to an individual student, it actually saves the mark to the associated *NumericGrade* object. In *GroupActivityMark*, there is a foreign key to *Group* and a foreign key to *activity*, basically telling the mark is for which group on which activity. To give mark to a group, it actually finds all the members of the group and sets the mark in the *NumericGrade* corresponding to each of the group members. So for each individual student, the valid mark version is stored in the *NumericGrade* of that student. Since this object only contains the latest version of the grade, if we want to view the history marks, we have to keep a copy of the mark in every *ActivityMark* object because a new instance of this class (either as *StudentActivityMark* or *GroupActivityMark*) is created every time a mark is assigned.

To view the details of a marking record, we have fields `created_at`, `created_by` in an *ActivityMark* object. Their meaning is obvious as their names indicate. The *created_at* helps to sort the marking records for one student in time order, so it is used in selecting the valid marking record (i.e., the latest one) and showing the marking history. When viewing a marking summary, the user wants to know what mark is given to each activity component. Since the information is stored in *ActivityComponentMark* object, in it we put a foreign key to *ActivityMark* so that given an *ActivityMark* object we can easily find the marking information on all the components by following the foreign key relationship in reverse.

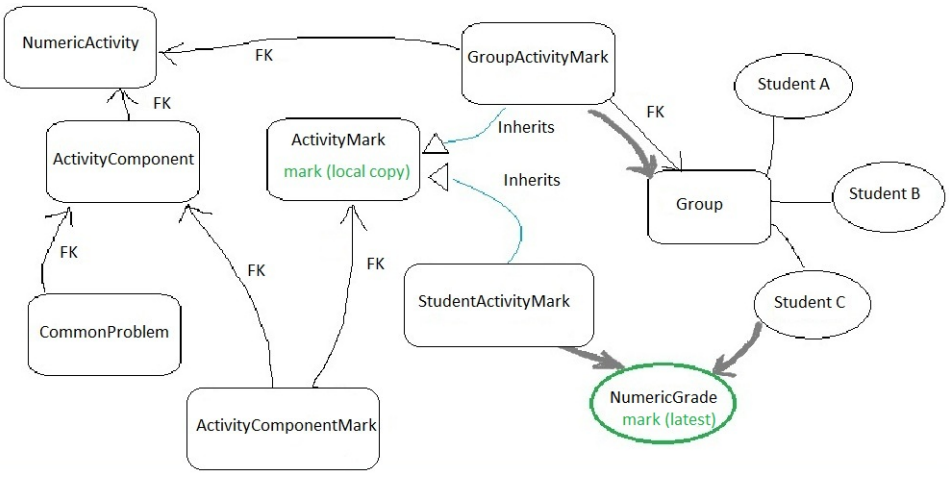

Figure 4.1: A sketch of data models and their relationships

All these relationships are shown in Figure 1. Particularly, as shown by the bold arrows, we can see that for both individual marking and group marking the mark values are always

finally saved to the *NumericGrade* objects. However, because a *NumericGrade* object only holds the mark value assigned most recently, the *ActivityMark* object also keeps a local copy of the mark value as a history record.

Data models are always crucial and they are the backbone of the whole marking module. With all these classes of data models defined, we now know how the tables in our database will look like. Now when user web requests arrive, we consider how to interpret them, which query to make on the database and which data we need to process and save to the database. These tasks are performed in the view layer.

Chapter 5

Making Requests

In this chapter, I am going to talk about the characteristics of Http requests. As we know, one request is always for one URL and it either gets data from the server or sends data to the server.

5.1 URL design

Recall that Django maintains a one to one mapping from URL's to views. To design neat and meaningful URL's, we usually insert meaningful words into URL's. For example, `/students` or `/groups` is inserted as a portion to URL's to distinguish marking for individual students from marking for groups. The general words do not refer to any particular resource or object we want to access. So we need identifiers of the objects in URL's and a view can retrieve the objects of interest from database.

