KING'S College LONDON

University of London

# An Architecture for Automated Assessment and Feedback of Programming Tasks

Final Project Report

Author: Sam White

Supervisors: Martin Chapman; Steffen Zschaler

Student ID: 1262820

April 2016

## Abstract

Assessment and feedback are core elements of the teaching process. The ability to automate these processes would be incredibly powerful, and potentially allow staff to make much better use of their teaching time. In the context of programming assignments at an undergraduate level, many previous attempts to automate marking have been made, however almost all of them have been greatly limited or short-lived.

This project develops a highly flexible and scalable architecture for the automation of assessment and feedback. This architecture consists of a central component, responsible for the management of data in the system; and the ability for a number of 'Tools' to be developed which can interact with this central component - according to well-defined interfaces - to provide the automation of generating marks and/or feedback.

This project delivers a detailed description of the architecture, an implementation of the central component, and an example 'Tool' which can be used as a reference for others to create their own. It is hoped that the implementation of the system will demonstrate the power of automation, and inspire continued development in the area.

## Originality Avowal

I verify that I am the sole author of this report, except where explicitly stated to the contrary. I grant the right to King's College London to make paper and electronic copies of the submitted work for purposes of marking, plagiarism detection and archival, and to upload a copy of the work to Turnitin or another trusted plagiarism detection service. I confirm this report does not exceed 25,000 words.

Sam White

April 2016

**Acknowledgements**

I would like to express my gratitude to Mr Martin Chapman and Dr Steffen Zschaler for their assistance and supervision throughout the project.

I would also like to extend my thanks to the entire Department of Informatics at King's College London. Their excellent and enthusiastic staff have provided me with many fantastic opportunities in addition to an outstanding level of teaching.

Finally, I wish to thank my family and friends for their continued support and encouragement throughout my studies.

# Contents

**List of Appendices**

# Chapter 1

# Introduction

Assessment and feedback are core elements of the teaching process. The notion of automating the processes involved with assessment and feedback are hardly new. Indeed, in recent years many such efforts have been made in the context of teaching computer programming. However, almost all of these efforts have been short-lived, and short-sighted: they focussed only on particular programming languages; they used assessment methods that were limited in their power; and they were only intended as small, atomic parts as part of a much larger assessment process.

This project aims to develop a flexible, highly-powerful and scalable architecture and system for the automation of assessment and feedback for programming assignments. Tools which carry out the tasks of assessment and feedback generation will be able to be developed separately then 'plugged in' to a central management system. There will be very little restriction on how these Tools can be developed, offering a great amount of customisation and flexibility.

As the size of undergraduate Computer Science classes continues to increase in institutions across the UK, the time and resources of those involved in the teaching process - mainly course leaders and teaching assistants - are further stretched. It is hoped that the use of the system developed in this project will allow teaching staff to rapidly begin to automate the workflows regarding assessing students' work and providing them with feedback. Teaching staff will be able to achieve more during their time with students: promoting discussion, answering queries, and offering invidual support; rather than monotonously assessing several students' work.

Certainly there is evidence throughout history that automation - powered by computers - can help improve efficiency, accuracy and ultimately trust in systems [14].

A key aim of the project is to develop a software system which is fully ready to be deployed in a real-world production environment, such as a university for the use of aiding undergraduate teaching.

This report will outline the background surrounding the area of automated assessment, and critically review a comprehensive range of research on the matter - stating the importance and clear need of the work that is to be undertaken. A brief case study will examine how easily existing assessment practices could be automated.

A more detailed analysis of requirements of the system will be gathered, then used to create a full specification of the software which is to be developed. The development process and implementation will be presented; along with reflective evaluations and critical conclusions.

# Chapter 2

# Background & Context

This chapter will introduce the background surrounding assessment, feedback, and the automation of these in the context of teaching computer programming.

## 2.1 Background

**Assessment** is a vital part of both the teaching and learning processes. For the teacher, it provides a channel of feedback which allows them to study how well learning goals are being met. It also provides a good metric for student progress, as well as that of the whole class [26].

For students, assessment is a way of monitoring their own progress and getting confirmation of their understanding. However, the existence of assessment itself seems to affect the way students learn - students are often found directing a large proportion of their efforts to elements of the course which are assessed and will ultimately affect their final grade [4]. For this reason, it is absolutely vital that assessments are well suited to the course, and that they are marked accurately.

However, for students, assessment alone is only so useful for the development of their knowledge and understanding. **Feedback** is vital in order for students to understand *why* it is they have met - or not met - the requirements for what they have been asked to do. High-quality feedback is more likely to keep students engaged and motivated in a course [17].

For even small classes, it is hard to give high-quality, individual feedback to students in a feasible manner with respect to the resources available (the human hours available from the teachers and teaching assistants). Thus, it is even harder to scale good feedback to larger classes that are becoming more common amongst universities. For a class of perhaps 250 undergraduates, the level of staff that would be required to give one-to-one feedback is simply impractical.

A common response to feedback forms provided to students during lab sessions is that students feel there are not enough TAs available to answer their questions fully.

As a result, systems which provide *automated* methods for assessment and/or feedback have become very desirable. They allow for the time of teaching staff to be better utilised - they are able to give most attention to students who require it most [8].

Creating a system to provide assessment and feedback of a high quality, however, is challenging. These challenges arise in the difficulty of describing to a machine how work should be assessed, and what feedback students should receive. In the domain of programming assignments, these challenges are somewhat easier to overcome; it is easier to describe to a machine how to assess a computer program than, for example, a literature essay. Even in this domain, though, providing useful feedback is still challenging. What is meant by ‘useful feedback’ is that which can be used to further the student’s understanding of where they could improve, delivered at a level they are able to understand.

Many systems have been developed for these purposes in the domain of programming assignments [18] - but most have been very short-lived or have rather large short-comings. Many of these short-comings lie in the narrowness of describing the assessment criteria, and how to provide feedback. Many systems are limited to one method of marking. In addition, such systems do not provide any sort of feedback more than just a numerical grade for that assignment. The capabilities and limitations of current systems will be explored further in the Literature Review.

## 2.2 Motivation

As the size of undergraduate programming classes increases, it is not always possible for a university department to simply hire extra teaching staff accordingly. It is therefore inevitable that (at least some) students will receive less individual attention from teaching staff; and it’s very possible that the students who do require the most help are less likely to receive it.

As a result, it would be useful for students to be able to get feedback on their programming from more sources than just teaching staff. A system which could provide feedback automatically is one ideal solution; provided the feedback is at a good standard, at least to some degree akin to what a TA (Teaching Assistant) would provide.

This would reduce the strain on the vital human resource of teaching staff member's time, allowing them to focus their efforts on the students which they identify as requiring extra assistance. In a later chapter it will be discussed how several marking tasks TAs undertake could be automated.

A system that is online would also allow students to gain feedback on their programming 'out-of-hours', when teaching staff are not readily available.

## 2.3 Problem Domain

The primary problem areas to be addressed as the main focusses for this project are: the automated assessment - or 'marking' - of programming assignments (in particular those given to university students as part of programming courses); automatically generating and providing high-quality feedback for students; and having a 'modular' system which can be centrally managed but also be expandable in terms of its capabilities.

### 2.3.1 Context

Most automated feedback systems work as follows [18]:

- Teaching staff set up *assignments* for a group of students. These comprise of a list of *requirements* and the means to assess these requirements - for example, an assignment may be to produce a program which takes as an input an integer, and outputs the square of that integer.
- Students can then *submit* their *solution* to the system, at which point it is run against the requirements and a grade for that student is stored, perhaps immediately viewable by the student. Their solution is usually in the form of source code file(s).

- Some systems allow for multiple attempts - i.e., for multiple solutions to be submitted sequentially up until a certain deadline.
- Some facility is available on the system to export the results of all submitted solutions, for analysis or to put towards student's overall grades.

### 2.3.2 Automated Marking

One such area - of particular interest to this project - is the lack of a system where setting up the automatic marking procedure for assignments is as simple as possible for the teaching staff. Ideally a solution where a minimal amount of coding has to be carried out. This aims to make the system easy-to-use, as well as saving the vital resource of time for teaching staff.

At present there exist systems which attempt to streamline this process, but they are currently limited in what they can achieve. For example, one such system only allows for the output of a solution to be examined for certain keywords or regular expressions [5].

It is to be expected that - at least for more advanced assignments - that the teacher will have to spend some time setting up how the assignment will be marked. The goal is that this time spent still results in a saving compared to marking each submission manually or with an existing system.

### 2.3.3 Automated Feedback

Another area focuses on the feedback that is automatically generated for students when they submit their solutions to the system. A study which examined the the behaviour of first-year Computer Science students has found that some of the biggest predictors of negative achievement include 'confusion' and 'frustration' at their coding tasks [28].

Feedback should be as meaningful and relevant as possible to the specific assignment. Feedback should also be presented in a manner which is appropriate for their level of understanding (for example, a student in a course which is requires no prior programming knowledge may receive different feedback than a course which does assume prior knowledge, if both make the same mistake).

### 2.3.4 Modular System

As previously mentioned, systems that currently exist are usually tailored to one specific method of marking - for example input/outping checking or unit testing. Being tied to one method does not provide a great deal of flexibility. It would be desirable to have a system where multiple methods for marking or generating feedback are available, or could be created and 'plugged in' to the system, then these methods would be collated by some central management system.

# Chapter 3

# Literature Review

This section aims to present a systematic and comprehensive literature review of recent (ideally 2005 onwards) automated assessment and/or feedback tools and systems, for programming exercises. It will discuss the similarities and differences in the way different tools define tests, provide feedback, allow for different solutions to problems, and how they deal with related concerns such as security and integrating with existing learning systems. After these have been discussed, conclusions will be drawn on what works well amongst current systems, what does not work well, and where there is room for improvement in the future.

## 3.1 Methodology

The main questions that aim to be addressed in this review are:

1. What are the features of automated marking systems reported in literature?
2. What features are commonly missing and could be future research directions?

The review will focus on systems where students submit code for an assignment, then are provided with automatically generated assessment results and/or feedback. Systems which assess anything other than this (e.g. qualitative questions about the work) will not be included. Data used will be of various formats: conference proceedings, journals, articles; and will come from various sources, for example the *ACM Digital Library* and *IEEE Explore*. Aggregation and search tools such as *Google Scholar* will also be utilised in order to find articles published elsewhere. No sources that date to before 2005 will be considered, and sources that date prior to 2007 will not be highly regarded. The volume of sources found relating to this topic should be enough to justify excluding these as most likely obsolete. Sources will be collected by searching for phrases including *'automatic'/'automated'* and *'assessment'/'marking'/'grading'* or *'feedback'*.

## 3.2 Results

In this section, the results will be discussed in terms of the areas in which the features identified are best categorised.

### 3.2.1 Programming Languages

Most systems tend to use C, Java, or Python as the supported programming language. Java was the most common, which is unsurpsing given Java's popularity for teaching introductory programming courses. Some systems were language independent, due to the way they assessed work: by running a program with a given command and seeing if the output is as expected. There appeared to be little to no intentional support, however, for more than one language within a system. This is most likely due to the scope of many of these systems being for use at a single university.

### 3.2.2 Defining Tests

By far the most common method for defining tests was by running programs with some predetermined (or randomly generated) input data and checking whether the output matched what should be expected. Unit tests were also used in many systems. [22] Some systems used Domain Specific Languages (DSLs) to define tests - these are languages designed to make it easy to describe what a test should be checking for [27].

Whilst checking input/output is perhaps the easiest way to assess a program, it does have its limitations. Mainly that code cannot be introspectively analysed, therefore acting as somewhat of a 'black box'. As a result, it is impossible to determine *how* a student's code achieved its results, which is often an important factor when teaching programming. Unit tests can achieve this more easily.

### 3.2.3 Assessment

Almost all sytems awarded marks in a very simple, standard way: tests are each given a mark for being correct, and the sum of all possible marks adds to a round number such as 100. Some, however, did experiment with strategies such as limiting the marks that can be achieved with each resubmission [23].

A common feature of many systems worth noting is the ability for submissions to be manually assessed. This is often achieved by simply allowing the user to view the full code of a submission. One system even combined the use of manual and automated assessment [9].

### 3.2.4 Feedback

Many systems exist solely for the use of teaching staff. Submissions are collated from elsewhere, run through the system, then the results are collated in terms of raw marks. Some systems do output reports of which tests passed and which failed, which could be forwarded to students. However more recent systems do offer graphical user interfaces for students to submit, and receive feedback such as test results for their submissions.

Most feedback does not go in to much more depth than test results, or stack traces in cases of errors, though.

### 3.2.5 Architecture

The vast majority of systems employ a monolithic architecture. Even those with support for multiple assessment methods or multiple languages appear to have this support integrated quite substantially in to their codebases. Almost all systems have very strictly-defined processes for marking; there is little to no scope for customisation beyond the differing test parameters for distinct tasks [15, 19].

## 3.3 Conclusions

A huge variety of systems exist to perform automated assessment. Most have had very short lifespans, due to being bespoke solutions for a particular course or university, or for a niche area of research; and are therefore not suitable for production deployment. As such, there is a lot of overlap in work carried out, even as the volume of research published in this area seems to be accelerating in recent years.

As most systems are bespoke, their processes for assessment and feedback are often very rigid. Customising the way assessment is carried out beyond the methods already employed by a system would often involve a substantial change in its architecture. The architecture of systems is almost always monolithic, too, increasing the difficulty of customising the system.

It was interesting to note that virtually no papers discussed automated assessment systems working well, at scale, in the context of a university.

It seems apparent that there is a need for a more unified approach. A common architecture which attempts to harmonise a lot of duplicated effort in terms of managing submissions, sandboxing, collating reults, would be ideal. This architecture would need to be very expandable in terms of the assessment methods, however, as this is what would vary the most depending on the use case.

Some attempts have been made to unify assessment systems, but they seem to have very little adoption, perhaps due to their rigid architiecture [30]. If more systems were made open source, however, this could encourage further collaboration.

There is an overwhelming sense that many in the field can see the potential for automated assessment, but a tendency to reinvent the wheel seems to be slowing progress towards a production-ready solution which can cater to a wide range of needs and integrate well with existing workflows.

# Chapter 4

# Case Study: Mark Scheme Analysis

A brief case study of a mark scheme - and how elements of it could be automated - will be presented in this chapter. It is hoped that this will demonstrate the power and potential time savings of automating the marking process.

#### Context

The mark scheme to be examined is for a coursework exercise as part of a ‘Programming Practice’ module, a mandatory module part of undergraduate Computer Science degrees at King’s College London. The module takes place in the first year of study, and is designed to be appropriate for students who have no prior programming experience. This coursework exercise is the third in a series of four as part of the module.

The module uses the Java programming language throughout and therefore all submissions must be in Java.

#### Mark Scheme Format

The mark scheme itself consists of several entries, each of which specify a particular criteron which a student’s code has to meet. Not each entry is worth an equal amount of marks - indeed the marks for each entry range from 1 to 8 (of a total 28).

The vast majority of marks can be determined by examining a student's code. Some, however, are determined by asking the student questions about the code. This is to ensure that the student fully understands the programming concepts the coursework was designed to test, and that their solution hasn't simply been copied from another student or the Internet.

Teaching Assistants carry out this process, and marks are then moderated by one of the course lecturers. Is will usually take 10-15 minutes to mark a single student's code; and of course the student has to be present during the marking process, in order for the Q&A section of the marking to be completed.

An abbreviated version of the mark scheme can be found in Table 4.1.

| # | **Description** | **Marks** |
|---|---|---|
| 1 | Four classes are used: NetworkDevice, Packet, Channel, Network | 4 |
| 2 | Correct fields exist in respective classes: e.g. address, channel, and key in NetworkDevice | 8 |
| 3 | Fields have appropriate access modifiers (private; protected; final; static) | 1 |
| 4 | Accessor and mutator methods are present only for certain fields | 1 |
| 5 | NetworkDevice constructor sets 'address' field | 1 |
| 6 | Packet constructor sets 'source address' and 'dest. address' fields | 1 |
| 7 | Inheritance is used properly: e.g. Client extends NetworkDevice | 1 |
| 8 | Subclasses have correct fields | 1 |
| 9 | Object instances are instantiated correctly | 1 |
| 10 | Method for adding an 'AccessPoint' to a 'Network' (mutator method) | 1 |
| 11 | Program behaves as expected with specified parameters (checks program output) | 4 |
|  | **Ask the student to explain how their code achieves the following:** |  |
| 12 | Sending a 'Packet' from a 'Client' to an 'AccessPoint' | 1 |
| 13 | AccessPoints accept valid handshakes as per the protocol specified | 1 |
| 14 | Clients accept valid handshakes as per the protocol specified | 1 |
| 15 | Unsuccessful handshakes are handled as per the protocol | 1 |

Table 4.1: Abbreviated Mark Scheme used for programming task

**Analysis**

In this section, each different type of criterion will be presented, alongside a possible way in which their marking could be automated.

**Checking for the existence of Classes** *Example: 'Four classes are used: NetworkDevice, Packet, Channel, Network.'* The existance of classes can easily be tested for. This can be achieved in a crude way using a shell script to check for the existance of the relevant file name (e.g. `NetworkDevice.java`). A more sophisticated solution would be to use a unit test. This way the existance of the file is not just asserted, but also that the class itself has been defined properly. Example code to test this using the popular xUnit-based JUnit[1] is given in Listing 1.

**Checking for the existence of Fields and their Access Modifiers** *Example: 'Correct fields exist in respective classes: e.g. address, channel, and key in NetworkDevice.'* Checking for the existance of certain fields and methods within classes is trivially achieved through the use of unit tests. In Java, this is not entirely straightforward for `private` fields and methods, but is possible through the use of `java.lang.reflect`, part of the standard library. This library also allows for the Access Modifiers of fields or methods to be examined. Listing 2 gives an example of how to access methods and fields within a class. JUnit tests could then be built to check these.

**Constructor methods behave as expected** *Example: 'NetworkDevice constructor sets 'address' field'.* This can also be checked through the use of unit tests. An instance of the object is constructed, and then the fields can be checked through the `java.lang.reflect` class as described above.

**Program output is correct for a given input** *Example: 'Program behaves as expected with specified parameters (checks program output)'.* This can be tested through unit tests, but perhaps more simply by executing the program and checking its output. If the program terminates with a zero exit code (completed successfully), examine the `STDOUT` and ensure the output matches what is expected. This can all be automated through a shell script such as *bash*.

**Asking the student questions about their work** *Example: 'Ask the student to explain how their code achieves the following...'.* This style of question is by far the most dif-

[1] http://junit.org/

ficult to fully automate. As described above, these questions are assessed by a TA having a conversation with the student about their work. Automating a fully interactive conversation would be rather complicated, especially for the use of determining whether marks should be awarded. However, there is a way that this style of question could be partially automated: A system is created where students are presented (through a Graphical User Interface) with a list of questions - derived from the marking criteria - regarding the coursework. They input their answers in text boxes (possibly under timed conditions), and their answers are stored by the system. These answers can later be reivewed by a member of teaching staff, to decide how many marks can be awarded. This solution still requires manual input and time to be taken by a member of teaching staff, but hopefully this process would take much less time than interviewing students individually.

```java
@Test
public void testNDClassExists {
  try {
      Class.forName("NetworkDevice");
  } catch (ClassNotFoundException e) {
      Assert.fail("A NetworkDevice class was not found.");
  }
}
```

Listing 1: Testing for the existance of a class using JUnit.

```java
import java.lang.reflect.Field;
import java.lang.reflect.Method;
// ...
Class<?> reflectedClass = MyClass.new().getClass();
Method methods[] = reflectedClass.getDeclaredMethods();
// getName(), getReturnType(), setAccessible(), and invoke() can be used on Method objects
Field fields[] = reflectedClass.getDeclaredFields();
// getName() and setAccessible() can be used on Field objects
```

Listing 2: Finding all fields and methods that exist in a class in Java.

