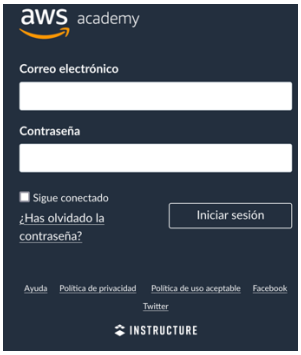
Grau Bioinformàtica
Curs 2024-2025
Distributed systems and web development
Django app lab – AWS Cloud9 version
Based in Django documentation – Release 2.2.29

1. Work environment preparation

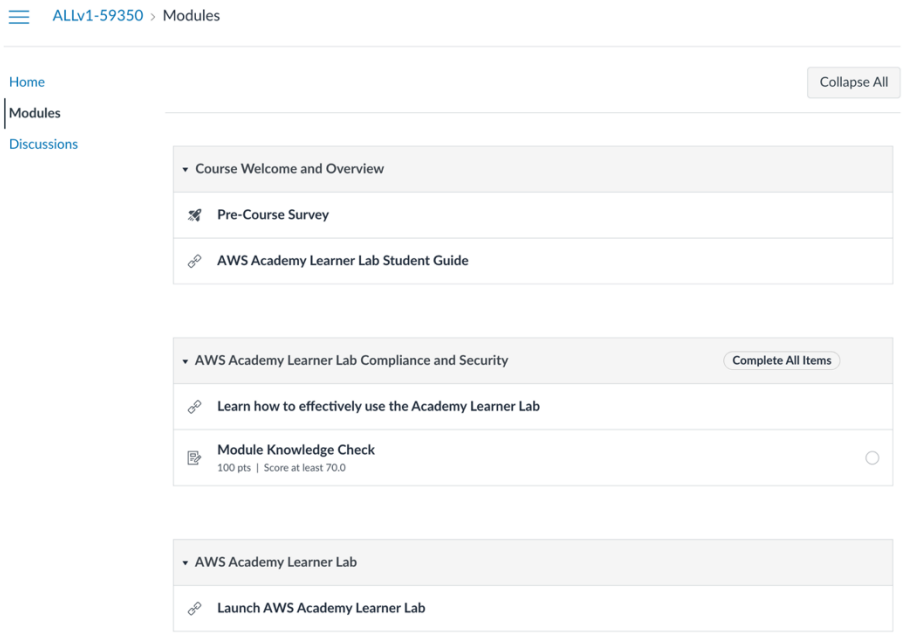
Django is a Python-based web development framework to simplify common development tasks. This tutorial is an overview of how to write a simple web app with Django.

First, we need to start a new AWS Academy Learner Lab by login into AWS Academy portal:

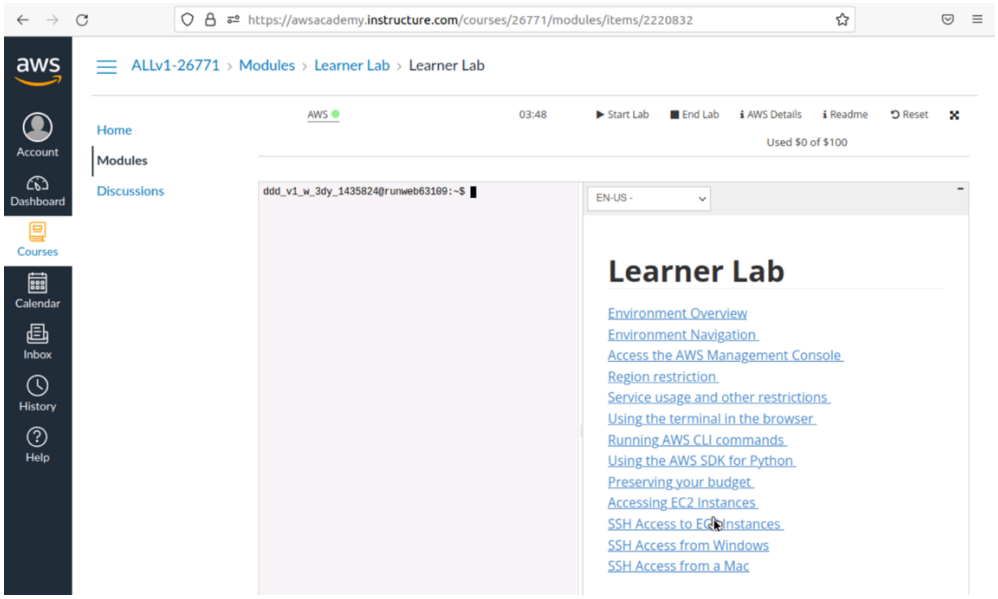
https://awsacademy.instructure.com/login/canvas