5.1.1 Use identifiers in URL's

Most of the time identifiers should be as readable as possible. We use a python package called *autoslug* that produces a unique identifier from a data model object (virtually a database row) and we specify which fields are to be used. For instance, in the *CourseOffering* data model, we use all its fields as the source to populate a unique string and it gives us `1101-cmpt-165-d100` for CMPT 165 D100 in spring 2011. To identify an activity belonging to some course offering, we use the *slugs* both of the course offering and the activity in the URL like `http://server_base/1101-cmpt-165-d100/a1/marking/`.

5.1.2 View function parameters

Every type of identifier (e.g., `slug`, `id`, `user-id`, etc.) usually has one fixed regular expression pattern. This is a very important property because in the mapping from URL's to views, we specify the type of URL's that match certain pattern. We can also specify which portion of the URL should be a *parameter*, meaning it can be extracted and passed as a parameter to the corresponding view that this URL maps to. In almost all cases in the marking module, we need at least two portions of the URL to be used as the parameters: the course slug and the activity slug. Typically, when these two are passed to a view function, it would first use them to access the database to get the *CourseOffering* and *NumericActivity* objects.

5.1.3 URL parameter list

A view function may be invoked in different scenarios. For example, a user may want to see the marking summary either of a marking record shown in the marking history or of the marking record associated with the current valid mark (i.e. most recently created). In my implementation, in both cases, the requests go to the same view called `marking_summary`. For the first case only, we could just put the `id` of the *ActivityMark* object into the URL. But obviously this method does not work for the latter case because we don't really know the `id` beforehand. So instead, we append a URL parameter list (although here only one parameter) like `http://base/1101-cmpt-165-d100/a1/marking/students/0aaa0/?activity_mark=5` to our URL to get the marking record whose `id` is 5. The URL-to-view mapping ignores this portion and once the correct view gets invoked, it examines the URL parameter list in the *request* object (which is always the first parameter passed to view functions): if an `id` is present it just fetches that *ActivityMark* object; otherwise it finds the most recently created *ActivityMark* object according to the *created_at* field.

5.1.4 URL referencing

A good practice is that to obtain the URL that maps to some view, we use function `reverse` and pass the name of the view function as the parameter. Rationale behind this is that we may change URL's much more often than view names. So we should avoid hard coding URL's as far as possible.

5.2 Http GET and POST requests

There are two basic types of Http requests. One type asks for data from the server and the other sends data to the server. We call them Http GET and POST requests respectively.

5.2.1 Http GET

In the marking module, a GET request only carries information in its URL and its parameter list. Generally, we need to first do database queries to retrieve the objects we need. One complicated query in the marking module is to get all the marking records for one student on one group activity. Since we know that the marks may be assigned directly to that individual student or via the group he/she is in, we need to query the mark records from tables for both data models *StudentActivityMark* and *GroupActivityMark*. Assuming we have the *NumericActivity* object named (`activity`) and the *Member* object named (*student_membership*, the queries might be:

- num_grade = NumericGrade.objects.get(activity = activity, member = student_membership)

  marks_to_individual = StudentActivityMark.objects.filter(numeric_grade = num_grade)

- group_mem = GroupMember.objects.get(student = student_membership, activity=activity, confirmed = true)

  if group_mem exists:

  marks_via_group = GroupActivityMark.objects.filter(group = group_member.group)

Here, we use method `get` expecting exactly one object returned from the table and use method `filter` to get potentially any number of objects. The parameters to these specify the conditions that each qualified row in the table must satisfy at the same time. The variable `marks_to_individual` will contain all the marking records created for this student individually on this activity. If the student does join some group on this activity, the variable `marks_via_group` will contain all the marking records for that group on this activity.

After these objects have been retrieved, the view will organize and encapsulate them into an Http response that will be rendered with a Django template. We will talk more about this process in the next chapter.