## Conclusions

As demonstrated, all of the marking criteria in the mark scheme could be at least partially automated. In fact, all questions that were directly regarding the code and its semantics could be fully automated through the use of shell scripts or unit tests. Only the criteria involving a conversation with the student could not be fully automated; however a partially automated method which would hopefully save time was given, using an interactive Question & Answer system.

Although, this hinges on the time spent writing the Unit Tests and configuring the Q&A tool being significantly less than the time that would be spent marking each submission individually. Additionally, Unit Tests and shell scripts are not perfect in their behaviour - edge cases will exist where students use unconventional methods for implementing their code. These edge cases could still be awarded marks, but the code will need to be examined manually. Therefore a system for reviewing the marks would need to exist also.

Another downside to using unit tests is that they require knowledge of *specific* names for classes, methods and fields in order to function correctly. Therefore students will need to be told - or at least strongly suggested - of what names to use for these in their solutions. It could be argued that giving away these names would give students less scope for flexibility in their answers; or that they would have less opportunity to discover the *'best'* solutions themselves.

An efficient way to organise the running of the unit tests, shell scripts, etc., and then collate the results data for each student, would also be required to ensure the system was efficient in terms of teaching staff time.

In summary, there is a strong case for suggesting that mark schemes for programming tasks can be automated - fully for some criteria; partially for others. The potential time savings would be significant, especially for larger class sizes, provided that configuring the automation apparatus (e.g. unit tests) and collating results can be undertaken in an efficient way.

# Chapter 5

# Requirements

This section aims to outline the requirements upon the solution that is to be proposed. A general overview of the proposed system will be given first. User stories will then be presented: to demonstrate some potential use cases of the system, and to extract extra requirements. Finally, a clear and detailed scope will be defined, along with a full list of requirements.

## 5.1 Proposed Solution: an Overview

The solution I am proposing is an modular system to be used by both teaching staff and students. It is comprised of a Central Management Component, along with defined interfaces with which marking and/or feedback components (herein referred to as *'Tools'*) can be created.

These interfaces will communicate using a widely-supported protocol. This will allow the tools to be created using a variety of different implementations and frameworks; giving them a large amount of flexibility over how they operate, their environment, their parameters, and so on. As such the tools can operate as independent *microservices* [13].

The tools will provide one interface which the central component can use to trigger the processing of a student's submission. After they have completed their respective process, they can send back a mark and/or feedback to the Central Management Component through interfaces provided to them.

The Central Management Component (herein referred to as the *'CMC'*) will - in addition to providing and consuming the interfaces mentioned above - also provide a user interface for system administrators, teaching staff, and students. These UIs will facilitate the management of students, courses, assignments, and submissions - entities common in the context of Virtual Learning Environment (VLE) systems.

System administrators will be able to use the CMC to add new tools to the system, as well as manage security and users.

For teaching staff, the system should provide a straight-forward way to set up their courses; and create assignments for them. The UI should allow them to select (and configure) which tools to use for this assignment - examples tools could include compilers, linters, unit test suites, static analysis tools, and so on. In the case of unit testing, ideally the system will help with the generation of the test code; but realistically it may only be achievable that the teacher has to create the unit testing code then input it to the tool. There will also be a number of configurable parameters the assignments; such as deadlines, an optional limit on submissions allowed, grade capping for late submissions, among others. There should also be some facility to perform basic analysis on students' submissions for an assignment - or at the very least a way to export data to a common format (such as Comma-Separated Value (CSV)) for analysis elsewhere.

The system will keep track of which students belong to which courses. Teachers for courses will be able to set assignments, described above, and then the students belonging to that Course will be able to submit their solutions (Submissions). Their submissions will consist one or more source code files. The system should then notify the tools that there is a new submission for them to process, through some common interface provided by the tools. After the tools have finished, they can report back a grade and/or some feedback about that submission (through the interfaces provided by the CMC).

When assignments are configured, each of the assigned tools can be given 'weighting'. These weightings will be used to calculate a weighted arithmetic mean which constitutes the final grade for a submission. Tools which only provide feedback, and not a grade, can be given a weighting of zero. If a tool provides both a grade and feedback, but the course leader only wants that tool to be used for feedback, it can similarly be given a weighting of zero. The sum of all non-zero weightings used in an assignment must, of course, add up to 100%.

Formally, the final mark for a submission will be calculated using the individual marks from non-zero weighted tools $\{m_1, m_2, \ldots, m_n\}$, and respective weights $\{w_1, w_2, \ldots, w_n\}$, as:

$$\bar{m} = \frac{\sum_{i=1}^{n} w_i m_i}{\sum_{i=1}^{n} w_i}.$$

As independent microservices, tools are free to generate feedback in any way they wish. Typically this feedback will be from parsing the output of command line tools or files from running tests or other analysis. However, this feedback must be presented in some commonly-agreed format when it is sent back to the CMC. This is so that multiple pieces of feedback can be uniformly stored and displayed without the need for additional processing. Some sort of markup language - such as Markdown or HTML - would be ideal for this.

The system should be easy and intuitive for students to use. This is an important factor in ensuring students maintain interest in using the system and engage with the feedback they are provided with. The system will provide a way of students to get an overview of what assignments they have, when they are due, how many attempts they have left, and so on.

The user interface should be web-based. This allows for students to access the system on a multitide of different platforms and devices; and also accommodates for the wide variety of environments universities are likely to have for their IT setup.

It should be noted that some elements of the CMC - mainly the management of courses, assignments, and users - already feature as part of many existing systems. The choice has been made to develop our own solution rather than attempting to utilise an existing system because:

- It may be cumbersome to extend the data models used in an existing system, especially if the data is stored in a format which is hard to manipulate. By creating our own system we can ensure all the data models are tailored to our requirements, and are stored in a way which is easily customisable and configurable for the programming language/framework which is chosen.

- In order to expand an existing system to one which can be modularised and expanded as we wish, a compatibility layer may have to be developed to bridge functionality with another programming language. This would result in added complexity to the architecture.

## 5.2 User Stories

The following user stories will be titled using a template developed by Connextra in 2001[1], then described in some more detail.

### 5.2.1 As a professor, I want to be able to have assignments marked automatically, so that I can save time

To a course leader or member of teaching staff, having student's assignments marked automatically offers several benefits. The greatest benefit is perhaps the time saved by not having to mark each submission manually. The time saved could then be better spent giving students more individual attention where required.

Intuitively, a requirement for this is that the time spent setting up the assignment on the system (and perhaps configuring Tools) is less than the time it would take to manually mark submissions individually.

As such, a professor would want a system where not only are the assignments relatively easy to set up, but also *accurate*. All software is imperfect - and there will be cases where automated marking systems will make mistakes and give people incorrect grades. In these cases, the system should offer a simple facility to manually assess that student's submission and override the automated grade.

### 5.2.2 As a teaching assistant, I want students to receive feedback automatically, so that my time can be spread less thinly

As was discussed earlier, simply hiring extra teaching assistants (TAs) as a means to cope with increasing class sizes is an impractical solution.

Currently issues can arise where teaching assistants have to spread their teaching time between too many students with too many queries, and thus the quality of the feedback they can

[1]"As a {role}, I want {goal/desire} so that {benefit}" [6]

provide in such a small amount of time is diminished. This is exacerbated in classes where a large proportion of students do not have prior programming experience, as these students will be less confident to search the web or other resources to find the answers to their problem.

In some cases TAs spend a large amount of lab time assessing work, in addition to answering students' queries. If assessment was largely or entirely automated, this would free up a lot of time for TAs.

As such, if students were able to receive grades/feedback on their work from an automated system, which they are able to use at any time, then perhaps some students would be able to continue to develop their solution and overcome their problems simply by utilising the feedback given from this system. This would hopefully result in fewer questions being asked during teaching sessions, and therefore the TAs could devote more of their time to the students who do require further help.

### 5.2.3 As a student, I want to get useful and timely feedback on my work, so that I can develop my skills

Most courses offer students multiple sources of help for their assignments: lab sessions with TAs present, office hours for the course leader(s), and occasionally additional 'drop-in' sessions. Across many institutions, class sizes in undergraduate Computer Science faculties are trending upwards. As a result, resources (mostly staff time) are stretched more than before.

An automated, web-based system would allow students to submit work and receive feedback at any time, anywhere. Students could use this feedback to develop their skills and improve their understanding.

## 5.3 Scope

Due to the limited resources available to deliver this system (one developer, over the course of approximately six months), the scope will have to be limited. The following limitations seem sensible:

- The system will only allow for assignments written in the Java programming language.

This is an appropriate choice given it is by far the most used language taught in Computer Science courses, according to two recent surveys [24, 29].

- Feedback can only be presented in the common markup langauge. There will not be support for hosting artefacts (such as images) on the CMC. However the feedback could link to externally-hosted artefacts.
- Students' submissions will have to consist of one ZIP file only. A much more elegant solution would be for submissions to point to a Version Control System (e.g. Git[2] or Mercuiral[3]) repository; however this would take significantly longer to develop.
- The system need not concern itself about performing detailed analysis on students' results. The system should, however, provide a way to export the results of assignments in some standard way, for instance to a CSV file.

Desirable features which could be implemented at a later date to improve the system will be discussed in a later chapter.

[2]https://git-scm.com/
[3]https://www.mercurial-scm.org/

## 5.4 Requirements List

| **1. Users** |
|---|
| 1.1 Users can be of type Student, Staff, or Administrator |
| 1.2 Administrative accounts (Admins and Teaching Staff) can be seeded to the database through a configuration file |
| 1.3 Students can register an account for themselves |
| **2. Administration** |
| 2.1 Tools can be added and removed from the system through an administrative section of the Web UI |
| 2.2 Access Tokens (randomly generated base64 strings) can be generated and revoked through the administrative section of the Web UI |
| 2.3 Only Admin accounts can access this part of the UI |
| **3. Tools & Interfaces** |
| 3.1 Tools provide an interface on which they listen for requests to invoke their marking/feedback process when a submission is made |
| 3.2 The address of this interface is provided to the CMC through the administrative UI discussed above |
| 3.3 The CMC provides an interface which tools can consume to report marks for a submission |
| 3.4 The CMC provides an interface which tools can consume to report feedback for a submission |
| 3.5 Once tools complete their respective process, if a mark is generated this is reported back to the CMC through the provided interface |
| 3.6 Once tools complete their respective process, if feedback is generated this is reported back to the CMC through the provided interface |
| **4. Courses** |
| 4.1 Teaching Staff can create courses |
| 4.2 Courses have a Title and optional Description |
| 4.3 Students are able to enroll themselves on to a course |
| 4.4 Teaching staff can view an ‘Enrollment List’ of all students currently enrolled on to one of their courses |
| 4.5 Once enrolled, students can unenroll from a Course |

| **5. Assignments** |
|---|
| 5.1 Teaching Staff can create assignments for their courses |
| 5.2 Assignments have a Title and an optional Description |
| 5.3 Assignments have a Start datetime and a Deadline datetime |
| 5.4 Assignments can optionally accept late submissions |
| 5.5 Assignments have an optional 'Late Cap', a percentage at which late submissions will be capped (e.g. 40%) |
| 5.6 Assignments use one or more tools, selected by the Teaching Staff upon creation |
| 5.7 Each Tool can be given a marking weight between 0% and 100% |
| 5.8 Each Tool selected with a non-zero weighting will be used to calculate the final mark (a weighted arithmetic mean) |
| 5.9 Tools with a zero weighting will still be used but purely for feedback, a grade - if reported - will be ignored |
| 5.10 The sum of all weights must add to 100%. |
| **6. Submissions** |
| 6.1 Students can make submissions for assignments whose courses the Student is enrolled on |
| 6.2 Submissions can only be made after the start date |
| 6.3 Submissions can only be made before the deadline, unless late submissions are available for that assignment |
| 6.4 Submissions must take the form of a single ZIP file |
| 6.5 Once a Submission is made, the CMC must inform all the tools associated with that assignment that a new submission has been made, through the tools' provided interfaces |
| **7. Marking** |
| 7.1 Tools can report a mark for a given submission |
| 7.2 When this mark is reported through the interface provided by the CMC, it is stored and can be viewed by the student (provided the CMC is expecting a mark from this tool, that is to say that this tool is registered as part of that assignment and has been given a non-zero weight) |
| 7.3 Once marks for all non-zero weighted tools have been received, a final mark is calculated |
| 7.4 This mark is capped if late capping is enabled for that assignment |
| 7.5 The final mark can be viewed by the student |
| 7.6 A mark can only be reported once for a given submission/tool pair |
| 7.7 Marks can be overridden by Teaching Staff in the case of an error |

| |
|---|
| **8. Feedback** |
| 8.1 Tools can report feedback for a given submission |
| 8.2 When this feedback is reported through the interface provided by the CMC, it is stored and can be viewed by the student (provided the CMC is expecting feedback from this tool, that is to say that this tool is registered as part of that assignment) |
| 8.3 Feedback must be presented in the expected markup format |
| 8.4 Feedback can only be reported once for a given submission/tool pair |
| **9. Analysis** |
| 9.1 Data for an assignment (all submissions, their marks, etc.) should be able to be exported by Teaching Staff in a commonly-used format such as CSV |
| **10. Miscellaneous** |
| 10.1 A secure way to log in and log out of the Web UI is available |
| 10.2 As such, passwords are stored in a secure manner |
| 10.3 Reports of marks and feedback will only be accepted if the Tool can provide a valid Access Token as part of their request |

# Chapter 6

# Specification & Design

This chapter will give a full and detailed specification of the data models to be implemented; the architecture along with its components and their interactions; the processes involved in the operation of the system; and some requirements for different sections of the user interface.

## 6.1 Data Model

Various entities which make up the core of the data model can be extracted directly from the problem domain - namely **Users** (Students, Staff), **Courses**, **Assignments**, and **Submissions**.

Other entities, such as **Marking Tool**, are somewhat more subtle to identify, and their structure more heavily dependent on the implementation chosen.

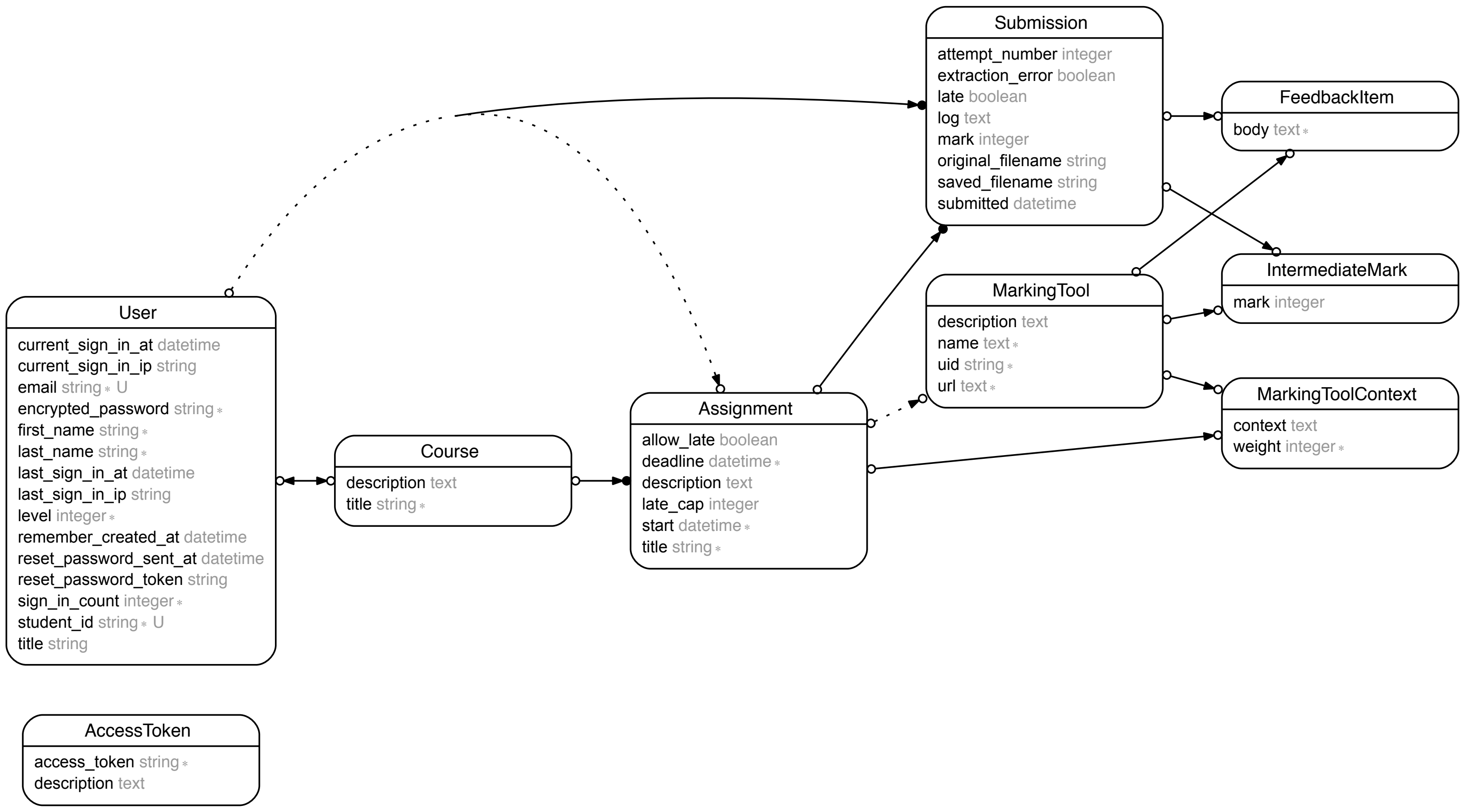


Figure 6.1: Entity-Relationship diagram (Bachman notation [1])

Figure 6.1 shows a Bachman Diagram for all the entities which will exist in the system. Some of these entities are partial JOINs upon two entities with some extra attributes. Each entity along with its main attributes and its relationships are presented below.

**User**

A user can be either a student, a member of teaching staff, or a system administrator. Student IDs are stored here so that they can be provided to Tools, if the student's ID is required for their marking/feedback process.

- Has and belongs to many **Courses**
- Has many **Assignments**, through **Courses**
- Has many **Submissions**

Key Attributes:

- `email` : string
- `first_name` : string
- `last_name` : string
- `student_id` : string
- `level` : enum (0, 1, or 2 for Student, Staff, or Admin respectively)
- `encrypted_password` : string

**Course**

A series of lessons/topics on a particular subject. This will usually correspond directly to a teaching module at the University.

