CS390 Python - Final Project

Writing a Capstone Project Marketplace in Python/Django

Grading Form

Turn In

- You will be using Github for this project. You should fork the Starter Project, and share it with **cs390python**.
- Tasks 0-2 are due **Monday, November 14th at 11:59pm.** Tasks should be committed to your repository you forked from the Starter Project.
- Final presentations are **Monday, December 5th.** All work should be **regularly committed** to GitHub.
- **You will work in teams of 3 students.** Record your team members here by: **Monday, November 14th at 11:59pm**
  https://docs.google.com/document/d/1fcOe1KOr2oHvOqT8LJc_qXstXeC7NgYFg1ZaFsVJZHE/edit?usp=sharing

Goal

Project Goal

A capstone project, or also called a senior design project, is one that some students do in the last year of their Computer Science major. The goal of this project is to implement a marketplace for students/groups to find creative and challenging capstone projects to work on. The projects will be usually provided by industry engineers. Our goal is to provide an easy-to-use system that students, instructors, and industry engineers can collaborate with to make the process of finding, selecting, and completing capstone projects much smoother for everyone.

General Goal of learning python…

Python and Django are commonly used web development technologies that, once learned, provide a good medium to effectively design and develop web applications with databases. These development principles can also be used with many other web application frameworks, as well as integrated with technologies such as IBM Watson.

Teams

This will be a team project with a maximum of three members. You may work individually but it is recommended that you work in a team. Teams should record their team members here. All students are individually responsible for doing the Django Setup, and Task 2. You may use any team members repo for Task 3.

Task 0: Django Setup

Before we get started, you will need to install a few Python packages to use Python and Django. You may use your own computer or a lab computer like data.cs.purdue.edu to do the Django Setup.

1. **Python (v3.5.2)** (https://python.org)
   Install python version 3.5.2 from the download page.
2. **Pip** (https://pypi.python.org/pypi/pip)
   It is a package management system used to install and manage software packages written in Python.

We will be using it to install Django.

3. **Virtualenv** (https://virtualenv.pypa.io/en/stable/)

Any change in an application's libraries or the versions of these libraries can break the application. To prevent this, we use virtualenv to create a virtual isolated environment for our Django server to run in. Any change in the libraries of the host computer will not affect our server's environment.

4. **Django (v1.10.2)** (https://www.djangoproject.com/)

Django is a free and open-source web framework, written in Python, which follows the model-view-template (MVT) architectural pattern.

Installing python

Python is already installed in data.cs.purdue.edu and in MacOS. In Windows computers you will have to install python from python.org/downloads.

Installing pip

(If installing on a Purdue CS lab machine)

```
wget --no-check-certificate https://bootstrap.pypa.io/get-pip.py -O - | python - --user
echo "PATH=\$PATH:~/.local/bin" >> ~/.bashrc
source ~/.bashrc
```

IMPORTANT: Make sure to include --user (pip install --user __) when installing anything on the lab machines, or it won't work (since you don't have admin) :)

(If installing on an Ubuntu machine)

```
sudo apt-get -y install python-pip
```

(If installing on an Mac)

```
sudo easy_install pip
```

(If installing on Windows)

```
Follow the instructions from https://pip.pypa.io/en/latest/installing/
```

Installing virtualenv

(If installing on a Purdue CS lab machine)

```
pip install --user virtualenv
```

(If installing on an Ubuntu machine or Mac)

```
sudo pip install virtualenv
```

Setting up a new virtual environment and activation

```
virtualenv <name_of_folder>
cd <name_of_folder>
source bin/activate
```

See the effects of virtualenv. Run this command **before and after** you run 'source bin/activate'

```
pip freeze
```

Install Django

Lorem Inside your activated virtual environment, run this command

```
pip install Django==1.10.2
```

Making sure Django is installed correctly

```
django-admin startproject <name_of_project>
cd <name_of_project>
python manage.py migrate
python manage.py runserver 8080
```