A GET request can ask for file data, too. As mentioned in the Section 3.2.4, this happens when a user downloads the file attachment in a marking record. The View acts differently for this type of request. It writes the data in the file to Http response object and sets special headers: `Content-Disposition=‘‘attachment;filename=[name of the file]’’` and `Content-Length=‘‘[the size of the file in bytes]’’`. For exporting students' grades into CSV files, we set the Http response object with `mimetype='text/csv'` and use Python *csv* package to write data to the Http response object [3]. Rather than rendering a new Html webpage, the browser pops up a dialog prompting users to save or open the download file.

5.2.2 Http POST

Other than the information in the URL and its parameter list, a POST request also carries the input contents submitted with an Html form by the user. For instance, you can submit one text input, one drop down selection and an uploaded file (e.g., by clicking a `Browse...` and selecting a file) all together in an Html form. To allow file uploading, we need to set `enctype=‘‘multipart/form-data’’` for the Html form element [5]. A POST request object has a dictionary-like dataset of the non-file input contents called *POST* and another dataset called *FILES* for file data [9]. We use these two datasets to construct a Django form and then get thevalue of a field in the form with its name as the key. We will talk more about the Django form in the next chapter.

Usually, it is a good practice to redirect to an appropriate page after we successfully submitted the form [11]. So one problem arises: we may go to the same page from different pages, so how can we return to the correct page? To solve this, we use a parameter like `?from_page=pageA` in the URL and the view can check what the previous active page is and return to it. This is a technique commonly applied in the grades module as well.

A POST request usually corresponds to a GET request and they are addressed to the same URL. A GET request is sent first to get a web page asking the user to input data when a `Submit` button is clicked, a POST request is sent addressed to the same URL. Hence, the view that the URL maps to needs to handle both types of requests. It decides the request type by the `method` attribute of the request object. For both types of requests, the view first retrieves the objects identified by the URL from the database. Then for the GET request, the view creates a Django form (or forms) that will be presented to the user for input. For the POST request, the view constructs a Django form (or forms) from the data contained

in the request object, analyzes them and saves updates to the database if needed. Chapter 8 will give a detailed example of this process.

Chapter 6

Generating and Rendering Responses

We have seen how requests are sent to a view. Now let us see how responses are generated by a view and how the information in the response is passed and rendered with Django templates. Particually, we will see how the Django model form and fromset provide a fast solution for representing objects from the database with Html forms.

6.1 Generating responses using model form and formset

As discussed in the Section 2.3, we define a form with various types of fields which will be automatically rendered by Django as a group of Html input elements (or widgets). The most exciting thing is that we can associate a model form with our data model. The fields of such a form will correspond to the fields of the model. We can choose which fields in the data model are actually used in the form. Although you can override the input widget for each field, in the marking module, I just use the default choices by Django because they suffice.

In the marking module, I define *ActivityMarkForm* which is a model form for Activity-Mark and *ActivityCompnoentMarkForm* which is a model form for *ActivityComponentMark*. You may wonder why I do not define a model form for *ActivityComponent* and *Common-Problem*, the reason is that there is an even more powerful thing: model formset. A formset by its name is a collection of forms. For a model, we can use a model formset factory to

define a class of formset for our model. Basically it's a collection of model forms, each one corresponding to one object of that model. We can specify the fields of the model we want to include and the base class of the formset if desired (we will see this usage shortly in next section), a *queryset* (list of the model objects we fetch from the database satisfying certain conditions) we want to initially fill into the formset. This is very useful considering that if we want to just display a list of components belonging to activity `Assignment 1`, we can pass in a queryset telling we just want the components of `Aassignment 1`. We can also pass in the parameter `extra` to tell how many empty rows to display. In marking module, I use this technique for the purpose of adding new activity components and common problems.

6.2 Validation

The validation is used when a user sends an Http POST request containing the data of a form or a bunch of forms (e.g., a formest). We have to validate that the data is legal and can be saved to the database safely. Model form and formset do their default validation automatically according to the restrictions on their associated model (e.g., you cannot enter letters into a text field representing a decimal number field of the model). We can define extra checking logic for a single model form by overriding its method called *clean* (e.g., we can rule out negative numbers for the maximum mark of a component which is legally decimal by default). This is validation on form level [6].