- Belongs to a **User**, as 'Teacher'
- Has and belongs to many Students (**Users**)

- Has many **Assignments**

Key Attributes:

- `title` : string
- `description` : text

**Assignment**

An Assignment is a piece of work assigned to students undertaking a particular Course.

- Belongs to a **Course**
- Has many **Marking Tools**, through **Marking Tool Contexts**

Key Attributes:

- `title` : string
- `description` : text
- `start` : datetime
- `deadline` : datetime
- `allow_late` : boolean (should Submissions be accepted after the `deadline` has passed)
- `late_cap` : integer (optional cap for late submissions, in %)

**Submission**

Represents a single attempt at an Assignment, submitted by a User.

- Belongs to an **Assignment**
- Belongs to a **User**
- Has many **Intermediate Marks**

- Has many **Feedback Items**

Key Attributes:

- `submitted` : datetime
- `late` : boolean (was it submitted after the deadline)
- `attempt_number` : integer (how many times has the user attempted this submission previously)
- `original_filename` : string
- `saved_filename` : string (filename as stored on the system)
- `extraction_error` : boolean (was there an error extracting the ZIP file)
- `mark` : integer (the final mark for the Submission, 0-100%)
- `log` : text (log of all events occuring related to that Submission, mainly used for debugging purposes)

**MarkingTool**

A Marking Tool represents a Tool (a microservice which provides and consumes the interfaces described previously) that can give a mark and/or feedback on a Submission.

Key Attributes:

- `description` : text
- `name` : text
- `uid` : string (canonical (unique) identifier used by Tools when reporting data back to the CMC)
- `url` : string (the location to send a request to when a Submission is made where this Marking Tool is part of the Assignment)

**MarkingToolContext**

This is a JOIN on **Marking Tools** and **Assignments**, with two extra attributes:

- `context` : text (a textual description of how this Marking Tool is used in the context of that Assignment)
- `weight` : integer (the weight given to this Tool when used to calculate the final mark of a Submission for that Assignment)

**IntermediateMark**

This is a JOIN on **Marking Tools** and **Submissions**, where a non-zero weight has been given (i.e. this Tool's reported mark will be used in calculating the final mark), with one attribute:

- `mark` : integer (the mark returned from the Tool)

**FeedbackItem**

This is a JOIN on **Marking Tools** and **Submissions**, where feedback is reported from a Tool about a particular Submission, with one attribute:

- `body` : text (the feedback in some standard markup format)

**AccessToken**

Access Tokens are used to authenticate messages coming from Tools to the interfaces provided by the CMC regarding marks and feedback. Any requests which do not contain an Access Token present on the system will be rejected.

Key Attributes:

- `access_token` : string (randomly generated base64 string)
- `description` : text

### 6.1.1 Possible Enhancements

Possible enhancements that could be made to the data models in future include:

- Adding integrations between Courses and similar models in external systems, such as Virtual Learning Environments (VLEs) like Moodle[1]. These integrations could be simple URLs for ease of navigation between systems, or more thorough integrations such as pulling Course titles and descriptions from a VLE.
- Having PDFs attached to Assignments, allowing students to be provided with a much more thorough set of instructions without having to leave the CMC. Having the ability for the Description attribute of Assignments to be in some sort of markup - such as Markdown - could also improve how expressive Assignments can be.
- The option of federating parts of the User model to an external server (Single Sign-On, or SSO). This way User can authenticate themselves through a system such as RADIUS [31] which has already been set up and secured by their education establishment.

These enhancements are deemed out-of-scope for the project due to the time limitations meaning development time would be better focussed on the more technical aspects of the system, such as implementing the interfaces.

## 6.2 System Architecture

### 6.2.1 Overview

An overview of the architecture to be used can be seen in Figure 6.2. It comprises one central component - the Central Management Component (CMC) - along with several tools. These tools act as microservices are able to provide a mark/feedback for given submissions.

In order for the system to scale well, and be suitably customisable and expandable in terms of functionality, it is apparent that the tools will have to be able to be developed somewhat independently from the main system.

[1] https://moodle.org/

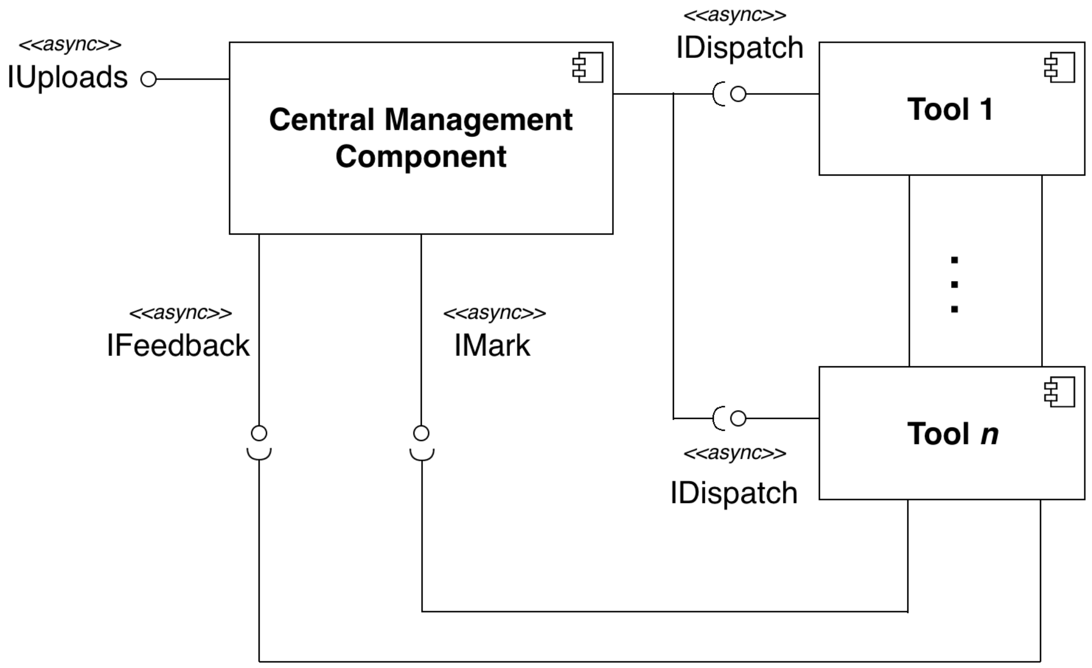


Figure 6.2: UML Diagram of System Architecture

A microservice-based architecture achieves this: any number of tools can be developed and added to the system, as long as their behaviour is correct with respect to the defined interfaces.

A more monolithic architecture would make it much harder to develop new tools. This would be due to a much less well-defined separation of concerns; and having to work on a (single) codebase which could easily become messy and congested.

An overview of the different components, their roles, and their respective interfaces will now be given.

### 6.2.2 Central Management Component

The main role of the CMC is to facilitate the management of data in the system (more detail on the data model can be found earlier in this chapter). Mostly this data will be managed through the web-based user interface, but data pertaining marks or feedback will be made through two provided interfaces.

The CMC will provide three interfaces:

**Submissions Interface** which students can use to upload attempts at a given assignment.

**Marking Interface** which tools can consume to report marks back about a given submission.

**Feedback Interface** as above, but for feedback rather than marks.

The shape (i.e. what data is required) of these interfaces will be discussed in more detail later.

### 6.2.3 Tools

Tools are microservices which exist to generate a mark and/or feedback on a student's submission.

In order to generate this mark/feedback, the tools may require a number of parameters. As microservices it is up to the invidivual tools to provide a way for these parameters to be configured.

Tools will provide one interface:

**Dispatch Interface** which triggers the marking/feedback process for a given submission.

The shape (i.e. what data is required) of this interface will be discussed in more detail later.

The process of generating a mark/feedback is triggered when the CMC invokes the **Dispatch Interface**. The invocation may contain futher parameters made available from the CMC, if the Tool requires them - these could include the student's ID, or the assignment ID.

Once a tool has completed the process of generating a mark and/or feedback for a given submission, it then consumes the **Marking Interface** and/or **Feedback Interface** respectively in order to submit this data back for collation.

#### Accessing Submissions

Once uploaded through the **Submissions Interface**, students' code will be extracted from the ZIP file and stored on a specific location on the filesystem of the CMC. Tools will need to mount this location remotely, so that they also have access to extracted submissions.

This is not the most ideal approach. A better approach would be to have students submit the repository location for some distributed Version Control System (such as Git); then this location could be communicated to tools who could then fetch the code themselves. However, given the scope of of the project, it was decided that submissions will be made in ZIP format. This will be discussed more in a later chapter.

#### Tool Configuration

If Tools require some configuration by teaching staff - for example, if their processes differ depending on specific assignments - then it is the responsibility of tools, as microservices, to provide a way for staff to achieve this. This could be through a static configuration file, or some sort of graphical user interface. In this instance, tools would have to ensure that teaching staff can configure their tool in a managable, secure way.

### 6.2.4 Submissions Interface

This interface is provided by the CMC, and will be used by students to upload a submission (for a given assignment) to the CMC.

It will require the student to provide:

- One valid **ZIP-compressed** archive file

This interface will be presented as a form on the web-based user interface. Students can upload a submission for any current (i.e. between the start and deadline) assignment they have (through their courses).

The process which will proceed once an upload is complete is detailed in the *Processes* section.

### 6.2.5 Dispatch Interface

This interface is provided by the individual tools, and will be automatically invoked by the CMC every time an assignment which is configured to use a tool receives a new submission.

It will require, at the minimum:

- The **Submission ID**, so that the tool can identify where to find the student's code; and report back marks and/or feedback correctly

Optionally, tools could ask for other parameters to be provided, including but not limited to:

- The **Assignment ID**, in the likely case where the tool's process differs for distinct assignments.
- The **Student ID** of the submitter, in the case where the tool's process differs for distinct students.

The CMC should define a standard list of all the extra parameters which tools are able to ask for.

What parameters tools wish to be given to them will then need to be specified when they are added to the CMC through the administrative UI.

### 6.2.6 Marking Interface

This interface is provided by the CMC, and is intended for use by tools to report a mark back to the system for a given submission

It will require the tool to provide:

- The **Mark** itself, a numerical value between 0 and 100 inclusive
- The **Submission ID**, along with
- the **Tool's UID** (canonical name), so the CMC knows which submission/tool pair to associate the mark with
- A valid **Access Token** to authenticate the request

The CMC will then store the mark, and if necessary (all expected marks now reported), calculate the final mark for that submission. This occurs if *all* of the following conditions are met:

- The access token is valid (i.e. belongs to the list of tokens stored by the CMC)
- A mark is expected from this tool for this Submission. This means that the assignment this submission belongs to actually uses this tool, with a non-zero weight assigned to it

- A mark for this submission/tool pair has not been received already (repeat reports will be ignored)

Note that tools could potentially report themselves to be a different tool (giving a different tool's UID if known). Any tool supplying a valid access token is expected to be trusted. Therefore the system is dependent on the access tokens being kept secret, and those responsible behaving responsibily.

The system could be further improved by having access tokens assigned to specific tools. This will be discussed as part of other potential improvements in a later chapter.

### 6.2.7 Feedback Interface

This interface is very similar to the Marking Interface, except tools can report items of feedback through it, rather than marks.

It will require the tool to provide:

- The **Feedback** itself, in some standard markup format
- The **Submission ID**, along with
- the **Tool's UID** (canonical name), so the CMC knows which submission/tool pair to associate the mark with
- A valid **Access Token** to authenticate the request

The CMC will then store the feedback. This occurs if *all* of the following conditions are met:

- The access token is valid (i.e. belongs to the list of tokens stored by the CMC)
- Feedback is expected from this tool for this Submission. This means that the assignment this submission belongs to actually uses this tool
- Feedback for this submission/tool pair has not been received already (repeat reports will be ignored)

Note that any tool assocaited with an assignment can report feedback, unlike marks which can only be reported by the subset of tools which are given a non-zero weight for calculating a final mark.

## 6.3 Processes

This section aims to outline how some of the key processes of the system are to be carried out - most importantly how Submissions are handled by the CMC and tools.

### 6.3.1 System Administration

#### Managing Users

Students can register themselves on the system using a form available to them through the UI. Assuming a standard database is used to store user credentials, system administrators could also bulk import students using a database query.

Administrative and teaching staff accounts will be loaded initially from a configuration file upon first deployment, but administrators will then be able to manage (add/edit/remove) Teaching Staff accounts using an ‘Admin Panel’ section of the UI.

Administrators will also be able to manage individual student accounts through the UI.

#### Tools and Access Tokens

Tools and access tokens can be managed through an admin panel section of the UI, accessible only to users who are administrators.

To create a tool, a administrator will need to give the tool a hame, optional description, a canonical ID (a name unique within the system), and also the remote address of the tool to trigger its dispatch interface. Any extra parameters required by the tool will also need to be specified.

Access Tokens can be revoked at any time by an administrator, and new ones can be generated as long as a Description is given. The Token itself should be randomly generated in a cryptographically secure way, to ensure that Access Tokens are hard to 'guess' for any potential adversaries.

### 6.3.2 Managing Courses & Configuring Assignments

#### Course Management

Creating a course is available to any administrator or teaching staff account. A course requires the user to specify a title, and optional description text.

Once created, it is possible to manage courses by viewing enrolment lists; creating assignments; and exporting data relating to the course (e.g. enrolment, marks, etc.). Teaching staff can manage their own courses. administrators can manage any course.

#### Assignment Configuration

To create an assignment, the user must specify:

- A title,
- A description,
- A start datetime,
- A deadline,
- Whether or not to accept late Submissions, and if so
- an optional Mark Cap for them.

The user will also have to specify the strategy they want the assignment to be marked with. That is, what tools do they wish to use for this assignment, and how do they want the final grade to be calculated (what weightings to give to each tool). A more detailed description of these weights is presented earlier in Chapter 5.

Assignments can use as many tools as they desire, and the user is free to choose from any tool which has been added to the system by an administrator.

When creating an assignment through the CMC only the tools and their weights are specified. As tools are independent services, if they require their own assignment-specific configuration, they will have to provide their own procedures to achieve this.

### 6.3.3 The life of a Submission

Figure 6.3 shows the activities that are undertaken as part of a submission's life cycle: from the point of being uploaded, to the point where feedback and marks are received. It also demonstrates what part of the system is responsible for each activity. The stages are as follows:

#### Upload

Firstly, a user uploads a submission through the web-based UI. They can reach this stage by visiting a link from the respective assignment's page.

They choose a single ZIP file to upload, and select a 'Submit' (or similar) button.

The file is then uploaded to the CMC, and stored. The user is redirected to a 'Submission Summary' page created for that submission.

#### Extraction

Upon a successful upload, the CMC will attempt to extract the ZIP file to a unique directoy based on the unique ID of that submission.

If the extraction fails for some reason - perhaps not being a valid ZIP file; a password being required; the file is corrupt in some way - then the `ExtractionError` field of the submission model is set to `true`, and the process terminates.

The fact the extraction failed will be displayed to the user on the 'Submission Summary' page.

#### Notification of Tools

Once a successful extraction is made, the dispatch interface of each tool associated with the Submission (through its Assignment) needs to be invoked.

The CMC will asynchronously send requests to each of the tools - using the addresses configured by an administrator - passing that tool the submission ID along with any extra parameters it needs from the CMC to begin its process of generating a mark and/or feedback.

Each tool with a non-zero weight (i.e. the ones that will be used to calculate the final mark) will be displayed on the ‘Submission Summary’ page with a ‘Pending’ status.

#### Receiving Marks

Once a tool has finished its process for generating a mark, it sends this information back to the CMC through the ‘Marking Interface’ (outlined previously).

This mark is stored by the CMC as an ‘Intermediate Mark’ and displayed to the user on the ‘Submission Summary’ page.

#### Final Mark Calculation

Each time a mark is recorded by the CMC from a tool, it checks whether there are any more ‘Pending’ marks for that submission. If there are not, then the final mark for that submission is calculated from all of the intermediate marks; stored; and displayed to the user on the ‘Submission Summary’ page.

#### Receiving Feedback

Once a tool has finished its process for generating feedback, it sends this information back to the CMC through the ‘Feedback Interface’ (outlined previously).

This feedback (in some common markup format) is stored by the CMC and displayed to the user on the ‘Submission Summary’ page. Any necessary processing for the markup format to be displayed properly will need to be carried out.

## 6.4 UI Requirements

The requirements of various different pages to be displayed on the web-based UI will be outlined below, along with brief descriptions.

As a web-based UI has been chosen, many elements of the UI will appear familiar to users; given the exponential growth of web-based applications in recent years. [11]

### 6.4.1 Admin Panel (System Management)

An 'Admin Panel' page should exist which allows for the management of tools and access tokens. It should feature:

- A list of tools, their names, descriptions, and URLs
- The ability to remove tools
- A button to add a new tool
- A list of access tokens
- A button to revoke an access token
- A button to generate a new access token

### 6.4.2 Registration & Login

Registration and Login pages should feature the following:

- Username (or email address) field
- Password field
- 'Remember Me' checkbox
- Link to registration page on the login form
- Registration page includes fields for name, email, and student number
- Registration page also includes both a password and password confirmation fields

### 6.4.3 Course View

The course overview should feature the following:

- Title and description clearly displayed
- The ability to enroll if not already
- The ability to unenroll if currently enrolled
- A list of current and upcoming assignments, along with their due dates

### 6.4.4 Assignment View

The assignment overview should feature the following:

- Title and description clearly displayed
- Marking breakdown table featuring weights and contextual descriptions of tools used
- A list of current submissions for this assignment
- A button to upload a new submission (if allowed)

### 6.4.5 Uploading a Submission

The page for uploading a submission should be faily simple, including:

- Information about what types of file are allowed (i.e. ZIP)
- A button to open the Operating System's file dialog for selecting the ZIP file
- A submit button
- Once pressed, the submit button should be disabled so that accidental duplicate submissions are avoided

### 6.4.6 Submission Overview

The assignment overview should feature the following:

- The attempt number
- The final mark (or ‘Pending’ if not yet calculated)
- The marking breakdown (i.e. a list of all tools used and their currently returned marks, or ‘Pending’)
- A list of all feedback returned by tools so far (if any)

### 6.4.7 Dashboard

The dashboard page should give a clear overview of items that may be requiring the user’s attention. It should therefore feature:

- A list of recent submissions (for all assignments)
- A list of current assignments or those and those that are due to start soon, sorted with the nearest deadlines at the top

### 6.4.8 Miscellaneous

Each of the pages should also adhere to the following rules, so that the user interface is intuitive to use, easy to navigate, and information is displayed clearly:

- Tables are used whenever relevant (i.e. listing submissions or a marking breakdown)
- A global navigation bar is available with links to core elements of the UI, such as logging in/out, the dashboard, courses list, and assignments list.
- Whenever an element such as a course, assignment or submission is included as part of a list, clicking on its name (or the most appropriate field) should link to the page containing that element. This way navigation between different elements on the site is simple, and the user will never be more than one click between accessing data related to their current page.

