

# CS> System User Manual

Jeb Brooks

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
<b>2</b>	<b>Basic Use</b>	<b>6</b>
2.1	General Use . . . . .	6
2.1.1	Logging in/Password Recovery . . . . .	6
2.1.2	The Main Navigation Bar . . . . .	6
2.2	Student Use . . . . .	8
2.2.1	Submitting Homework . . . . .	8
2.2.2	Viewing Grades . . . . .	9
2.3	Grader Use . . . . .	10
2.3.1	Grading Submitted Homework . . . . .	10
2.3.2	Entering grades in the Grade-book . . . . .	15
2.4	Instructor Use . . . . .	17
2.4.1	General Course Settings . . . . .	17
2.4.2	Adding Problems . . . . .	18
2.4.3	Problem Settings . . . . .	19
2.4.4	Tests . . . . .	21
2.4.5	Test Settings . . . . .	22
2.4.6	Writing Tests . . . . .	22
2.5	Wiki Use . . . . .	30
2.5.1	Wiki page permissions . . . . .	30
2.5.2	Wiki syntax . . . . .	31
<b>3</b>	<b>Administration</b>	<b>33</b>
3.1	Server Structure . . . . .	33
3.2	Set-up . . . . .	34
3.2.1	MongoDB Set-up . . . . .	34
3.2.2	File-system Set-up . . . . .	35
3.2.3	RabbitMQ Set-up . . . . .	36

3.2.4	Create config.py . . . . .	36
3.2.5	Flask Front-end Set-up . . . . .	37
3.2.6	Celery Worker Back-end Set-up . . . . .	38
3.2.7	Bootstrap the Administrator Account . . . . .	38
3.3	Managing Courses . . . . .	39
3.3.1	Creating Courses . . . . .	39
3.3.2	Adding Instructors . . . . .	39
3.3.3	Deactivating Courses . . . . .	39
3.4	Creating User Accounts . . . . .	40
3.4.1	Creating via web . . . . .	40
3.4.2	Creating via command line . . . . .	40
<b>4</b>	<b>Command Line Reference</b>	<b>42</b>
4.1	Setting up the Environment . . . . .	42
4.1.1	The scripts directory . . . . .	43
4.2	Assigning Points . . . . .	43
4.2.1	The giveFullPoints Script . . . . .	43
4.2.2	Removing points from removed rubric sections . . . . .	43
4.3	Changing a User Password . . . . .	44
4.4	Adding a user to courses . . . . .	44
4.5	Making an administrator user . . . . .	44
4.6	Adding batches of users . . . . .	45
4.7	Marking Submissions as Done . . . . .	45
4.8	Regrade all submissions . . . . .	46
<b>5</b>	<b>Development</b>	<b>47</b>
5.1	Auto-grader Plug-in Development . . . . .	47
5.1.1	testFileParser . . . . .	47
5.1.2	runTests . . . . .	48
5.2	Late calculator Development . . . . .	49
5.2.1	The Grade List Structure . . . . .	49
5.3	Core Development Practices . . . . .	51
5.3.1	Repository Layout . . . . .	51
5.3.2	One file per page policy . . . . .	52
5.3.3	AJAX vs Redirect . . . . .	52
<b>A</b>	<b>Example Auto-grader plugin</b>	<b>53</b>
<b>B</b>	<b>Example Late Calculator plugin</b>	<b>56</b>

# List of Figures

2.1	The header of the main page before logging in. . . . .	6
2.2	The login page. Reset password is at the bottom. . . . .	7
2.3	The main navigation bar . . . . .	7
2.4	A list of problems for a course. Problems are grouped by <b>Assignment Group</b> . . . . .	8
2.5	The page for submitting files for a problem . . . . .	9
2.6	The My Grades page . . . . .	10
2.7	A list of all currently assigned problems in CS Test and the current grading status of the problems . . . . .	10
2.8	The current status of grading for all submissions in this problem	11
2.9	The top of the grading page . . . . .	12
2.10	The comments and files section of the grading page . . . . .	13
2.11	The grade-book for the course CS Test . . . . .	15
2.12	The grade column for Quiz 1 in CS Test . . . . .	16
2.13	The main course settings page, minus the users lists . . . . .	18
2.14	The problem settings page . . . . .	19
2.15	The problem tests lists . . . . .	21
2.16	The test point allocation page . . . . .	23
2.17	The various maps available to the Picobot auto-grader . . . . .	29
2.18	The wiki page permissions table . . . . .	30
2.19	An example wiki page . . . . .	32
3.1	Possible server configuration for <b>CS&gt;</b> . . . . .	34
3.2	Red: The user Dropdown, Blue: The admin dashboard link . . . . .	39
3.3	The admin users panel . . . . .	41
5.1	HMC-Grader repository structure . . . . .	51

# Chapter 1

## Introduction

The **CS>** system is an all in one submission system and grading platform designed primarily for use with CS coursework. It is designed to support basic grading functionality and auto-grading capabilities. Additionally it is designed to be extended to support new testing systems through a simple plug-in system which can hot-load new plugins on a live instance.

This document will attempt to get you familiar with the **CS>** system and guide you through using and maintaining its various features. **CS>** provides the following major features:

- Logical assignment/problem grouping
- Fully featured grade-book system. Supports adding groups and columns to the grade-book which are not based on student submitted work (e.g.. Participation Points or Quizzes)
- Markdown enabled grader comment system.
- Automatic grading based on unit tests (modifiable via plug-ins)
- Automatic late grade calculation (modifiable via plug-ins)
- Built-in wiki support for problem descriptions and notes for graders. The wiki also supports various permissions levels.
- Support for command line control of the system.
- Support for listing currently active tutors and locations so students can get help on problem sets.

This document will attempt to guide you through the use of these features. Because not all features are useful to every class of user this document is divided into three chapters. The [Basic Use](#) chapter should be enough for instructors, tutors, and students. The [Administration](#) chapter will cover topics required to set-up and maintain the **CS>** system. Finally, the [Development](#) chapter will cover topics required to develop new plug-ins and modify the core functionality of the system.

## Chapter 2

# Basic Use

This chapter will attempt to cover all of the basic use cases for the various user situations. In addition it will describe the built in wiki system which ships with the **CS>** system.

### 2.1 General Use

This section covers general account management.

#### 2.1.1 Logging in/Password Recovery

Most of the features for the site are restricted to registered users. The login button is located in the top right corner of the page, see Figure 2.1.

If you forget your password (and the system has been configured to send emails, see Section 3.2.4) you may recover your password on the login page, shown in Figure 2.2.

#### 2.1.2 The Main Navigation Bar

Most navigation will be done through the use of the main navigation bar at the top of the screen, see Figure 2.3. The navigation bar's content will change based on the permissions of your account. Figure 2.3 shows an account that occupies all three roles, student, grader, and instructor.



Figure 2.1: The header of the main page before logging in.

## Sign In

**Username**

**Password**

☐ **Remember Me**

Sign In

Forgot your password?

Reset Password

Figure 2.2: The login page. Reset password is at the bottom.

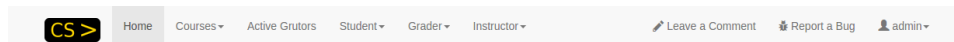


Figure 2.3: The main navigation bar

From right to left the links on the navigation bar are:

**Home** Takes you to the main page.

**Courses** Presents a drop-down menu which can take you to the homepage for active courses in the system.

**Active Grutors** Takes you to a page which shows where the current active grutors are for courses you are involved with (enrolled, grading, or teaching).

**Student** This only appears if you are a student in a course and will present a drop-down that can take you to the problem submission page for your course or a summary of your grades.

**Grader** This only appears for graders and instructors of a course. It presents a drop-down for choosing which course you want to grade assignments for.



**Instructor** This only appears for instructors. It presents a drop-down for which course you want to edit settings for.

**Leave a Comment** Takes you to a feedback page. Feedback provided through this page is sent to the admin.

**Report a Bug** Takes you to the github issues page for the **CS>** system.

**<username>** Presents a drop-down for account settings and logging out.

## 2.2 Student Use

The primary actions students will take in **CS>** is submitting homework assignments and checking grades.