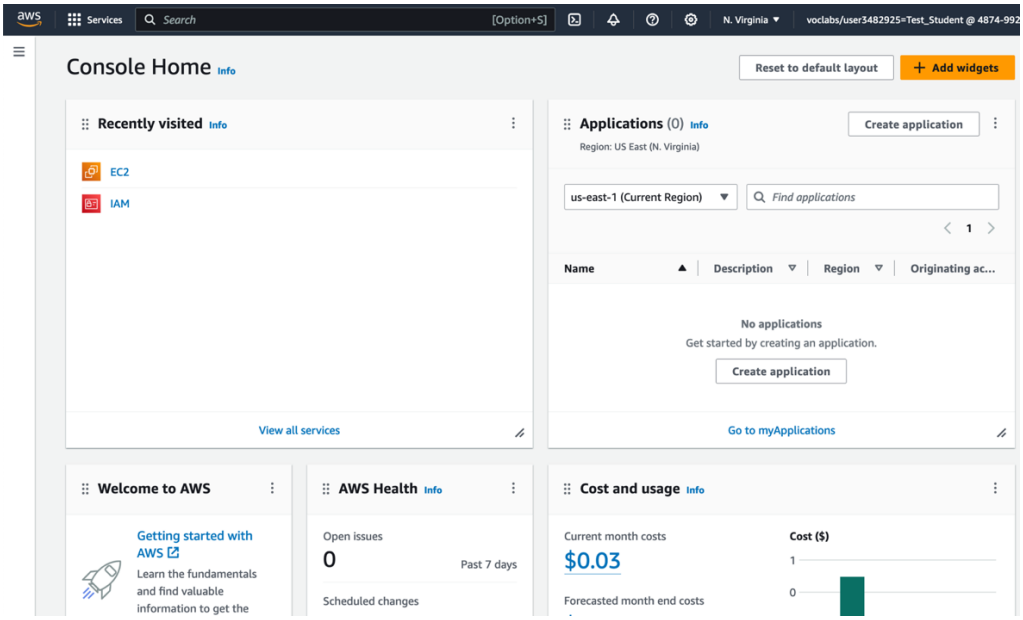
Enter the **AWS Academy Learner Lab** course and go to **modules** option in the left menu. Scroll down to the bottom of the list of modules to find AWS Academy Learner Lab and click on the Launch AWS Academy Learner Lab link.



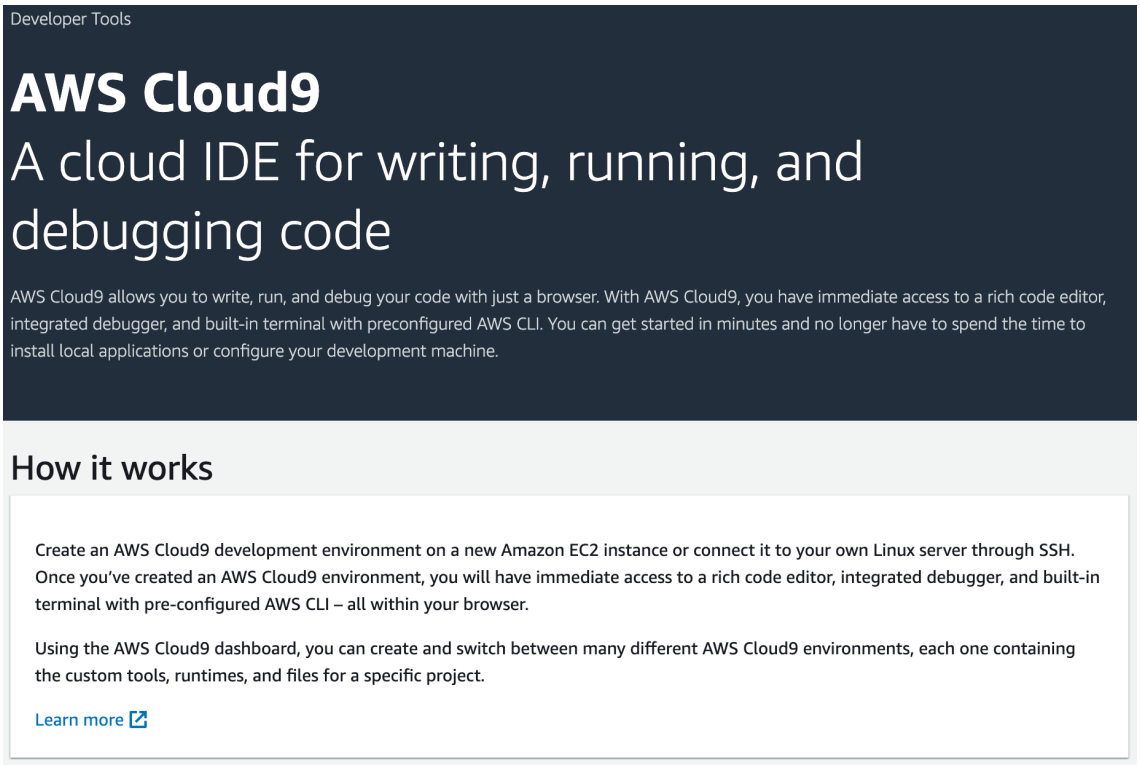
Now you must start your lab using the "Start Lab " button and wait for AWS bullet to turn green.