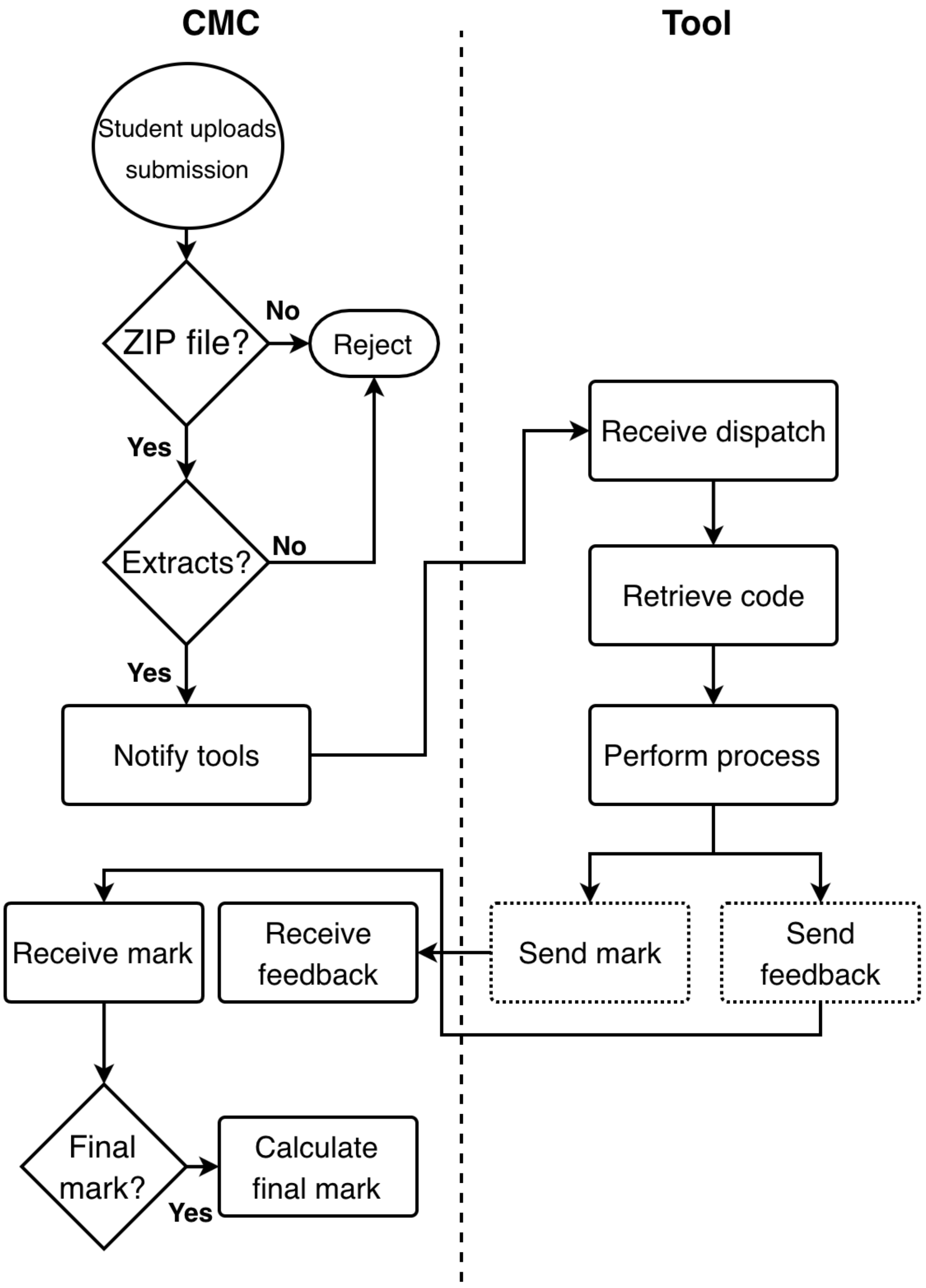


Figure 6.3: Flowchart demonstrating the activities that occur in a Submission's lifecycle (*Send mark* and *Send feedback* are optional, depending on the tool, but at least one should be performed)

# Chapter 7

# Development Process

## 7.1 Agile Software Development

*Agile* software development encompasses a large variety of different principles, methodologies, and philosophies; all with a focus on being adaptive, self-organising, rapid, and flexible. This is in contrast to methodologies regarded as *Waterfall*.

Many of the development procedures regarded as agile can be traced back to the Toyota production system [25], which heralded continous improvement and evolution at its core.

As such, an agile approach is an appropriate choice for the development of this system. Given the relatively short time in which the project is to be completed, being able to develop rapidly and adapt to change quickly are imperative; as requirements are likely to change as ideas develop.

## 7.2 Testing

Extreme Programming (XP) [2] and Test-Driven Development (TDD) [3] are agile software development processes which revolve around the idea of writing code 'test-first': first of all, the developer writes a (failing) unit test for what the code to be written should achieve; then they develop the minimal amount of code required to make the test pass; then code is refactored.

This process is reapeated for each atomic piece of code (usually a single method).

The principles of TDD will be utilised throughout the project. The aim is to have as high a test coverage[1] as possible. As long as the tests are well-written, this allows us to have confidence in the code that is written, that it behaves as expected, and allows us to notice if any changes to the code could break functionality.

In practice, this will mean that code is broken up in to smaller methods wherever possible[2], which are independently testable with a full range of inputs with expected outputs.

## 7.3 Coding Standards

All code written will adhere to standard style practices wherever they exist; and this can be enforced through the use of linting tools, which automatically scan through code for improvements that can be made.

The use of open-source software packages and platforms will be used wherever possible, and it is planned for the software behind this project to be open-sourced at some point in the future, too.

## 7.4 Continuous Integration

Continous Integration is the process of continually testing code and merging it to a shared 'mainline' very often. This becomes trivially easy with Version Control Systems such as *git*; utilising the process of 'branching' off of a mainline to create a feature, then merge that branch back once development is completed and the feature meets all the requirements.

Git - specifically the GitHub platform[3] - will be used throughout the development of this project. Branches will be created for each atomic feature, and branches will not be merged back in to the main branch unless unit tests which properly test the function of the feature are passing.

[1]Test coverage is the proportion of code which has unit tests written for it
[2]Namely the 'Don't repeat yourself' (DRY) software principle
[3]https://github.com

The cloud-based continous integration system CircleCI[4] will be used to enforce this. Each time code is committed and pushed to GitHub on a branch, CircleCI will build the software, run the full unit test suite, and then run the software and run some basic integration tests against it. CircleCI will also be configured to run linting tools against the code to ensure coding standards are maintained. Only if these tests all pass can a branch be merged back in to the main branch.

Acceptance tests (verifying that the requirements, as understood by the relevant stakeholders, of a feature are met) will also be present through regular meetings with the two project supervisors to demonstrate development progress.

## 7.5 Professional Issues

The development of this project will be undertaken with a full awareness and appreciation of the relevant professional issues. The BCS Code Of Conduct[5] will be used as a reference for these. Issues that will be understood and acted upon throughout development include:

**Professional Competence and Integrity** It is firmly believed that the work that is to be carried out is within our professional competence, and that the undertaking of this work will help to develop our skills and understandings of the technology used.

**Duty to Relevant Authority** Work will be carried out fully within the operational requirements set by the College. No attempts to deceive stakeholders on the performance or capabiltiies of the product will be made.

**Duty to the Profession** It is believed that this software will contribute back to the profession by developing something which can potentially be used to enhance the teaching and understanding of Computing.

[4]https://circleci.com
[5]http://www.bcs.org/category/6030

# Chapter 8

# Implementation

In this chapter, the specifics of how the system was implemented will be discussed in detail. First, what pieces of technology were chosen will be presented, and justified. Concerns such as security and maintenance will also be discussed.

The finalised web-based UI will be presented, with screenshots, and the various interfaces will be given detailed specifications for their behaviour.

An example tool was created and will be demonstrated as a reference implementation, of how tools can interact with the CMC.

Finally, options and concerns for how to deploy the system will be briefly discussed, including containerisation, scalability, and security.

## 8.1 Technology

The technologies that will be used for the implementation include:

**Ruby on Rails** [1] (better known as just 'Rails') for the core of the CMC. Rails is a Web Application Framework, written in the Ruby[2] programming language, and based around the MVC (Model-View-Controller) software design paradigm. It allows for rapid development

[1]http://rubyonrails.org/
[2]https://www.ruby-lang.org/

of web applications and offers many libraries to facilitate the easy manipulation of data models. It is a good choice for the CMC given its proven stability, ease of use, and the short time it takes to develop an application is appropriate given the scope of time for the project.

**PostgreSQL** [3] will be used as the database. It is stable, scalable and operates with high performance with a low memory footprint. Additionally, it integrates very well with Rails making it a perfect fit.

**JavaScript** [4] will be used for interactions in the web UI front-end. Specifically, the yet unreleased ECMAScript 7 will be used for development, then transpiled to regular, browser-supported JavaScript through the use of a transpiler called Babel[5]. JavaScript is supported in all major browsers and is therefore the only realistic choice for complex client-side interactions.

**Sass** [6], an extension of CSS (Cascading Style Sheets) will be used to style the web UI. It compiles down to browser-supported CSS, and offers many features to ease development of stylesheets. This will help to keep the time developing the UI at a minimum, so the more technical aspects of the system can be focussed on.

**Webpack** [7] will be used to transpile the ECMAScript 2015 and Sass down in to JavaScript and CSS respectively, so it can be consumed by users' web browsers. Webpack is a high-performance asset bundler which will help to give users a fast and responsive experience when using the UI.

**Node.js** [8] is a JavaScript runtime which will be used to create the example tool. It is simple to respond to and send HTTP requests; code is easy to read; and it is a very popular piece of technology; making it an ideal choice for a reference tool.

The Ruby libraries **devise**[9] and **delayed_job** [10] will be utilised to help with secure authentication and asynchronous task queueing respectively. **Rspec**[11] will be used as the Unit Testing framework for the application.

[3] http://www.postgresql.org/
[4] http://www.ecmascript.org/
[5] https://babeljs.io/
[6] http://sass-lang.com/
[7] https://webpack.github.io/
[8] https://nodejs.org/
[9] https://github.com/plataformatec/devise/
[10] https://github.com/collectiveidea/delayed_job
[11] http://rspec.info/

The Ruby linter **Rubocop**[12] will be used to ensure all code written consistently follows the Ruby Style Guide[13], a community-driven style guide.

**Twitter Bootstrap**[14] will be used as the basis for front-end styling. It is well established as having excellent UX (User Experience) when utilised properly. This is so that development time can be spent on the more technical aspects of the system, rather than Design and UX which are not the main focal points of the system.

A full list of libraries used, their versions, and their authors will be documented and credited in an Appendix.

Note that all software frameworks, languages, and runtimes are fully open source. This has its obvious benefits, such as giving us more control over the implementation; having more confidence in the software we use and its security.

### 8.1.1 Tools

Note that although the example tool will be developed in Node.js, as microservices tools can be developed in any programming language or framework, so long as they can **listen for and send requests over HTTP**. A crude survey of the most popular programming languages - according to a recent survey of developers[15] by Stack Overflow - finds that all offer libraries to interact via HTTP. Almost all offer this as part of their standard library. This gives developers a lot of flexibility when it comes to chosing what to create their tools with.

### 8.1.2 HTML as the standard format for Feedback

It makes sense to use HTML[16] as the standard markup format for items of feedback. As the language of the web, the CMC would not have to perform any extra processing on feedback before presenting it to users - it can be sent by tools as HTML, stored as HTML in the database, then displayed directly as HTML to be rendered by users' web browsers.

HTML is simple to learn and incredibly widespread given the vast adoption of the web as a

[12]https://github.com/bbatsov/rubocop
[13]https://github.com/bbatsov/ruby-style-guide
[14]http://getbootstrap.com/
[15]http://stackoverflow.com/research/developer-survey-2015
[16]https://www.w3.org/TR/2014/REC-html5-20141028/

platform.

HTML offers a huge variety of ways to display information, and even offer complex levels of interaction. This gives a huge amount of power and flexibility to tools on how they wish to display items of Feedback. For example, images can be encoded as base64 as part of HTML, or external images can be linked to.

### 8.1.3 Alternatives

Popular alternatives for Rails include Sinatra[17] and Express[18]. Rails is preferable to these as it requires less work to get the application going: Rails deals with a lot of the scaffolding behind-the-scenes, especially when it comes to interactions with the Database. Its strong foundation in MVC also makes it easy to expand later on.

For feedback items, an alternative to HTML would be Markdown[19]. Markdown is arguably a much simpler syntax to learn, and it compiles to HTML so could be displayed on the UI very easily. However, its simplicity also limits its capability. Beyond basic formatting elements such as headers, lists, and tables, Markdown offers limited to no support for images, dropdown lists, and other more complex elements which HTML offers. In order to give Tool developers the most flexibility, HTML is the most appropriate choice.

## 8.2 Rails (and MVC) in More Depth

**Model-view-controller** is a software architecture pattern which enforces separation between business logic and presentation logic (the UI). In the domain of web applications, business logic typically involves the manipluation of data models for entities such as Courses and Assignments, and the UI is just HTML rendered by a web browser [20].

When a browser sends a request to a Rails application, this request is handled by a *controller*. Which controller handles the request is decided by the *route* of the request. For example, if a browser were to visit `http://cmc/submissions/3`, the route would be `/submissions/3`. Based on a list of rules Rails would match this route to a corresponding controller.

[17]http://www.sinatrarb.com/
[18]http://expressjs.com/
[19]https://daringfireball.net/projects/markdown/

Listing 3 displays an example of some routing rules for the CMC, consisting of pairs: a pattern to match, and the controller (and action within the controller) to handle the request.

```
# config/routes.rb
get 'courses/all' => 'course#index'
get 'assignments/:id' => 'assignment#show'
```

Listing 3: Example of routes in the CMC.

Once a controller receives a request, it usually interacts with a model - an object representing an entity such as a User, stored in the database. After gathering or modifying data in the model, the controller then renders a *view* and returns this view to the browser as HTML. Figure 8.1 shows a detailed diagram of how a request is handled in Rails.

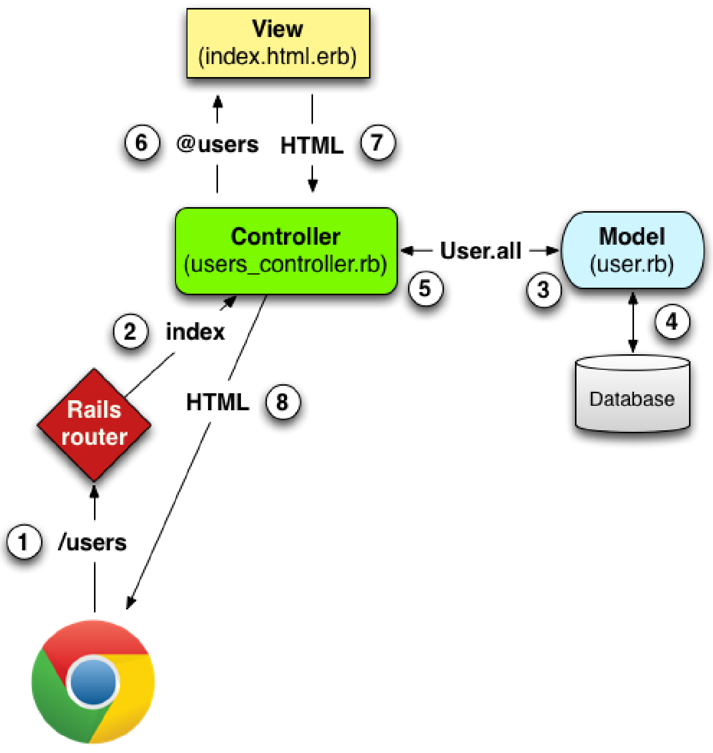


Figure 8.1: How a request is handled (with MVC) in Rails. Credit: Michael Hartl [16]

Views in rails come are in the form of a language called *erb* (embedded Ruby): HTML with Rails snippets embedded inside. Listing 4 shows an example of this. Note the Ruby code is contained within the <% %> tags.

```
<li>
  <%= link_to(logout_path) do %>
    Logout
  <% end %>
</li>
```

Listing 4: Example of ERB (Embeddable Ruby) within HTML.

As previously mentioned, Rails models correspond to entities within the database. Rails uses a DSL (Domain-specific language) to make it simple for us to define restraints and relationships on our models. Listing 5 demonstrates how this DSL can easily be used to validate attributes of the `User` model, as well as define its relations to other models in the database. It is this simplicity which allows applications to be developed so rapidly with Rails.

```
# app/models/user.rb
class User < ActiveRecord::Base
  VALID_EMAIL_REGEX = /\A[\w+\-.]+@[a-z\d\-]+(\.[a-z]+)*\.[a-z]+\z/i

  validates :email, presence: true,
                    format: { with: VALID_EMAIL_REGEX },
                    uniqueness: { case_sensitive: false }

  validates :first_name, presence: true
  validates :last_name, presence: true
  validates :level, presence: true
  validates :student_id, presence: true, uniqueness: true, if: :student?

  has_and_belongs_to_many :courses
  has_many :assignments, through: :courses
  has_many :submissions
end
```

Listing 5: Part of the User model in the CMC, defining constraints and relationships.

Another example of the power of the Rails DSLs can be seen in Listing 6, which is an extract of the `create` action for an Access Token. The methods `AccessToken.new` and `.save!` are provided to us by Rails once we have created an Access Token model. All of the interactions with the database itself are handled by Rails.

```ruby
# app/controllers/access_tokens_controller.rb
class AccessTokenController < ApplicationController
  def create
    @access_token = AccessToken.new(access_token_params)
    @access_token.save!
    redirect_to admin_panel_path
  end
end
```

Listing 6: The `create` action on the Access Token controller.

## 8.3 Interfaces

In this section, the various interfaces and their behaviour will be outlined in detail: what parameters are required, and what they return based on those parameters.

### 8.3.1 REST

The Submissions, Marking, and Feedback Interfaces will all be implemented as RESTful API calls. REST [12] is an architectural style used on the web to simplify interfaces which operate over HTTP.

The Dispatch Interface will take the form of a Webhook [21] (an HTTP request, typically using the `POST` method). This can be thought of as a RESTful call when used in the purposes of queuing a message/task to be processed, as we are doing here.

The advantages of using REST are clear:

- Naturally very easy to implement in Rails.

- It allows the system to scale easily, especially horizontally. RESTful API requests sent over HTTP can be routed to one of many instances of the CMC, which will then process that request by updating a single transactional database.
- The interfaces themselves become easy to define and document. There are several tools available for documenting RESTful APIs, for example Swagger[20].
- Most popular programming languages/frameworks have mature, straight-forward ways to interact over HTTP. This is convenient in our context as Marking Tools are supposed to be independent microservices which can operate in any way they want - thus, ideally, in any programming language they want.
- The system would be easy to integrate with other systems - such as VLEs - by opening up the RESTful API for other actions such as directly creating Assignments and Courses.

### 8.3.2 Specificiations

The detailed behaviour of each interface will now be defined. Note that `<CMC Base URL>` represents the URI of the CMC, and will vary depending on deployment. Example values could include `https://nexuscmc.com` or `http://193.112.1.32:41201`.