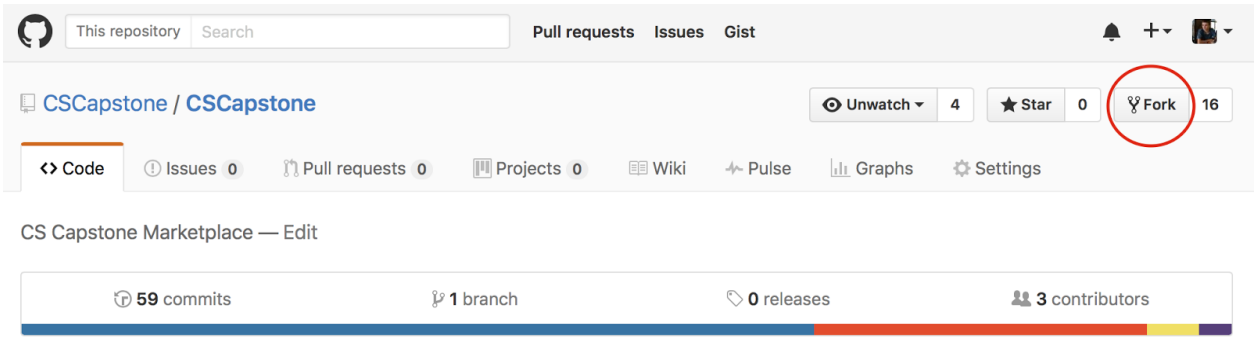
Open a web browser and go to 127.0.0.1:8080

Task 1: Introduction to the Starter Project

In this task, you will get and become familiar with the CS Capstone starter project.

Getting the Starter Project

We will be using GitHub for this project . You will fork the git repository at: https://github.com/CSCapstone/CSCapstone . To fork the git repository , click the “Fork” button on the repository page. You should also add **cs390python** as a collaborator on your forked repo.



For more info on forking, visit: https://help.github.com/articles/fork-a-repo/

After forking, you’ll need to clone your forked repo to get the files on your local machine. To do this, copy the url for the repo (e.g. http://github.com/<your_github_name>/CSCapstone) and run

```
git clone <url>
```

Whats Included?

A few modules have been included in the starter project to provide some basic functionality . A list and description of these modules can be found below .

Modules included in Starter Project

1. CSCapstone: Django Root Module (contains primary urls and settings config)
2. CSCapstoneApp: Project Root Module (contains home page)
3. AuthenticationApp: Handles account creation, login, and logout for our project
4. CompaniesApp: Handles company creation, management, and profiles.
5. GroupsApp: Handles group creation, management, and profiles.
6. ProjectsApp: Handles project creation, management, and profiles.
7. UniversitiesApp: Handles university and class creation and management.

Task 2: Introduction to Django: Creating a new app and building a comment system (10 pts)

In this task you will be adding a new Django app to our starter Capstone project, as well as creating new pages in the app.

First, let's take a look at manage.py . Manage.py is a extension of the django-admin command-line utility but takes care of some nitty gritty stuf f for you. Read more here.

Go ahead and run the following command and look at the output:
(Make sure you are under the CSCapstone directory)

```
python manage.py
```

This command will display all available commands you can use with manage.py . Here's the important ones to know for now:

- makemigrations: Generates migrations
- migrate: Runs all migrations
- startproject: Creates a new Django project
- startapp: Creates a new Django app
- runserver: Runs the server
- createsuperuser: Generates a new super user

You should've already used "startproject" and "runserver" in T ask 0. These two are pretty self-explanatory , but "python manage.py startproject <name>" will create a new project <name> in a new folder inside the current directory. You can then use "runserver" to run your web application (and access it at localhost:8000 by default).

The "createsuperuser" command will allow you to create an admin user that can access the built-in admin pages for your Django models. T ype in the following command and follow the prompts to create a new superuser that you can use to access the information in the Django database.

```
python manage.py createsuperuser
```

Now you can access the admin pages for your Django database by going to localhost:8000/admin and logging in with your superuser credentials. Make sure your application is running when you do this!

Now we'll look at the "startapp" command. Similar to the "startproject" command, the "startapp" command will create a new app inside of the project you're currently working in. Run the following command:

```
//Inside CSCapstone Root
python manage.py startapp CommentsApp
cd CommentsApp
```

In this task we're going to be creating an application that is just a basic comment system. Once you're in the CommentsApp directory you'll be able to see the files that the "startapp" command created. The important ones to note are "views.py" and "models.py". When making our Django app we will be defining the routing to our html files and such in "views.py" and our models (database stuf f) in "mode ls.py".

