COMS3002: Software Engineering III

AVI SYSTEM

08 October 2018

Contents

Abstract

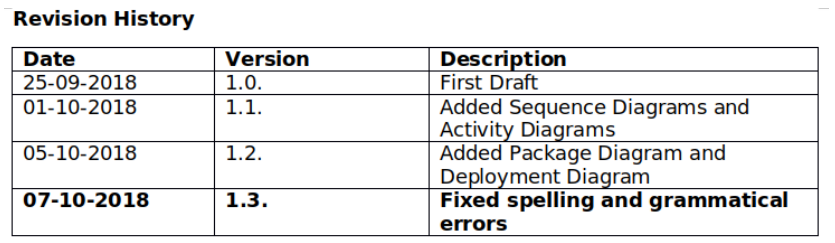
It is becoming more and more difficult, particularly for students entering graduate schools, to make decisions on courses that subsequently impact on their successful completion of graduate studies. Often, students have to choose from a number of electives in their specific programmes. Selecting wrong courses means mismatch between students aptitude, capability and personal interest. Thus our project idea is to develop a system for helping the students to choose elective courses as well as to predict grades which would be best suited for them based on their previous courses and grades. To attain this, we will use some of the machine learning techniques which will combine all the features and output all the recommended courses as well as predicted grades for those recommended courses.

1 Introduction

The purpose of this document is to detail the requirements for the Avi system to be developed. It will describe the features, functions and interfaces of the system. This document is intended for developers of the system as this document will serve as a guideline for the development of the Avi system. It is also intended for the client(The University of Witwatersrand) and serves as an indication of what the client should expect from the developed system. The approval of this document will be interpreted as permission to go ahead with the development of the system.

Avi is a web application that will recommend elective courses to prospective postgraduate students and further predict the grades the students are likely to get in the recommended courses based on their grades in previously completed courses. A student will start by creating an account which includes specifying their firstname, lastname, student number,current level of study and their login password. Once a student has created an account they can only be able to get recommendations once they have added the courses they have done and the grade they obtained in each course. This information can be updated by the student for cases where they may have made a mistake or their mark has changed. The student can now request to get recommendations. After requesting to get a recommendation, the system will present to them the recommended courses along with the predicted grade for each course.

2 Avi System Software Design and Architecture



Revision History

Date	Version	Description
25-09-2018	1.0.	First Draft
01-10-2018	1.1.	Added Sequence Diagrams and Activity Diagrams
05-10-2018	1.2.	Added Package Diagram and Deployment Diagram
07-10-2018	**1.3.**	**Fixed spelling and grammatical errors**

2.0.1 Purpose

The main purpose of this section is to provide the design and architectural overview of the Avi System. It presents the architectural decisions made in designing and developing the system. This document can be used by the project stakeholders (developers and client) to better understand the problem being solved and how the Avi System will represent the solution.

2.0.2 Scope

Avi is a web application that will recommend elective courses to prospective postgraduate students and further predict the grades the students are likely to get in the recommended courses based on their grades in previously completed courses.

A student will start by creating an account which includes specifying their first name, surname, student number,current level of study and their login password.

Once a student has created an account they can only be able to get recommendations once they have added the courses they have done and the grade they obtained in the courses. This information can be updated by the student for cases where they may have made a mistake or their mark has changed.

The student can now request to get recommendations. After this the system will present to them the recommended courses along with the predicted grade for each course.

2.0.3 Definitions, Acronyms and Abbreviations

PC: Personal computer

Client: University of Witwatersrand

2.0.4 References

- Source: https://en.wikipedia.org/wiki/4
- Source: https://djangobook.com/model-view-controller-design-pattern/
- Source: http://www.ece.uvic.ca/ itraore/seng422-05/notes/arch05-5.pdf

- The System Requirements Specification Document

2.0.5 Overview

The next section, describes the goals and constraints of designing the systems architecture. Section 2.2 describes the architectural representation of the system. Section 3 describes the 5 different ways in which the system architecture can be viewed. Section 3.6 describes the systems size and performance and Section 3.7 talks about the systems quality concerns.

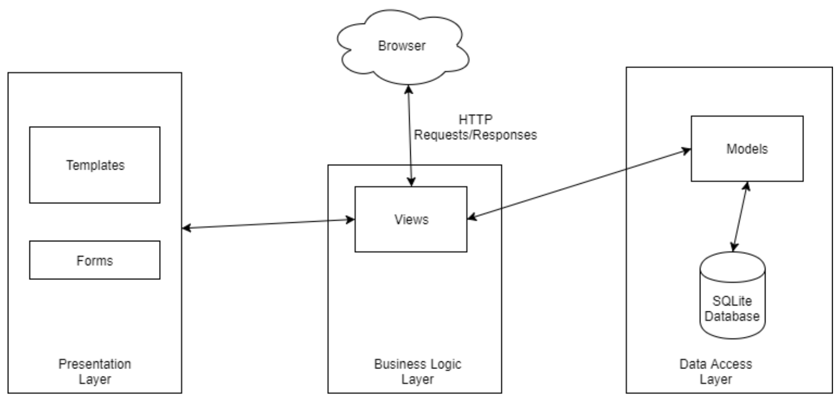
2.1 Architectural Goals and Constraints

The architecture design of the Avi System was influenced by the requirements specified in the System Requirements Specification document and it was constrained by the Django framework architecture.

2.2 Architectural Representation

2.2.1 Architectural Design Pattern and Architectural Style

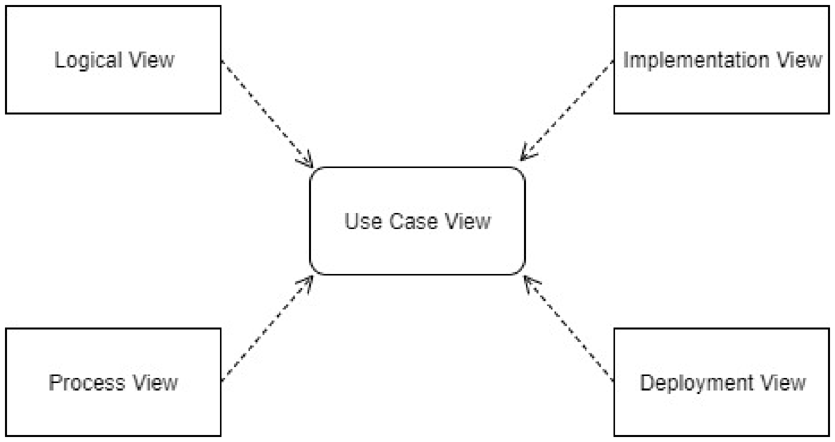
The Avi System follows the Model-Template-View (MTV) design pattern. The model is the data access layer, it is a representation of the database and contains everything about how to access the data. The template is the presentation layer, it is what the user sees and interacts with. The view is the business logic layer, it controls the flow of information between the models and templates and is responsible for any processing.



Avi System: Arichitecture Diagram

2.2.2 Architectural Views

This section describes the system from different perspectives of the project stakeholders. The image below illustrates the views considered.



Architectural Views

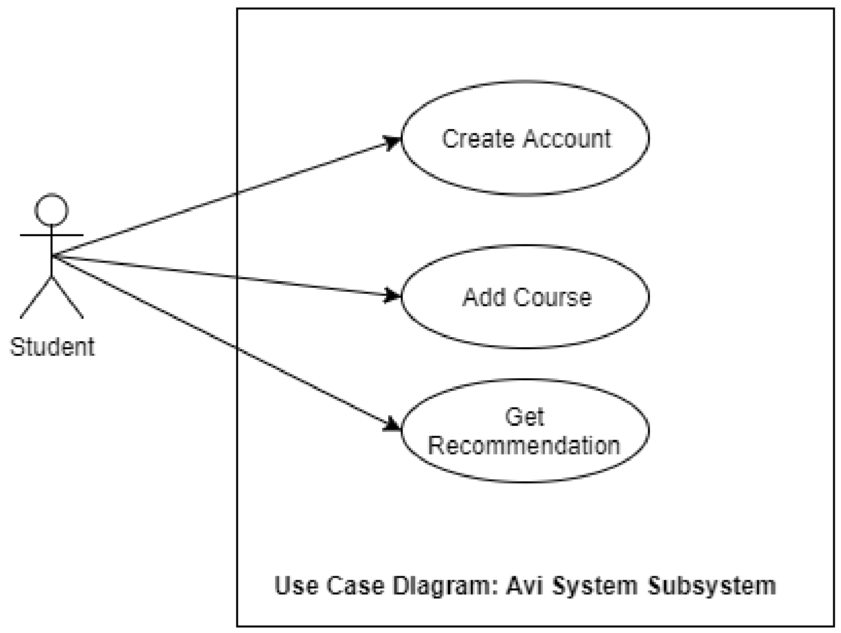
3 Architectural View Decomposition

3.1 Use Case View

The use case view depicts the system from the users perspective. In this section we have focused on three use cases which we believe are crucial to the running of the system, these use cases are Create Account, Add Course and Get Recommendation.

3.1.1 Use Case Diagram

This Use Case Diagram is a subset of the final Use Case Diagram found in the System Requirements Specification document.



3.1.2 Use Case Descriptions

Create Account

Main Success Scenario:

The use case starts when the user requests to create an account. The system will prompt the user to enter their details (i.e. their first name, surname, student number, current level of study and their login password). The user will enter these details and commit the details by pressing the Create Account button. The system will check that the password and confirmation of password match. If they match, the system will create the account and the user will be redirected to the login page.

Alternative Scenario:

S1: If the password and confirmation of password do not match the use case will be restarted.

Add Course

Main Success Scenario:

The use case starts when the user requests to add a course. The system will prompt for a course code and the grade obtained in the course. The user will enter these details and commit the details by pressing the Add Course button. The system will validate (i.e. Check if the course code exists and if the entered grade is a number between 0 and 100) the details and if they are valid it will then add the course and the added course will appear under the Your Courses section of the page.

Alternative Scenario:
S1: If the details are invalid the use case will be restarted.

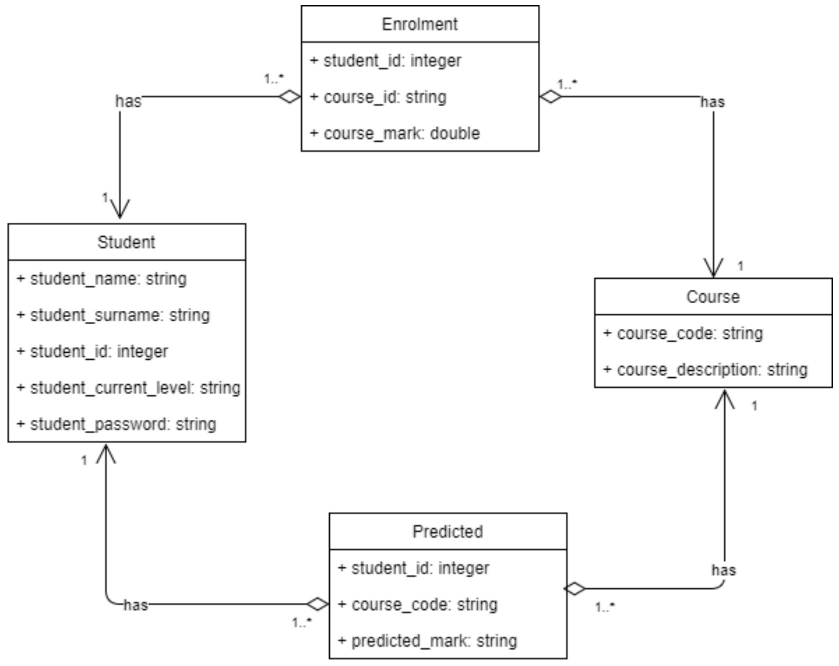
Get Recommendation

Main Success Scenario:
The use case is started when the user requests to get a recommendations. The system will check if the user has added courses to their profile. If they have, the system will generate the recommendations and they will appear under the Your Recommendations section of the page.

Alternative Scenario:
S1: If the user does not have any courses added to their profile, the use case will halt and the user will be advised by the system to add their courses before requesting a recommendation.

3.2 Logical View

This view is concerned with depicting the functionality provided to the users by the system. This is done using class and state diagrams.



Class Diagram

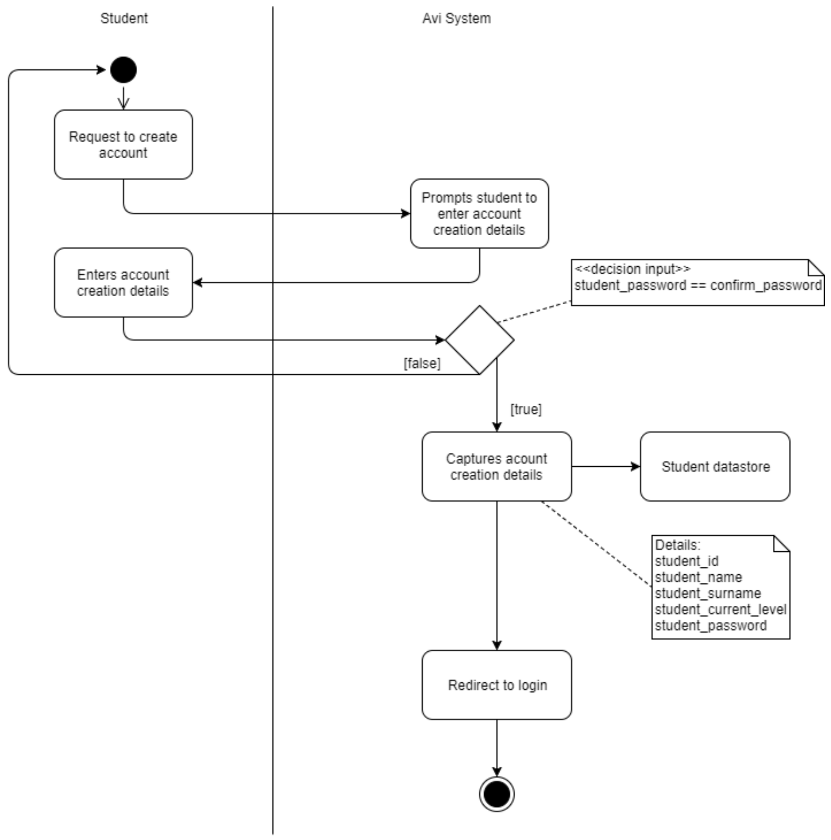
Note: The AVI System does not contain any state changes. Thus there is no state diagram.

3.3 Process View

This view demonstrates the processes that form part of the system and how they interact with each other. It focuses largely on concurrency and the sequence of processes.