6.2.1 Formset validation

There is another level of validation: formset level validation. We need to validate some attributes that all the forms in the formset as a whole should not violate. For example, each of the components for one activity should have a unique title. With each model form in the formset associated with a component, even if each single form is valid, the titles of two or more components may still conflict. This validation is defined in the formset class and that is the very usage of the formset base class: we define the validation logic in that base class. By passing it to the model formset factory, we can generate a model formset that inherits this extra validation ability.

6.2.2 Typical work flow

So the typical work flow of a view handling Http POST request is as follows: If the validation passes, we process and save data to the database if needed, and finally redirect to an appropriate page. On that page a success message is displayed. If there are any warning messages they get displayed, too (e.g., a mark value higher than the max mark is entered which may or may not be the user's intention). If the validation fails, the erroneous forms is passed in the response and the user will see an error message indicating that submission fails. There also will be a specific error message beside each invalid form field so that the user will know how to correct it.

6.3 Rendering response with templates

Let us now look at how a view passes the response for rendering and how a template uses the contents in the response. Templates are Html files, but because they are specific to Django they need to be processed first before the browser can understand them.

6.3.1 Tags and variable references

A view produces a dictionary-like context object encapsulating all the information the template needs. The key-value pairs in the dictionary are the variable name and the actual variable that are referenced in the template. In the templates, these variables as well as their fields and methods (only those with no parameters) can be referenced. Tags in a template like `for` loops and `if-else` clauses are surrounded by {% and %}. References to a variable or its method or field are surrounded by {{ and }}. So Django interprets and executes the tags, replaces the variables by their string presentation (i.e., the `str` method, replaces the variable field references with their values and replaces variable method references by their output. Then the resulting Html is handed to the browser. For example, when displaying a list of components of an activity, in the template we loop over the list variable and for each element in it, we access its title, description and maximum mark. Django expands the loop, replaces the references to the fields with their actual string values. The browser then will receive the resulting regular Html file.

6.3.2 Filters for displaying forms

There is another concept called *filter* that we can apply to a variable in templates. A filter is just a python function we define that takes a variable and returns an Html code of how it should be displayed. We have defined several filters for displaying a form. A form can be displayed as a list with each field being an item (within) or as a row in a table (within <tr></tr>) with each field being a table field. Furthermore, the filter also displays a `cross` icon beside any field in a form that has an error such as ‘‘`this field is required`’’. We use filters to avoid repeating writing the same Html code on multiple templates which is more time consuming and difficult to maintain.

Chapter 7

Other Specific Topics

7.1 Security

7.1.1 Decorators for authorization

We have seen that a course staff can give marks to students and view their marking history. Whenever a user makes a request to some URL, the user's role in the course should be checked: he/she must be the instructor or one of the TA's of that course. The course can be queried by the course slug portion in the URL. This is the basic authorization mechanism and is implemented by python *decorators* which we put before the views. The mechanism of *decorators* is as follows: "the original function is compiled and the resulting function object is passed to the decorator, which does something to produce a function-like object that is then substituted for the original function"[12]. That essentially means that the decorator tells the runtime compiler to modify the code of the view function to actually do the authorization checking logic first. Another way is to imagine that the decorator has its own "body" and the code in it is for authorization checking. It is executed before the view is entered. In our system, all modules share the same set of authorization decorators.