Before we begin to add anything to our new app, however , we need to update a few things in our project' s "urls.py" and "settings.py" files. Navigate back to the main project directory and then "cd" into your "CSCapstone" folder .

```
cd ..                       #cd into project's root
cd CSCapstone               #cd into settings directory
```

Now, open up the "urls.py" file and find the following snippet of code:

```
from django.conf.urls import url, include
from django.contrib import admin
```

```
urlpatterns = [
    url(r'^admin/', admin.site.urls),
    ...
]
```

What this does is tells our web project where to look for urls so that requests to our server can be properly fulfilled. Change the snippet from above so that it includes urls from our new app inside our project:

```
from django.conf.urls import url, include
from django.contrib import admin

urlpatterns = [
    url(r'^admin/', admin.site.urls),
    ...
    url(r'^', include('CommentsApp.urls')),
]
```

Now whenever someone using our web application requests a specific page it will look in our "CommentsApp/urls.py" file to figure out what to do. Note that the server will check for urls in descending order, so if you have an "index.html" in both the original project app and our ne w CommentsApp then it will take the original "index.html" over our new one. This is best fixed by just keeping .html files named dif ferently. More information on the Django URL dispatcher is here.

In the same directory we have our "settings.py" file. Go ahead and open this file and find the following snippet:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    ...
    'UniversitiesApp',
]
```

This is a list of all of the installed apps that our web application will be using. Having our project set up as a sort of collection of "sub-applications" allows it to be modular and for changes to be made more easily . Edit the code above to include our new CommentsApp in our project:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    ...
    'UniversitiesApp',
    'CommentsApp',
]
```

Now we're all set and ready to begin working with our new app!

First, let's give our app a fresh new loo k. The first thing we will need to do is create our app' s first html page. This page will just be a simple home page for our application. Let' s quickly set up some directories so that we can stay organized.

Inside your "CommentsApp" directory , create a new folder called "templates". This folder will be the home for any .html files that we have that are used by our particular app (in this case, the CommentsApp).

Our project uses templates to more easily create pages of our app so that they all have a similar look. Quickly look at the 'base.html' file in the 'CSCapstone/templates' folder . This file will serve as our homepage (similar to how index.html is typically used) but also as a base for our other .html files we will be creating. This makes creating new .html files much simpler .

Now, before we're able to run our application and view the page we need to a dd a route to our page in a file called "views.py". Create this new file in our "CommentsApp" folder and open it for editing. Inside "views.py", paste the following code:

```
from django.shortcuts import render

from . import models
#from . import forms

# Create your views here.
def getComments(request):
    return render(request, 'comments.html')
```

What this does is defines a function we can call when our server receives a request for "localhost:8000/comments". Note that the function name "getComments()" is not a standard name, it could be anything. It's best to keep your function names descriptive, however , so that you can keep track of which function is doing what. In this case our "getComments()" function is simply rendering the "comments.html" in the user's web browser.

We still have one thing left to do, which is define when your server should call our "getIndex()" function! Inside your "CommentsApp" folder create a new file "urls.py" and paste the following code inside:

```
from django.conf.urls import url

from . import views

urlpatterns = [
    url(r'^comments$', views.getComments, name='Comments'),
]
```

This code is similar to the "urls.py" we edited earlier , but this one is specific to our CommentsApp. What this code does is define URL regular expressions that the Django URL dispatcher will check the requested URL against. If there is a match, it will execute the function provided as the second argument to url(). Here, we are importing our "views.py" file and using the "getComments()" function we defined previously whenever a user requests "localhost:8000".