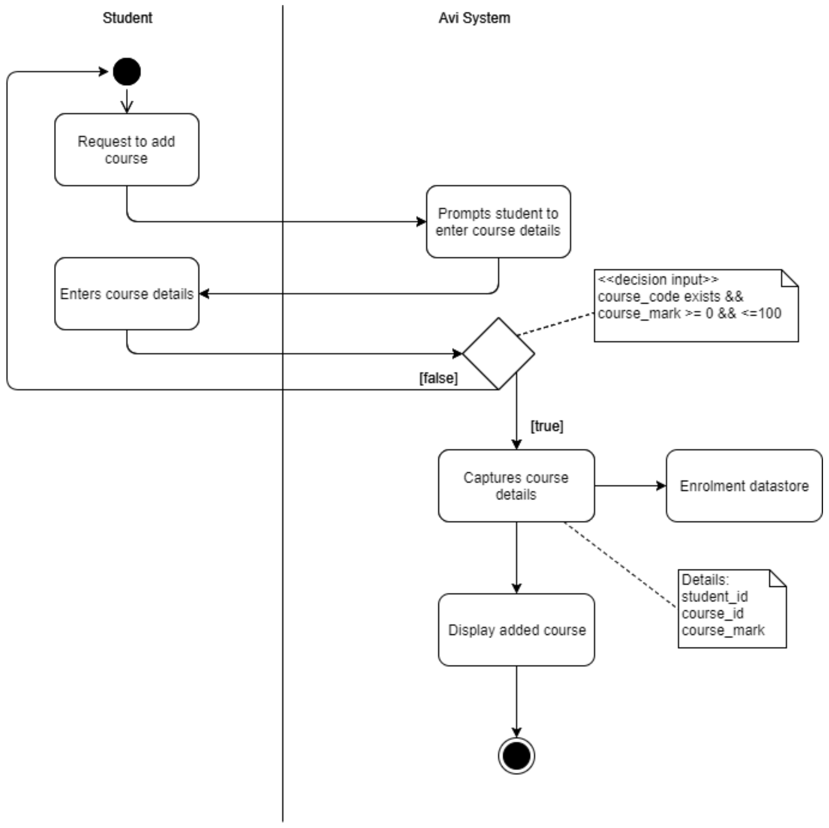
Activity Diagrams

These diagrams are a graphical representation of multiple business processes, specifically, the Create Account, Add Course and Get Recommendation business processes. They show the data flow and the control flow of the business processes.

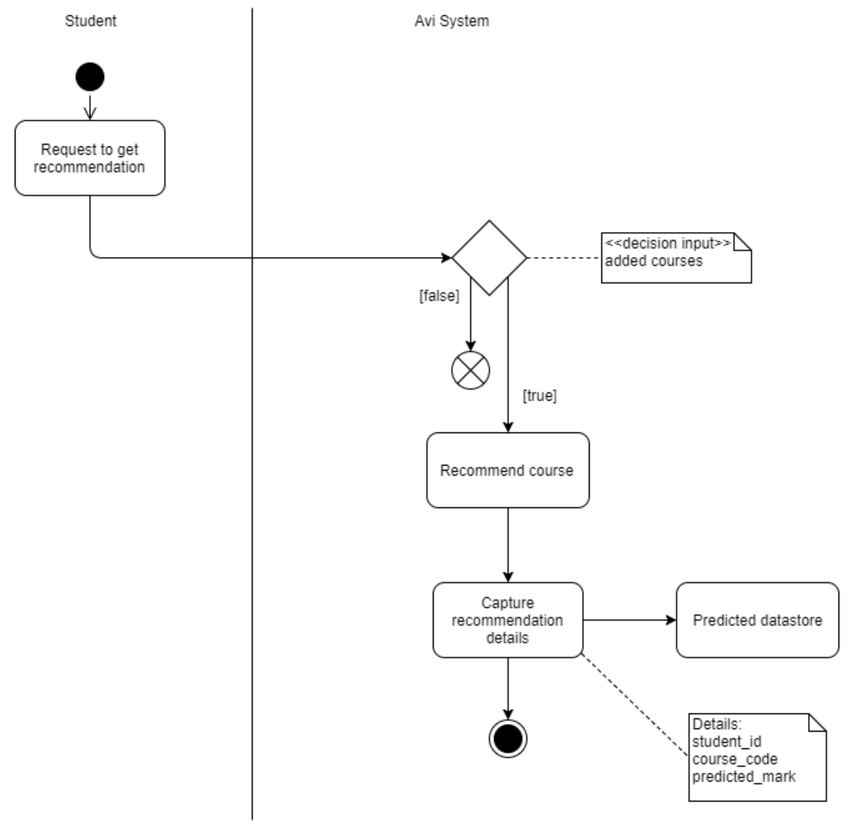


Student
Avi System
Request to create account
Prompts student to enter account creation details
Enters account creation details
<<decision input>>
student_password == confirm_password
[false]
[true]
Captures acount creation details
Student datastore
Details:
student_id
student_name
student_surname
student_current_level
student_password
Redirect to login

Activity Diagram: Create Account


Student
Avi System
Request to add course
Prompts student to enter course details
Enters course details
<<decision input>> course_code exists && course_mark >= 0 && <=100
[false]
[true]
Captures course details
Enrolment datastore
Display added course
Details: student_id course_id course_mark

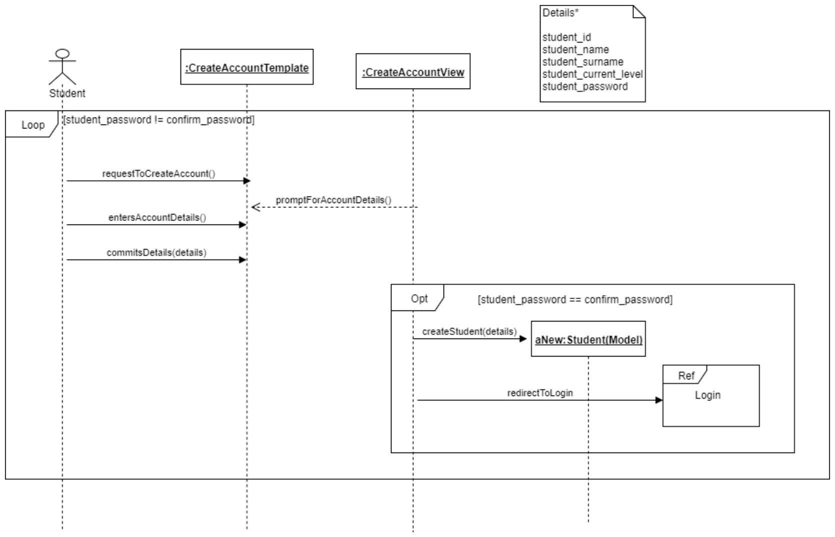
Activity Diagram: Add Course



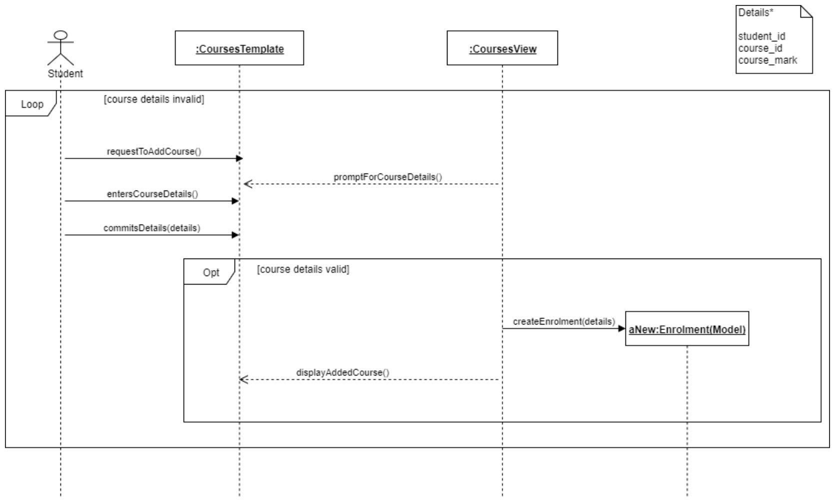
Activity Diagram: Get Recommendation

Sequence Diagrams

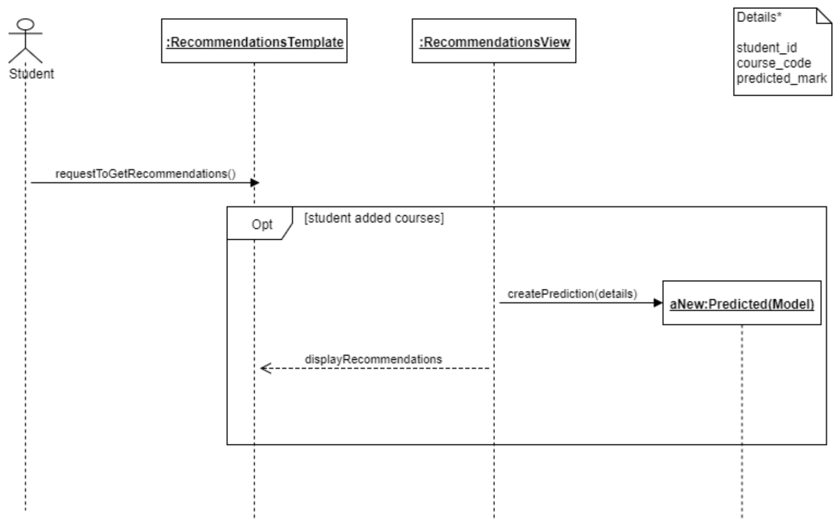
The purpose of these diagrams is to illustrate the communication between the actor and system during the Create Account, Add Course and Get Recommendations processes.



Sequence Diagram: Create Account



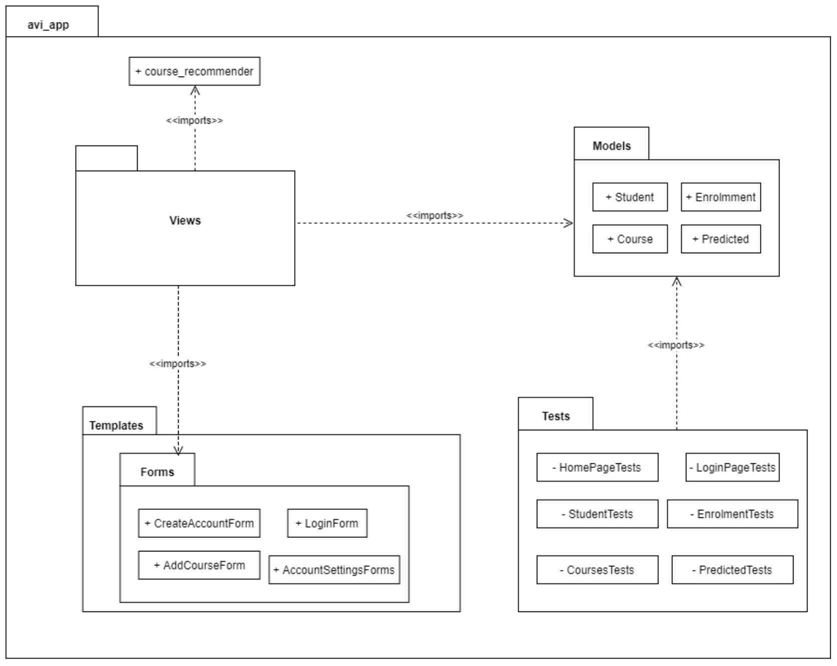
Sequence Diagram: Add Course



Sequence Diagram: Get Recommendation

3.4 Implementation View

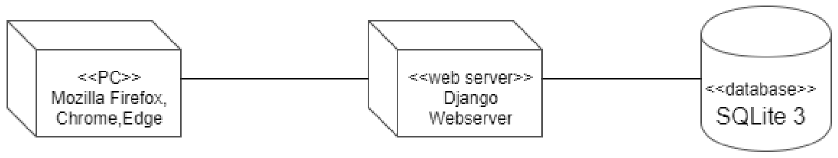
This view depicts the system from the developers perspective. It describes the systems modules in terms of packaging, layering and configuration.



Package Diagram: Avi System

3.5 Deployment View

This view depicts the system from the engineers point of view. It depicts the physical components on which the system runs on. The diagram below only depicts the systems deployment diagram in the development environment since we do not move it to a production environment



Deployment Diagram: Avi System

3.6 Size and Performance

Below is a summary of the systems size and performance.

- The size of the system (classes, packages etc.) is approximately 2.2 MB.
- The size above does not include the external libraries and software that need to be installed.

3.7 Quality

This system is not entirely reliable mainly because this was a prototype to demonstrate how the final system would operate.

- Password encryption was not implemented.
- It depends on the operating system and database security features.
- Difference users data is protected i.e. No user has access to another users data.

4 Description and Demonstration of Modules

This section intends to thoroughly explain the implementation of the important modules and functions of the system.

4.1 Form Modules

4.1.1 AVI Module Forms

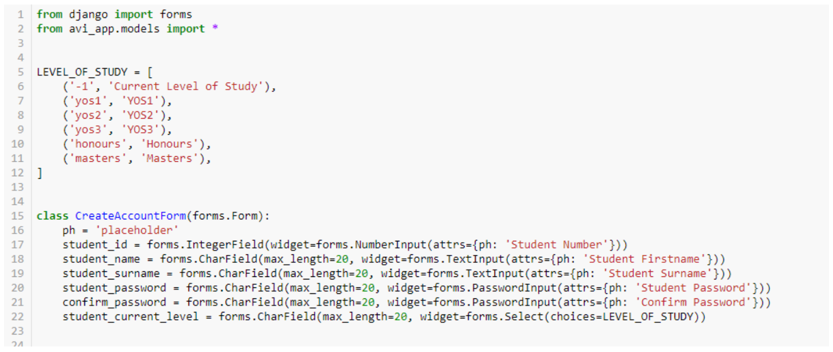
We use Django built in form creator to create a basic form which will then have to pass our required attributes(which invoke database usage- to be discussed later) to it - from Django import forms.

4.1.2 The Django Form class

At the heart of this system of components is Djangos Form class. In much the same way that a Django model describes the logical structure of an object, its behavior, and the way its parts are represented to us, a Form class describes a form and determines how it works and appears.

4.1.3 Avi App Form Functions

The Create User Form: Extracted from avi_app/forms.py

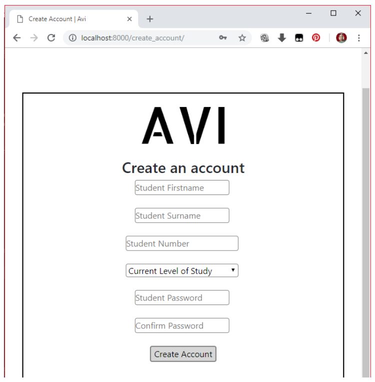


```python
from django import forms
from avi_app.models import *


LEVEL_OF_STUDY = [
    ('-1', 'Current Level of Study'),
    ('yos1', 'YOS1'),
    ('yos2', 'YOS2'),
    ('yos3', 'YOS3'),
    ('honours', 'Honours'),
    ('masters', 'Masters'),
]


class CreateAccountForm(forms.Form):
    ph = 'placeholder'
    student_id = forms.IntegerField(widget=forms.NumberInput(attrs={ph: 'Student Number'}))
    student_name = forms.CharField(max_length=20, widget=forms.TextInput(attrs={ph: 'Student Firstname'}))
    student_surname = forms.CharField(max_length=20, widget=forms.TextInput(attrs={ph: 'Student Surname'}))
    student_password = forms.CharField(max_length=20, widget=forms.PasswordInput(attrs={ph: 'Student Password'}))
    confirm_password = forms.CharField(max_length=20, widget=forms.PasswordInput(attrs={ph: 'Confirm Password'}))
    student_current_level = forms.CharField(max_length=20, widget=forms.Select(choices=LEVEL_OF_STUDY))

```

forms.py

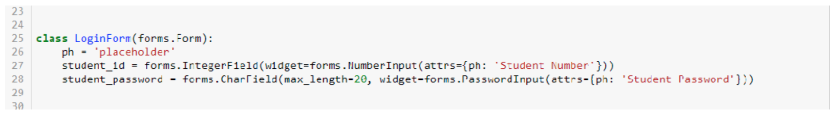
The class CreateAccountForm(form.Form) instantiates the Student Class attributes therefore enabling them to be used in the form. We specify the type of input we can expect for each attribute which enables us to check whether the user input is valid. This class is used in incorporation with html code and CSS for the create_account.html file.