7.1.2 URL integrity checking

Another type of security checking is the URL integrity checking. It ensures the information represented by the URL as a whole is consistent. This checking is implemented in every view function and it needs to be done only when the authorization checking

has passed. Integrity checking usually is the first step the view does. The view continues only after this checking has passed. For example, when an instructor marks a student's assignment, instead of following links on the web page (which guarantees the correctness of the URL), he manually types the student's user-id `abc12` in the URL like `http://server_base/1101-cmpt-165-d100/a1/marking/new/students/abc12`. We need to check that the activity does belong to that course and the student does participate in the course. This checking usually involves retrieving *CourseOffering*, *NumericActivity* and *Member* objects with proper query conditions first. If they can be retrieved successfully, the checking passes. Otherwise a `''Page Not Found''` error with Http code 404 is returned.

7.1.3 Claim ownership for submissions

When a staff wants to assign a mark to a student's electronic submission, he has to first claim the ownership of the submission so that if someone else (i.e., another TA or instructor) wants to mark this student on the same activity, they will get warning messages saying this student's submission is being or has been marked by another staff. This will avoid the situation where one staff is marking or has marked and another one comes along and begins marking without being aware that someone else is marking or has marked it. So whoever finishes later will override the marking results made earlier, which is somewhat unusual. Actually the steps of checking, claiming ownership and issuing warning messages are implemented in the submission module and the request will first be sent to a proxy view in the submission module. If conflicts are found, the user will get noticed. If there is no conflict or the user wants to take over the ownership and continue to mark, the request will then be redirected to the view for actual marking in the marking module.

This is an important place where marking system interacts directly with submission system. When you mark, you can click a link to view the submission details, which is convenient to users.

7.2 Client-side presentation and interaction

In designing user interfaces, one principle is that we should present data neatly and interactively to users.

7.2.1 Layout styles

As intruduced in Section 2.3, the *Datatables* library offers a clean style to present table data with rich interactive features including searching, sorting and pagination. These features can be disabled optionally. For example, when displaying activity components in a table, since normally there will be no more than a couple of components for a single activity, I can disable the pagination feature. We also use another jquery user interface package that contains animations and images for styling. In many places, we use an element called *fieldset* to encase an Html form for nicer appearance.

The background styles of our system references to the *CSS* files for websites in SFU. We also add more customized styles for the layout. We also unify the styles for the whole system by defining classes that are used across different web pages. For instance, we design styles associated with `button` class of <a> (i.e., Html link element) for status including `hover`, `active`. This type of link is used across all the four modules.

7.2.2 A tricky issue

As mentioned in Section 3.1.1, to reorder activities for a course or marking components for an activity, there is an issue as to how to make it interactive. We display an `up` arrow and a `down` arrow in the first cell of each row in the table (each row represents an activity or marking component). When the `up` arrow is clicked by the user, the current row will be swapped with the previous row; when the `down` arrow is clicked, the current row will be swapped with the next row.

My first approach is to make these arrows act as links and whenever a click on an arrow occurs, an Http request with the position information of the two rows is sent to the server. The view corresponding to this URL updates the positions of these two components and returns an Http response showing the new ordering of the components. So the new result will be presented when the page is reloaded. While this is a working approach, it is not satisfactory in terms of the degree of user interaction. Reloading the webpage means the user has to wait for response until the page suddenly moves to its top location once reloaded. This is very unnatural to user's eyes.

So instead of reloading the page every time, I decided to use *Ajax* approach. It sends asynchronous POST request containing the identifiers of the two rows to the server and the server does almost the same work as the first approach. Specifically, this is realized by `ajax`

`post` method in *jquery* library. In a *call-back* function, I need to tell what action to take when the server has done the update (i.e., the server returns a successful Http response) [14]. Intuitively, the action should be to swap the two rows using the Datatables API (i.e., update one row's content to that of the other and vice versa). I introduce this method when reordering activities for grades module. However, for reordering the marking components, I use another slightly different method: once any arrow is clicked, change happens on client side only. When the user clicks an `Update` button, the new position information of all the components will be sent to the server in an *Ajax* POST request and the server updates the ordering. In this case, when sever sends back a successful response, the action of the call-back function is to pop up an alert window telling the user the new ordering has taken effect.