Let's start making our "comments.html" page. Create a new .html file inside our CommentsApp' s template folder: we'll call it "comments.html". Open up our new file and put the following code in:

```
{% extends "body.html" %}
{% block content %}

    <!-- Content -->
    <div class="container" role="main">
        <div class="table-responsive">
            <table class="table table-striped sortable">
                <thead>
                    <tr>
                      <th style="width: 20%">Time</th>
                        <th style="width: 80%">Comment</th>
                    </tr>
                </thead>
                <tbody>
                   {% for item in comments %}
                   <tr>
                       <td>{{ item.time }}</td>
                       <td>{{ item.comment }}</td>
```

```
                </tr>
                {% endfor %}
            </tbody>
        </table>
    </div>
</div>

{% endblock %}
```

You may be wondering what some of this code is, as not all of it looks like htm l. The very first line of our new .html file simply states that this file is an extension of our “base.html” file that we created earlier . By being an extension we are able to keep the same format as our original page, but change certain portions of it to suit the needs of our new page. Here we are using **blocks** to replace the “content” portion of “base.html” with our content block in “comments.html”.

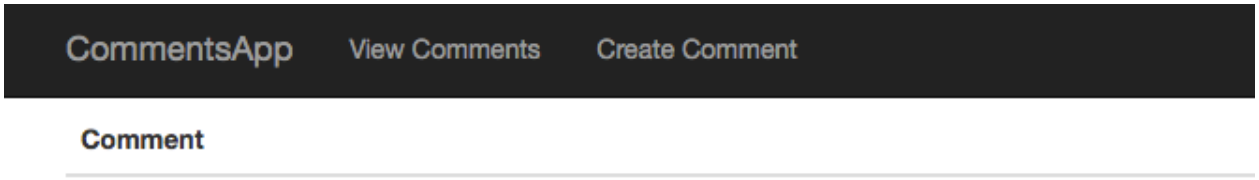
```
{% block content %}
    <!-- Content -->
    <div class="container theme-showcase" role="main">
        <div class="jumbotron">
            <h1>Our first Django app!</h1>
            <p>Hello World!</p>
        </div>
    </div>
{% endblock %}
```

This chunk of code is what our original content block contains. By overriding it in our “comments.html” we display the block specified in “comments.html” instead.

```
<div class="table-responsive">
    <table class="table table-striped sortable">
         <thead>
            <tr>
                <th style="width: 20%">Time</th>
                 <th style="width: 80%">Comment</th>
            </tr>
         </thead>
         <tbody>
            {% for item in comments %}
            <tr>
                <td>{{ item.time }}</td>
                <td>{{ item.comment }}</td>
            </tr>
            {% endfor %}
        </tbody>
    </table>
</div>
```

Now, the above portion of code (“comments.html”) specifies a table that will ho ld the comments made by users. The “thead” tag allows us to specify the header and columns for our table and the “tbody” lets us specify the values that will go in each of the table entries. When a request is made for our “comments.html” file we will send a “comments” parameter that our .html file will use to fill in the table with information. The code in our “tbody” will iterate through every entry in the comments array and will place the information “item.time” and “item.text” in our table.

Save your “comments.html” file, run the server , and check out the result at localhost:8000/comments . You should see a page like this (with a dif ferent nav bar):



If you see a neatly formatted but empty table then you're on the right track. Now , we need a way to store and display the comments that users make! For this, we'll use models.

We will need to define a new model for our app. With Django each model you define creates a new table to store all the database entries of each model with the needed fields. For example, if you defined a "Person" model with the fields "Name" and "Age" you could imagine the table looking like this when it has three entries:

Name	Age
Phil	27
Sally	24
Bob	33

So, for our CommentsApp we're going to need a model that stores the comments that users create! Navigate to and open your "CommentsApp/models.py" file. Here' s how the file should lo ok:

```
from django.db import models

# Create your models here.
```

Where it says "Create your models here" is where we'll define our Comment model! Put the following snippet into the "models.py" file:

```
class Comment(models.Model):
    time = models.DateTimeField(auto_now=True)
    comment = models.CharField(max_length=500)
```

We have now defined our Comment mo del. A quick explanation of the code…

Line 1 - defines our new Comment model as an extension of the models.Model class
Line 2 - gives our Comment model a field that will act as a time stamp
Line 3 - gives our Comment model a character field with a maximum size of 500

Now we have a new table in our database that we can use to store Comments. Whenever you make changes to your models be sure to run the following commands:

```
python manage.py makemigrations
python manage.py migrate
```

Migrations are Django' s way of propaga ting changes from the models to the database. It comes in handy when we want to make changes to a database schema with existing data, without dropping the table every time we change things.