This form allows users to register to use the application. This is what final form looks like:



4.1.4 The Login Form

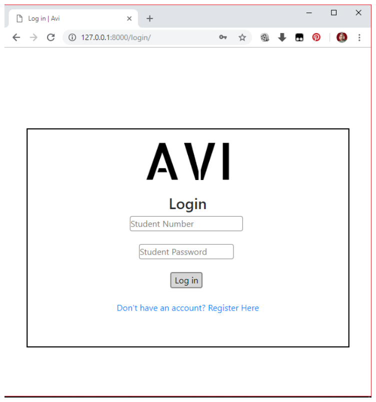
The following lines of code instantiates the login form attributes. This form allows user to be able to login to the application.



```
class LoginForm(forms.Form):
    ph = 'placeholder'
    student_id = forms.IntegerField(widget=forms.NumberInput(attrs={ph: 'Student Number'}))
    student_password = forms.CharField(max_length=20, widget=forms.PasswordInput(attrs={ph: 'Student Password'}))
```

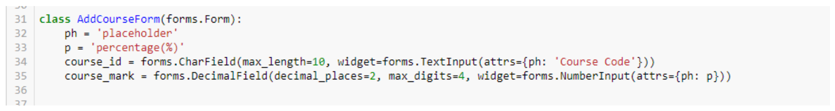
Login Form

This along with the html and CSS code leads to the following form being created:



4.1.5 Add Courses Form

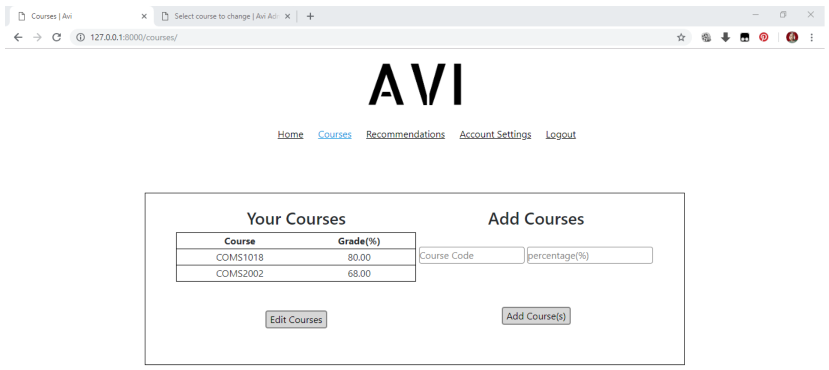
Extracted from avi_app/forms.py.
This form allows users to be add courses to their profile.



```
class AddCourseForm(forms.Form):
    ph = 'placeholder'
    p = 'percentage(%)'
    course_id = forms.CharField(max_length=10, widget=forms.TextInput(attrs={ph: 'Course Code'}))
    course_mark = forms.DecimalField(decimal_places=2, max_digits=4, widget=forms.NumberInput(attrs={ph: p}))
```

Add Course

The above code along with the html and CSS code leads to the following form being created:


AVI

Your Courses

Course	Grade(%)
COMS1018	80.00
COMS2002	68.00

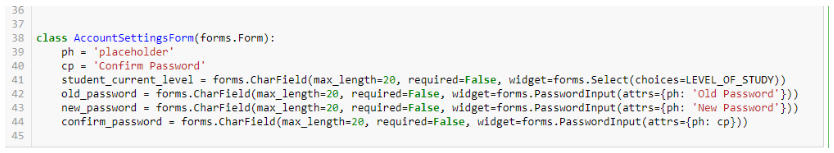
Add Courses

Course

4.1.6 The Account Settings Form

Extracted from avi_app/forms.py.

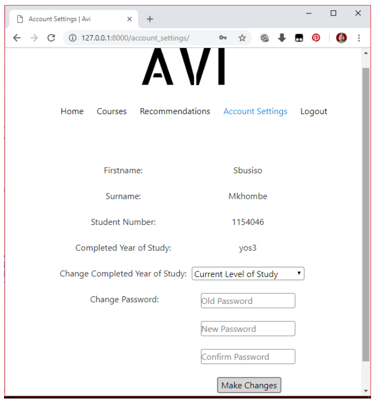
The following lines of code instantiates foreign key attributes. This form allows user to be able to edit their personal details in the application.



```python


class AccountSettingsForm(forms.Form):
    ph = 'placeholder'
    cp = 'Confirm Password'
    student_current_level = forms.CharField(max_length=20, required=False, widget=forms.Select(choices=LEVEL_OF_STUDY))
    old_password = forms.CharField(max_length=20, required=False, widget=forms.PasswordInput(attrs={ph: 'Old Password'}))
    new_password = forms.CharField(max_length=20, required=False, widget=forms.PasswordInput(attrs={ph: 'New Password'}))
    confirm_password = forms.CharField(max_length=20, required=False, widget=forms.PasswordInput(attrs={ph: cp}))

```

Account Settings

The above code along with the html and CSS code leads to the following form being created:


AVI

Firstname:	Sbusiso
Surname:	Mkhombe
Student Number:	1154046
Completed Year of Study:	yos3
Change Completed Year of Study:	Current Level of Study
Change Password:	Old Password
	New Password
	Confirm Password

4.2 Models

Also known as model attributes. A model is the single, definitive source of information about your data. It contains the essential fields and behaviors of the data you are storing. Generally, each model maps to a single database table.

Once you have defined your models, you need to tell Django you are going to use those models. Do this by editing your settings file and changing the INSTALLED_APPS setting to add the name of the module/ app that contains your models.py.

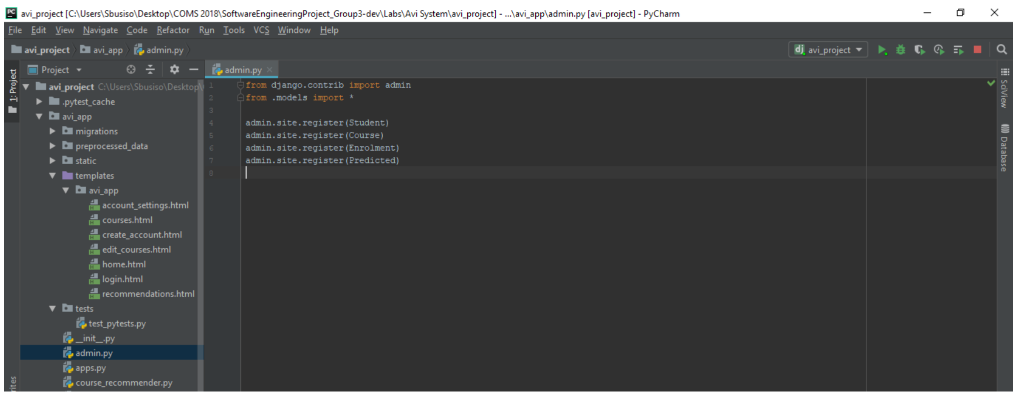
The most important part of a model and the only required part of a model is the list of database fields it defines. Fields are specified by class attributes. Examples of fields in our Student Model (Below) would be student_name, student_surname and student_id.

Each field takes a certain set of field-specific arguments (documented in the model field reference). For example, CharField (and its subclasses) require a max_length argument which specifies the size of the VARCHAR database field used to store the data.There is also a set of common arguments available to all field types. All are optional.

4.2.1 Registering Models

Django comes with an admin.py file which allows us to register our model - basically add tables to our database.

We register our tables with the following code:



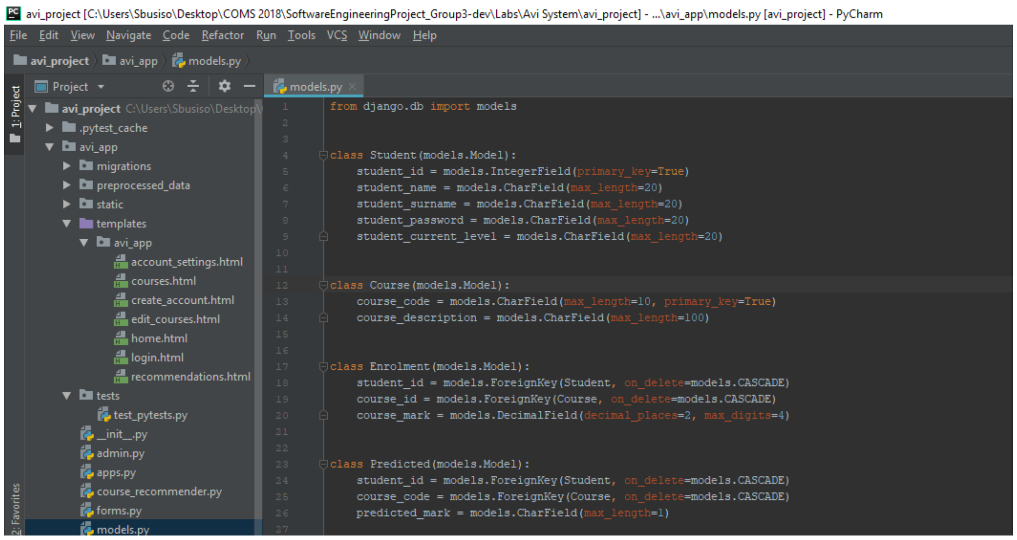
```
from django.contrib import admin
from .models import *

admin.site.register(Student)
admin.site.register(Course)
admin.site.register(Enrolment)
admin.site.register(Predicted)
```

We then use the following commands from our shell to save our database changes:

python manage.py makemigrations avi_app

python manage.py migrate

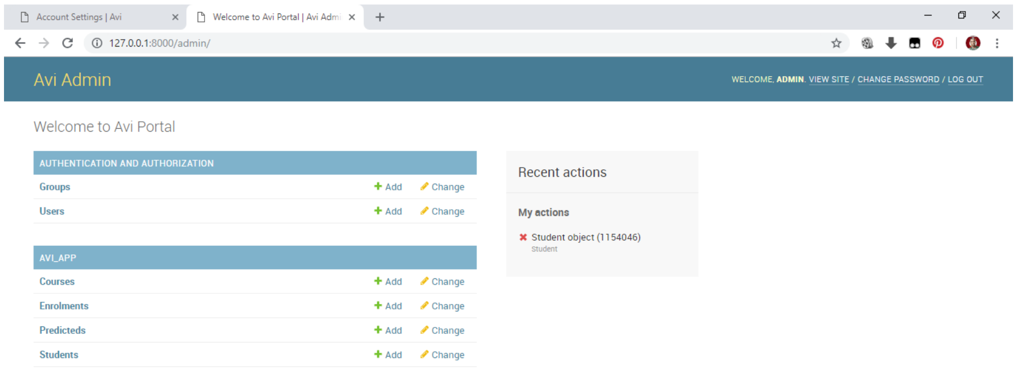
Defining and initializing our models(tables):



```python
from django.db import models


class Student(models.Model):
    student_id = models.IntegerField(primary_key=True)
    student_name = models.CharField(max_length=20)
    student_surname = models.CharField(max_length=20)
    student_password = models.CharField(max_length=20)
    student_current_level = models.CharField(max_length=20)


class Course(models.Model):
    course_code = models.CharField(max_length=10, primary_key=True)
    course_description = models.CharField(max_length=100)


class Enrolment(models.Model):
    student_id = models.ForeignKey(Student, on_delete=models.CASCADE)
    course_id = models.ForeignKey(Course, on_delete=models.CASCADE)
    course_mark = models.DecimalField(decimal_places=2, max_digits=4)


class Predicted(models.Model):
    student_id = models.ForeignKey(Student, on_delete=models.CASCADE)
    course_code = models.ForeignKey(Course, on_delete=models.CASCADE)
    predicted_mark = models.CharField(max_length=1)
```

The shell migration command leads to these tables being created:

(These can be accessed from http://127.0.0.1:8000/admin with the admin as mpinane and nane as password for admin access).



Avi Admin

WELCOME, ADMIN. VIEW SITE / CHANGE PASSWORD / LOG OUT

Welcome to Avi Portal

AUTHENTICATION AND AUTHORIZATION		
Groups	Add	Change
Users	Add	Change

AVI_APP		
Courses	Add	Change
Enrolments	Add	Change
Predicteds	Add	Change
Students	Add	Change

Recent actions

My actions

Student object (1154046)
Student

4.2.2 URL Dispatcher

A clean, elegant URL scheme is an important detail in a high-quality Web application. Django lets you design URLs however you want, with no framework limitations.

To design URLs for an app, you create a Python module informally called a URLconf (URL configuration). This module is pure Python code and is a mapping between URL path expressions to Python functions (your views).

This mapping can be as short or as long as needed. It can reference other mappings. And, because its pure Python code, it can be constructed dynamically.