### 2.2.1 Submitting Homework

To submit homework start by clicking the *Student* link on the navigation bar and selecting the course which you want to submit work to. This will bring you to a page like in figure 2.4.

Assignment Group	Problem	Submit	Status	Is Late	Due Date	Maximum Points
Week 1	Problem 0	<a href="#">Submit</a>	Unsubmitted		May 6, 2015 11:59 PM	20.00
Week 0	Problem 0	<a href="#">Submit</a>	Graded <a href="#">View</a>		May 6, 2015 11:59 PM	5.00
	Problem 1	<a href="#">Submit</a>	Unsubmitted		May 6, 2015 11:59 PM	10.00

Figure 2.4: A list of problems for a course. Problems are grouped by **Assignment Group**

Problems are grouped by **Assignment Group** which may refer to single weeks or any other logical grouping of multiple problems. Assignment groups are displayed in reverse order so that the most recently assigned group is at the top of the list and thus easier to find.

Once you have located the problem you wish to submit click the blue *Submit* button. This will direct you to a page like in figure 2.5.

On the submit page there is information about what files are expected, a place for you to choose the files you are submitting and a drop-down to select an optional partner.

Required Files For Submission: foo\_results.png

Required Files For Auto-Grading: foo.py

CS Test/Week 1/Problem 0

Files

Choose Files No file chosen

Partner

None

Submit

Figure 2.5: The page for submitting files for a problem

**Required Files For Submission** These files must be present in the set of files you select or else the system will reject the submission.

**Required Files For Auto-Grading** If any of these files are not present the submission will be accepted but the auto-grader will not run.

**Files** Will open a file selection window. To submit multiple files you may either select multiple files in the window, or submit a zip file. If you need to preserve directory structure submit a zip file.

**Partner** A drop-down of all the students in the course for selecting a partner. If you select a partner your partner does not have to submit the solution as well.

### 2.2.2 Viewing Grades

There are two ways to view grades. First on the problem list page, see figure 2.4, you can click the *view* button to see the grade and grader comments for an individual problem.

If you want an overview of all of your grades you can go to the *My Grades* page which is under the student drop-down menu. The My Grades page will display something like in figure 2.6.

### Test Semester/CS Test

	Week 0		Week 1	Total
	Problem 0	Problem 1	Problem 0	
Max Score	5.00	10.00	20.00	35.00
Your Scores	0.00	0.00	0.00	0.00

Figure 2.6: The My Grades page


## 2.3 Grader Use

Graders can assign grades in one of two ways, they can grade submitted assignments, or they can enter external grades into the grade-book. This section will cover both of these options.

### 2.3.1 Grading Submitted Homework

First to grade a student's submitted work click the *Grader* drop-down from the navigation bar (see Figure 2.3) select the course you wish to grade. This will bring up a list of all the problems currently assigned as seen in Figure 2.7.

### CS Test

 Grade Book

### Assignments

Assignment Group	Problem	Due Date	Submissions	Status	Grade
Week 1	Problem 0	May 6, 2015 11:59 PM	0	Done	<a href="#">Grade</a>
Week 0	Problem 0	May 6, 2015 11:59 PM	1	Done	<a href="#">Grade</a>
	Problem 1	May 6, 2015 11:59 PM	1	Unfinished (1-0)	<a href="#">Grade</a>

Figure 2.7: A list of all currently assigned problems in CS Test and the current grading status of the problems

On this page click the *Grade* button for the problem you wish to grade. This will bring up a list of all of the students in the class with the status of their submission, see Figure 2.8.

There are two ways to claim a submission to grade. First, you can click the *Random Grade* button. This button will atomically select a currently

## Submissions for: CS Test/Week 0/Problem 1

Not Graded		In Progress		Done Grading		
1		0		0		
<div>🔄Random Grade</div>						
Username	Submission Status	Is Late	Submission Time	Scored Points	Grade Submission	Graded By
admin	Submitted (Auto-graded, Waiting for Grader)		May 6, 2015 4:39 PM	0.00/10.00	<div>Grade</div>	None
jdoe	No Submission		N/A	0.00/10.00	<div>Grade</div>	N/A

Figure 2.8: The current status of grading for all submissions in this problem

ungraded assignment for you. Second, you can click the *grade* button for the student you wish to grade. Once you have selected a problem to grade you will see a page like in Figures 2.9 and 2.10.

Grading is fairly simple, using the list of student files (Bottom of Figure 2.10) you can view most textual, image, and PDF files. Files that cannot be directly viewed in browser may be downloaded.

To help during grading the professor can supply grading notes. These notes are in the *Submission Properties* table in the row labelled *Grade Notes* (See Figure 2.9). If notes are supplied a button will be presented which will open the specified link in a new tab.

Once you have assessed the students submission and left comments (Figure 2.10) you must save your changes and mark the submission as graded. All of this is handled by clicking the green *Finish Grading (save all)* button. Clicking this button will save all of the fields on the page and take you back to the list of submissions (Figure 2.8).

**Note:** If the submission you are working on has a partner that submission will receive the exact same grades and comments that you put on this submission.

Additional things you can do on the grade page are modifying the late status of a submission, sending the submission through the auto-grader again, and viewing old versions of the submission. These functions are all found in the *Submission Properties* table (Figure 2.9).

Submission for: CS Test/Week 0/Problem 1

User: admin

### Submission Properties

Submission Number	1 ▾
Submission Time	May 6, 2015 4:39 PM
Partner	None
Submission Status	Grading in progress
Toggle Late	<a href="#">Toggle</a>
Grade Notes	None
Regrade Submission (Removes comments and scores for this student)	<a href="#">Regrade</a>
<a href="#">Cancel Grade</a> <a href="#">Finish Grading (save all)</a>	

### Rubric

Section	Score	Max Points
Points	<input type="text" value="0.0"/>	10.00
<a href="#">Save Grades</a>		

Figure 2.9: The top of the grading page

Grader Comments

Save

No Comments

Grader Comments (Preview)

Auto-Grader Comments

Save

Auto-Grader Comments (Preview)

No tests provided. Testing complete.

Student Files


Filename	View	Download
 admin_header.png	<div>View</div>	<div>Download</div>

Figure 2.10: The comments and files section of the grading page

## Status of Grading

It is important to understand the different stages of grading for a submission. Each submission can be in one of six stages:

1. Unsubmitted
2. Submitted: No auto-grading or grading
3. Auto-grading in progress
4. Submitted: Auto-graded but not graded
5. Grading in progress
6. Graded

Along with the status a submission also has a grader associated with it. For the most part submissions will always be in state 4 when a grade gets to them. The largest area of confusion is how the *Cancel Grade* and *Finish Grading* buttons affect the status of the submission. There are simple rules for how this works:

- You are listed as the grader of the submission.

**Finish Grading** Sets the status to 6. You are still the grader.

**Cancel Grading** Sets the status to 4. The submission has no grader.

**Leaving the page (other links)** Submission status does not change.  
Grader does not change.

- Another grader is listed as the submission's grader

**Finish Grading** Sets the status to 6. You are now the grader.

**Cancel Grading** Submission status does not change. Grader does not change.

**Leaving the page (other links)** Submission status does not change.  
Grader does not change.

It is important to remember that if you leave the page via *Cancel Grade* or any other link none of your changes will be saved. If you want to make changes and leave with these methods you must use the explicit blue save buttons for each section you modify.

Gradebook: Test Semester/CS Test					
Username	Week 0		Week 1	Tests	Total
	Problem 0	Problem 1	Problem 0	Quiz 1	
Total Points	5.00	10.00	20.00	50.00	85.00
admin	0.00	3.00	0.00	0.00	3.00
jdoe	0.00	0.00	0.00	0.00	0.00

Figure 2.11: The grade-book for the course CS Test

### 2.3.2 Entering grades in the Grade-book

The other way graders can assign grades is through the grade-book. The grade-book is intended for assigning grades to assignments or other parts of the class that were not submitted through the system, (e.g. Participation, Quizzes, Tests...).

To get to the Grade-Book click the Grader drop-down menu and select the course you wish to grade. This brings up a page like in Figure 2.7. On this page click the *Gradebook* button. This will display a page like in Figure 2.11.