Next, we need to make a way for users to make their comments! Let' s create a new .html file in our templates folder called "commentForm.html". On this page we'll have a form that users can use to submit comments to our application. Inside our new .html file, put the following code:

```
{% extends "body.html" %}
```

```
{% block content %}
<!-- Content -->
<div class="container" role="main">
    <div class="panel panel-default">
        <form method="post" enctype="multipart/form-data" action="/addcomment" class="panel-
body validate">
        {% csrf_token %}
             <label for="comment">Comment</label>
             <input type="text" name="comment" id="comment" placeholder="Comment" class="form-
control" data-bvalidator="required" data-bvalidator-msg="Comment Required">
             <br>
             <input type="submit" value="Submit" class="btn btn-primary">
        </form>
    </div>
</div>
{% endblock %}
```

This file also extends our “base.html” file and replaces the content block within it. This form has a text field for the comment itself and a submit button. When the user clicks the submit button the form will perform the specified “action”, which in this case is a redirect to “/addcomment”.

In our “urls.py” file we will need to add two new entries:

```
url(r'^commentform$', views.getCommentForm, name='CommentForm'),
url(r'^addcomment$', views.addComment, name='AddComment'),
```

The first entry will be our route for our “commentForm.html” page and the second is the route for when a comment form is submitted. Now we need to add the getCommentForm() and addComment() functions to our “views.py” file.

```
def getCommentForm(request):
    return render(request, 'commentForm.html')

def addComment(request):
    if request.method == 'POST':
        form = forms.CommentForm(request.POST)
        if form.is_valid():
            new_comment = models.Comment(comment=form.cleaned_data['comment'])
            new_comment.save()
            comments_list = models.Comment.objects.all()
            context = {
                'comments' : comments_list,
            }
            return render(request, 'comments.html', context)
        else:
            form = forms.CommentForm()
    return render(request, 'comments.html')
```

Our “addComment()” function does the following:

- Checks if the request method is POST (that is, there is data being sent from the form)
- Checks the request data against a specified form (we will define this in a moment)
- If the form is valid, we will create a new entry in our Comment table and save it
- Finally, it will render the “comments.html” page where the user will be able to s ee their submitted comment.

As you can see, our “addComment()” function uses a form to validate the information sent. First, we need to make sure we can access the form by including the following code at the top of “views.py”:

```
from . import forms
```

Now we’ll to make a “forms.py” file to define our form in. Create a new file name “forms.py” and put the following code inside:

```
from django import forms

class CommentForm(forms.Form):
    comment = forms.CharField(label='Text', max_length=500)
```

Now, “views.py” will use this form to check the data sent by the html form.

We also need edit our “getComments() ” function in “views.py” so that rather than simple rendering a static page it actually renders the page with the comments from our database. Change your “getComments()” function so that it looks like the following:

```
# Create your views here.
def getComments(request):
    comments_list = models.Comment.objects.all()
    context = {
        'comments' : comments_list,
    }
    return render(request, 'comments.html', context)
```

Now the app should be ready to go! Go ahead and run the app and check that both localhost:8000/comments and localhost:8000/commentform work properly. If they do, you’re all done with creating this basic CommentsApp! For practice, consider how you can improve our new app and check out the Bonus section below .

Bonus: Nested Comments (5 pts)

Extend your CommentsApp to allow nested comments, similar to what you might find on Facebook, Y outube, and Reddit. Users should not only be able to leave a comment in the thread, but also reply to other users comments as a sub-comment.

This will require:

- Modifying your comment model to support nested comments
- Modifying your addComment logic to support nested comments.
- Modifying your comments.html template to correctly display nested comments. Y ou will also need to create a method for the user to reply to any comment. This can be accomplished with many <form> elements, or using Javascript to cleverly render a reply form and indicate what comment is parent.

Task 3 (90 pts) on next page

Task 3: CS Capstone Portal (90 pts)

In this task you will be extending the starter project to build a CS Capstone Portal using Django. Please use a single repository for your team, and remember to commit frequently .

Capstone Portal: User Stories

The Capstone Portal serves as a platform for industry engineers to collaborate and share projects with academic students working on group projects. Students will be able to view , select, and post updates on projects posted by the industry engineers.