4.2.3 Request Process

When a user requests a page from your Django-powered site, this is the algorithm the system follows to determine which Python code to execute:

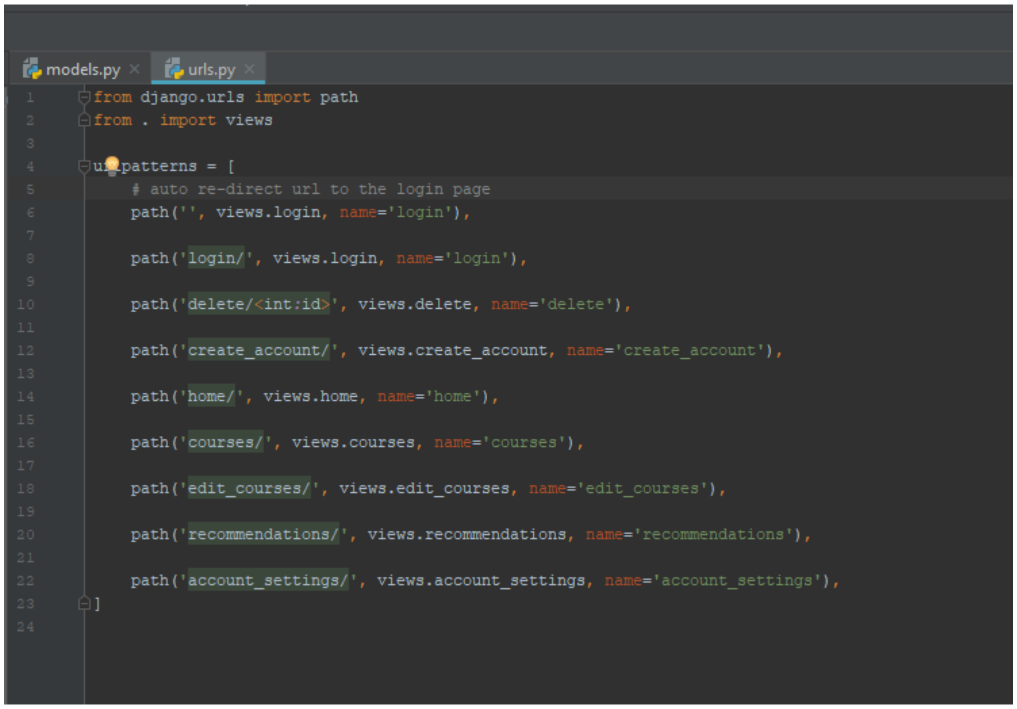
- Django determines the root URLconf module to use. Ordinarily, this is the value of the ROOT_URLCONF setting, but if the incoming HttpRequest object has a urlconf attribute (set by middleware), its value will be used in place of theROOT_URLCONF setting.
- Django loads that Python module and looks for the variable urlpatterns. This should be a Python list of django.urls.path() and/or django.urls.re_path() instances.
- Django runs through each URL pattern, in order, and stops at the first one that matches the requested URL.
- Once one of the URL patterns matches, Django imports and calls the given view, which is a simple Python function (or a class-based view). The view gets passed the following arguments:

  - An instance of HttpRequest.

  - If the matched URL pattern returned no named groups, then the matches from the regular expression are provided as positional arguments.

  - The keyword arguments are made up of any named parts matched by the path expression, overridden by any arguments specified in the optional kwargs argument to django.urls.path() or django.urls.re_path().
- If no URL pattern matches, or if an exception is raised during any point in this process, Django invokes an appropriate error-handling view.

From avi_app/urls.py:
We initialize all urls we want our site to expect, for example localhost:8000/login would be a valid url because we have declared it in line 8.

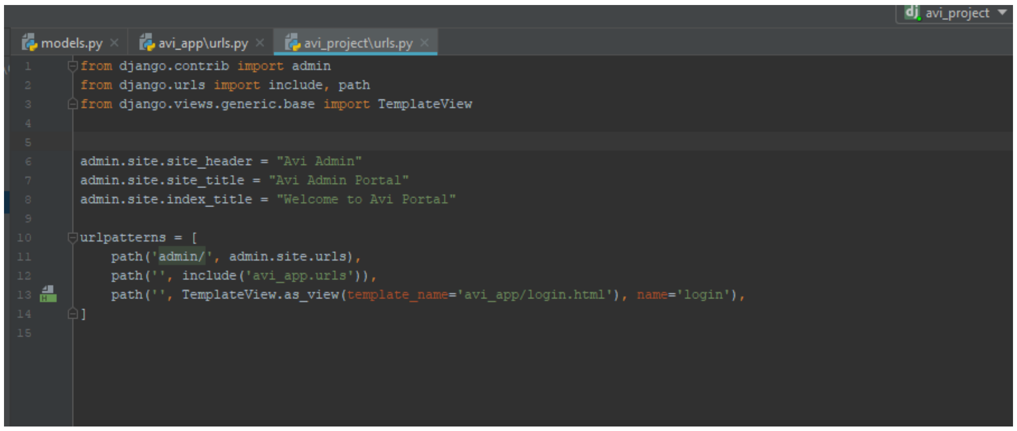


```
1   from django.urls import path
2   from . import views
3
4   urlpatterns = [
5       # auto re-direct url to the login page
6       path('', views.login, name='login'),
7
8       path('login/', views.login, name='login'),
9
10      path('delete/<int:id>', views.delete, name='delete'),
11
12      path('create_account/', views.create_account, name='create_account'),
13
14      path('home/', views.home, name='home'),
15
16      path('courses/', views.courses, name='courses'),
17
18      path('edit_courses/', views.edit_courses, name='edit_courses'),
19
20      path('recommendations/', views.recommendations, name='recommendations'),
21
22      path('account_settings/', views.account_settings, name='account_settings'),
23  ]
24
```

Line 8 declares the path/url that localhost:8000/login redirects to views.login then ensures that the login method in the views.py file is associated with this url and used in its execution.

from . import views allows us to use methods from our views.py file which are then translated to html files and forms which are then displayed to the user.

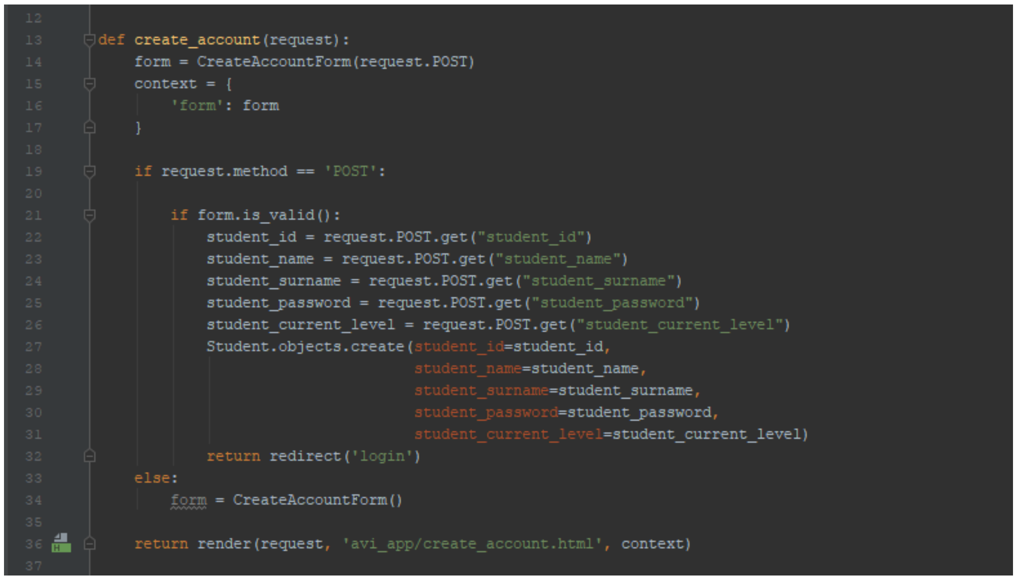
We declare our overall usage of urls in our main url file url.py in line 12 by basically declaring the inclusion of all the other urls in the avi_app.



```python
from django.contrib import admin
from django.urls import include, path
from django.views.generic.base import TemplateView


admin.site.site_header = "Avi Admin"
admin.site.site_title = "Avi Admin Portal"
admin.site.index_title = "Welcome to Avi Portal"

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('avi_app.urls')),
    path('', TemplateView.as_view(template_name='avi_app/login.html'), name='login'),
]
```

4.3 Views

4.3.1 Create Account Module

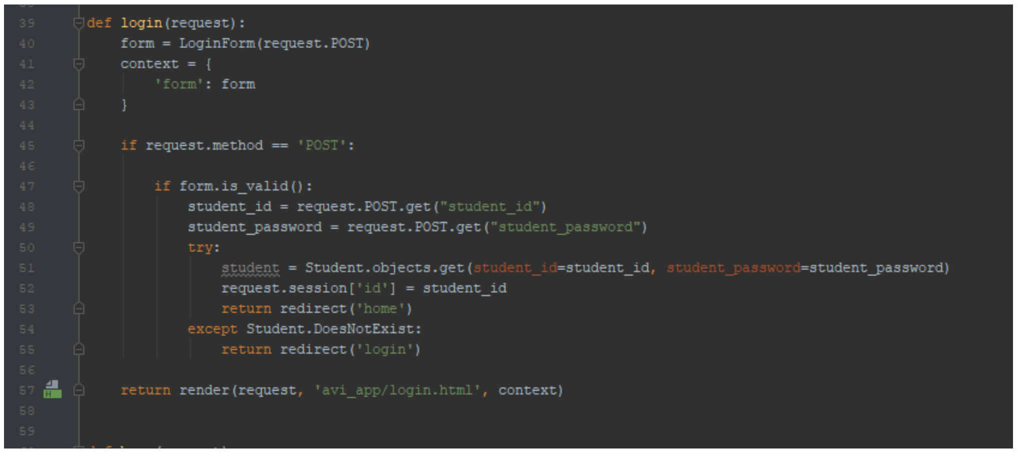


```python
12
13  def create_account(request):
14      form = CreateAccountForm(request.POST)
15      context = {
16          'form': form
17      }
18
19      if request.method == 'POST':
20
21          if form.is_valid():
22              student_id = request.POST.get("student_id")
23              student_name = request.POST.get("student_name")
24              student_surname = request.POST.get("student_surname")
25              student_password = request.POST.get("student_password")
26              student_current_level = request.POST.get("student_current_level")
27              Student.objects.create(student_id=student_id,
28                                     student_name=student_name,
29                                     student_surname=student_surname,
30                                     student_password=student_password,
31                                     student_current_level=student_current_level)
32              return redirect('login')
33      else:
34          form = CreateAccountForm()
35
36      return render(request, 'avi_app/create_account.html', context)
37
```

- This module is essential in the create account form functionality.
- It creates a form instance to display (line 13-17).
- checks if the request sent by the user is a POST request i.e. It want to add something (line 19).
- If yes, checks if user has submitted valid input to the form (line 21).
- If the form is valid, it then creates and stores the data into variables.

- A student object is then stored in our database table.
- It then redirects the user to the login page where they are then able to login with their credentials.

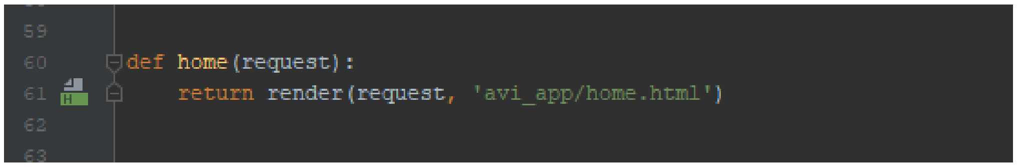
4.3.2 Login Module



```
39 def login(request):
40     form = LoginForm(request.POST)
41     context = {
42         'form': form
43     }
44
45     if request.method == 'POST':
46
47         if form.is_valid():
48             student_id = request.POST.get("student_id")
49             student_password = request.POST.get("student_password")
50             try:
51                 student = Student.objects.get(student_id=student_id, student_password=student_password)
52                 request.session['id'] = student_id
53                 return redirect('home')
54             except Student.DoesNotExist:
55                 return redirect('login')
56
57     return render(request, 'avi_app/login.html', context)
58
59
```

- This module is essential in the login form functionality.
- It creates a form instance to display (line 40-43).
- checks if a POST request has been sent from the user (line 45).
- If yes, checks if user has submitted valid input to the form (line 47).
- If the form is valid, it then creates variables which are used for authentication by checking if they match the values of the corresponding attribute in our database. (line 50-52).
- If they do, the student is then logged-in and redirected to the home screen. (line 53).
- Else they have to try logging in again. (line 54-55).

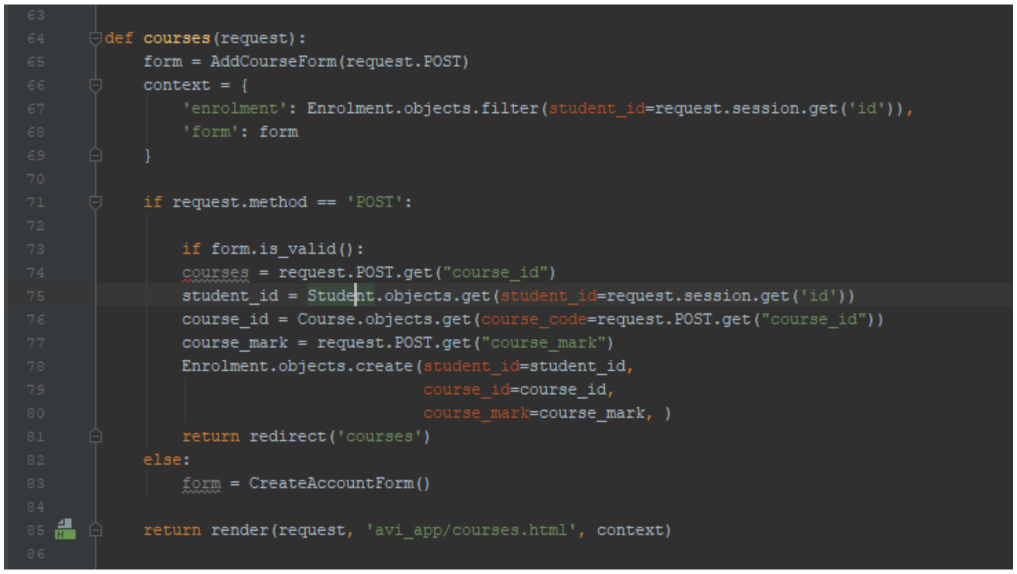
4.3.3 Home Module



```
def home(request):
    return render(request, 'avi_app/home.html')
```

This module displays the home page.

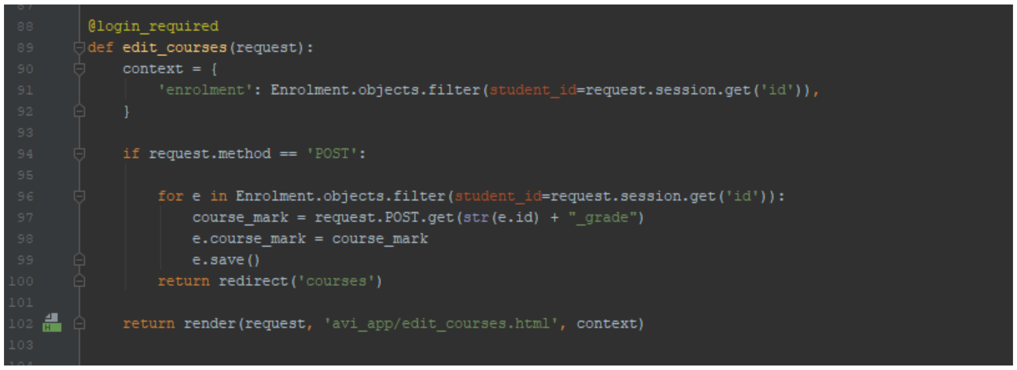
4.3.4 Course Module



```
63
64  def courses(request):
65      form = AddCourseForm(request.POST)
66      context = {
67          'enrolment': Enrolment.objects.filter(student_id=request.session.get('id')),
68          'form': form
69      }
70
71      if request.method == 'POST':
72
73          if form.is_valid():
74          courses = request.POST.get("course_id")
75          student_id = Student.objects.get(student_id=request.session.get('id'))
76          course_id = Course.objects.get(course_code=request.POST.get("course_id"))
77          course_mark = request.POST.get("course_mark")
78          Enrolment.objects.create(student_id=student_id,
79                                   course_id=course_id,
80                                   course_mark=course_mark, )
81          return redirect('courses')
82      else:
83          form = CreateAccountForm()
84
85      return render(request, 'avi_app/courses.html', context)
86
```

- This module is essential in the adding courses form functionality.
- It creates a form instance to display (line 65-69).
- same as create account (line 71).
- If yes, checks if the user has submitted valid input to the form (line 73).
- If the form is valid, it then creates and stores the data into variables.
- A course object is then stored in our database table which uses the variables we declared as column attributes.
- It then redirects us to the courses page where we are then able to see our added courses.
- If the form info passed is invalid, the form is recreated (line 83) and the user can retry adding his or her courses. (line 85).