Using *Ajax* approach, both methods avoid reloading the page while the later one is more responsive because the user does not need to wait for the server to finish processing the request in order to see the swap of rows happen. But the trade off is that the user has to click the `Update` button in order to save the new ordering on the server.

Chapter 8

A Concrete Example

If you are interested, here is a good example for your reference because it embodies many important concepts described in Chapters 5, 6 and 7.

8.1 View function

Let us first have a glance of the view function:

```
1. @requires_course_staff_by_slug
2. def change_grade_status(request, course_slug, activity_slug, userid):
3.     course = get_object_or_404(CourseOffering, slug=course_slug)
4.     activity = get_object_or_404(NumericActivity, offering=course, slug=activity_slug)
5.     member = get_object_or_404(Member, offering=course, person__userid=userid, role='STUD')
6.     numeric_grade = get_object_or_404(NumericGrade, activity=activity, member=member)

7.     error = None
8.     if request.method == 'POST':
9.         status_form = GradeStatusForm(data=request.POST, activity=activity)
10.        if not status_form.is_valid():
11.            error = 'Error found'
12.        else:
13.            new_status = status_form.cleaned_data['status']
14.            comment = status_form.cleaned_data['comment']

15.            if new_status != numeric_grade.flag:
16.                numeric_grade.save_status_flag(new_status, comment)
17.                messages.add_message(request, messages.SUCCESS,
18.                    'Grade status for student %s on %s changed!' % (userid, activity.name,))
19.            return _redirct_response(request, course_slug, activity_slug)
20.    else:
21.        status_form = GradeStatusForm(initial={'status': numeric_grade.flag})
```

```
22.    if error:
23.        messages.add_message(request, messages.ERROR, error)
24.    context = {'course':course,'activity' : activity, 'student' : member.person,
25.               'current_status' : FLAGS[numeric_grade.flag], 'status_form': status_form}
26.    return render_to_response("marking/grade_status.html", context,
                                       context_instance=RequestContext(request))
```

This piece of Python code is a view function handling user requests for changing a student's grade status on some activity as mentioned in Section 3.2.7.

It has four parameters (line 2): the first one is an Http request object and the following three are identifiers of the course, activity and the student. These identifiers are extracted from the URL of the request (recall section 5.1.1). An example URL maps to this view is `http://server_base/1101-cmpt-165-d100/a1/gradestatus/abc12/` with the course's slug (`1101-cmpt-165-d100`) with the activity's slug (`a1`) and the student's user-id (`abc12`) in it. We use slugs to identify a course and an activity while we use user-id to identifier a student. Line 1 is a decorator for authorization that checks the user is a staff for this course. The code of the view is run only when the checking has passed.

In the view function, the first step is to fetch the *CourseOffering*, *NumericActivity* and *Member* objects from the database (lines 3 - 6) using the identifiers. At the same time, by specifying proper query conditions, it does URL integrity checking to ensure the activity does belong to this course (line 4) and the student is a member in this course (line 5). Since we assume the grade status can be changed only when a grade has been given, that means in the database there should already be a *NumericGrade* object for this student on this activity, we obtain it, too (line 6). If any one of these four queries fails, which indicates inconsistency in the database, we return ``page not found'' error with Http code 404.

As described by Section 5.2, the request is either GET or POST. Recall that for a view handling both GET and POST requests, typically the GET request is sent asking for a form where user can input data and the POST request is sent once the input is submitted by the user. If it is GET, we switch to line 20. We create a status form object (this is a django form type we have defined which contains a drop-down box for selecting status and a text area for comment). In line 21, the current status (the `flag` field of the *NumericGrade* object) is passed to initialize the selection. We then construct a dictionary object called `context` which contains the course, activity and student objects together with the form object (lines 24, 25). We encapsulate it into the Http response which will be rendered with the template

called `grade_status.html` (line 26). Django will resolve the references to the variables in the `context` object and return the resulting Html file to the browser.