Below are the User Stories for the CS Capstone Marketplace. T ry to cover as many of the user stories as possible while still covering them well.

Users: students, group (group of student(s)), teachers, engineers, admin, bystander

1. As a user, I can create an account
2. As a user, I can view available projects
3. As a user, I can bookmark projects
4. As an engineer, I can post projects
5. As a student, I can join groups
6. As a group, I would like a matching system to assist me in picking a project
7. As a group, I can apply for a project
8. As a group, I can select a project
9. As a group, I can post my progress on a project
10. As a group, I can post my completed project
11. As a teacher, I can view my students
12. As a teacher, I can view my students projects and their status
13. As an engineer, I can view my projects and their status
14. As an engineer, I can view students working on my projects
15. As a privileged user, I can comment on the projects
16. As an admin, I can view and edit all projects
17. As an admin, I can view and edit all users

Final Project Grading Form

A list of project features, and the point distribution for completing each, can be found on the grading form. Descriptions of each feature are included below .
https://docs.google.com/document/d/1owkuHpkWHiZVT_yX7PPE0SZMQ5wUxfQFdkWwn8CsJHo0/edit?usp=sharing

Task 3 Features

A description of each of the features, and tips to complete each, is included below . Point distribution can be found on the grading form.

3.1	**Teachers**
	• Create “Teacher” model • Create Teacher Profile (university contact info, etc) • Allow creation of Teacher users • Teacher profiles can be edited by the associated teacher, or any site admin.

3.2	**Engineers**

- Create “Engineer” model
- Create Engineer Profile (Alma Mater, About, Contact Info, etc)
- Allow creation of Engineer users
- Engineer profiles can be edited by the associated teacher, or any site admin.

3.3	**Universities and Classes**

- Create “University” and “Class” models.
- All students and teachers should belong to 1 University, and can belong to multiple classes.
- Teachers can create classes.
- Teachers can manage who is enrolled in their class (add/remove students)

3.4	**Groups**

- Any student can create a group.
- Group members may be added to the group by any current group member, by entering their email address.
- A group can either be assigned to 1 project **(Assigned)**, or not assigned to any project **(Unassigned)**. A group may not be assigned to more than 1 project. For students to work on multiple projects, they should create multiple groups.

3.5	**Projects**

- Projects can be only created by engineer/corporate users.
- Projects are visible to everyone.
- Projects should be editable by users of the company which posted the project.
- Projects should have a list of qualifications, which are used for matching. At minimum, qualifications should include:
    - Programming Language Required
    - Years of Experience Required
    - Speciality (for example: Windows, Mac, iOS, Android, Web, Computer Vision, AI)

3.6	**Bookmarks**

- Create the “Bookmark” model, which relates user_id and project_id.
- Add and implement a button on the project page to “Bookmark” a project. Once bookmarked, the button should function as an “Unbookmark” button.
- Add and implement a bookmark page or list.
- All users should be able to bookmark projects.

3.7	**Matching System**

- Implement a group -> project matching system.
- Your system should intelligently suggest projects which fit the combined student properties of each group.

3.8	**Group Profiles**

Each group should have a profile page containing:

- Team Member Details
- Chosen Project (Include brief overview pulled from Project Model, and link to actual company project profile page)
- Comments Section: Group members, their teachers, and the project corporate users may post comments on the group profile page, allowing discussion. (Hint: Use what you learned from Task 2.)

3.9	Deletion

Allow privileged users to delete groups, projects, classes, and comments.

A privileged user is any user who should have permission to perform the action. In this case:

- Groups: May be deleted by group members, or site admin
- Projects: May be deleted by users of company who created, or site admin.
- Classes: May be deleted by teachers, or site admin.
- Comments: May be deleted by poster, or site admin.

3.10	WYSIWYG

Use a WYSIWYG editor for any multi-line textarea (user profile about, group profile comments, etc). There are many WYSIWYG options available, just google "WYSIWYG editor". One options is TinyMCE

Verify that you can post feature-rich content to the page using your WYSIWYG editor. Additional changes will be necessary to your django backend and templates.

When implementing a WYSIWYG editor, be sure to manage security vulnerabilities you may be introducing to your site, such as XSS.