#### Submissions

**Provided by:** CMC

**Purpose:** for students to upload Submissions for an Assignment they are given (this will be called from the upload form in the UI)

**HTTP Method:** `POST`

**URL:** `<CMC Base URL>/submissions/create`

**Required Parameters:**

- `assignment_id` - the ID of the Assignment which the Submission belongs.
- `code` - the ZIP archive containing the source code of the Submission

[20]http://swagger.io/

**Required Headers:** A valid CSRF token will need to be given as part of the request, as this interface is invoked through the web UI. Rails handles this automatically when rendering the webpage with the form. This is to prevent Cross-Site Request Forgery.

**Responses:**

- `HTTP 303 See Other`[21] if the submission has been accepted, with a redirect URI containing the Submission Summary page
- `HTTP 401 Unauthorized` if the CSRF token does not match what Rails expects
- `HTTP 422 Unprocessable Entity` if the MIME time of the uploaded file does not equal `application/zip`

**Dispatch**

**Provided by:** Tools

**Purpose:** to notify the Tool that a new Submission has been made that should be processed

**HTTP Method:** `POST`

**Notes:** The URL, required parameters, and responses will all vary depending on the implementation of the specific Tool.

For simplicity, any parameters that are required will be encoded in to the URL. This means the CMC just has to encode that URL and send the request, and does not have to do any additional processing such as building an XML/JSON request body or adding extra HTTP headers.

Of course, the Submission ID will always be a required parameter - so that the Tool knows which Submission to process - but the CMC makes available various other parameters should the Tool require them to carry out its mark/feedback generation process.

The tool "asks" for these parameters by providing its URL in the form of a *template string*. For example, a tool which requires both the Submission ID and Assignment ID could have the URL:

`https://marking.tool/invoke?submission=%{sid}&assignment=%{aid}`

[21]Since HTTP/1.1, see https://tools.ietf.org/html/rfc7231#section-6.4

Note the `%{sid}` and `%{aid}` parts of the above string: the CMC will use these to insert the appropriate parameters (here `sid` for Submission ID, and `aid` for the Assignment ID to which the Submission belongs), creating a final URL for which to dispatch the request.

So, for the Submission ID `41`, belonging to Assignment `17`, the final URL the CMC will dispatch the request to will be:

```
https://marking.tool/invoke?submission=41&assignment=17
```

A full list of available parameters, the token to use in the template string and their descriptions are listed in Table 8.1.

| Token | Description |
|---:|---|
| `sid` | Submission ID |
| `aid` | Assignment ID |
| `teacherid` | The User ID of the Teacher who created the Assignment |
| `studentid` | The Student ID who uploaded the Submission |
| `atitle` | The title of the Assignment |
| `astart` | A datetime string of when the Assignment started |
| `adeadline` | A datetime string of when the Assignment's deadline |

Table 8.1: Available parameters to use in the template string for a Tool's dispatch URL

**Marking**

**Provided by:** CMC

**Purpose:** for Tools to report back a mark for a Submission to the CMC

**HTTP Method:** `POST`

**URL:** `<CMC Base URL>/report_mark`

**Required Parameters:**

- `sid` - the ID of the Submission the mark belongs to
- `tool_uid` - the unique canonical name associated to this Tool (this is set when the Tool is added to the system by an Administrator)

- `mark` - the value of the mark (between 0 and 100 inclusive)

**Required Headers:**

- `Nexus-Access-Token` - a valid Access Token known by the CMC

**Responses:**

- `HTTP 200 OK` if the mark is received and stored successfully
- `HTTP 400 Bad Request` if the mark is not within the range [0, 100]
- `HTTP 401 Unauthorized` if the Access Token is not present on the system
- `HTTP 404 Not Found` if a mark from this Tool is not expected for this Submission ID, or if the Submission is not known to the CMC
- `HTTP 409 Conflict` if a mark from this Tool for this Submission has already been received

**Notes:** each time a mark is received, the CMC will check to see whether the all Intermediate Marks for that Submission have been received, and therefore the final mark can be calculated. Listing 7 demonstrates this procedure.

```ruby
# app/models/submission.rb
def calculate_final_mark_if_possible
  return if intermediate_marks.any?(&:pending?)
  final_mark = 0
  intermediate_marks.each do |im|
    final_mark += im.mark * assignment
                              .marking_tool_contexts
                              .find_by(marking_tool_id: im.marking_tool_id)
                              .weight / 100
  end
  final_mark = final_mark.floor
  if late
    self.mark = [assignment.late_cap, final_mark].min
  else
    self.mark = final_mark
  end
  save!
end
```

Listing 7: Method to calculate and store the final mark for a Submission, if possible. Lines of code related to logging are removed for clarity.

**Feedback**

**Provided by:** CMC

**Purpose:** for Tools to report back feedback for a Submission to the CMC

**HTTP Method:** `POST`

**URL:** `<CMC Base URL>/report_feedback`

**Required Parameters:**

- `sid` - the ID of the Submission the mark belongs to

- `tool_uid` - the unique canonical name associated to this Tool (this is set when the Tool is added to the system by an Administrator)
- `body` - the body of the feedback, in HTML format

**Required Headers:**

- `Nexus-Access-Token` - a valid Access Token known by the CMC

**Responses:**

- `HTTP 200 OK` if the feedback is received and stored successfully
- `HTTP 400 Bad Request` if the `body` is not present
- `HTTP 401 Unauthorized` if the Access Token is not present on the system
- `HTTP 404 Not Found` if the Submission or Tool are not known to the CMC
- `HTTP 409 Conflict` if a feedback from this Tool for this Submission has already been received

**Notes:** Listing 8 demonstrates how the Feedback Interface is handled in Rails

```ruby
# app/controllers/feedback_item_controller.rb
def report
  (render_unauthorized_json && return) unless verify_access_token_header

  @submission = Submission.find(params[:sid])
  @marking_tool = MarkingTool.find_by!(uid: params[:tool_uid])
  @feedback_item = FeedbackItem.find_by(submission_id: @submission.id,
                                        marking_tool_id: @marking_tool.id)

  if @feedback_item.nil?
    render json: { response: 'Feedback received successfully!' }.to_json
    @feedback_item = FeedbackItem.create(submission: @submission,
                                         marking_tool: @marking_tool,
                                         body: params[:body])
  else
    render json: { response: 'Feedback for this tool and submission has already been received.' },
           status: 409
  end
rescue ActiveRecord::RecordNotFound
  render json: { response: 'Could not find matching record(s) for request.' },
         status: 404
end

# app/helpers/application_helper.rb
def verify_access_token_header
  token = request.headers['HTTP_NEXUS_ACCESS_TOKEN']
  return false if token.nil?
  AccessToken.find_by(access_token: token).present?
end
```

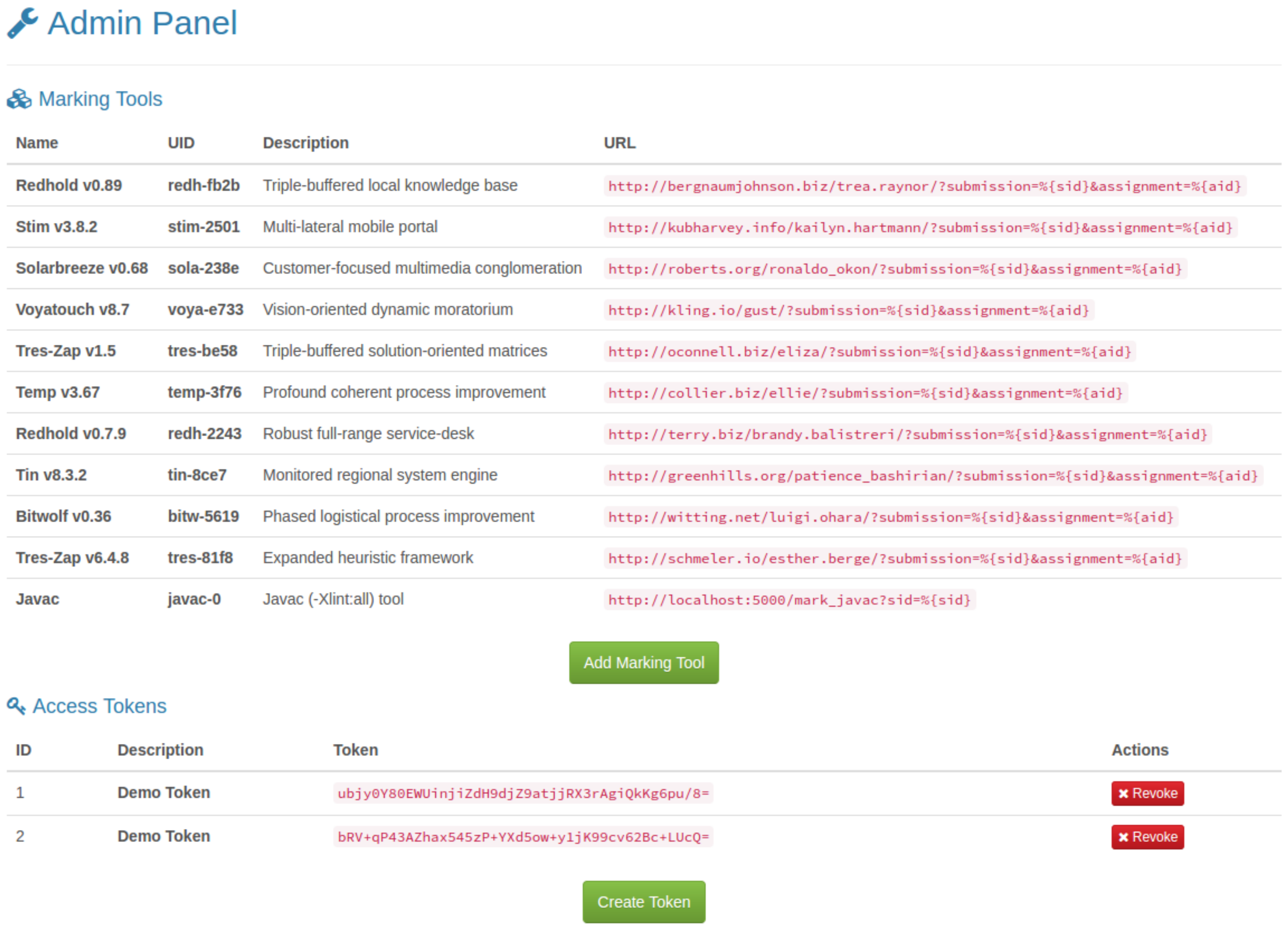
Listing 8: Method to handle requests to report feedback. Lines of code related to logging are removed for clarity. The `verify_access_token_header` method is included for completeness.

## 8.4 User Interface

Listed below are several screenshots of the finalised user interface.

Note the use of the Bootstrap CSS library. Various typography elements (multiple levels of headings, among others) are used to demonstrate the hierarchical navigational structure of the site. A navigation bar is present at the top of every page allowing for global site navigation, and HTML elements such as tables are used to best display information to the user.

Various different colours are also used to give context to particular elements - for example, yellow colours (indicating 'warning') are used to indicate Assignments with near due dates; and red colours (indicating 'danger') are used to indicate that an Assignment is past its deadline (and a Submission now would be deemed as late).



Admin Panel

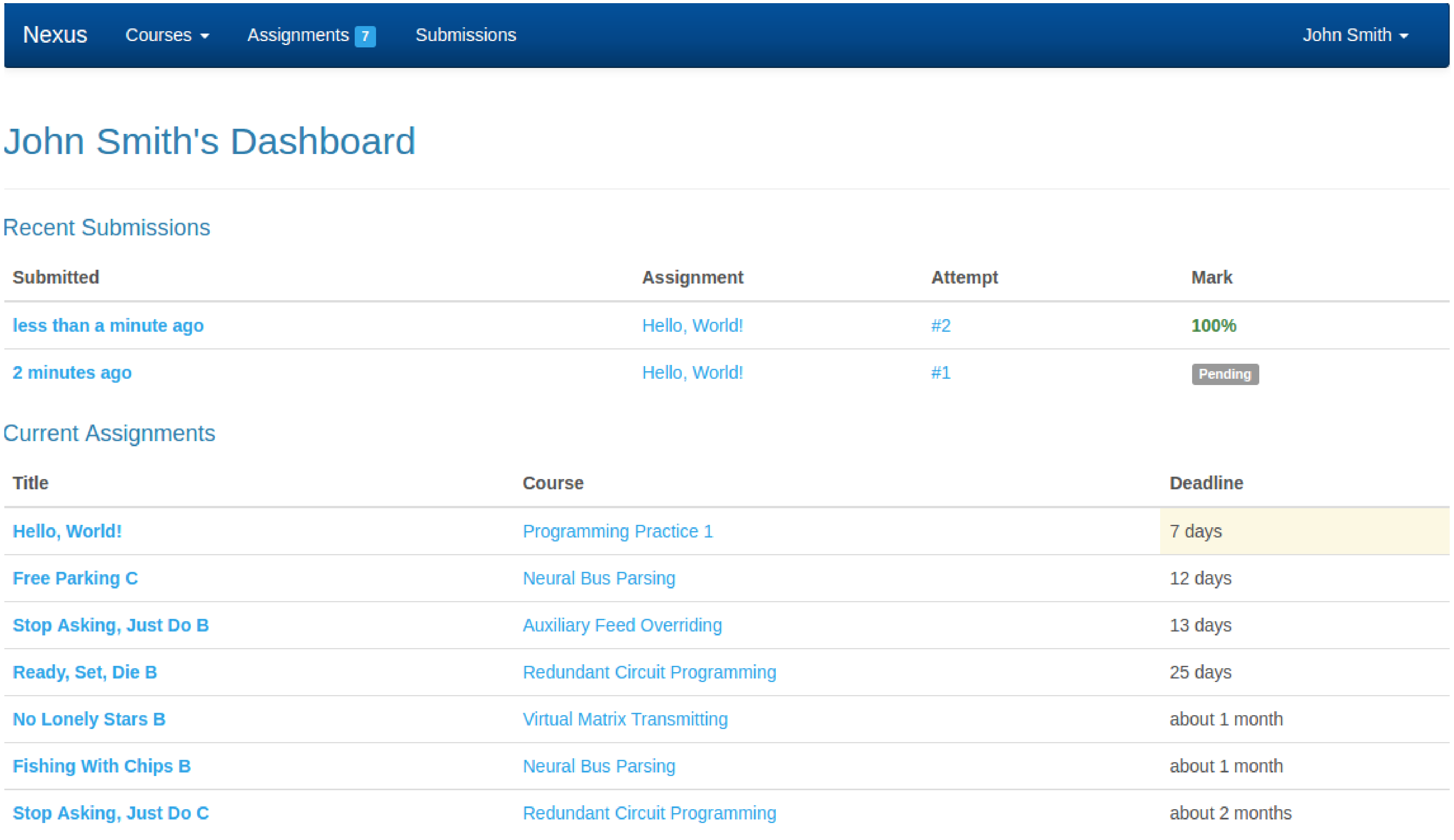
Marking Tools

| Name | UID | Description | URL |
|---|---|---|---|
| **Redhold v0.89** | **redh-fb2b** | Triple-buffered local knowledge base | `http://bergnaumjohnson.biz/trea.raynor/?submission=%{sid}&assignment=%{aid}` |
| **Stim v3.8.2** | **stim-2501** | Multi-lateral mobile portal | `http://kubharvey.info/kailyn.hartmann/?submission=%{sid}&assignment=%{aid}` |
| **Solarbreeze v0.68** | **sola-238e** | Customer-focused multimedia conglomeration | `http://roberts.org/ronaldo_okon/?submission=%{sid}&assignment=%{aid}` |
| **Voyatouch v8.7** | **voya-e733** | Vision-oriented dynamic moratorium | `http://kling.io/gust/?submission=%{sid}&assignment=%{aid}` |
| **Tres-Zap v1.5** | **tres-be58** | Triple-buffered solution-oriented matrices | `http://oconnell.biz/eliza/?submission=%{sid}&assignment=%{aid}` |
| **Temp v3.67** | **temp-3f76** | Profound coherent process improvement | `http://collier.biz/ellie/?submission=%{sid}&assignment=%{aid}` |
| **Redhold v0.7.9** | **redh-2243** | Robust full-range service-desk | `http://terry.biz/brandy.balistreri/?submission=%{sid}&assignment=%{aid}` |
| **Tin v8.3.2** | **tin-8ce7** | Monitored regional system engine | `http://greenhills.org/patience_bashirian/?submission=%{sid}&assignment=%{aid}` |
| **Bitwolf v0.36** | **bitw-5619** | Phased logistical process improvement | `http://witting.net/luigi.ohara/?submission=%{sid}&assignment=%{aid}` |
| **Tres-Zap v6.4.8** | **tres-81f8** | Expanded heuristic framework | `http://schmeler.io/esther.berge/?submission=%{sid}&assignment=%{aid}` |
| **Javac** | **javac-0** | Javac (-Xlint:all) tool | `http://localhost:5000/mark_javac?sid=%{sid}` |

Access Tokens

| ID | Description | Token | Actions |
|---|---|---|---|
| 1 | **Demo Token** | `ubjy0Y80EWUinjiZdH9djZ9atjjRX3rAgiQkKg6pu/8=` | ✖ Revoke |
| 2 | **Demo Token** | `bRV+qP43AZhax545zP+YXd5ow+y1jK99cv62Bc+LUcQ=` | ✖ Revoke |

Figure 8.2: Screenshot demonstrating the admin panel for managing tools and access tokens



John Smith's Dashboard

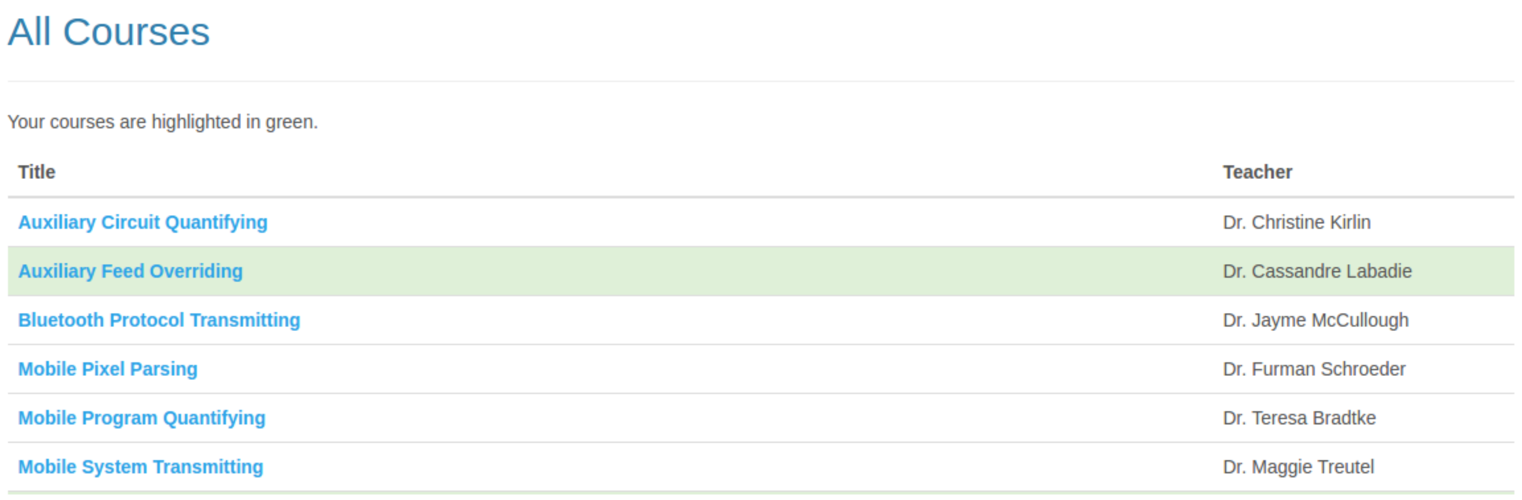
Recent Submissions

| Submitted | Assignment | Attempt | Mark |
|---|---|---|---|
| less than a minute ago | Hello, World! | #2 | 100% |
| 2 minutes ago | Hello, World! | #1 | Pending |