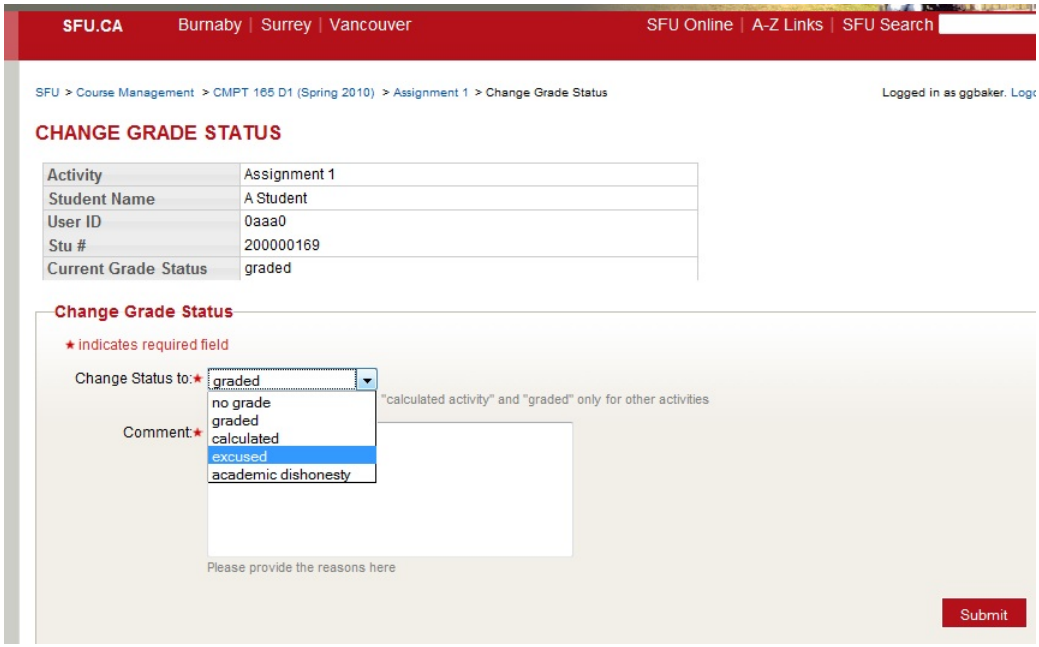

Figure 8.1: web page for changing grade status

Figure 8.1 shows the web page that will be presented to the user as the response to the GET request. We can see that it has information about the activity, the student and the current grade status. That means the template makes use of the variables `activity` and `student` in the `context` object. All these are referenced as fields in the templates. The drop-down box and the text area can be submitted as Html form input data.

If it is POST, the work flow is a little more complicated. First we construct a form object from the dataset contained in the POST request. If the form object passes validation (line 12), we get the two fields (lines 13, 14): the status the user selected and the comment the user input. Now we compare this status with the current one, if they are the same we do nothing because it is unnecessary to save to the database. Otherwise, we save the new status (line 15, 16). The method `_save_status_flag` of *NumericGrade* saves the update to the database and use the comment to create an object for notifying the student about this status change. We then finish by adding a success message and redirecting to another page (lines 17 - 19) where the success message will be shown. If the form validation fails (e.g., the user selected an empty status entry), we fall to line 22 and set an error message (which

will be shown near the top of the page). Then as we do for a GET request, we construct the *context* object and return the same web page `grade_status.html`. But this time the form object contains errors in it because we want the user to correct the invalid fields before submitting again.

8.2 Template

Now, let us also look at a small portion of the template `grade_status.html`:

```
1. {% block content %}
...
2. <div id="form_container">
3.     <form action="" method="post">
4.       <fieldset>
5.        <legend>Change Grade Status</legend>
6.        {{status_form|display_form}}
7.     </fieldset>
8.     </form>
9. </div>
...
10.{% endblock %}
```

The template inherits from `base.html` and overrides the `content` block (lines 1, 10). This part basically shows how we display the `status_form` passed in by the `context` object. We use a filter called `display_form`. It is a Python function we have defined which takes a django form object as the parameter. The syntax is to put this filter after the form object with a — in between. What this filter does is to loop over every input field of the form and output its default Html code plus some styling to it and its error message (e.g., appends a red icon) if there is an error with this field. It also adds a `Submit` button at the end. The filter saves a lot of work because we display many different form objects across many web pages in this manner. With it we just need one line of code.