Non-submitted grade-book columns are always the right most columns in the grade-book. For example in Figure 2.11 The column **Quiz 1** in the category **Tests** is a non-submitted column.

To assign grades click the name of the column, in this case **Quiz 1**. This will take you to a page like seen in Figure 2.12.

From here you may edit each students score. When you are done you can press save and leave. **Important Note:** Two graders cannot be editing the same grade-book column at the same time once one grader saves their changes it will overwrite the other graders changes.



# Gradebook Column: Quiz 1

Maximum Points

50.00

Save

---

Username	Score
admin	<input type="text" value="0.00"/>
jdoe	<input type="text" value="0.00"/>

Figure 2.12: The grade column for Quiz 1 in CS Test

## 2.4 Instructor Use

Instructors are responsible for configuring problems and adding columns to the grade-book as well as grading. Since grading was covered in Section 2.3 this section will focus on configuring courses and problems.

### 2.4.1 General Course Settings

Once an administrator has created a course and assigned you as an instructor you will need to begin configuring your course. The course settings page is shown in Figure 2.13, this section will describe the functionality of this page.

**Note:** The figure does not show the entire page. At the bottom of the page is the list of students, grutors, and instructors. This document will explain later how to use these lists.

First let us examine the *Settings* panel near the bottom of the page. There are three major settings for a course.

**Course Homepage** This setting changes where the link in the *Courses* drop down from the nav-bar goes. By default it links to an empty built-in wiki page (use of the wiki is described in Section 2.5) but it can be changed to link to an external site.

**Use Anonymous Grading** This boolean setting changes whether or not usernames are made anonymous to graders. If it is active any page under the *Grader* drop-down for this course will use anonymous usernames. All pages in the *Instructor* drop-down for this course will display both usernames.

**Late Work Policy** The late work policy changes how the system handles scoring assignments submitted late. The default is *Highlighter* which simply highlights late assignments red in the grade book. This policy can be replaced via plug-in to modify scores based on how late or how many late assignments a student has turned in.

**Note (Possible feature request):** Currently the modified grade calculated by the late policy is only displayed in the grade-book or the students *My Grades* page. If a student views a problem individually it will show the original grade for that problem. This may be fixed in future versions of the CS> system.

## CS Test

[Grade Book](#) [Random Student](#)

### Assignments

eg. Week 0 [+ Add Assignment Group](#)

Assignment Group	Problem	Due Date	Total Points	Edit/Delete
Week 1	<a href="#">+</a>			<a href="#">Edit</a>
	Problem 0	May 6, 2015 11:59 PM	20.00	<a href="#">Edit</a> <a href="#">Delete</a>
Week 0	<a href="#">+</a>			<a href="#">Edit</a>
	Problem 0	May 6, 2015 11:59 PM	5.00	<a href="#">Edit</a> <a href="#">Delete</a>
	Problem 1	May 6, 2015 11:59 PM	10.00	<a href="#">Edit</a> <a href="#">Delete</a>

### Settings

Course Homepage (include [http://](#) for external sites)

☐ Use anonymous grading

Late Work Policy

[Save](#)

Figure 2.13: The main course settings page, minus the users lists

## 2.4.2 Adding Problems

Once you have configured the course to have the settings you want you will want to add problems. Within the **CS>** system problems are always contained within an **assignment group**. Assignment groups are ways to logically group related problems. For example assignment groups could be based on which week the problems material relates to, which chapter of the book they relate to, or what problem set they are from. Assignment groups are always displayed with the most recently assigned at the top of the page. This helps students quickly locate the most relevant assignments.

To add a problem you must first create an assignment group. To create an assignment group type in the text-box that displays the place-holder text *eg. Week 0* and press the *Add Assignment Group* button. In Figure 2.13 we can see that this course has assignment groups **Week 0** and **Week 1**.

Once you have created an assignment group you can add a problem by clicking the blue plus button in the problem column for the assignment group you want. This will take you to the **Problem Settings** page, explained in Section ??, you can also get to the problem settings page by clicking the grey cog button for the problem you wish to modify.

Problem Information

Problem Name

Problem 0

Due Date

05/06/2015

Due Time

11:59 PM

Do not accept without: (filenames separated by commas)

foo\_results.png

Do not autograde without: (separated by commas)

foo.py

Problem Description URL [+ Make Page](#)

eg. cs.hmc.edu/cs5/fhw3.html

Grading Notes URL [+ Make Page](#)

eg. cs.hmc.edu/cs5/fhw3grading.html

Problem Rubric

Name	Points	Add/Remove
Points	20.00	
Total	20.00	
eg. Auto-grader points	eg. 5.00	

Problem Settings

☒ Allow Partners ☒ Is the problem open for submissions

☒ Release autograder comments ☐ Finish grading after autograder (Releases scores and comments)

Saved

Figure 2.14: The problem settings page

### 2.4.3 Problem Settings

The problem settings page has four major sections: Problem Information, Problem Rubric, Problem Settings (all seen in Figure 2.14), and Tests. The first three sections are inter-related and are all saved at once using the Save button at the bottom of the Problem Settings section. The tests are handled separately and will be described later in the chapter.

#### Problem Information

The Problem Information section of the page allows you to enter basic information about the assignment. Most of the sections are fairly self explanatory, but there are subtle distinctions about some:

**Do not accept without** A comma separated list of files which the system should reject a submission if any of them are missing. This is useful to prevent students from turning in partial assignments.

**Do not auto-grade without** A comma separated list of files which the system should look for to be allowed to run the auto-grader. This is good if you want to accept partial submissions but some files may be required to not crash the auto-grading system.

**Problem Description URL** A link to the specification of the problem. This can be an external link (preceded with `http://`) or a wiki page (See section 2.5). The *Make Page* button will generate a wiki page where you can write the specification for the problem.

**Grading Notes URL** A link to the specification for graders to use to grade problems. This link is only made available to graders. Similarly to the Problem Description URL it can be a wiki page generated with the *Make Page* button.

### Problem Rubric

The problem rubric is where you can specify how many points a problem is worth. Rubrics may contain multiple sections to help both graders and students better understand what deductions are for.

Sections can be added to the rubric by typing a name and a number of points into the fields at the bottom of the table then clicking the *plus* button. Similarly you can remove a section by clicking the *minus* button. Modifications are color coded and only applied when the save button is pressed

**Note:** To change the number of points assigned by a section simply re-add that section.

**Note:** Removing a section of the rubric does not remove any points assigned to a student in that particular field of the rubric. If you need to remove a section of the rubric that has already been graded you will need to use the command line to remove or rename the assigned points. Example in section 4.2.2.

### Problem Settings

The settings section contains four toggles to change the way the problem handles submissions.

**Allow Partners** If this box is checked the student submission page allows students to select one partner to submit with. Otherwise that box is disabled.

**Is this problem open for submissions** When this box is checked the students can click the submit button. when it is unchecked that button is removed.

## Tests

Test Filename	Test Language	Edit	Remove
<div>Choose File No file chosen</div>	Jflap (Turing Machine) ▼	<div>+</div>	

Figure 2.15: The problem tests lists

**Release Autograder Comments** When this box is checked any comments that the auto-grader makes are shown to the students before their grade is marked as done. Otherwise a student looking at their grade will see a place-holder comment telling them to check back when the grading is finished.

**Finish grading after autograder** When this is checked the auto-grader will mark a submission as done grading once it has finished. If the auto-grader fails to grade an assignment it will be left in the *Waiting for Autograder* state.

### 2.4.4 Tests

One of the major features of the system is its ability to auto-grade student submissions. To allow this each problem has a section in its settings page to add multiple test files (See figure 2.15).

#### Adding Tests

Adding a test is fairly simple. First select the unit-test file you want to upload. Next select the type of unit test that the file is for. The currently implemented test types are:

**Python (pyunit)** This type runs a pyunit test file.

**Java (junit4)** This type runs a Junit4 style test file.

**Prolog (plunit)** This type runs a Prolog file that implements plunit tests.

**Racket (rackunit)** This type runs a rackunit test file. **Note:** There are several subtleties to running the rackunit tests which are described in Section 2.4.6.