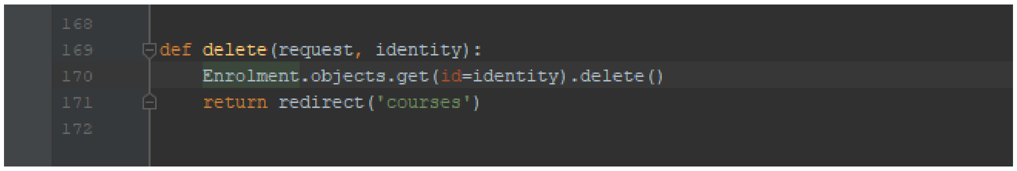
4.3.5 Edit Courses Module



```
 88  @login_required
 89  def edit_courses(request):
 90      context = {
 91          'enrolment': Enrolment.objects.filter(student_id=request.session.get('id')),
 92      }
 93
 94      if request.method == 'POST':
 95
 96          for e in Enrolment.objects.filter(student_id=request.session.get('id')):
 97              course_mark = request.POST.get(str(e.id) + "_grade")
 98              e.course_mark = course_mark
 99              e.save()
100          return redirect('courses')
101
102      return render(request, 'avi_app/edit_courses.html', context)
103
```

- Line 88 declares that login is required for user to be able to access this view (form). This method is extracted from the from the Django contrib.auth class and we import the login required from it. (check imports).
- Line 90 to 92 we define the form context and filter student_id to ensure that the request is from a valid student-object (user).
- same as create account (Line 94).
- If yes, we then iterate through the enrolment table for the user and update their marks for every updated mark input they pass.
- e.save() achieves this for every updated mark input. (Line 99).
- The user is then redirected to the courses page which now displays all their updated marks.
- If the user has passed invalid input or no input at all, they are redirected to the edit courses page and given a new form to try updating their course or course mark again.

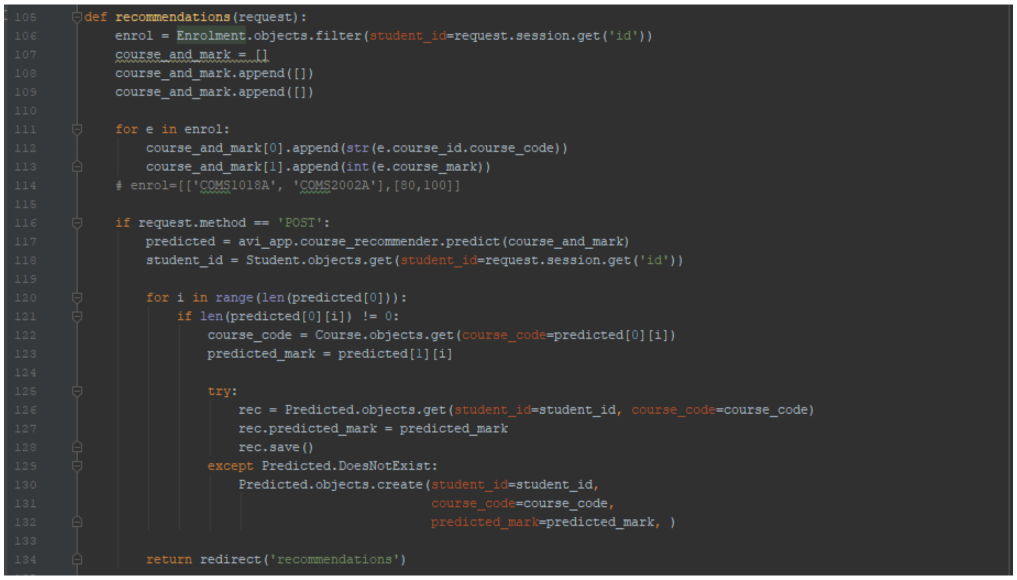
4.3.6 Delete Module



```

def delete(request, identity):
    Enrolment.objects.get(id=identity).delete()
    return redirect('courses')

```

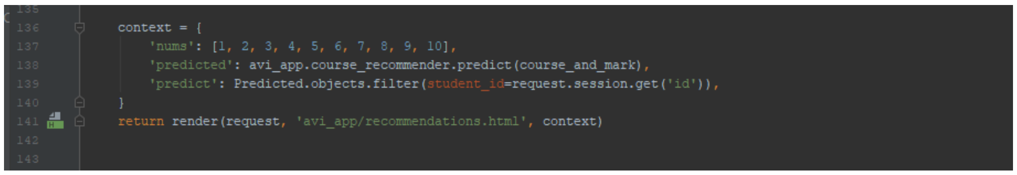
- The delete courses module deletes a user course from our database.
- The course id is passed as identity into the function and the corresponding course is deleted from the users courses.
- The courses view is then updated accordingly.

4.3.7 Recommendations Module



```
105 def recommendations(request):
106     enrol = Enrolment.objects.filter(student_id=request.session.get('id'))
107     course_and_mark = []
108     course_and_mark.append([])
109     course_and_mark.append([])
110
111     for e in enrol:
112         course_and_mark[0].append(str(e.course_id.course_code))
113         course_and_mark[1].append(int(e.course_mark))
114     # enrol=[['COMS1018A', 'COMS2002A'],[80,100]]
115
116     if request.method == 'POST':
117         predicted = avi_app.course_recommender.predict(course_and_mark)
118         student_id = Student.objects.get(student_id=request.session.get('id'))
119
120         for i in range(len(predicted[0])):
121             if len(predicted[0][i]) != 0:
122                 course_code = Course.objects.get(course_code=predicted[0][i])
123                 predicted_mark = predicted[1][i]
124
125                 try:
126                     rec = Predicted.objects.get(student_id=student_id, course_code=course_code)
127                     rec.predicted_mark = predicted_mark
128                     rec.save()
129                 except Predicted.DoesNotExist:
130                     Predicted.objects.create(student_id=student_id,
131                                              course_code=course_code,
132                                              predicted_mark=predicted_mark, )
133
134         return redirect('recommendations')
```

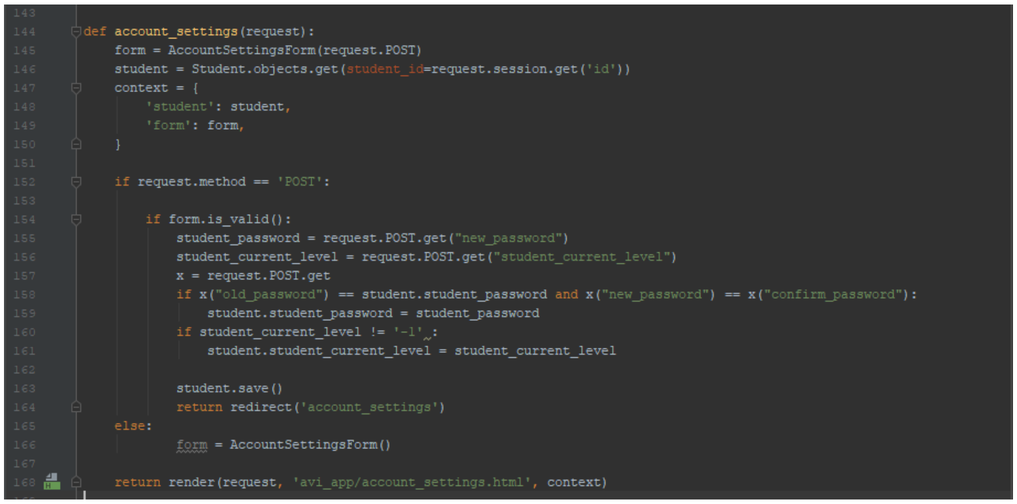
- We redefine a new enrolment called enrol which ensures that user is logged in.(Line 106).
- We define a new python list variable called course and mark which will store a course and its corresponding mark.
- A for-loop is used to append the users courses and marks into the list. (line 111-113).
- A check is used to ensure that the user is trying to get recommendations. (line 116).
- We use the course_recommender module from recommender.py to get our final recommendations for courses.
- An if statement is used to check that our courses marks list is not empty, if not we then return the predicted marks for their respective courses.
- The recommendations are then returned to the student through the recommendations view(Line 134).



```
135
136     context = {
137         'nums': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
138         'predicted': avi_app.course_recommender.predict(course_and_mark),
139         'predict': Predicted.objects.filter(student_id=request.session.get('id')),
140     }
141     return render(request, 'avi_app/recommendations.html', context)
142
143
```

- If the user didnt pass valid form values or the form was empty we define a new context.
- We then pass it through the recommendations form which enables the user to try getting a recommendation again.

4.3.8 Account Settings Module



```
143
144 def account_settings(request):
145     form = AccountSettingsForm(request.POST)
146     student = Student.objects.get(student_id=request.session.get('id'))
147     context = {
148         'student': student,
149         'form': form,
150     }
151
152     if request.method == 'POST':
153
154         if form.is_valid():
155             student_password = request.POST.get("new_password")
156             student_current_level = request.POST.get("student_current_level")
157             x = request.POST.get
158             if x("old_password") == student.student_password and x("new_password") == x("confirm_password"):
159                 student.student_password = student_password
160             if student_current_level != '-1' :
161                 student.student_current_level = student_current_level
162
163             student.save()
164             return redirect('account_settings')
165     else:
166             form = AccountSettingsForm()
167
168     return render(request, 'avi_app/account_settings.html', context)
```

- This module lets the user update their current year of study and password through the account settings form.
- A form is created which then checks if user is logged in and requires user input - (line 145).
- We define our default context which we will pass to our form to be displayed in our html - (line 147-150).
- same as create account (line 152 and 154).
- If the user is attempting to update their current level the new value is stored in the variable student_current_level - (line 156) and then updated in the database if its a valid value (line 160 and 161).
- If the user is attempting to update their password, the passed in current password is checked if it matches the one in the database and if valid it changes the password to new_password value.

5 User Acceptance Testing

User Acceptance Testing is the final stage of testing where the target user can check the system for its compliance with business requirements and allows the end user to familiarize themselves with the functionality of the system.

5.1 Objective

To allow the end user to run through the Avi System basing their test on the business requirements. To get feedback from the user with any questions or changes that they require and also to assess usability/intuitiveness of the system.

5.2 Justification

The User Acceptance testing has an important role in which the end user validates the system whether it meets the business requirements before getting deployed.

5.3 Participant

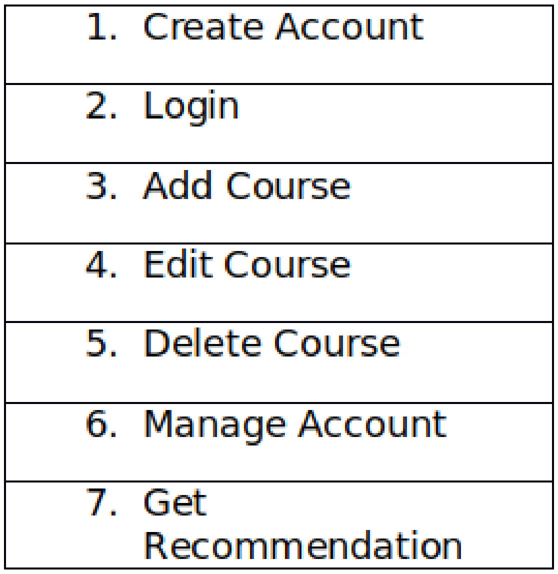
This testing will be carried out by an end user (a third year Computer Science Student) that is briefed about the business requirements so that they may have knowledge about the clients needs of the system.

Test Date: 05/10/2018

Tester: Katleho Mokoena

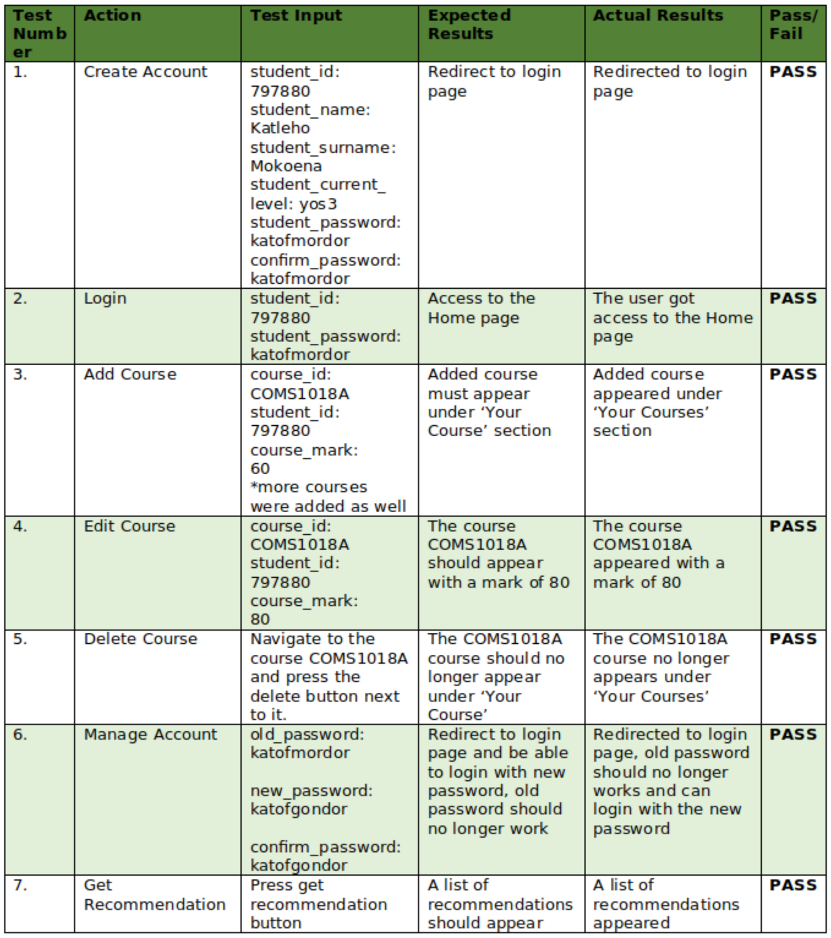
5.4 Methodology

The user will participate in the following use cases:



1. Create Account
2. Login
3. Add Course
4. Edit Course
5. Delete Course
6. Manage Account
7. Get Recommendation

Use Cases



Test Number	Action	Test Input	Expected Results	Actual Results	Pass/ Fail
1.	Create Account	student_id: 797880 student_name: Katleho student_surname: Mokoena student_current_ level: yos3 student_password: katofmordor confirm_password: katofmordor	Redirect to login page	Redirected to login page	**PASS**
2.	Login	student_id: 797880 student_password: katofmordor	Access to the Home page	The user got access to the Home page	**PASS**
3.	Add Course	course_id: COMS1018A student_id: 797880 course_mark: 60 *more courses were added as well	Added course must appear under 'Your Course' section	Added course appeared under 'Your Courses' section	**PASS**
4.	Edit Course	course_id: COMS1018A student_id: 797880 course_mark: 80	The course COMS1018A should appear with a mark of 80	The course COMS1018A appeared with a mark of 80	**PASS**
5.	Delete Course	Navigate to the course COMS1018A and press the delete button next to it.	The COMS1018A course should no longer appear under 'Your Course'	The COMS1018A course no longer appears under 'Your Courses'	**PASS**
6.	Manage Account	old_password: katofmordor new_password: katofgondor confirm_password: katofgondor	Redirect to login page and be able to login with new password, old password should no longer work	Redirected to login page, old password should no longer works and can login with the new password	**PASS**
7.	Get Recommendation	Press get recommendation button	A list of recommendations should appear	A list of recommendations appeared	**PASS**

Actions

5.5 Results

The user found the system easy to use and was able to perform all the tasks.