To improve the layout, we encase the contents of the form object into a *fieldset* element (lines 4, 7) because it gives a nicer style as mentioned in Section 7.2.1. Then we in turn encase this *fieldset* element into an Html form element (lines 3, 8). By setting its method of it to be `post` and leave the action URL empty (line 3), we tell the browser to send a POST request to the same URL of the current page whenever the `Submit` button is clicked. Finally, we enclose the whole Html form element into a `div` element (lines 2, 9). This `div` element has a particular `id` so that it subjects to the styles we specify in our CSS for this

`id`.

Chapter 9

Conclusion

In summary, this report presents the course management web system that our team has been working on during this semester. In particular, it focuses on the marking module of the system which is the part I mainly contributed to. I am glad that I have implemented all the functionalities well.

This project is of big practical value. On one hand, it aims at creating an application to bring convenience to instructors and students in the School of Computing Science in their daily campus life. On the other hand, I hope our adventure with *Django* could be a valuable experience that people can benefit from. I think as an open-source project *Django* is a very powerful, fast and clearly structured web development framework.

However, the system still needs to be improved. Firstly, testing is not solid enough. We have done unit tests for each module, but we are short of systematic testing plans for integration between modules and for various security issues. Secondly, in my implementation, I have not used the Django cache system which lets us save dynamic objects and pages to speeds up response time of server. In my opinion, it should be one of the priorities for future extension.

Bibliography

[1] Greg Baker. Courseman project use cases, 2010.

[2] Django documentation, 2010. http://www.djangoproject.com/.

[3] Genereating csv overview, 2010.
http://docs.djangoproject.com/en/1.1/howto/outputting-csv/#howto-outputting-csv.

[4] Databases supported by django, 2010.
http://docs.djangoproject.com/en/1.1/ref/databases/#ref-databases.

[5] file uploding overview, 2010.
http://docs.djangoproject.com/en/1.1/topics/http/file-uploads/#topics-http-file-uploads.

[6] Django cusotmized form validation documenation, 2010.
http://docs.djangoproject.com/en/1.1/ref/forms/validation/#ref-forms-validation.

[7] Django model syntax documentation, 2010.
http://docs.djangoproject.com/en/1.1/topics/db/models/#topics-db-models.

[8] Django executing query overview, 2010.
http://docs.djangoproject.com/en/1.1/topics/db/queries/#topics-db-queries.

[9] Request and response objects documenation, 2010.
http://docs.djangoproject.com/en/1.1/ref/request-response/#ref-request-response.

[10] Django template documenation, 2010.
http://docs.djangoproject.com/en/1.1/topics/templates/#topics-templates.

[11] Django first tutorial part 4, 2010.
http://docs.djangoproject.com/en/1.1/intro/tutorial04/#intro-tutorial04.

[12] Bruce Eckel. Decorators i: Introduction to python decorators, 2008.

[13] Grade book version 5.00, school of computing science, sfu, 2010. https://gradebook.cs.sfu.ca/.

[14] Jquery ajax post method documentation, 2010. http://api.jquery.com/jQuery.post/.

[15] Jquery basics tutorial, 2010. http://docs.jquery.com/How_jQuery_Works.

[16] Mark Pilgrim. *Dive Into Python*, pages 183–190. 2004.

[17] Mark Pilgrim. *Dive Into Python*, pages 15–28. 2004.

[18] Server-side scripting, 2010. http://en.wikipedia.org/wiki/Server-side_scripting.

[19] What is scrum?, 2010. http://www.scrumalliance.org/pages/what_is_scrum.