**Picobot** This type runs a picobot program in various mazes from various starting positions. An online picobot simulator can be found at the HMC Picobot site <http://www.cs.hmc.edu/picobot/>.

**JFLAP** This type runs various inputs through DFAs and NFAs defined using the JFLAP program. **Note:** The JFLAP simulator that this type uses is based off of a python script that was developed over many years for use internal to HMC. It may contain some bugs in edge cases that don't occur in standard HMC assignments.

**JFLAP (Turing Machine)** This type runs various inputs through Turing machines defined using the JFLAP program. **Note:** The JFLAP simulator that this type uses is based off of a python script that was developed over many years for use internal to HMC. It may contain some bugs in edge cases that don't occur in standard HMC assignments.

**Supplemental File** Supplemental files are never executed. Instead these files are meant to allow the test files to get external data or link in extra code pieces.

### 2.4.5 Test Settings

By default when you add a test file to a problem it will parse the file to look for tests. Each test it finds it will assign one point to an unspecified rubric section. This functionality provides a useful default but it is often useful to have more power when specifying which tests go to what rubric sections and how many points they are worth.

To allow for this functionality the site provides a test settings page, seen in Figure 2.16.

To allocate points you must first create a *Test Section*. Once you have a test section you can define how many points it is worth, which rubric section it assigns points to, and which tests are part of that section.

**Note:** Any test that is not in a section will not be displayed to the student. If you want a test to be run but not have any points assigned to it you should put it in its own 0 point section.

### 2.4.6 Writing Tests

There are some idiosyncrasies with some of the auto-grader systems (mostly stemming from the fact that some languages do not easily support external

Rubric Section	Allocated Points	Total Points
Points	0	20.00

Timeout:
✔ Saved

Reupload File

File

No file chosen

Sections

Section: Basic Tests ✖

Points:

Rubric Section:

Test Name	Remove
testA	<span>✖</span>
<input type="text"/>	<input type="button" value="✚"/>

Figure 2.16: The test point allocation page



unit-tests). This section covers some of the test writing issues that attempt to combat those idiosyncrasies.

### **Generic Test Features**

One of the biggest features of the testing framework is being able to provide external output. To help the system differentiate between student code output, test framework output, and output intended to be presented to the student, the system uses a simple annotation system.

When a test wants to produce output that is presented to the student in the web interface the print statement should begin with the name of the test and a colon. This is probably best shown in an example:

```
def testThatPrints(self):  
    x = studentCode()  
    print "testThatPrints: Your code produced %d" % (x)
```

This will print out `Your code produced #` to the student in the grading comments (where `#` is whatever number their code produced).

## Python

Python has relatively few issues but there are some things to point out. First it is recommended to use `import foo as bar`. This helps prevent name conflicts with the unit-testing framework or your test classes. Below is a simple example file:

```
import hw3pr2 as hw
import unittest

class BasicTests(unittest.TestCase):
    def testA(self):
        self.assertEqual(hw.foo(3), 2)

if __name__ == '__main__':
    unittest.main()
```

Another important point is that a python test file must be executable on its own. This means that it must contain a "main".

## Java

Java, like python, has relatively few issues with the auto-grader. Currently the only tested style of test writing is using the `@Test` decorator. Below is a simple test file:

```
import static org.junit.Assert.*;
import org.junit.Test;

public class BasicTests {

    /* This tests whether the foo function multiplies by 3 */
    @Test
    public void testA() {
        assertTrue( foo(2) == 6 );
    }
}
```

The biggest issue with the Java auto-grader is the way its test detection regex works. Even though Java will let you put a comment between the decorator and the function the regex doesn't do a very good job detecting the test name without the decorator being directly adjacent to the function definition.

## Racket

Racket is one of the more complicated testing systems. Because students submit self contained racket files racket doesn't want to import the student's code into the test file. To combat this the auto-grader performs temporary modifications to the students code, such as removing the `#lang racket` declaration.

Additionally if a student has failed to implement a function to in their submitted code the grading will fail. To combat this it is recommended to create a different test file for each function you wish to test. Because each test file is executed independently a failure in one file will not affect the others.

The Racket auto-grader uses the *rackunit* framework. This framework allows us to add comments to the tests which we use as test names. Because of limitations of the test name extraction regular expression tests must be at most split across 2 lines. This makes for terribly unattractive test files and we are working on improving its capabilities. An example test file is below:

```
;;;Racket - Unit Tests Example

#lang racket
(include "change.rkt")
(require rackunit)

(check-equal? (min-change 1 '(1 5 10 25 50)) '(1)
"one-cent")
(check-equal? (min-change 42 '(1 5 10 25 50)) '(1 1 5 10 25)
"42-cents us coins: your test didn't make 42 cents change with us coins")
(check-equal? (min-change 42 '(1 5 21 35)) '(21 21)
"42-cents weird coins")
(check-equal? (min-change 0 '(1 5 21 35)) '()
"no change")
```

In this case the names of the tests are anything in in the comment strings before a colon. If there is no colon the name is the whole string.

## Prolog

Prolog has relatively few issues with the auto-grader. The test system utilizes the basic Prolog unit-testing framework. When importing the student's submitted code it is important to turn off loading unit-tests. An example file is below:

```
:- set_test_options([load(never)]).
:- include('sort.pl').
:- set_test_options([load(always)]).

:- begin_tests(selectionSort).
test(selectionSortT1) :- selectionSort([1, 2, 3], [1, 2, 3]), !.
:- end_tests(selectionSort).
```

## JFLAP and JFLAP (Turing Machine)

The JFLAP and JFLAP (Turing Machine) use a simple text file format to specify tests. The format consists simply of an input string to be fed to the automaton and then if that input should reject. The biggest issue is that the auto-grader only knows what file to run the tests on based on the name of the solution file. For example the file `part1.sols` will grade the JFLAP file `part1.jff`. An example file follows

```
reject
100101
100100 reject
1
00 reject
```

Inputs that reject are followed by a space and the word reject. All others accept. To indicate the empty string as an input simply leave a blank line or a just the space and the word reject.

## Picobot

Similar to JFLAP, Picobot uses a text input file. The format consists of which map to test the program on, what file the program is contained in and then a list of starting positions. An example file is below

```
emptyMap
hw0pr3.txt
```

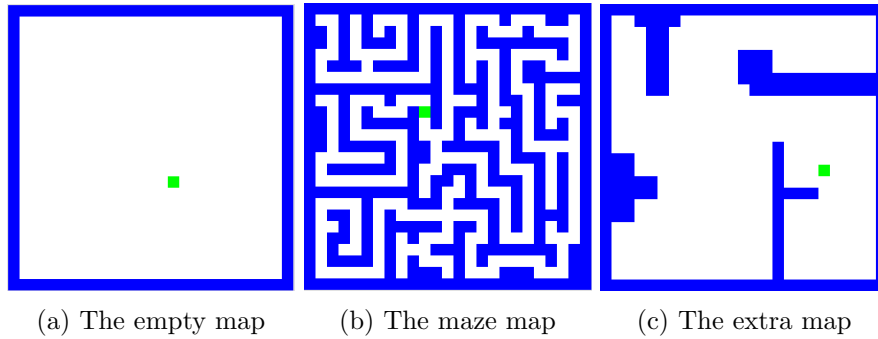


Figure 2.17: The various maps available to the Picobot auto-grader

```
###
topleftcorner: 1,1
toprightcorner: 23,1
bottomrightcorner: 23,23
middle: 10, 10
othermiddle: 9,9
```

There are three available maps:

1. `emptyMap` Figure [2.17a](#)
2. `mazeMap` Figure [2.17b](#)
3. `extraMap` Figure [2.17c](#)

Permissions

Mouse over headers for tool tips

Type	Anyone	Any User	Course Users	Any Grutor	Course Grutors
View	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Edit	N/A	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>

Figure 2.18: The wiki page permissions table

## 2.5 Wiki Use

The **CS>** system provides a wiki like system for providing problem descriptions, grading notes, and even full course websites.

**Note:** The wiki system saw relatively little use during our first semester of testing. Because of this the wiki system may contain more issues than other parts of the **CS>** system.