5.6 User Acceptance Testing Summary

Avi System passed all the user acceptance tests conducted on the specified use cases.

6 Unit Testing

Unit Testing is a level of software testing where individual units or components of a software are tested. The purpose is to validate that each unit of the software performs as designed. A unit is the smallest testable part of any software. It usually has one or a few inputs and usually a single output.

Unit Testing is the first level of software testing. These tests are independent of each other.

This is achieved through white box testing which is essentially testing based on the analysis of the internal structure of the system.

6.1 Unit Testing Benefits

- Code is reusable.
- Development is faster in the long run.
- Code maintenance is easier.
- Code becomes more reliable.
- Debugging becomes easier.

In our project we used unit tests to ensure that the individual components in our code work accordingly for various possible inputs. We don't create tests for everything, we instead focus on testing the part of our code that impact the behavior of our system.

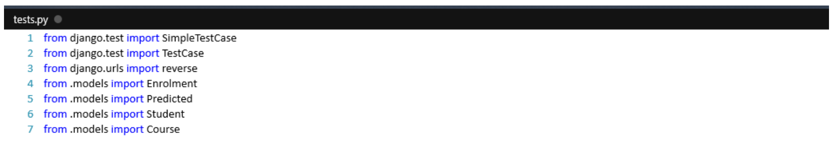
These test cases are split into classes where a class contains test methods which test many features of one Module, for example we have a class TestUrls which tests all the possible urls which have been defined to be handled.

Its always a good idea to test both expected and unexpected behavior. These are Commented as Test and Counter Test for most individual test cases. It is essential that both instances are considered whether they seem trivial or not. This is how our tests are conducted.

The tests are divided into two file in which one is conducted using the Django built in test modules in the file tests.py in the main app directory and the other is conducted using Pytest modules in the test_pytest.py file under the tests folder.

6.2 Part One: Using Django Test modules

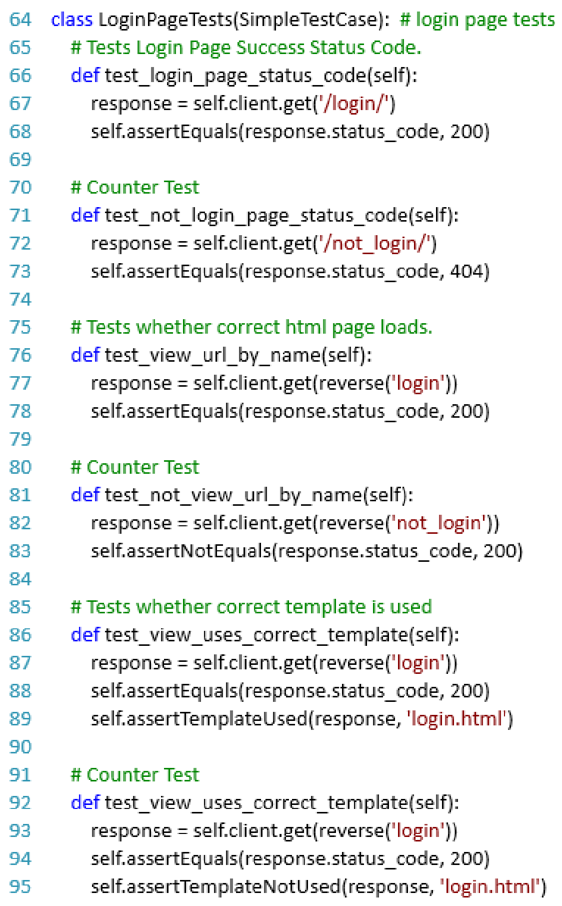
Codes extracted from tests.py.



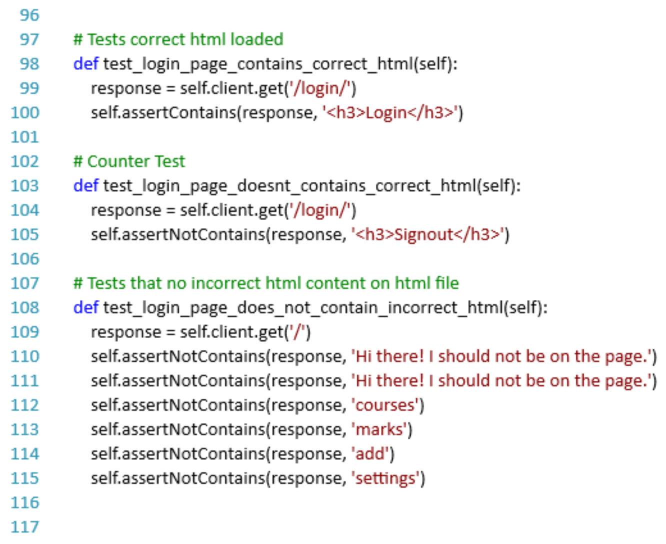
```
tests.py
1  from django.test import SimpleTestCase
2  from django.test import TestCase
3  from django.urls import reverse
4  from .models import Enrolment
5  from .models import Predicted
6  from .models import Student
7  from .models import Course
```

Imports We import the models we are going to test as well. (Line 4-7)

6.2.1 Login Page Tests



```python
class LoginPageTests(SimpleTestCase):  # login page tests
    # Tests Login Page Success Status Code.
    def test_login_page_status_code(self):
        response = self.client.get('/login/')
        self.assertEquals(response.status_code, 200)

    # Counter Test
    def test_not_login_page_status_code(self):
        response = self.client.get('/not_login/')
        self.assertEquals(response.status_code, 404)

    # Tests whether correct html page loads.
    def test_view_url_by_name(self):
        response = self.client.get(reverse('login'))
        self.assertEquals(response.status_code, 200)

    # Counter Test
    def test_not_view_url_by_name(self):
        response = self.client.get(reverse('not_login'))
        self.assertNotEquals(response.status_code, 200)

    # Tests whether correct template is used
    def test_view_uses_correct_template(self):
        response = self.client.get(reverse('login'))
        self.assertEquals(response.status_code, 200)
        self.assertTemplateUsed(response, 'login.html')

    # Counter Test
    def test_view_uses_correct_template(self):
        response = self.client.get(reverse('login'))
        self.assertEquals(response.status_code, 200)
        self.assertTemplateNotUsed(response, 'login.html')
```

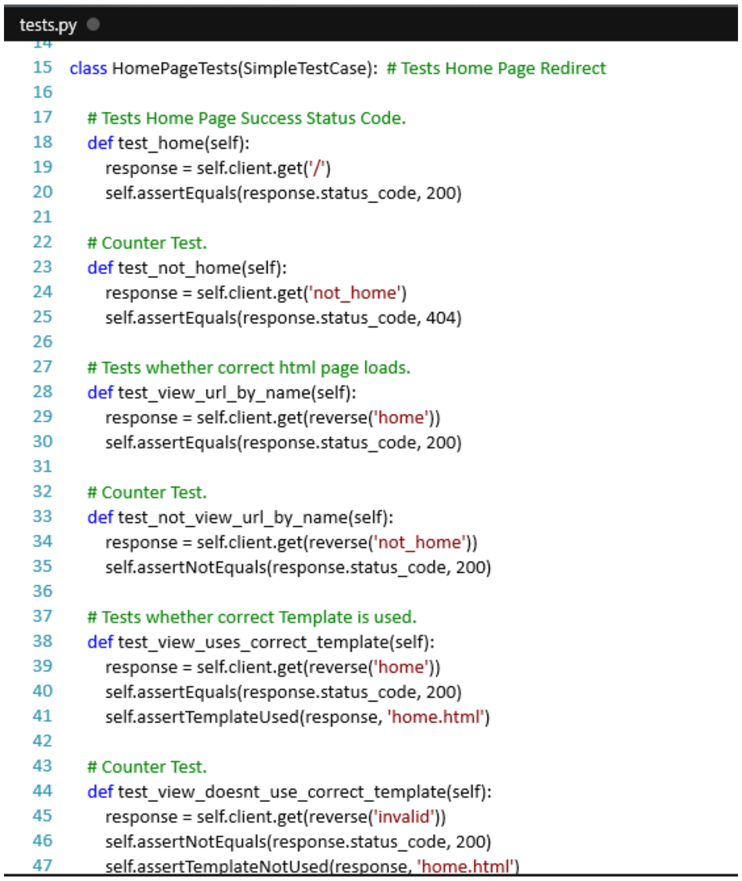


```python

    # Tests correct html loaded
    def test_login_page_contains_correct_html(self):
        response = self.client.get('/login/')
        self.assertContains(response, '<h3>Login</h3>')

    # Counter Test
    def test_login_page_doesnt_contains_correct_html(self):
        response = self.client.get('/login/')
        self.assertNotContains(response, '<h3>Signout</h3>')

    # Tests that no incorrect html content on html file
    def test_login_page_does_not_contain_incorrect_html(self):
        response = self.client.get('/')
        self.assertNotContains(response, 'Hi there! I should not be on the page.')
        self.assertNotContains(response, 'Hi there! I should not be on the page.')
        self.assertNotContains(response, 'courses')
        self.assertNotContains(response, 'marks')
        self.assertNotContains(response, 'add')
        self.assertNotContains(response, 'settings')


```

Login Page Tests

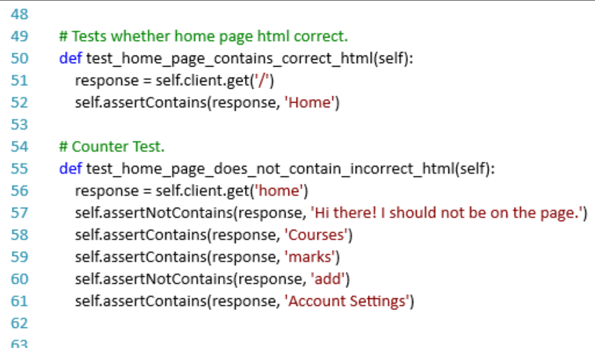
- The login tests are defined in the class LoginPageTests - (line 64).
- The first test and counter test ensures that the login page status code is returned whenever the login url is invoked and that an error status code is returned whenever an invalid login request (possibly due to a typing error) is passed.- (line 65-73).
- The second test ensures that the correct html file is returned whenever login url is passed, the counter test ensures that ensures that its not returned anywhere else.-(lines 75-83).
- The third test ensures that the correct template is used by the login url, its counter test ensures that that template is not used elsewhere. - (line 86-95).
- The forth test ensures that the html file contains the word Login heading, its counter test ensures that the html doesnt contain any other random headings. (line 97-105).
- The fifth test ensures that the html doesnt contain any strings from other html files. - (line 107 - 115).

6.2.2 Sign Up Page Tests



tests.py

```
14
15 class HomePageTests(SimpleTestCase):  # Tests Home Page Redirect
16
17   # Tests Home Page Success Status Code.
18   def test_home(self):
19     response = self.client.get('/')
20     self.assertEquals(response.status_code, 200)
21
22   # Counter Test.
23   def test_not_home(self):
24     response = self.client.get('not_home')
25     self.assertEquals(response.status_code, 404)
26
27   # Tests whether correct html page loads.
28   def test_view_url_by_name(self):
29     response = self.client.get(reverse('home'))
30     self.assertEquals(response.status_code, 200)
31
32   # Counter Test.
33   def test_not_view_url_by_name(self):
34     response = self.client.get(reverse('not_home'))
35     self.assertNotEquals(response.status_code, 200)
36
37   # Tests whether correct Template is used.
38   def test_view_uses_correct_template(self):
39     response = self.client.get(reverse('home'))
40     self.assertEquals(response.status_code, 200)
41     self.assertTemplateUsed(response, 'home.html')
42
43   # Counter Test.
44   def test_view_doesnt_use_correct_template(self):
45     response = self.client.get(reverse('invalid'))
46     self.assertNotEquals(response.status_code, 200)
47     self.assertTemplateNotUsed(response, 'home.html')
```



```
48
49   # Tests whether home page html correct.
50   def test_home_page_contains_correct_html(self):
51     response = self.client.get('/')
52     self.assertContains(response, 'Home')
53
54   # Counter Test.
55   def test_home_page_does_not_contain_incorrect_html(self):
56     response = self.client.get('home')
57     self.assertNotContains(response, 'Hi there! I should not be on the page.')
58     self.assertContains(response, 'Courses')
59     self.assertContains(response, 'marks')
60     self.assertNotContains(response, 'add')
61     self.assertContains(response, 'Account Settings')
62
63
```

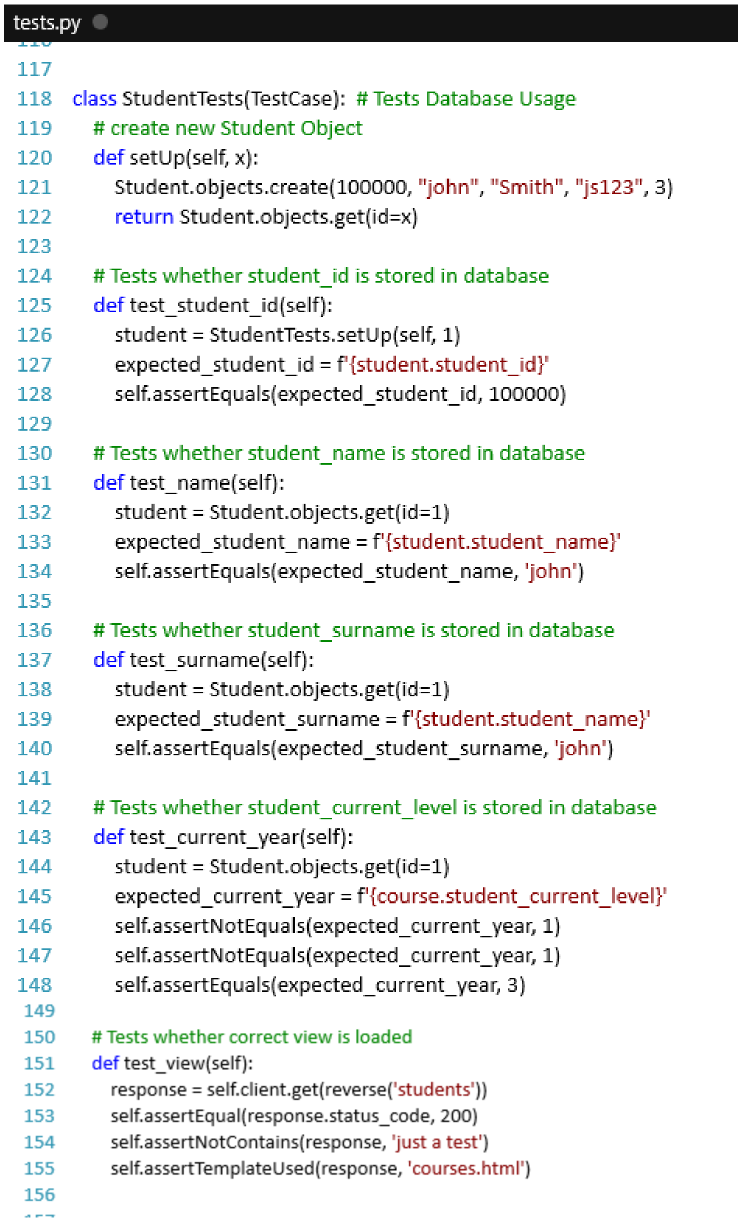
<u>Sign Up Page Tests</u>

- The HomePage tests are defined in the class HomePageTests - (line 15).
- The first test and counter test ensures that the home page status code is returned whenever the home url is invoked and that an error status code

is returned whenever an invalid home request (possibly due to a typing error) is passed.- (line 17-25).