Current Assignments

| Title | Course | Deadline |
|---|---|---|
| Hello, World! | Programming Practice 1 | 7 days |
| Free Parking C | Neural Bus Parsing | 12 days |
| Stop Asking, Just Do B | Auxiliary Feed Overriding | 13 days |
| Ready, Set, Die B | Redundant Circuit Programming | 25 days |
| No Lonely Stars B | Virtual Matrix Transmitting | about 1 month |
| Fishing With Chips B | Neural Bus Parsing | about 1 month |
| Stop Asking, Just Do C | Redundant Circuit Programming | about 2 months |

Figure 8.3: Screenshot demonstrating the ‘Dashboard’ page

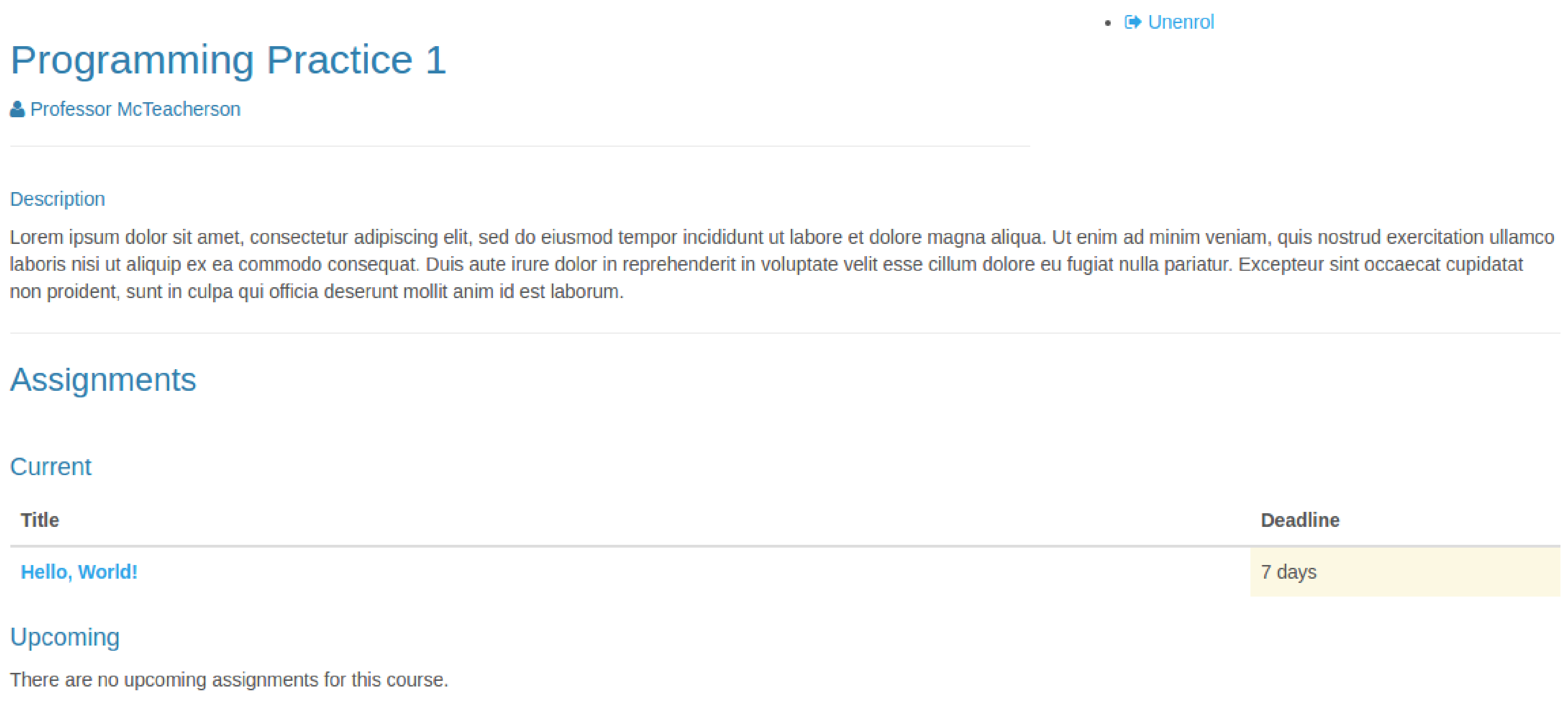


## All Courses

Your courses are highlighted in green.

| Title | Teacher |
|---|---|
| **Auxiliary Circuit Quantifying** | Dr. Christine Kirlin |
| **Auxiliary Feed Overriding** | Dr. Cassandre Labadie |
| **Bluetooth Protocol Transmitting** | Dr. Jayme McCullough |
| **Mobile Pixel Parsing** | Dr. Furman Schroeder |
| **Mobile Program Quantifying** | Dr. Teresa Bradtke |
| **Mobile System Transmitting** | Dr. Maggie Treutel |

Figure 8.4: Screenshot demonstrating the courses list



• Unenrol

Programming Practice 1

Professor McTeacherson

Description

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Assignments

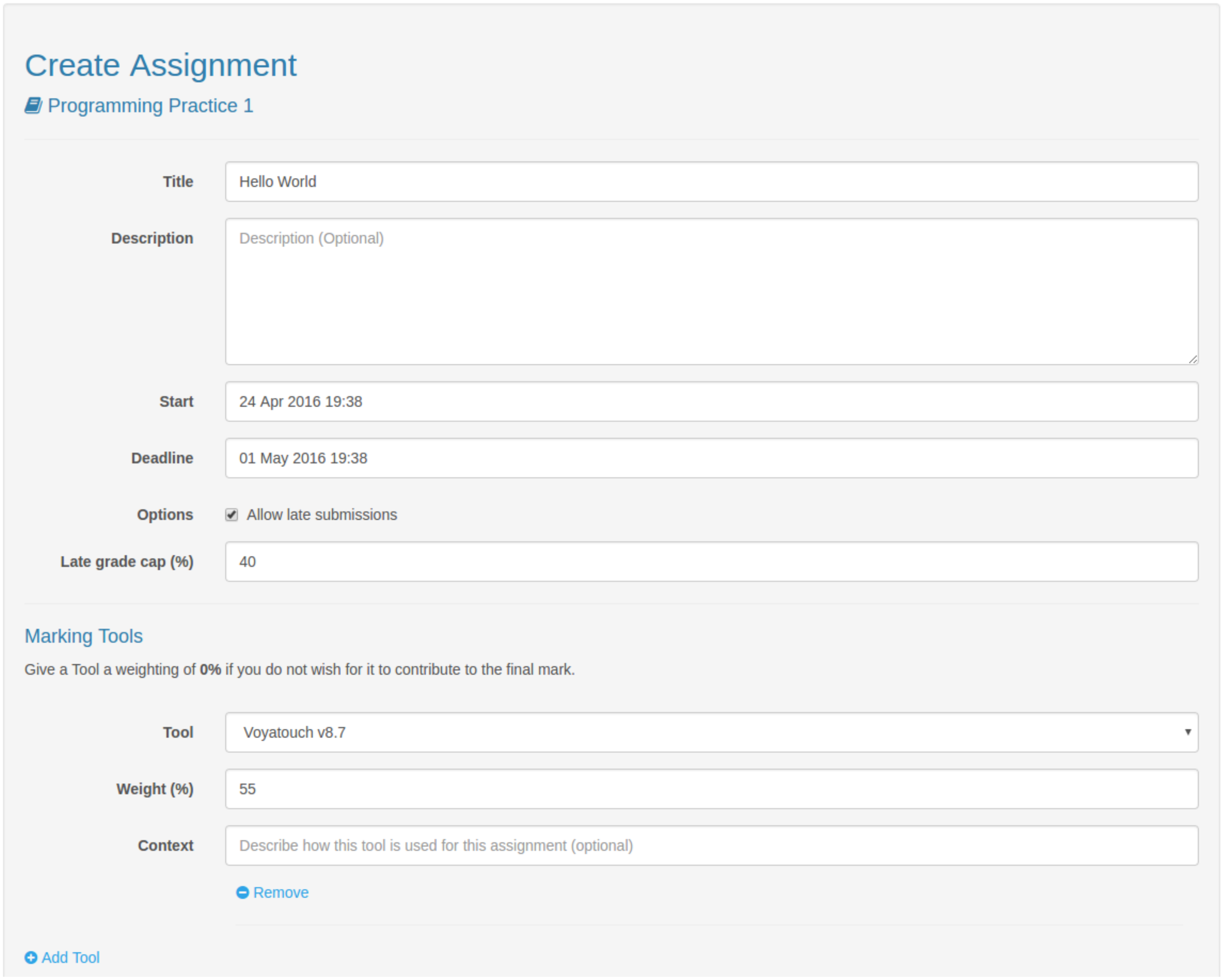
Current

| Title | Deadline |
|---|---|
| Hello, World! | 7 days |

Upcoming

There are no upcoming assignments for this course.

Figure 8.5: Screenshot demonstrating the course view



## Create Assignment

Programming Practice 1

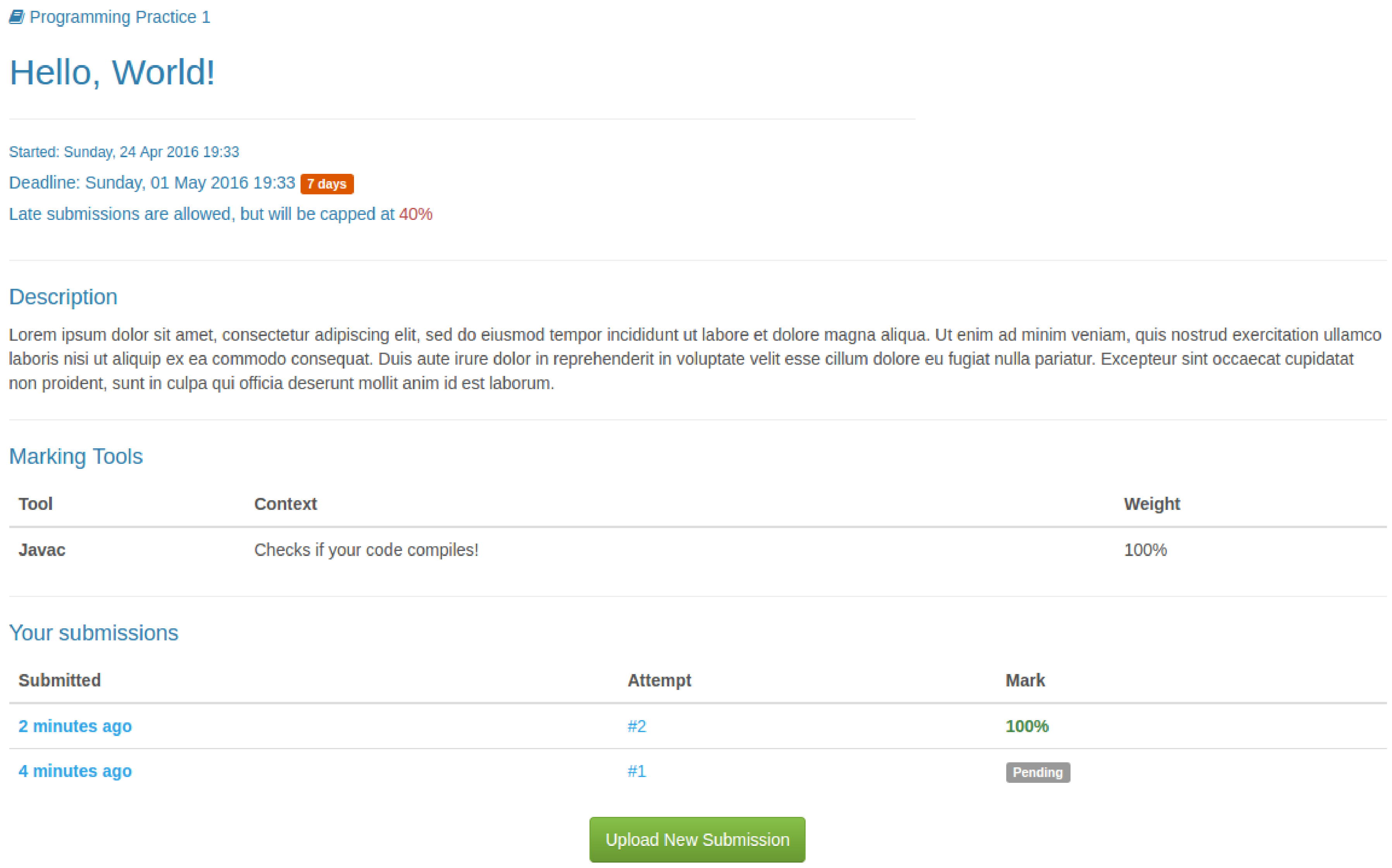
| | |
|---|---|
| **Title** | Hello World |
| **Description** | Description (Optional) |
| **Start** | 24 Apr 2016 19:38 |
| **Deadline** | 01 May 2016 19:38 |
| **Options** | ☑ Allow late submissions |
| **Late grade cap (%)** | 40 |

### Marking Tools

Give a Tool a weighting of **0%** if you do not wish for it to contribute to the final mark.

| | |
|---|---|
| **Tool** | Voyatouch v8.7 |
| **Weight (%)** | 55 |
| **Context** | Describe how this tool is used for this assignment (optional) |

Figure 8.6: Screenshot demonstrating the form for creating an assignment



Programming Practice 1

Hello, World!

Started: Sunday, 24 Apr 2016 19:33

Deadline: Sunday, 01 May 2016 19:33 7 days

Late submissions are allowed, but will be capped at 40%

Description

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

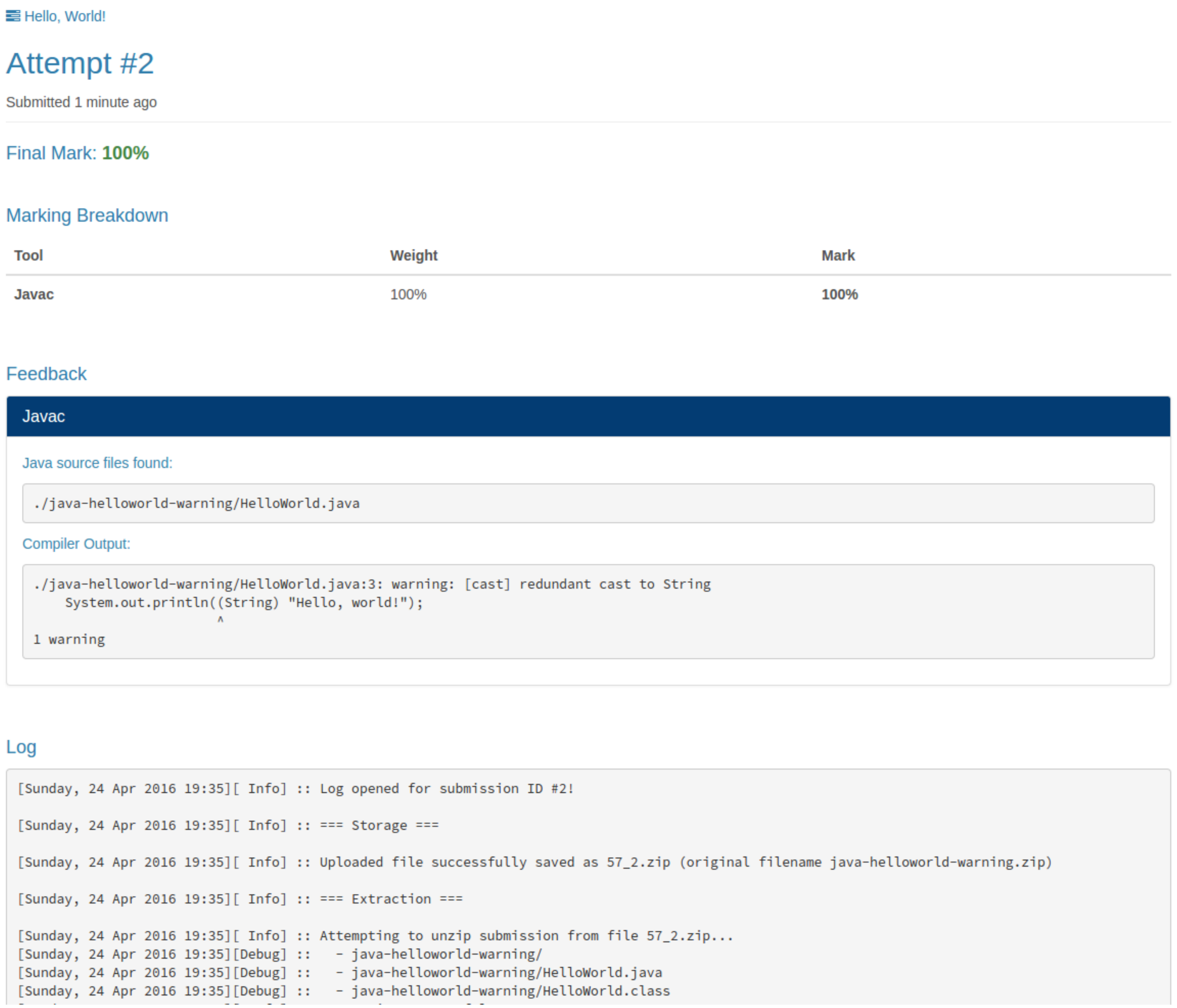
Marking Tools

| Tool | Context | Weight |
|---|---|---|
| **Javac** | Checks if your code compiles! | 100% |

Your submissions

| Submitted | Attempt | Mark |
|---|---|---|
| **2 minutes ago** | #2 | **100%** |
| **4 minutes ago** | #1 | Pending |

Figure 8.7: Screenshot demonstrating the assignment view



Hello, World!