### 2.5.1 Wiki page permissions

One of the most critical points of the wiki system is the ability to set the view and edit permissions. The permissions table is seen in Figure 2.18.

For each category of users you can assign either *view* or *edit* permissions. The only restrictions are that if a group does not have view permission they cannot have edit permissions and you cannot give edit permissions to users who are not registered for the site. The vertical columns have the following meanings:

**Anyone** Any person on the internet. The page does not check for credentials.

**Any User** Any person who has a login with this instance of the **CS>** system.

**Course Users** Any person who is registered as a student in the course that owns this page.

**Any Grutor** Any person who is registered as a user for any course.

**Course Grutors** Any person who is registered as a grutor for the course that owns this page.

Course instructors always have access and the ability to edit a page.

### 2.5.2 Wiki syntax

The basic wiki syntax is based on Markdown (<http://daringfireball.net/projects/markdown/syntax>) but there are several extensions to allow for designing nicer looking pages.

The first major extension enabled is the Attribute list extension ([http://pythonhosted.org/Markdown/extensions/attr\\_list.html](http://pythonhosted.org/Markdown/extensions/attr_list.html)). This extension allows you to specify attributes such as CSS classes and styles. Additionally you can specify IDs to use as anchors for links.

The second major extension is a custom wiki link syntax extension. Default links in markdown allow you to link to external sites. Because links are long strings of UUID's for the wiki it is impractical to use them to create links (**Note:** It is planned to add easier links in the future). To solve this the markdown syntax has been expanded to include a wiki link as follows:

```
[Text]{Semester Name, Course Name, Page Title}
[Text]{Course Name, Page Title}
[Text]{Page Title}
```

The difference in the three lines is that the less information you provide the more it will grab from the course that owns the current page.

Like in most wikis if a page you reference doesn't exist that page will be created. That page will inherit the permissions of the page that created it.

Along with links and text you can upload an image and load that into a page. To access an image you use something like the link syntax but starting with a "!".

```
![Alt Text]{Semester Name, Course Name, Page Title, Image Name}
![Alt Text]{Course Name, Page Title, Image Name}
![Alt Text]{Page Title, Image Name}
![Alt Text]{Image Name}
```

Like links it will load information not provided from the page that the link exists in. You can also use the attribute list to specify a width and height of the image.



## Home

Welcome to the homepage of the CS Test course for Test Semester! Test

HI

This is stuff that is blue

Test code blocks

Text

Other [Course Not Found]



Figure 2.19: An example wiki page

Below is a small example page:

Welcome to the homepage of the CS Test course for Test Semester!

# HI IM A RED HEADER # {style="color:red;"}

This is stuff that is blue

{style="color:blue;"}

<pre>

Test code blocks

</pre>

[Text]{Test Semester, CS Test, Test Page}

[Other]{Fake Semester, CS Test, Test Page}

[Not created yet]{Test Semester, CS Test, Not Created}

![Alt Test]{Alien.PNG>

The rendered page is shown in Figure 2.19.

# Chapter 3

## Administration

### 3.1 Server Structure

The **CS>** system is designed to be modular and expandable as the number of users of the system grows. To accommodate this each of the five major components are designed to be operated as a separate server. The five components are:

1. Python-Flask Front-end Server
2. Python-Celery Grading Worker
3. File-system for submission storage (pick your favorite one)
4. MongoDB for storing grades
5. RabbitMQ for passing grading requests to workers

In the most basic configuration, which is most useful for testing, all of the required components will be on one machine. As the requirement for redundancy and scalability increases components may be moved to separate servers and replicated. Figure 3.1 shows one possible configuration of the servers which provides redundancy on both the front and back end. It is important to note that a load-balancer is required to support multiple front-ends.

**Note:** Currently even though MongoDB and rabbitMQ support distributed options the system has never been tested using that configuration. This is a feature which may be researched in the future if it becomes needed.

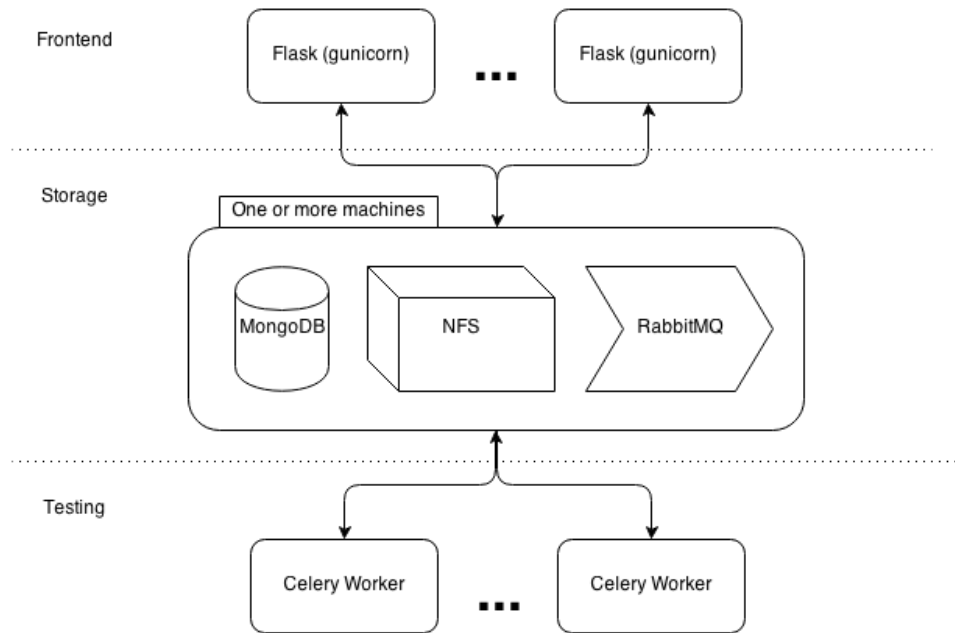


Figure 3.1: Possible server configuration for **CS>**

## 3.2 Set-up

Once you have decided on the basic configuration for your servers setting up the requisite systems requires a little work. For each server type I will provide basic set-up instructions. These instructions are listed in order to prevent dependency issues.

### 3.2.1 MongoDB Set-up

To set-up a brand new MongoDB instance you can use the instructions below. If you already have a running version of MongoDB you can adapt the instructions or request help from your database administrator.

1. Get the latest version of MongoDB from [here](#).
2. Follow [these](#) instructions to create an administrator account.
3. Edit `/etc/mongodb.conf` so that `bind_ip` is commented and uncomment `auth = true`. Additionally it is recommended to change the port that the server listens on to prevent attacks.

4. Reload *mongod* and log into the *mongo* shell with  
\$ `mongo -u {username} -p --authenticationDatabase admin`
5. Type `> use submissionsite` to create the *submissionsite* database
6. Add a user for this database by typing

```
> db.createUser(  
  { user: "grader",  
    pwd: "{your password}",  
    roles: [{role: "dbOwner", db: "submissionsite"}]  
  }  
)
```

7. Finally `> quit()` the mongo shell.

### 3.2.2 File-system Set-up

Setting up the file-system is pretty easy. There are two major requirements for the file-system, one, it must be accessible from all machines in the cluster, and two it must have a minimal required directory structure.

Because exporting a directory is complicated and other better resources exist on the internet I leave figuring out how to export your directory as an exercise to the reader.

Once you decide how to export your chosen storage directory you must give it some basic structure. This structure is as follows:

- submissions
- photos
- plugins
  - autograder
  - latework

Once these paths exist in your storage directory you are good to go.

### 3.2.3 RabbitMQ Set-up

RabbitMQ is a distributed queue which handles sending auto-grading requests to the various celery workers. Setting up this part of the system can be very easy but depending on your security requirements you may want to enforce more stringent permissions. To set-up RabbitMQ do the following:

1. Install *rabbitmq-server*.
2. Run  

```
$ sudo rabbitmqctl add_user {username} {password}
```
3. Run  

```
$ sudo rabbitmqctl set_permissions -p / {username} ".*" ".*" ".*"
```

RabbitMQ is now in a very basic workable configuration.