- The second test ensures that the correct html file is returned whenever home url is passed, the counter test ensures that ensures that its not returned anywhere else.-(lines 27-35).
- The third test ensures that the correct template is used by the home url, its counter test ensures that that template is not used elsewhere. - (line 37-47).
- The fourth test ensure that the correct home url is loaded and returned.

6.2.3 Model Tests



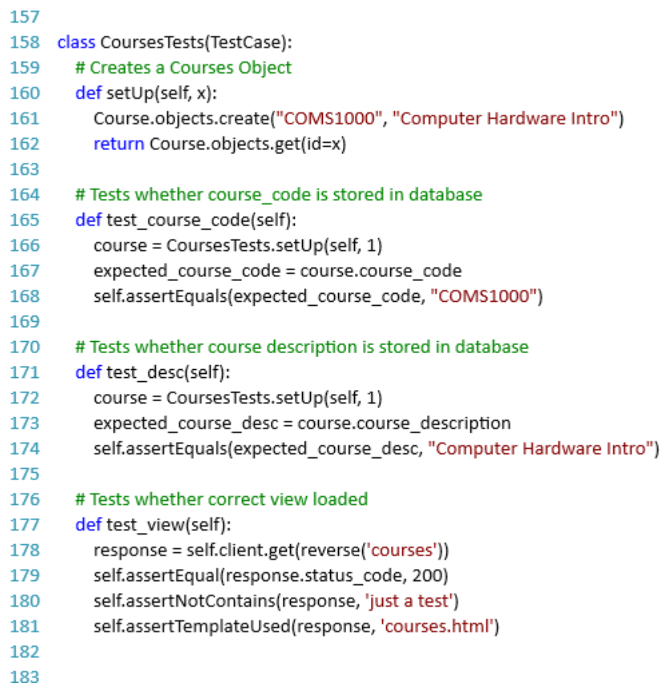
tests.py

```
117
118 class StudentTests(TestCase):  # Tests Database Usage
119     # create new Student Object
120     def setUp(self, x):
121         Student.objects.create(100000, "john", "Smith", "js123", 3)
122         return Student.objects.get(id=x)
123
124     # Tests whether student_id is stored in database
125     def test_student_id(self):
126         student = StudentTests.setUp(self, 1)
127         expected_student_id = f'{student.student_id}'
128         self.assertEquals(expected_student_id, 100000)
129
130     # Tests whether student_name is stored in database
131     def test_name(self):
132         student = Student.objects.get(id=1)
133         expected_student_name = f'{student.student_name}'
134         self.assertEquals(expected_student_name, 'john')
135
136     # Tests whether student_surname is stored in database
137     def test_surname(self):
138         student = Student.objects.get(id=1)
139         expected_student_surname = f'{student.student_name}'
140         self.assertEquals(expected_student_surname, 'john')
141
142     # Tests whether student_current_level is stored in database
143     def test_current_year(self):
144         student = Student.objects.get(id=1)
145         expected_current_year = f'{course.student_current_level}'
146         self.assertNotEquals(expected_current_year, 1)
147         self.assertNotEquals(expected_current_year, 1)
148         self.assertEquals(expected_current_year, 3)
149
150     # Tests whether correct view is loaded
151     def test_view(self):
152         response = self.client.get(reverse('students'))
153         self.assertEqual(response.status_code, 200)
154         self.assertNotContains(response, 'just a test')
155         self.assertTemplateUsed(response, 'courses.html')
156
```

Model Tests

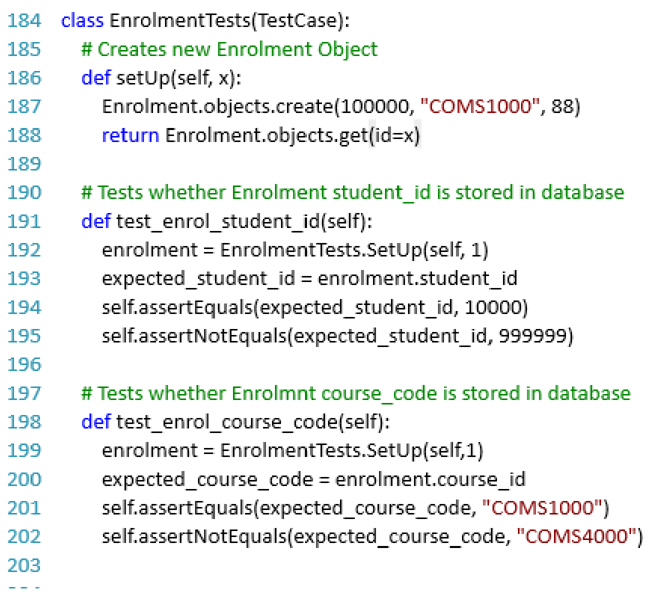
- The Student Model tests are defined in the class StudentTests - (line 118).
- The first test ensures that the student_id is stored in the database. (line 125-128).

- The second test ensures that the student_name is stored in the database. (line 131-134).
- The third test ensures that the student_surname is stored in the database. (line 137-140).
- The fourth test ensures that the student_current_level is stored in the database. (line 143-148).
- The fifth test ensures that the correct view is loaded.- (line 151-156).



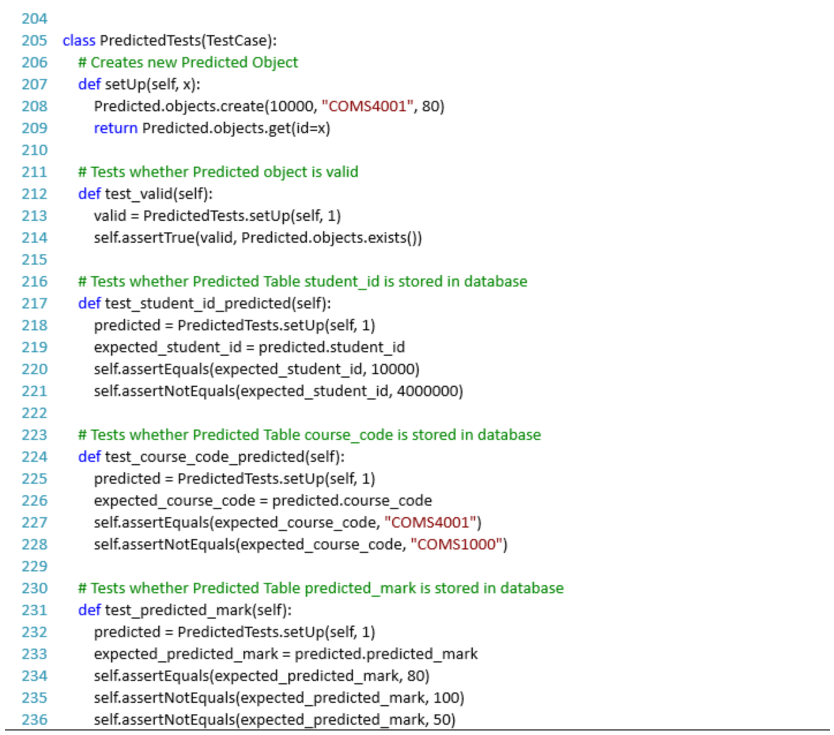
```
157
158 class CoursesTests(TestCase):
159   # Creates a Courses Object
160   def setUp(self, x):
161     Course.objects.create("COMS1000", "Computer Hardware Intro")
162     return Course.objects.get(id=x)
163
164   # Tests whether course_code is stored in database
165   def test_course_code(self):
166     course = CoursesTests.setUp(self, 1)
167     expected_course_code = course.course_code
168     self.assertEquals(expected_course_code, "COMS1000")
169
170   # Tests whether course description is stored in database
171   def test_desc(self):
172     course = CoursesTests.setUp(self, 1)
173     expected_course_desc = course.course_description
174     self.assertEquals(expected_course_desc, "Computer Hardware Intro")
175
176   # Tests whether correct view loaded
177   def test_view(self):
178     response = self.client.get(reverse('courses'))
179     self.assertEqual(response.status_code, 200)
180     self.assertNotContains(response, 'just a test')
181     self.assertTemplateUsed(response, 'courses.html')
182
183
```

- The Course Model tests are defined in the class CourseTests - (line 158).
- The first test ensures that the course_code is stored in the database. (line 165-168).
- The second test ensures that the course_desc is stored in the database. (line 171-174).
- The third test ensures that the correct view is loaded.- (line 177-181).



```
184 class EnrolmentTests(TestCase):
185     # Creates new Enrolment Object
186     def setUp(self, x):
187         Enrolment.objects.create(100000, "COMS1000", 88)
188         return Enrolment.objects.get(id=x)
189 
190     # Tests whether Enrolment student_id is stored in database
191     def test_enrol_student_id(self):
192         enrolment = EnrolmentTests.SetUp(self, 1)
193         expected_student_id = enrolment.student_id
194         self.assertEquals(expected_student_id, 10000)
195         self.assertNotEquals(expected_student_id, 999999)
196 
197     # Tests whether Enrolmnt course_code is stored in database
198     def test_enrol_course_code(self):
199         enrolment = EnrolmentTests.SetUp(self,1)
200         expected_course_code = enrolment.course_id
201         self.assertEquals(expected_course_code, "COMS1000")
202         self.assertNotEquals(expected_course_code, "COMS4000")
203 
```

- The Enrolment Model tests are defined in the class EnrolmentTests - (line 184).
- The first test ensures that the student_id (foreign key) is stored in the database. (line 191-195).
- The second test ensures that the course_code is stored in the database. (line 198-202).



```
204
205 class PredictedTests(TestCase):
206     # Creates new Predicted Object
207     def setUp(self, x):
208         Predicted.objects.create(10000, "COMS4001", 80)
209         return Predicted.objects.get(id=x)
210
211     # Tests whether Predicted object is valid
212     def test_valid(self):
213         valid = PredictedTests.setUp(self, 1)
214         self.assertTrue(valid, Predicted.objects.exists())
215
216     # Tests whether Predicted Table student_id is stored in database
217     def test_student_id_predicted(self):
218         predicted = PredictedTests.setUp(self, 1)
219         expected_student_id = predicted.student_id
220         self.assertEquals(expected_student_id, 10000)
221         self.assertNotEquals(expected_student_id, 4000000)
222
223     # Tests whether Predicted Table course_code is stored in database
224     def test_course_code_predicted(self):
225         predicted = PredictedTests.setUp(self, 1)
226         expected_course_code = predicted.course_code
227         self.assertEquals(expected_course_code, "COMS4001")
228         self.assertNotEquals(expected_course_code, "COMS1000")
229
230     # Tests whether Predicted Table predicted_mark is stored in database
231     def test_predicted_mark(self):
232         predicted = PredictedTests.setUp(self, 1)
233         expected_predicted_mark = predicted.predicted_mark
234         self.assertEquals(expected_predicted_mark, 80)
235         self.assertNotEquals(expected_predicted_mark, 100)
236         self.assertNotEquals(expected_predicted_mark, 50)
```

- The Predicted Model tests are defined in the class PredictedTests - (line 184).
- The first test ensures that the predicted object which was created is valid.- (line 212-214).
- The second test ensures that the student_id (foreign key) is stored in the database. (line 217-220).
- The third test ensures that the course_code(foreign key) is stored in the database. (line 224-228).
- The fourth test ensures that the predicted_mark is stored in the database. (line 231-236).

7 Gather and Prepare Data

Task	Task Description	Tools Used and Description	Challenges	How were challenges fixed
Generate Data	Artifitial data was generated using courses from the faculty information booklet.	Python write(), Libre Office Spreadsheets(store generated data), import random.	Artificial data results in poor model performance, lack of cohesion.	Artificial data is relatively sufficient to test and run our prototype, real data is needed in later versions.
Feature Extraction	The model only needs course symbols as features and the target being the grade to be predicted	A python script was implemented to access desired features and pass the features to the Data processor.	The code for extracting features was order n which took time to execute.	The features were extracted manually from a file to a separate file thus the training data was in its own file
Data Preprocessing	All the feature variables were converted from symbols to numerical variables.	import Pandas, used the pandas library to replace symbols to numerical variables	The processed data had NULL variables.	All the NULL variables were replaced with a flag numerical value (-1).

8 Training the Model

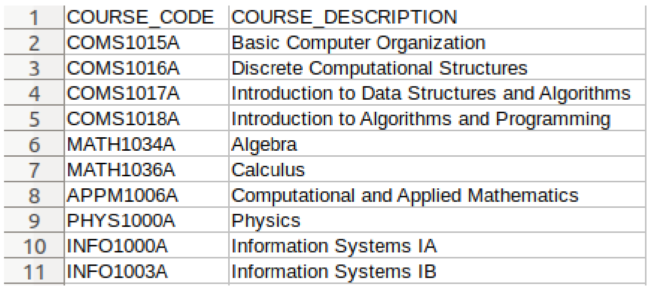
Task	Task Description	Tools Used and Description	Challenges	How were challenges fixed
Decision Algorithm	A Decision Tree algorithm was chosen as the preliminary machine learning technique. This was done to give a provisional method for predicting the recommended courses as well the expected grade for each course.	From the Scikit learn library, the built in Decision algorithm was utilized to decide whether a student will get good grades for courses that will soon be recommended, if the grades are indeed good the specific courses will be recommended.	The trained model did not perform in its optimal performance, because the data used to train the model was biased that is , there was no cohesion amongst features.	Artificial data is relatively sufficient to test and run our prototype, real data is needed in later versions.