Attempt #2

Submitted 1 minute ago

Final Mark: 100%

Marking Breakdown

| Tool | Weight | Mark |
|---|---|---|
| Javac | 100% | 100% |

Feedback

Javac

Java source files found:

```
./java-helloworld-warning/HelloWorld.java
```

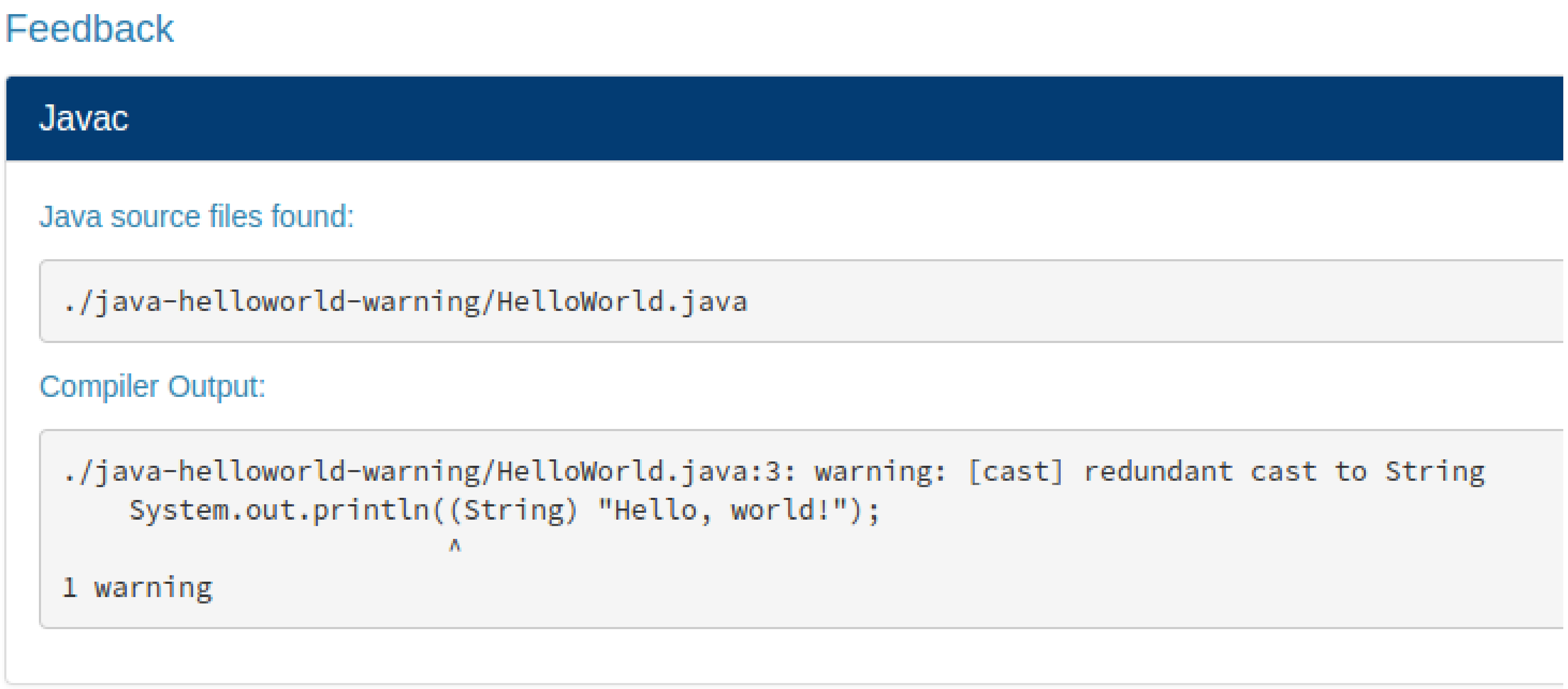
Compiler Output:

```
./java-helloworld-warning/HelloWorld.java:3: warning: [cast] redundant cast to String
    System.out.println((String) "Hello, world!");
                       ^
1 warning
```

Log

```
[Sunday, 24 Apr 2016 19:35][ Info] :: Log opened for submission ID #2!

[Sunday, 24 Apr 2016 19:35][ Info] :: === Storage ===

[Sunday, 24 Apr 2016 19:35][ Info] :: Uploaded file successfully saved as 57_2.zip (original filename java-helloworld-warning.zip)

[Sunday, 24 Apr 2016 19:35][ Info] :: === Extraction ===

[Sunday, 24 Apr 2016 19:35][ Info] :: Attempting to unzip submission from file 57_2.zip...
[Sunday, 24 Apr 2016 19:35][Debug] ::   - java-helloworld-warning/
[Sunday, 24 Apr 2016 19:35][Debug] ::   - java-helloworld-warning/HelloWorld.java
[Sunday, 24 Apr 2016 19:35][Debug] ::   - java-helloworld-warning/HelloWorld.class
```

Figure 8.8: Screenshot demonstrating the submission summary page



Feedback

Javac

Java source files found:

```
./java-helloworld-warning/HelloWorld.java
```

Compiler Output:

```
./java-helloworld-warning/HelloWorld.java:3: warning: [cast] redundant cast to String
    System.out.println((String) "Hello, world!");
                       ^
1 warning
```

Figure 8.9: Screenshot demonstrating feedback reported from the javac tool, in detail

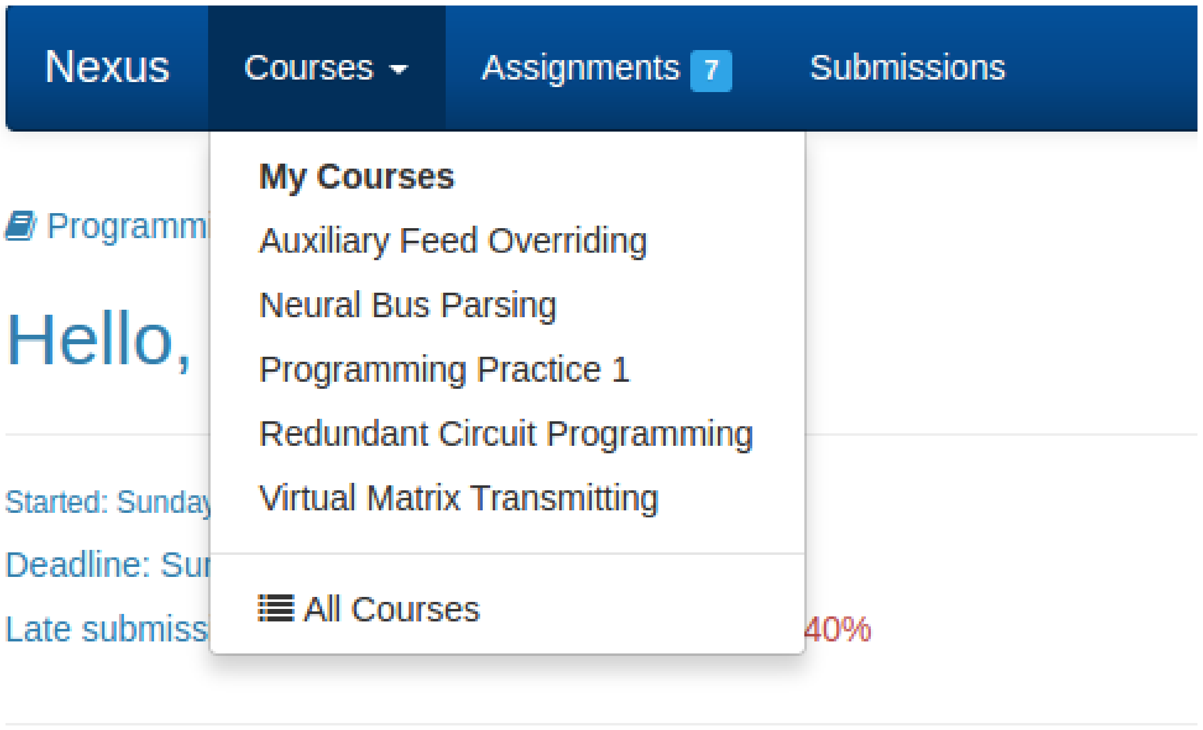


Figure 8.10: Screenshot demonstrating the global navigation

## 8.5 Unit test Examples

Listings 9 and 10 demonstrate some example of unit tests that have been written as part of the CMC. The code in Listing 9 confirms that the restrictions on data models are proplerly adhered to, and that instances will not be accepted if any of the restrictions (validations) are unsatisfied. Listing 10 checks the behaviour of a controller action for different types of user.

```ruby
RSpec.describe User, type: :model do
  describe 'model validations' do
    # ...
    it 'is invalid without an email' do
      expect(build(:student, email: nil)).not_to be_valid
    end
    # ...
    it 'has a level defaulting to "student"' do
      user = build(:student)
      expect(user.level).to eq('student')
    end
    # ...
    end
    it 'duplicate student IDs are invalid' do
      create(:student, student_id: '12312312')
      expect(build(:student, student_id: '12312312')).not_to be_valid
    end
  end
  # ...
end
```

Listing 9: Unit tests checking data model validations.

```
RSpec.describe CourseController, type: :controller do
  let(:s) { create(:student) }
  let(:t) { create(:staff) }
  describe 'GET #index' do
    it 'returns http success when logged in' do
      sign_in s
      get :index
      expect(response).to have_http_status(:success)
    end
  end
  # ...
  describe 'POST #create' do
    it 'returns http found when authorized and with valid parameters' do
      sign_in t
      post :create, course: attributes_for(:course)
      expect(response).to have_http_status(:found)
    end
    it 'raises error when logged in as a student' do
      sign_in s
      expect { post :create,
                course: attributes_for(:course) }.to raise_error(ActiveRecord::RecordInvalid)
    end
  end
end
```

Listing 10: Unit tests checking the behaviour of a controller.

A full set of unit tests have been written to check each validation on every data model; and for each controller action with a full set of different inputs (different user types, good data / corrupt data, etc.).

Unit tests are in particularly important for checking the behaviour of the Marking Interface and Feedback Interfaces. These must behave as expected, at all times, so that Tools are able to report their results correctly.

## 8.6 An example Tool

In addition to the CMC, an example tool has been developed. This can be used as a reference for other to develop tools.

It receives a submission ID, finds all the Java source code files within that submission, then runs the Java compiler `javac` on them. If the compiler succeeds, then a mark of 100 is reported. If any of the files fail to compile, a mark of 0 is reported.

The output of the compiler is reported as an item of feedback. Thus, any compiler warnings or errors will be visible to the student. The compiler is run with the `-Xlint:all` flag, instructing the compiler to be as verbose as possible with warnings.

The tool is written in **Node.js**, using the **Express** framework.

In total there are approximately 100 lines of code for the entire tool. This demonstrates how simple it can be to develop a basic tool.

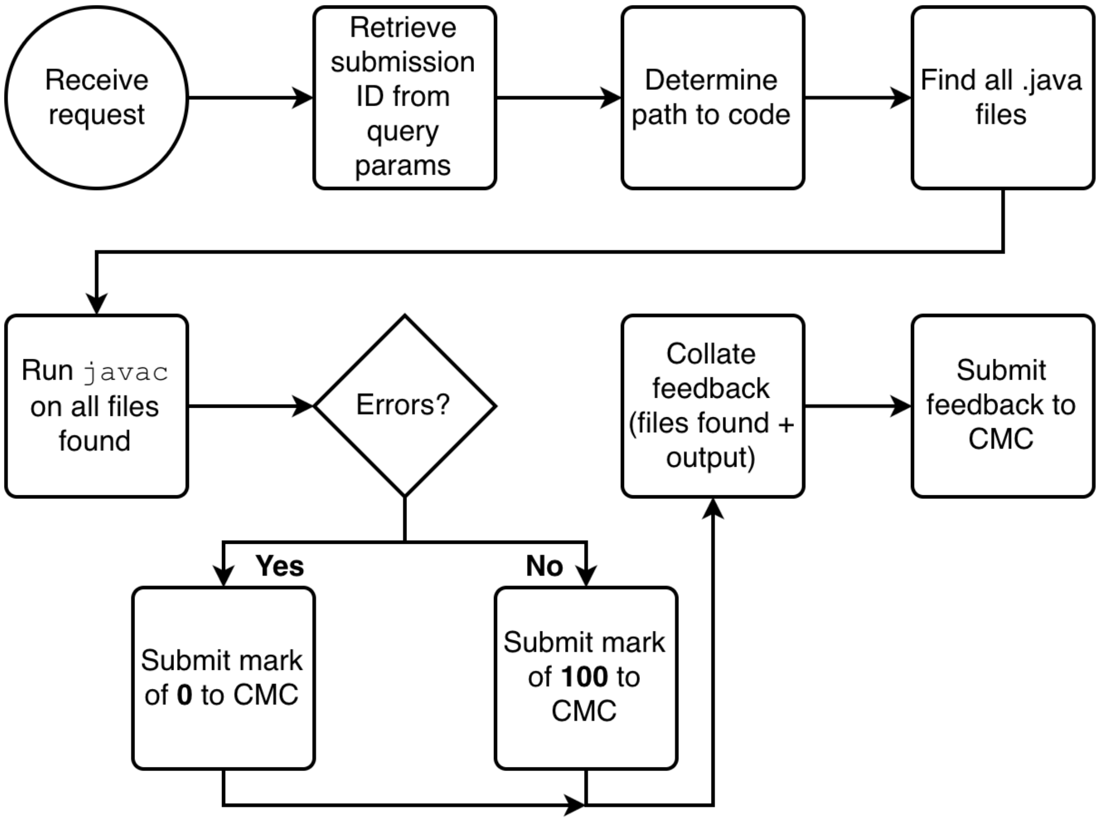

Receive request
Retrieve submission ID from query params
Determine path to code
Find all .java files
Run javac on all files found
Errors?
Yes
No
Submit mark of 0 to CMC
Submit mark of 100 to CMC
Collate feedback (files found + output)
Submit feedback to CMC

Figure 8.11: Flowchart with activities for javac tool

Figure 8.11 demonstrates the process this tool undertakes. Its various activites will now be discussed, along with appropriate code snippets:

### 8.6.1 Setup

The following environment variables are available to configure the tool:

| Name | Description |
|---|---|
| `PORT` | The port to listen for requests on |
| `SUBMISSIONS_DIRECTORY` | Path to the directory where submissions are kept |
| `NEXUS_ACCESS_TOKEN` | Access Token to use when sending mark/feedback to CMC |
| `NEXUS_BASE_URL` | Location of the CMC |
| `NEXUS_TOOL_CANONICAL_NAME` | Canonical name (UID) of the Tool as registered with the CMC |

Table 8.2: Environment variables for configuration of the javac Tool

For ease of use while developing, `PORT` defaults to `5000`; `NEXUS_BASE_URL` defaults to `http://localhost:3000`; and `NEXUS_TOOL_CANONICAL_NAME` defaults to `javac`.

#### Submissions directory

Note that it is assumed submissions will be available in some directory of the OS where the tool is being run. This is fine if both the tool and CMC are on running with the same OS/same filesystem; but if they are on separate systems then the directory will have to be mounted remotely through a utility such as NFS or SSHFS.

### 8.6.2 Listening for Requests

The tool listens for `POST` requests sent to the route `/mark_javac`, with a query parameter of `sid` (for the submission ID). Listing 11 demonstrates this.

```
// tools/javac/server.js
app.post('/mark_javac', (req, res) => {
  const submissionID = req.query.sid;
  console.log(`Request to mark submission ${submissionID} received.`);
  res.sendStatus(200);
```

Listing 11: Listening for a request to invoke the Tool.

For example, a POST request to the URL `http://localhost:5000/mark_javac?sid=17` would begin the compilation process for the Submission with ID 17.

### 8.6.3 Compilation process

Next, the directory to search for Java files is determined: this will be a join on the `SUBMISSIONS_DIRECTORY` environment variable and the submission ID. For example, if the submissions directory is `/mnt/submissions` and the submission ID is 513, the directory used for the searching will be `/mnt/submissions/513`.

Then, all `.java` files are found and listed in a file called `sources.txt`. This file is then passed to `javac`.

If the process exits without error (with an exit code of 0), then a mark of 100 is reported. If an error occured, then a mark of 0 is reported. If the call to `javac` takes more than 5 minutes (300,000ms), it will timeout and throw an error. In this case a mark and feedback will be reported as normal, and the feedback will specify that the error occured as the result of a timeout.

Finally, feedback is reported: the list of Java source files that were found, and the compiler output (including any error information if applicable).

Listing 12 demonstrates this process:

```
1  // tools/javac/server.js
2  const sourceDir = path.resolve(process.env.SUBMISSIONS_DIRECTORY, submissionID);
3
4  try {
5    const childFind = execSync(’find . -name "*.java" > sources.txt’, { cwd: sourceDir });
6    const childJavac = execSync(’javac -Xlint:all @sources.txt 2>&1’, { cwd: sourceDir,
7                                                                         timeout: 300000 });
8    // Success. Report 100 score
9    sendMark(100, submissionID, (err, res, body) => {
10     // ...
11   });
12 } catch (e) {
13   // Error. Report 0 score
14   sendMark(0, submissionID, (err, res, body) => {
15     // ...
16   });
17 } finally {
18   // Send output as feedback
19   sendFeedback(‘<div class="javac-feedback">${output}</div>‘,
20                submissionID,
21                (err, res, body) => {
22       // ...
23   });
24 }
```

Listing 12: Finding java files and running them against javac. (some parts removed for clarity)

### 8.6.4 Reporting a mark/feedback

As can be seen on line 19 of Listing 12, the value of the `output` variable (a String containing HTML) is what is returned as feedback. This variable is built up throughout the compilation process as demonstrated in Listing 13.

```
// tools/javac/server.js
let output = '';
// ...
const childCat = execSync('cat sources.txt', { cwd: sourceDir });
output += '<p class="text-info">Java source files found:</p>';
output += `<pre><code>${childCat.toString()}</code></pre>`;
// ...
const childJavac = execSync('javac -Xlint:all @sources.txt 2>&1', { cwd: sourceDir, timeout: 30000
output += '<p class="text-info">Compiler Output:</p>';
output += `<pre><code>${childJavac.toString()}</code></pre>`;
// ...
// in the case of an error 'e':
output += `<pre><code>${e.toString()}\\n${e.stdout.toString()}</code></pre>`;
```

Listing 13: Building up the HTML to return as feedback

The methods `sendMark` and `sendFeedback` are outlined in Listing 14. They construct a HTTP POST request to the relevant URL (constructed from the environment variables configuring the tool), then encode either the mark or feedback in JSON format, which Rails will be able to decode and process accordingly.

```
// tools/javac/utils.js
function sendRequest(body, url, callback) {
  const requestOptions = {
    url,
    method: 'POST',
    headers: {
      'Nexus-Access-Token': process.env.NEXUS_ACCESS_TOKEN
    },
    json: true,
    body
  };

  request(requestOptions, callback);
}

export function sendMark(n, submissionID, callback) {
  const url = `${NEXUS_BASE_URL}/report_mark/${submissionID}/${NEXUS_TOOL_CANONICAL_NAME}`;
  const body = { mark: n };

  sendRequest(body, url, callback);
}

export function sendFeedback(feedbackHTML, submissionID, callback) {
  const url = `${NEXUS_BASE_URL}/report_feedback/${submissionID}/${NEXUS_TOOL_CANONICAL_NAME}`;

  sendRequest({ body: feedbackHTML }, url, callback);
}
```

Listing 14: Sending the mark and feedback to the CMC

The screenshot (shown earlier) in Figure 8.9 demonstrates the feedback, rendered by the CMC on the Submission Summary page, for a submissions with a single file and a compiler warning.

Hopefully this tool serves as a good reference implementation for how tools can be built. It is designed to demonstrate the simplicity of generating a mark and feedback and sending it back to the CMC.

The process that is undertaken to generate the mark/feedback is also representative of how most Tools will operate:

- Find source files for Submission
- Run a series of commands/utilities/tools with a set of parameters
- Analyse the output of these tools in order to generate a mark and/or feedback
- Format the feedback in HTML
- Report the mark and/or feedback to the CMC

## 8.7 Security

There are several different attack vectors for web applications, each of which needed to be considered as the CMC was developed. There are also more general security issues such as the storage of passwords which needed to be addressed. A variety of security issues and action taken to mitigate them are outlined below.