### 3.2.4 Create config.py

The `config.py` file handles all of the connection configuration for the system. Inside `config.py` there are several sections that need to be modified using configuration information from the steps above.

1. Give your own `SECRET_KEY`
2. Configure MongoDB connection properties

```
MONGODB_SETTINGS = {  
    'DB': 'submissionsite',  
    'username': '{dbuser}',  
    'password': '{dbpassword}',  
    'host': '{db_ip}',  
    'port': '{db_port}'  
}
```

3. Configure the Celery broker URL  

```
CELERY_BROKER_URL="amqp://{rabbit_user}:{rabbit_pwd}@{rabbit_ip}"
```

4. Set the where the file-system will be mounted for the front/back-end servers

```
STORAGE_HOME="{path_to_mount_point}"
```

**Note:** If you are storing the data locally you still have to set this flag but instead want to set `STORAGE_MOUNTED=False`. This disables

a check to make sure that the directory is mounted before performing writes.

5. Set up the system email (This is used for sending password reset requests)

```
SYSTEM_EMAIL_ADDRESS = "{server_email}"
SMTP_SERVER = "{smtp_server}"
```

This configuration file should be passed to all front/back-end servers.

**Note:** There is an additional setting `GRADER_USER` which is currently set to `None`. This setting is supposed to set which user to switch to as the auto-grader but it is currently non-functional.

### 3.2.5 Flask Front-end Set-up

The Flask based Front-end of the system serves dynamically generated web-pages to the users. To set-up the flask front-end do the following:

1. Ensure that `python-dev` (or its equivalent package) is installed on your platform.
2. Check out the repository
3. `$ cd <your_repository>`
4. Copy `config.py` to the repository
5. Create a virtual environment using `virtualenv <venv_name>`
6. **Temporary Fix** The current version of mongoengine does not function with the latest version of pymongo. To solve this install version 2.8 `$ <venv_name>/bin/pip install pymongo==2.8`
7. Install the following packages with pip  
`$ <venv_name>/bin/pip install flask flask-script flask-bootstrap flask-login flask-markdown mongoengine flask-mongoengine WTForms python-dateutil celery psutil python-magic bleach gunicorn gevent flower`
8. Launch the front-end processes  
`$ <venv_name>/bin/celery flower -A app:celery`  
`$ <venv_name>/bin/gunicorn -w 4 -k gevent -b 0.0.0.0:80 app:app`

**Note:** You may have to modify some permissions to allow *gunicorn* to bind to port 80. Additionally if you are using a load balancer you may want to bind it to some other higher port.

### 3.2.6 Celery Worker Back-end Set-up

Setup for the Celery Worker is almost identical to the Flask front-end but requires fewer packages. The instructions are displayed below:

1. Ensure that `python-dev` (or its equivalent package) is installed on your platform.
2. Check out the repository
3. `$ cd <your_repository>`
4. Copy `config.py` to the repository
5. Create a virtual environment using `virtualenv <venv_name>`
6. **Temporary Fix** The current version of mongoengine does not function with the latest version of pymongo. To solve this install version 2.8 `$ <venv_name>/bin/pip install pymongo==2.8`
7. Install the following packages with pip  
`$ <venv_name>/bin/pip install flask flask-script flask-bootstrap flask-login flask-markdown mongoengine flask-mongoengine WTForms python-dateutil celery psutil python-magic bleach`
8. Launch the worker processes  
`$ <venv_name>/bin/celery worker -A app:celery`

### 3.2.7 Bootstrap the Administrator Account

All of the previous sections have gotten a working version of the system but currently there are no accounts in the database. To begin creating accounts there must be an administrator account.

The easiest way to create the `admin` account is to run

```
$ <venv_name>/bin/python run.py
```

then kill this process. `run.py` launches the debug server which also checks for an administrator account if no account is found it creates it. You can find the password for the default `admin` account in `run.py`.

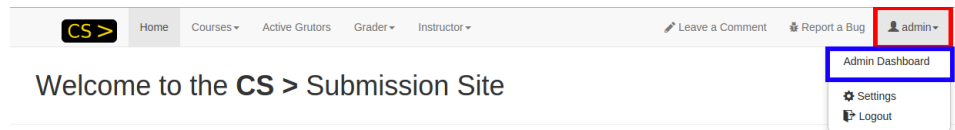


Figure 3.2: Red: The user Dropdown, Blue: The admin dashboard link

## 3.3 Managing Courses

### 3.3.1 Creating Courses

The administrator is required to create all of the courses for the system. When a course is created all administrators are invisibly added as instructors. Any other instructors must be explicitly added by an administrator.

To add a new course to the system first navigate to the *Admin Dashboard* (See Figure 3.2). Once in the Admin Dashboard click the link for the *Courses* panel. Fill in the required information and click the *Create Course* button.

### 3.3.2 Adding Instructors

Once a course has been created instructors (other than the current administrators) must be added to the course. This requires the following actions:

1. Navigate to the course settings page. If you just created the course find it in the *Active Courses* list and click *Administer*. If you are on the main page click the *Instructor* drop-down and click on the course link.
2. Scroll to the bottom of the course settings page.
3. Under the instructors heading enter the user-name of the desired instructor. (If the instructor does not have an account yet see Section 3.4). Click the plus button.
4. That user is now an instructor.

### 3.3.3 Deactivating Courses

When a course is finished it is desired to prevent that course from being modified. Additionally we want currently activated courses to be prioritized when displaying lists of courses to users. To accomplish this **CS>** allows the administrator to **Deactivate** a course. When a course is deactivated



it no longer accepts submissions and is moved from the main drop down menus into a separate page of old courses.

To deactivate a course first navigate to the Admin Dashboard and then to the Courses panel. Once on the courses panel find the desired course and click the *Deactivate* button.

**Note:** Some of the archival pages are still works in progress, but the framework is there to accomplish these features. If you run into problems with deactivated courses please submit a bug report.

## 3.4 Creating User Accounts

One of the primary roles of the `admin` account is to create user accounts. There are two methods for creating accounts for users.

### 3.4.1 Creating via web

Web creation is the easiest way to create a small number of accounts. The steps are as follows:

1. Navigate to the *Admin Dashboard* by clicking the user drop-down menu and then *Admin Dashboard*. Figure 3.2 shows the locations of the links.
2. Once in the *Admin Dashboard* navigate to the user panel using the top bar. (See Figure 3.3)
3. Fill in the information fields and click **Create User**. (See Figure 3.3)

### 3.4.2 Creating via command line

Creating new users via the command line is generally one of the easiest ways to add large batches of users. There are already several scripts (Located in `app/scripts` in the repository) which demonstrate how to add users from various files (CSV files with different column layouts). Below are general instructions for adding a user via command line:

**Note:** To correctly use the mongo interface for the system in python you must export `PYTHONPATH` as the root directory of the repository.

```
$ export PYTHONPATH=<path_to_repository>
$ <venv_name>/bin/python
```

Grader Admin Home Statistics Courses Users

### Add a User

**Username**  
eg. jdoe

**First Name**  
eg. Jamie

**Last Name**  
eg. Doe

**Email**  
eg. jdoe@example.com

**Password**  
eg. asdf (defaults to asdf)

Create User

Figure 3.3: The admin users panel

```
...python start message...
>>> from app.structures.models.user import *
>>> user = User()
>>> user.username = "<username>"
>>> user.firstName = "<firstName>"
>>> user.lastName = "<lastName>"
>>> user.email = "<email>"
>>> user.setPassword("<password>")
>>> user.save()
```

If we then want to add the user to a course we want to continue by doing

```
>>> from app.structures.models.course import *
>>> c = Course.objects.get(semester="<semester>", name="<name>")
>>> #One or more of these
>>> user.courseStudent.append(c)
>>> user.courseGrutor.append(c)
>>> user.courseInstructor.append(c)
>>> user.save()
```

## Chapter 4

# Command Line Reference

Some functionality is too infrequently used or too cumbersome to provide through the web interface. Including:

- Assigning all students a comment and a score for a given problem.
- Changing the password of a student.
- Adding a user to a large number of courses at once.
- Making a user an administrator.
- Creating a large number of user accounts from a file.