9 Detailed description of the dataset

9.1 Raw Data

- The raw data is structured in a .csv file. It consists of 1000 students who did their undergraduate courses.

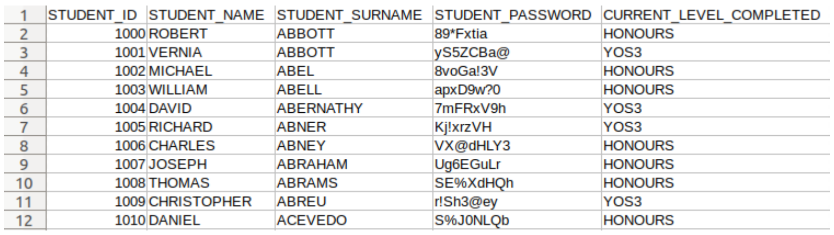
- The following files : FIRST_YEAR.csv, SECOND_YEAR.csv, THIRD_YEAR.csv, HONOURS.csv and MASTERS.csv have all the course codes and course descriptions for computer science as stated in the faculty of science information booklet.



1	COURSE_CODE	COURSE_DESCRIPTION
2	COMS1015A	Basic Computer Organization
3	COMS1016A	Discrete Computational Structures
4	COMS1017A	Introduction to Data Structures and Algorithms
5	COMS1018A	Introduction to Algorithms and Programming
6	MATH1034A	Algebra
7	MATH1036A	Calculus
8	APPM1006A	Computational and Applied Mathematics
9	PHYS1000A	Physics
10	INFO1000A	Information Systems IA
11	INFO1003A	Information Systems IB

Sample of FIRST_YEAR.csv

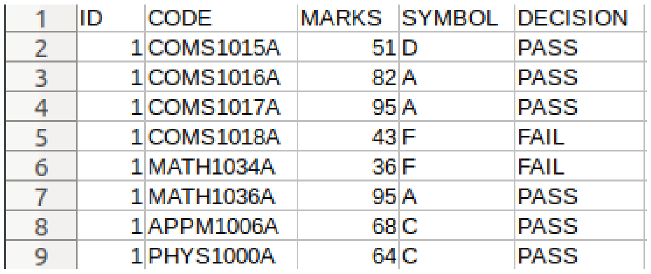
- The STUDENTS.csv contain the details of each user(student).



1	STUDENT_ID	STUDENT_NAME	STUDENT_SURNAME	STUDENT_PASSWORD	CURRENT_LEVEL_COMPLETED
2	1000	ROBERT	ABBOTT	89*Fxtia	HONOURS
3	1001	VERNIA	ABBOTT	yS5ZCBa@	YOS3
4	1002	MICHAEL	ABEL	8voGa!3V	HONOURS
5	1003	WILLIAM	ABELL	apxD9w?0	HONOURS
6	1004	DAVID	ABERNATHY	7mFRxV9h	YOS3
7	1005	RICHARD	ABNER	Kj!xrzVH	YOS3
8	1006	CHARLES	ABNEY	VX@dHLY3	HONOURS
9	1007	JOSEPH	ABRAHAM	Ug6EGuLr	HONOURS
10	1008	THOMAS	ABRAMS	SE%XdHQh	HONOURS
11	1009	CHRISTOPHER	ABREU	r!Sh3@ey	YOS3
12	1010	DANIEL	ACEVEDO	S%J0NLQb	HONOURS

Sample of STUDENTS.csv

- The following files : FIRST_YEAR_SUB1.csv, SECOND_YEAR_SUB1.csv, THIRD_YEAR_SUB1.csv, HONOURS_SUB1.csv contain all the students transcripts.



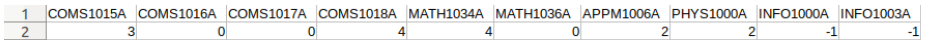
1	ID	CODE	MARKS	SYMBOL	DECISION
2	1	COMS1015A	51	D	PASS
3	1	COMS1016A	82	A	PASS
4	1	COMS1017A	95	A	PASS
5	1	COMS1018A	43	F	FAIL
6	1	MATH1034A	36	F	FAIL
7	1	MATH1036A	95	A	PASS
8	1	APPM1006A	68	C	PASS
9	1	PHYS1000A	64	C	PASS

Sample of FIRST_YEAR_SUB1.csv showing results for student 1.

9.2 Normalized Data

For the purpose of training our model, the above data needs to be normalized before it is fed into the built-in sklearn Decision Tree Algorithm. Since the 'SYMBOLS' column has been chosen to be the feature for our classifier, we need to convert all the symbols to numerical values. That is, A $\rightarrow 0, B \rightarrow 1, C \rightarrow 2, D \rightarrow 3, F \rightarrow 4$.

If a student did not take part in a particular course, we flag the grade with -1.



1	COMS1015A	COMS1016A	COMS1017A	COMS1018A	MATH1034A	MATH1036A	APPM1006A	PHYS1000A	INFO1000A	INFO1003A
2	3	0	0	4	4	0	2	2	-1	-1

Normalized first year results for student 1.

10 Database Tables

10.1 Student Table

<u>student_id</u>:

- Description: A unique number given to every student by the university
- Type: integer
- Format: N/A
- Note: This is a primary key; no non numeric values

<u>student_name</u>:

- Description: This is the students first name
- Type: 20-character string

- Format: N/A
- Note: No numeric values

student_surname:

- Description: This is the students surname or lastname
- Type: 20-character string
- Format: N/A
- Note: No numeric values

student_current_level:

- Description: This is the students current level of study
- Type: 20-character string
- Format: N/A
- Note: Can take the value yos1, yos2, yos3,honours,masters

student_password:

- Description: this is the students login password that they chose/set
- Type: 20-character string
- Format: N/A
- Note: N/A

10.2 Course Table

course_code:

- Description: This is a courses code chosen by the university
- Type: 10-character string
- Format: [A-Z][A-Z][A-Z][A-Z][1-9][0-9][0-9][0-9][A-Z] or [A-Z][A-Z][A-Z][A-Z][1-9][0-9][0-9][0-9]
- Note: This is the primary key

course_description:

- Description: This is the course description, usually the courses name
- Type: 100-character string
- Format: N/A
- Note: N/A

10.3 Enrolment Table

<u>student_id</u>:

- Description: this is the student_id as mentioned in the Student table
- Type: integer
- Format: N/A
- Note: this is a foreign key, a reference of a Student object

<u>course_id</u>:

- Description: this is the course_code as mentioned in the Course table
- Type: 9-character string
- Format: [A-Z][A-Z][A-Z][A-Z][1-9][0-9][0-9][0-9][A-Z] or [A-Z][A-Z][A-Z][A-Z][1-9][0-9][0-9][0-9]
- Note: this is a foreign key, a reference of a Student object

<u>course_mark</u>:

- Description: this is the mark(percentage) the student obtained in a course
- Type: double
- Format: N/A
- Note: N/A

10.4 Predicted Table

<u>student_id</u>:

- Description: this is the student_id as mentioned in the Student table
- Type: integer
- Format: N/A
- Note: this is a foreign key, a reference of a Student object

<u>course_code</u>:

- Description: this is the course_code as mentioned in the Course table
- Type: 10-character string
- Format: [A-Z][A-Z][A-Z][A-Z][1-9][0-9][0-9][0-9][A-Z] or [A-Z][A-Z][A-Z][A-Z][1-9][0-9][0-9][0-9]

- Note: this is a foreign key, a reference of a Course object

predicted_mark:

- Description: this is the mark(symbol) the system predicted the student will obtain in a course
- Type: 1-character string
- Format: N/A
- Note: Can take the value A, B or C

11 Iteration Planning

11.1 Overview

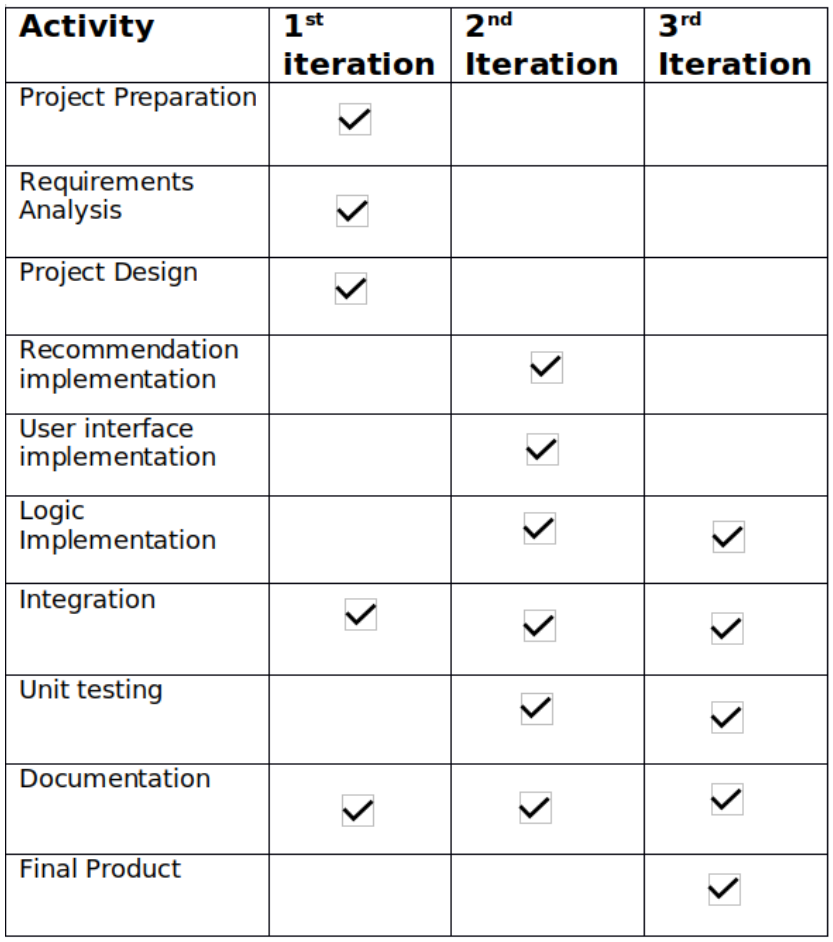
In the beginning of every iteration features are chosen for that iteration and are broken down into specific tasks that are then allocated to the team members. These tasks are also called the sprint backlog. Goals set for each sprint are used as a guideline for features that are selected for that sprint. Task take 4 hours to 4 days depending on their level of difficulty and level of knowledge of the member that the task is assigned to.

11.2 Roles

- Team leader - Mpinane Mohale
- Team member - Thulisile shipyana
- Team member - Sbusiso Mkhombe
- Team member - Lucky Mahlangu

11.3 Product Backlog

- As a student I want to be able to register on the system so that I can have access to the services provided by the systemion.
- As a student I want to be able to add a course that I have completed and also the marks obtained on them so that I can get a recommendation(s) of the courses I can take for my post graduate studies.
- As a student I want to get the predicted marks on the recommended courses so that I can decide on which courses to enroll for.



Activity	1st iteration	2nd Iteration	3rd Iteration
Project Preparation	✓		
Requirements Analysis	✓		
Project Design	✓		
Recommendation implementation		✓	
User interface implementation		✓	
Logic Implementation		✓	✓
Integration	✓	✓	✓
Unit testing		✓	✓
Documentation	✓	✓	✓
Final Product			✓

Iteration Table The table above shows a simplified activity plan that was carried out.

11.4 First Iteration

11.4.1 Objectives

This iteration aims to gather the requirements of the system and to design the system architecture together with the system interface also determining the overall feasibility of the project

11.4.2 Tasks

Documentation (all members worked on the documentation)

11.4.3 Sprint Retrospective

What worked: Goals of the sprint where achieved and everything was done on time, team collaboration What was challenging/ didnt work: having to learn the Django design patterns with limited time, generating the data for training the classifier How was it resolved: attacked the problem bit by bit, each member had to take Django tutorials in order to familiarize themselves with it.

11.5 Second Iteration

11.5.1 Objectives

In this iteration, we will focus on the implementation of three elements, namely the user interface, the login and the registration page with the functionality, and the course recommendation. We will also conduct unit test on the implemented tasks.

11.5.2 Task

As a student I want to be able to register on the system so that I can have access to the services provided by the system.

As a student I want to be able to add courses that I have completed together with the marks achieved so that I can get recommendations of the elective courses that I can take for my post graduate degree.

- Implementation of the login and registration Mpinane, Sbusiso.
- Add courses page, create the UI - Mpinane.
- Conduct unit testing - Sbusiso.
- Add the recommendation page, UI and Implementation - Mpinane.
- Generate data - Lucky Mahlangu.
- Create the recommendation function using machine learning techniques Lucky Mahlangu.

11.5.3 Acceptance Criteria

- User should be able to register.
- User should be to retrieve the saved courses.
- User should be able to delete and edit the courses.
- User should be able to get recommendations based on the added courses and their marks.

11.5.4 Sprint Retrospective

What worked: where a member struggled with a particular task, other members were able to assist or refer them where they can find help.

What was challenging/ didnt work:

- Other things took time to implement.
- Some tasks took longer than anticipated.
- Some meetings were not effective.
- Deciding on which features to implement and getting user stories right the first time.
- Members arriving late to the meetings.

How was it resolved: We broke the tasks further into smaller tasks which also helped in them being completed in the estimated time. Meetings time was reduced to an hour. What could have been done: We could have updated the documentation in line with the code changes.

11.6 Third Iteration

11.6.1 Objectives

In this iteration, we will continue to add functionality to the web pages created in the previous iteration and also to extend the course recommendation to include the marks predictions. Unit test was also conducted for the code implemented in this iteration.

11.6.2 Tasks

As a student I want to get the predicted marks on the recommended courses so that I can decide on which courses to enroll for.

- Implement the marks prediction - Lucky.
- Unit testing Sbusiso.
- Account Settings. UI and Implementation - Mpinane

11.6.3 Acceptance Criteria

- User should be able to get the predicted mark of the recommended courses.
- All the web pages should have the expected functionality, i.e. the user should be directed to the page they have requested.
- User should be able to update their account.

What worked: team collaboration, incorporating lessons learned in the previous iterations in to this one.

What was challenging/ didnt work: some members were not committing small changes.

How was it resolved: we introduced a rule that members have to commit no matter how small the commits.

What could have been done: We could have used a planning tool to track our progress, task time estimation.

11.6.4 Definitions

Product Backlog - these are the features to be implemented for the whole project, they are in a form of a user story, i.e. the sentence inside the " ".

User story is a tool used to capture a description of a feature from an end-user perspective, format: As a ¡user type¿, I want ¡goal¿ so that ¡ reason¿.

Sprint backlog - is a subset of the product backlog, that is chosen by the team and broken down into manageable tasks and prioritized and completed in the sprint.

Acceptance criteria - is when a task item is complete and working as expected.