**Password Storage**

All passwords are stored in the database as hashes, with salts attached; similar to how the UNIX `passwd` file is stored. This way, if the database is breached, it would not be trivial to recover user's passwords.

When passwords are transmitted (for example, when registering or logging in), a digest is sent instead of the plaintext password, protecting against Man-in-the-Middle attacks. Although, when deployed in a production setting, HTTPS should always be used.

**Cross-Site Request Forgery (CSRF) Attacks**

All POST requests on the system require either a CSRF token or valid Access Token in order to be accepted. POST requests that use forms from the web-based UI require a CSRF token;

and POST requests to API endpoints (e.g. to report some feedback) require a valid Access Token.

In the case of the CSRF token, Rails automatically generates this token when a web form is rendered (e.g. the form to create a new Course). Then, when this POST request is actually received, Rails checks to ensure the CSRF token is valid.

### Cross-Site Scripting (XSS) Attacks

The only place in the CMC where user input is rendered *without* being sanitised is when feedback items are displayed on the Submission Summary page. This is intentional, to allow Tools to produce rich and highly customisable output.

However, it does open up the possibilty for Cross-Site Scripting attacks, if a Feedback Item contains some malicious JavaScript as part of its HTML body.

Therefore, we have to rely on being able to trust that the Feedback Tools return is not to be used maliciously. This should not be an issue as long as the source code for a Tool is readily available to be reviewed by a trusted party.

This leaves one possible attack vector, though: that when request to send some feedback is sent from a Tool to the CMC, it is intercepted, and the interceptor modifies the `body` to contain the malicious code. This is an example of a Man-in-the-Middle attack. It can be mitigated by ensuring HTTPS (with strong encryption parameters[22]) is used for all requests sent to the CMC.

### Authorisation

Every request to a controller that requires escaled privileges ensures that these privileges are met as the first action taken, and if not the request is immediately responded to with an error.

For example, a method has been created to ensure that the logged in user is at least a member of staff (i.e. teaching staff or an administrator) - this method is called at at the beginning of many requests such as creating a course or assignment. If the logged in user is not a member of

[22]for example: 128-bit AES in Galois Counter Mode, with Elliptical Curve Ephemeral Diffie-Hellman used as the key exchange mechanism

staff, a `HTTP 401 Unauthorized` is the response sent, and the controller immideately `return`s - terminating any further processing.

Other similar methods have been created to authenticate that a user is logged in at all, to authenticate that the signed in user is a student, an administrator; and a method to check that an access token is valid is given earlier in Listing 8.

**Security of Tools**

As microservices, it is the responsibility of individual tools to implement security correctly. If the tools provide a UI for their configuration, then this will need to be properly secured. If tools wish to use access tokens (or similar) to authenticate that requests to their Dispatch Interface is indeed coming from the CMC, then they will need to provide a way for these tokens to be sent from the CMC as a URL paramater, similar to the way a submission ID is attached.

Regardless, HTTPS should certainly be used for communication to all tools. A reasonable argument could be formed to enforce the use of HTTPS when the system is used in a production environment.

## 8.8 Deployment

Docker[23] allows software to be wrapped up in to containers. A Docker container is essentially a filesystem containing everything an application needs to run: the code, the runtimes, any relevant system tools and libraries, and anything else that is required. This offers the assurance that the application will always be run on the same environment.

They differ from Virtual Machines because containers do not contain a full Guest OS - resources such as the kernel can be shared between containers. This makes Docker containers more lightweight than regular Virtual Machines, and helps with scaling.

Docker configurations for both the CMC and the example Tool have been created.

These allow the CMC and the example Tool to easily be deployed on any Linux host machine, or deployed to the cloud on a platform such as Amazon AWS EC2[24].

[23]https://www.docker.com/
[24]https://aws.amazon.com/ec2/

Deploying to the cloud would allow the system to be automatically scaled horizontally when there is increased demand (for example, near the deadline for a large Assignment). Other benefits of deploying to the cloud include more easily being able to adopt Continous Deployment, and potentially lower operating costs.

Another advantage to cloud deployment would be the ability to have all Tools and the CMC on the same Virtual Private Cloud (VPC). The CMC along with all the Tools would exist on a logically isolated network, and a strong security policy would only allow certain external requests through to the CMC. If requests to report marks and feedback were only accepted from hosts inside the VPC, then the security of the system as a whole would be strengthened (see discussion on XSS in the previous section).

### 8.8.1 System Requirements

As the CMC and example Tool can both be deployed through Docker containers, effectively the only technical System Requirement is:

- A Linux host machine,
- with Docker installed.

However, it is recommended that the system is run on a machine with a *minimum* of a modern 4-core processor; 8GB of RAM; and a solid-state drive.

# Chapter 9

# Results & Evaluation

This chapter aims to critically evaluate the system that has been developed, with respect to the requirements and scope that were determined. This will be achieved by analysing the strengths, weaknesses and limitations of the final system.

## 9.1 Strengths

This project has created an architecture and system which facilitates the management of automated assessment and feedback for programming assignments. This system has been created with flexibility and ease of expansion at the core. The latest technologies have been used to ensure that implementation will remain relevant for as long as possible; and sound development practices have provided high-quality, well-structured, stable code.

Developing using the Test-Driven Development (TDD) methodology, combined with a very high level of test coverage, has allowed us to have great confidence in the stability of the system produced.

The system developed can be used to manage the relevant organisational entities, namely Users, Courses, Assignments and Submissions; and an example tool which has been developed can be used as a reference to create much more powerful and ambitious ways to mark and provide feedback on assignments.

The system was always created with the notion of being deployable in a production environment, ready for new tools to be developed and integrated with the CMC. The system would be perfectly capable, in its current form, of being deployed at scale. Therefore it can be said with some confidence that this goal has been met. Some changes may need to be made for the system to properly fit in to existing assessment and feedback workflows that exist where it is deployed, however.

The way in which the system has been architectured - with a central component, and tools as microservices interacting with the central component via HTTP calls - has allowed the goals of flexibility and easy expansion to be met. A huge variety of programming langauges and frameworks can be chosen from in order to develop new Tools: in theory support for listening on and sending HTTP requests are the only two requirements.

Ease-of-use for students and teaching staff was just as important an aim as ease-of-use for tool developers (who are likely to also be teaching staff). The use of the web to provide the UI, with the help of a well-established CSS library, delivers a sleek and functional experience for users - students and staff alike. The management of courses, assignments, and submissions is intuitive and easy to learn.

Being able to develop tools and attach them to a central component which facilitiates the management of users, courses, assignments, etc. is not something that has been developed in this way before.

It does not take a great amount of thought to imagine how automation could aid with the process of assessment. Nevertheless, during the development of this project it was somewhat surprising to realise just *how* much could be automated. As was demonstrated in the Case Study, virtually every marking criterion could be automated by a tool as part of the system developed. I believe that the simplicity and relatively small size of the example tool also demonstrates that the time spent developing automated solutions would be worth the investment compared to marking students' work manually.

Looking back through the full Requirements List presented earlier, the system which has been developed meets every requirement listed. It is interesting to note that the scoping requirement of assignments and submissions only being written in Java was not actually enforced anywhere in the system. The system does not feature any way to enforce what language can and cannot be used. However it also does not provide any way to differentiate between different languages, or let tools know what language is being used, so operationally the system would have to work with a single language (not necessarily Java, though).

## 9.2 Weaknesses and Limitations

When creating Assignments, some way to integrate the configuration of individual tools (which require Assignment-specific configuration) as part of the User Interface of the CMC would greatly improve ease-of-use. This way, the user does not have to use several separate systems to ensure an Assignment is fully configured. A way to ensure that every tool used is also properly configured before an Assignment can be created - as part of the validation process, perhaps - would provide greater confidence. As microservices, tools should provide their own GUI for their configuration, but there is no reason that this GUI - if also web-based - could not be integrated with the assignment creation process in the CMC.

Although a key goal of the project was to create a system that could be deployed in a real institution such as a university, the developed solution offers very little in terms of integration with existing systems a university may have, such as VLEs or Student Records systems. It is likely that in many cases teaching staff will have to duplicate information such as course descriptions and assignments. This is an unfortunate duplication of effort which goes against the mantra of the system being used to save staff time.

Only having had one example tool developed is also somewhat disappointing. Although it demonstrates how the various interfaces work, and the general process for generating a mark and feedback, it barely scratches the surface for demonstrating just how powerful automation can be for programming assignments. It would definitely have been valuable to have more example tools, perhaps demonstrating the use of static analyers or unit test suites to run against students' code. It is discussed how tools are microservices and as such it is their responsibility to tailor their marking and/or feedback generation to paramters such as the specific assignment or student that made a submission; while this is all true, an example tool demonstrating this functionality would also have been of some value.

## 9.3 Conclusions

Overall, the system developed and the architecture behind it lay a solid foundation for the automation of assessment and feedback processes for programming assignments. The architecture is sound, sensible, and allows for a great deal of further expansion if required.

It is hoped that the example tool, along with some of the discussions in this report, are able to demonstrate the vast potential and power behind the notion of automating assessment and feedback; in addition to the secondary benefit this would provide of freeing up the time for teaching staff to focus more on students' needs, rather than manually marking dozens if not hundreds of submissions individually.

Our hope is that the system created could be developed and expanded further, to the point where eventually it is utilised - on a large scale - to its full potential.

In recent years, automation powered by technologies such as the Personal Computer and the Internet has greatly transformed the world around us. A great deal is also owed to the programming used to power such technologies. Hopefully the work produced by this project will help to introduce automation in the field of teaching programming, so that the next generations of students can utilise their skills to continue changing the world.

# Chapter 10

# Future Work

There are many ways in which the work completed in this project could be extended, to add features, integrations or improve ease-of-use. There is also room for more general work in the area of automating assessment and feedback. In this chapter, several ideas from both categories will be presented.

## 10.1 Potential Extensions

There are several ways in which the system developed as part of this project could be extended. These include:

### 10.1.1 More Tools

There is, of course, a great deal of potential for creating intricate and complex Tools to assess and provide feedback on students' work. Some ideas for Tools which could be developed are:

- Running students' code against Unit Testing frameworks. Unit tests would differ per assignment and be written by teaching staff to form a marking scheme. The results on which tests failed or passed (and hints on how to fix a failed test) could be sent as feedback.
- Running code through linting tools and other static analysers to identify code smells and

other ways code could be improved. Once identified this could be returned as feedback to the student.

- Performance analysis tools could be used, running code serveral times, in order to determine the efficiency of the code written. This could be used for marking or solely for feedback purposes.
- Plaigarism checkers could also be integrated as tools, as part of the regulatory process of submitting work that will count towards a formal award such as a degree.

### 10.1.2 Support for More Languages

As previously discussed, the system in its current form is agnostic to which programming langauge is used, as long as *only one* is used for all submissions and tools (as there is currently no way for the system to differentiate between what assignments use which languages; or which tools support what languages).

Expanding the system to support multiple languages would allow the system to be deployed in institutions where more than one programming language is used throughout teaching.

### 10.1.3 Integration with VCS

Another extension could be integrating the system with a Version Control System (VCS) such as *Git* or *Mercurial*. Students would upload their code to a remote repository, and then submissions would simply take the latest code - or a specific commit - from that repository. Tools would also be able to pull students' code from the repositories, which would eliminate the need for network mounting a remote submissions store. Deeper integration could involve these remote repositories automatically being set up for students when an assignment starts.

### 10.1.4 Opening up the API

Opening up the API to allow the access, creation, modification, and removal of resources (users, courses, assignments, and submissions) - in addition to marks and feedback - would allow integrations to be developed with a huge range of existing or newly developed systems.

The use of the Rails framework - which is well suited to building web-based APIs - means that this extension would not require any significant changes to the codebase.

**Integrations with Other Systems**

Examples of systems which could interact with the API include:

- Virtual Learning Environments (VLEs), such as Moodle
- Student/Staff Record Systems
- Course Record Systems (used to manage enrolment in different courses/modules available at an education institution)
- Web-based VCS systems which offer the ability for external integrations, for example GitHub

## 10.2 Areas for Further Research

Topics in the general areas of assessment and feedback for programming tasks, which could be further researched in future (and could theoretically be integrated as extensions to the architecture designed in this project), include:

### 10.2.1 Automating the Creation of Test Code

Automating the creation of unit test code is one area which could be further researched. If the time taken to write sufficiently powerful unit tests is not siginificantly less than the time it would take to mark a set of students' submissions manually, the case for automating assessment becomes much less convincing. Being able to automate the creation of test code, given some input parameters, would be highly valuable in the context of education.

### 10.2.2 Dynamic Assignment Generation

Being able to dynamically generate assignments based on certain parameters would be a powerful capability for use in education. There is much evidence to suggest that repetition of tasks is an effective way to learn something new [7].

It could also be used to generate slightly different assignments for each student, decreasing the potential for plagiarism and encouraging independent learning.

### 10.2.3 Automated Feedback Through Online Resources

Students code is checked for compiler warnings, errors, and runtime errors. If any exist, these specifics of each warning/error are used as search terms in one or more online knowledge bases; examples could include Stack Overflow[1] and the official documentation for the programming langauge used. The top results for resolving each warning/error are then parsed, and returned as feedback to be viewed by the student. This would be especially useful for new students who are not yet used to independendly searching for help with errors themselves.

### 10.2.4 Peer-to-Peer Assessment and Feedback

Peer assessment can be a powerful tool in education settings [10]. In addition, programming is a naturally collaborative process. A system enabling peer-to-peer assessment or feedback for programming tasks could be a benefit to all students: those more capable would be able to share their knowledge, whereas those less capable would be able to get help from others on a similar, hopefully approachable, level.

[1]http://stackoverflow.com/

# References

[1] Charles W Bachman. Data structure diagrams. *ACM Sigmis Database*, 1(2):4–10, 1969.

[2] Kent Beck. *Extreme programming explained: embrace change.* addison-wesley professional, 2000.

[3] Kent Beck. *Test-driven development: by example.* Addison-Wesley Professional, 2003.

[4] John B Biggs. *Teaching for quality learning at university: What the student does.* McGraw-Hill Education (UK), 2011.

[5] Michael Blumenstein, Steve Green, Ann Nguyen, and Vallipuram Muthukkumarasamy. Game: A generic automated marking environment for programming assessment. In *Information Technology: Coding and Computing, 2004. Proceedings. ITCC 2004. International Conference on*, volume 1, pages 212–216. IEEE, 2004.

[6] Connextra. A connextra story card. `http://agilecoach.typepad.com/photos/connextra_user_story_2001/connextrastorycard.html`, 2001. [Online; accessed 10-Dec-2015].

[7] Guy Cook. Repetition and learning by heart: An aspect of intimate discourse, and its implications. *ELT journal*, 48(2):133–141, 1994.

[8] Stephen H Edwards. Using test-driven development in the classroom: Providing students with automatic, concrete feedback on performance. In *Proceedings of the International Conference on Education and Information Systems: Technologies and Applications EISTA*, volume 3, 2003.

[9] Stephen H Edwards and Manuel A Perez-Quinones. Web-cat: automatically grading programming assignments. In *ACM SIGCSE Bulletin*, volume 40, pages 328–328. ACM, 2008.

[10] Nancy Falchikov. Peer feedback marking: developing peer assessment. *Programmed Learning*, 32(2):175–187, 1995.

[11] Jason Farrell and George S Nezlek. Rich internet applications the next stage of application development. In *Information Technology Interfaces, 2007. ITI 2007. 29th International Conference on*, pages 413–418. IEEE, 2007.

[12] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures.* PhD thesis, University of California, Irvine, 2000.

[13] Martin Fowler and James Lewis. Microservices. *Viittattu*, 28:2015, 2014.

[14] Mikell P Groover. *Automation, production systems, and computer-integrated manufacturing.* Prentice Hall Press, 2007.

[15] Surendra Gupta and Shiv Kumar Dubey. Automatic assessment of programming assignment. *Computer Science & Engineering*, 2(1):67, 2012.

[16] Michael Hartl. *Ruby on rails tutorial: learn Web development with rails.* Addison-Wesley Professional, 2015.

[17] Richard Higgins, Peter Hartley, and Alan Skelton. The conscientious consumer: reconsidering the role of assessment feedback in student learning. *Studies in higher education*, 27(1):53–64, 2002.

[18] Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppala. Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, pages 86–93. ACM, 2010.

[19] David Insa and Josep Silva. Semi-automatic assessment of unrestrained java code. 2015.

[20] Avraham Leff and James T Rayfield. Web-application development using the model/view/-controller design pattern. In *Enterprise Distributed Object Computing Conference, 2001. EDOC'01. Proceedings. Fifth IEEE International*, pages 118–127. IEEE, 2001.

[21] J Lindsay. Web hooks to revolutionize the web, 2014.

[22] Philip-Daniel Beck Lukas Ifflander, Alexander Dallmann and Marianus Ifland. Pabs-a programming assignment feedback system. `http://ceur-ws.org/Vol-1496/paper5.pdf`, 2015. [Online; accessed 10-Nov-2015].

[23] Lauri Malmi, Ville Karavirta, Ari Korhonen, and Jussi Nikander. Experiences on automatically assessed algorithm simulation exercises with different resubmission policies. *Journal on Educational Resources in Computing (JERIC)*, 5(3):7, 2005.

[24] Raina Mason, Graham Cooper, and Michael de Raadt. Trends in introductory programming courses in australian universities: Languages, environments and pedagogy. In *Proceedings of the Fourteenth Australasian Computing Education Conference - Volume 123*, ACE '12, pages 33–42, Darlinghurst, Australia, Australia, 2012. Australian Computer Society, Inc.

[25] Yasuhiro Monden. *Toyota production system: practical approach to production management*. Engineering & Management Press, 1983.

[26] Julie L Montgomery and Wendy Baker. Teacher-written feedback: Student perceptions, teacher self-assessment, and actual teacher performance. *Journal of Second Language Writing*, 16(2):82–99, 2007.

[27] Vreda Pieterse. Automated assessment of programming assignments. In *Proceedings of the 3rd Computer Science Education Research Conference on Computer Science Education Research*, pages 45–56. Open Universiteit, Heerlen, 2013.

[28] Ma Mercedes T Rodrigo, Ryan S Baker, Matthew C Jadud, Anna Christine M Amarra, Thomas Dy, Maria Beatriz V Espejo-Lahoz, Sheryl Ann L Lim, Sheila AMS Pascua, Jessica O Sugay, and Emily S Tabanao. Affective and behavioral predictors of novice programmer achievement. *ACM SIGCSE Bulletin*, 41(3):156–160, 2009.

[29] Robert Michael Siegfried, Daniel Greco, Nicholas Miceli, and Jason Siegfried. Whatever happened to richard reid's list of first programming languages? *Information Systems Education Journal*, 10(4):24, 2012.

[30] Matthew Thornton, Stephen H Edwards, Roy P Tan, and Manuel A Perez-Quinones. Supporting student-written tests of gui programs. *ACM SIGCSE Bulletin*, 40(1):537–541, 2008.

[31] Steve Willens, Allan C Rubens, Carl Rigney, and William Allen Simpson. Remote authentication dial in user service (radius). 2000.