Other functionality simply hasn't been made available yet in the web interface and will be added later. Including:

- Marking all submission grading status as "Done".
- Sending all submissions through the auto-grader again.

This section will demonstrate how to use the scripts provided to perform these functions as well as other useful tips for the command line.

### 4.1 Setting up the Environment

Because of the way the python import system works, before you can run any scripts or do work on the command line you must first export the environment variable `PYTHONPATH` to the location of the root of your repository. One easy way to do this is:

```
$ cd <your_repository>
$ export PYTHONPATH='pwd'
```

### 4.1.1 The scripts directory

The repository contains several useful scripts in the directory `app/scripts`.

## 4.2 Assigning Points

### 4.2.1 The giveFullPoints Script

The give full points script allows you to assign a specific number of points to a section of the rubric for all submissions, as well as leave a comment in the grader comment section.

To configure the script for your specific course there are a few variables to set.

1. `courseName` The name of the course you are working on.
2. `semester` The semester the course is in.
3. `comments` The comments string to leave.
4. `section` What rubric section to edit.
5. `score` How many points to give.

The `courseName` and `semester` variable can also be grabbed using `raw_input`.

```
courseName = raw_input("Course Name: ")
semester = raw_input("Course Semester: ")
```

The script will automatically enumerate all available assignments and problems and allow you to pick dynamically.

### 4.2.2 Removing points from removed rubric sections

When a section is removed from the rubric the system doesn't remove any assigned points from submissions. To handle this situation you must delete or move the assigned points in the submission's grade object.

```
>>> s = #A submission object obtained by magic (Python)
>>> #Move points from a badly named section
>>> s.grade.scores['Points'] = s.grade.scores['Pionts']
>>> #Remove an unneeded section
>>> del s.grade.scores['useless']
>>> s.save() #Don't forget to save
```

### 4.3 Changing a User Password

For security reasons there is no web-facing way to change an arbitrary users password. Instead this is accomplished through the command line as follows:

```
>>> from app.models.structures.user import *
>>> user = User.objects.get(username="someUsername")
>>> user.setPassword("new password")
>>> user.save()
```

### 4.4 Adding a user to courses

It is easy to add a user to a course through the web. But if you want to add a user to more than one course at a time it can be difficult and involves clicking a lot of links. It is sometimes easier to add a user to courses through the command line as follows:

```
>>> from app.models.structures.user import *
>>> from app.models.structures.course import *
>>> # Get the user
>>> user = User.objects.get(username="someUsername")
>>> # Get some courses
>>> course0 = Course.objects.get(semester="semester", name="name")
>>> course1 = Course.objects.get(semester="semester", name="othername")
>>> #Add the user
>>> user.courseStudent.append(course0)
>>> user.courseGrutor.append(course1)
>>> #Save the changes
>>> user.save()
```

### 4.5 Making an administrator user

Administrator users have the ability to create courses and add users. An administrator is automatically an instructor in all courses. To create a new administrator do the following:

```
>>> from app.models.structures.user import *
>>> from app.models.structures.course import *
>>> user = User.objects.get(username="someUsername")
>>> user.isAdmin = True
```

```
>>> #We need to retroactively add them to all courses
>>> user.courseInstructor = Course.objects
>>> user.save()
```

## 4.6 Adding batches of users

The three scripts `addUsersFromList`, `addUsersFromSakai`, `addGrutorsFromList` all provide examples for how to read CSV style files and creating user accounts from them.

These scripts were designed for the data format used by the Harvey Mudd College Registrar and so may not be immediately useful at other schools, but they will provide good reference.

For more info on how to add users see [Section 3.4](#).

## 4.7 Marking Submissions as Done

Currently the easiest way to mark a batch of submissions as done is through the command line. This functionality will eventually be added the web interface. This works as follows:

```
>>> from app.models.structures.user import *
>>> from app.models.structures.course import *
>>> course = Course.objects.get(semester="semester", name="name")
>>> #Find the assignment we want
>>> course.assignments[#].name
>>> #repeat above until we find the one we want
>>> #Get the assignment we want based on its index
>>> assignment = course.assignments[#]
>>> #Find the problem we want
>>> assignment.problems[#].name
>>> problem = assignment.problems[#]
>>> #loop through all submissions
>>> for submissionList in problem.studentSubmisssions.itervalues():
>>>     #Modify the latest submission
>>>     submissionList[-1].status = SUBMISSION_GRADED
>>>     submissionList[-1].save()
```

## 4.8 Regrade all submissions

The easiest way to do this is through the `regradeSubmissions` script. Simply run the script:

```
$ <venv>/bin/python ./app/scripts/regradeSubmissions.py
```

Then follow the instructions, it will guide you through the whole process.

# Chapter 5

## Development

One of the major benefits of the **CS>** system is that it is easily extensible. It is written in python (which is increasingly becoming a first language taught in college courses) so that it may be easier to pick up for many students.

### 5.1 Auto-grader Plug-in Development

The greatest feature of the **CS>** system is the auto-grader plug-in system. Auto-grader plug-ins consist of three major parts.

1. `PLUGIN_NAME` This variable sets the name of the plug-in that will be displayed on the web interface
2. `testFileParser(filename)` This function parses the given test file and extracts the test names.
3. `runTests(cmdPrefix, testFile, timeLimit)` This function is designed to run the test file given in `testFile`. It uses `cmdPrefix` when running any sub-processes. If the computation takes longer than `timeLimit` it will report a time-out.

A full example of an auto-grader plug-in for python is located in Appendix [A](#).

#### 5.1.1 testFileParser

The test file parser is a relatively simple function for most languages. This function is given a filename as input and as output it should produce a list of strings that contain the names of the tests contained in that file.



**Note:** This can be done with either simple regular expressions or a more complicated parser.

### 5.1.2 runTests

The `runTests` function can be more complicated than the `testFileParser`. It takes in three arguments

**cmdPrefix** A list of strings. This list is either empty or contains a list of commands to switch to a less privileged user help prevent some potential security issues with student code.

**testFile** The name of the file that contains the tests to be run.

**timeLimit** The maximum number of seconds that the test may take before reporting a time-out. This prevents non-halting code from consuming the system.

With these inputs the `runTests` function should produce two dictionaries as output, the `summary` dictionary and the `failedTests` dictionary. The `summary` dictionary provides information about the success or failure of the tests. This dictionary must contain the following values.

**died** This is a boolean value indicating if there was a crash or some other type of failure where some subset of tests failed to run. This is often used when there was a compiler error.

**timeout** This is a boolean value indicating if the tests ran longer than the time specified by `timeLimit`.

**totalTests** This is an integer representing the total number of tests that were run. This value is only used to provide some feedback to the user and so if it cannot be determined from the test output 0 may be passed.

**failedTests** This is an integer representing the number of failed tests. This is used to provide a summary to the user.

**rawOut** This variable should contain the raw test output gathered from the standard output of the test script.

**rawErr** This variable should contain the raw test output gathered from the standard error of the test script. If the system died due to a parse error or some other error in the `runTests` function this variable can be replaced with a different message for diagnostics.

Assuming that the tests did not die or incur a time-out the function must return a dictionary of failed tests. When the tests do die or time-out this dictionary can be empty. Otherwise it should only be empty if no tests have failed.

This dictionary is a mapping: Test Name  $\rightarrow$  Info Dictionary. Currently the *Info Dictionary* only contains one value `hint`. The reason it is a dictionary is for future extensibility.

**hint** This should be an error message explaining why the test failed. This can be extracted from the test output or can be generated by the `runTests` function.

## 5.2 Late calculator Development

The late calculator is another useful plug-in. This plug-in changes the way the system calculates the student's final score based on how many and which assignments were turned in late. This plug-in contains two major components:

1. `PLUGIN_NAME` This is a string which gives the name of the plug-in to the web-interface.
2. `calculateGrades(gradeList)` This gets given a grade list (which is a data structure explained below) and returns a modified grade list with the adjusted scores.

### 5.2.1 The Grade List Structure

The grade list structure is a list of lists. Each nested list represents one Assignment group and each element of that list is one problem in that Assignment Group. If a student didn't submit anything for one problem the element at that index will be `None`, otherwise it is a dictionary with the following values:

**rawTotalScore** This is the total score assigned by the combination of the auto-grader and the graders.

**maxScore** This is the maximum possible score for this problem.

**timeDelta** This is the time difference between when the problem is submitted and when the problem was due.

**isLate** This is a boolean representing if a problem should be considered late. Even if the **timeDelta** says an assignment is late if this is **False** that means an instructor manually unmarked the assignment as late.

**finalTotalScore** This is the score after penalties or extra credit are assigned for turning the problem in early or late. The initial dictionary doesn't have this value and must be added by the function.

**highlight** This is initially not present in the dictionary. It is a string representing what color the cell should be highlighted ("red", "blue", "green", "yellow"). If this is unset it won't be highlighted.

Consider the following set of problems and submissions:

Assignment Group	Week 0	Week 0	Week 1	Week 1
Problem Name	Problem 0	Problem 1	Problem 0	Problem 1
Points	10pts (late)	12pts	No Submission	11pts

This gets translated to the following grade list:

```
[
  [
    {'rawTotalScore':10, 'isLate': True, 'timeDelta':+2Days,
     'maxScore': 15},
    #This has been marked not late by an instructor
    {'rawTotalScore':12, 'isLate': False, 'timeDelta':+2Days,
     'maxScore': 15}
  ],
  [
    None,
    {'rawTotalScore':11, 'isLate': False, 'timeDelta':-1Day,
     'maxScore': 15}
  ]
]
```

An example function is shown in Appendix B and below:

```
def calculateGrades(gradeList):
    '''Allows for 3 late submissions and then gives 0'''
    lateCount = 0
    for assignment in gradeList:
        for problem in assignment:
            if problem['isLate']:
                lateCount += 1
                problem['highlight'] = "red" #Highlight all late hws
                red
            if lateCount > 3:
                problem['finalTotalScore'] = 0
    return gradeList
```

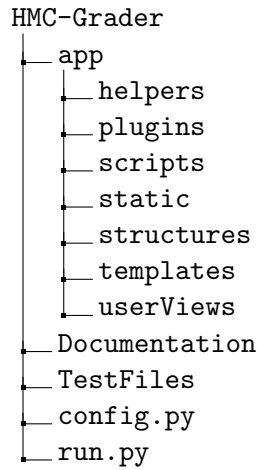


Figure 5.1: HMC-Grader repository structure

## 5.3 Core Development Practices

Beyond plug-ins the **CS>** system is designed to be relatively easy to maintain and extend. This section attempts to cover the major design decisions and coding practices for developing the **CS>** system further.

**Note:** Much of what follows in this section falls under the "Do what I say not what I do" practice of coding standards. This system was developed initially as I was learning to write python based websites. Some of that early code was rough but still exists in the system. I have tried to do several refactors to get rid of the old code. Unfortunately some still exists, sorry.

### 5.3.1 Repository Layout

The repository is structured as shown in Figure 5.1.

Within the **userViews** directory there are several sub-directories for storing code relevant to specific pages. Each directory has a category of pages it services as follows:

**admin** This folder contains code for pages that are only accessible to an administrator account.

**common** This folder is for pages that are accessible to many different classes of user.

**grutor** This folder is for pages that are accessible to the grader users.

**instructor** This folder is for pages that are accessible to the instructor only.

**student** This folder is for pages that are used by the student users.

The **templates** folder contains a mirrored set of directories to the **userViews** directory. If there is a view function in a folder in **userViews** the template file that it uses will be in the sub-folder with the same name in the **templates** directory.

### 5.3.2 One file per page policy

To make debugging easier each page in the site has its own individual python file in the **userViews**. For example the page **templates/grutor/gradeSubmissions.html** is implemented in **userViews/grutor/gradeSubmissions.py**. The python file is designed to implement all of the functions needed to handle that page. Such functions include

- The view function.
- Redirection functions to go from this page to other pages while performing actions.
- AJAX functions for performing manipulations of data on the page.

### 5.3.3 AJAX vs Redirect

For the best user experience any time information on the page is being transferred to or from the server it should be done with an AJAX call. This prevents the page from reloading (and losing any other modified data) as well as it makes it easier to use the forward and back buttons in the browser.

If you are attempting to change the page while performing an intermediate action a redirect should be used. For example when leaving the **templates/grutor/gradeSubmissions.html** page there is a redirect which marks the submission being that was being graded as "Done".

## Appendix A

# Example Auto-grader plugin

```
import re
from subprocess import Popen, PIPE
from os import environ
from datetime import datetime

from app.helpers.command import Command

import itertools, json

PLUGIN_NAME = "Python_(pyunit)"

PYTHON_TEST_REGEX=r"^.*?def_(.*?)\.(self\):$"

def testFileParser(filename):
    """
    Takes a python pyunit test file and attempts to extract all
    the tests.
    It may accidentally grab other helper functions you define
    because it is
    using a simple regex search.
    """
    with open(filename) as f:
        contents = f.read()

    testRegex = re.compile(PYTHON_TEST_REGEX, re.M)

    testNames = []

    for test in testRegex.findall(contents):
        if test != "setUp" and test != "tearDown":
            testNames.append(test)
```

```

    return testNames

def runTests(cmdPrefix, testFile, timeLimit):
    startTime = datetime.now()

    testProc = Command(cmdPrefix + ['python', testFile])

    timeoutReached, testOut, testError = \
        testProc.run(timeout=int(timeLimit), env=envron)

    #Check for a timeout
    if timeoutReached:
        print "Timeout_reached"
        summary = {}
        summary['totalTests'] = 0
        summary['failedTests'] = 0
        summary['timeout'] = timeoutReached
        summary['died'] = False
        summary['rawOut'] = ""
        summary['rawErr'] = ""

        return summary, {}

    #testOut, testError = testProc.communicate()

    summary = {}
    summary['rawOut'] = testOut
    summary['rawErr'] = testError

    #Parse the results
    testSummarySearch = re.search("Ran_([0-9]+)_tests_in",
        testError)

    #If we don't find the test summary the tests died so we
    report that
    if not testSummarySearch:
        summary['totalTests'] = 0
        summary['failedTests'] = 0
        summary['timeout'] = timeoutReached
        summary['died'] = True

        return summary, {}

    # Get the list of failed tests

    failedSections = testError.split('= '*70)

    splitList = []
    for section in failedSections:

```

```

        splitList.append(section.split('-'*70))

#flatten the lists
sections = list(itertools.chain.from_iterable(splitList))

#Create the summary
summary['totalTests'] = int(testSummarySearch.group(1))

failedSections = sections[1:-1]
summary['failedTests'] = len(failedSections)/2
summary['died'] = False
summary['timeout'] = False

#Extract the messages from the failed tests
failedTests = {}
for i in range(0, len(failedSections), 2):
    header = failedSections[i]
    headerParts = header.split()
    tname = headerParts[1]

    pyunitmsg = failedSections[i+1]

    failedTests[tname] = {'hint': pyunitmsg}

return summary, failedTests

```



## Appendix B

# Example Late Calculator plugin

```
from decimal import Decimal

PLUGIN_NAME = "Euros_(Optimized_Grade)"

def calculateGrades(gradeList):
    """
    Basic_(and_default)_late_calculator_only_colors_late_
    submissions_red
    so_that_they_can_be_noticed_in_the_gradebook
    """
    lateAssignments = []
    for i, assignment in enumerate(gradeList):
        for j, problem in enumerate(assignment):
            #check for no submission and skip
            if problem is None:
                continue
            if problem['isLate']:
                lateAssignments.append((i,j))

    lateAssignments.sort(reverse=True, key=lambda x: gradeList[x
        [0]][x[1]]['rawTotalScore'])
    num = 0
    while num < 3 and num < len(lateAssignments):
        pos = lateAssignments[num]
        gradeList[pos[0]][pos[1]]['highlight'] = "yellow"
        num += 1
    while num < len(lateAssignments):
        pos = lateAssignments[num]
        gradeList[pos[0]][pos[1]]['highlight'] = "red"
```

```
    gradeList[pos[0]][pos[1]]['finalTotalScore'] = Decimal('
        0.00')
    num += 1
return gradeList
```