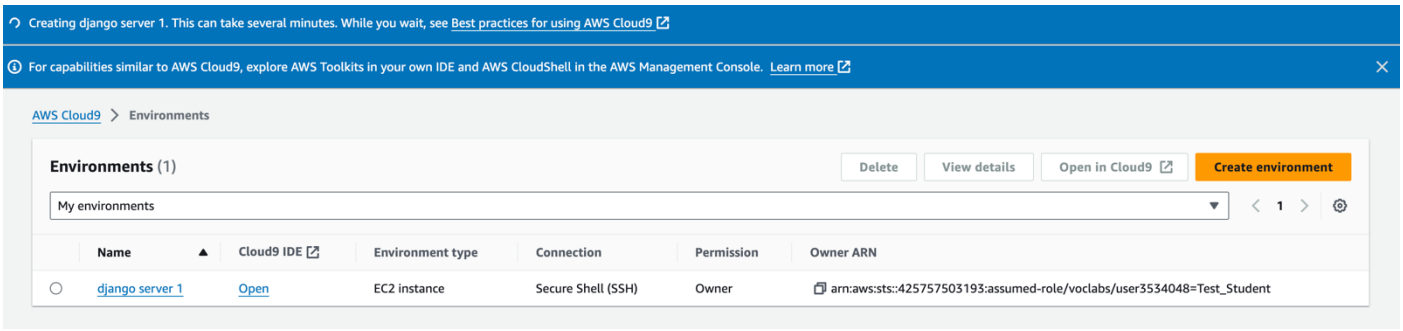
After some minutes, the lab environment will be ready for use. Use the AWS green bullet to enter the AWS services console home:



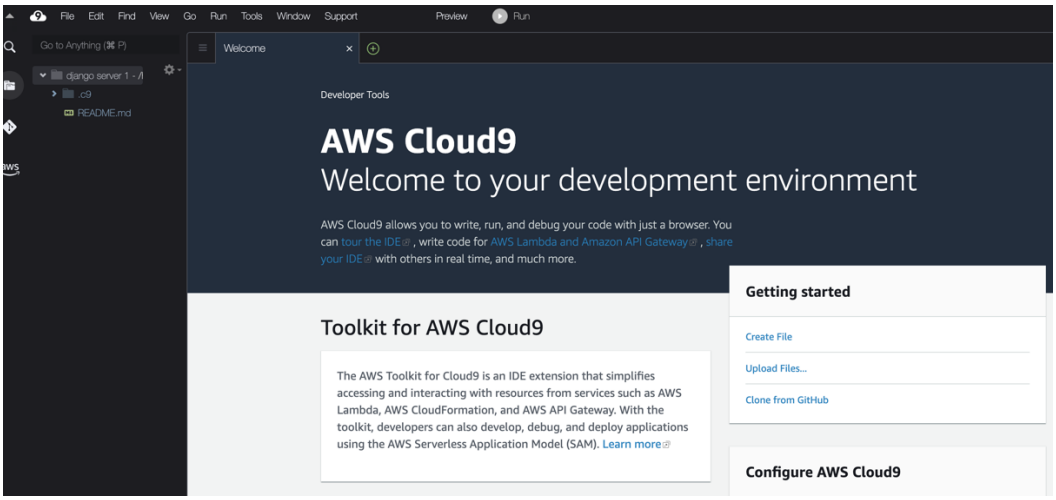
Now you have to search for cloud9 IDE environment to access AWS Cloud9:



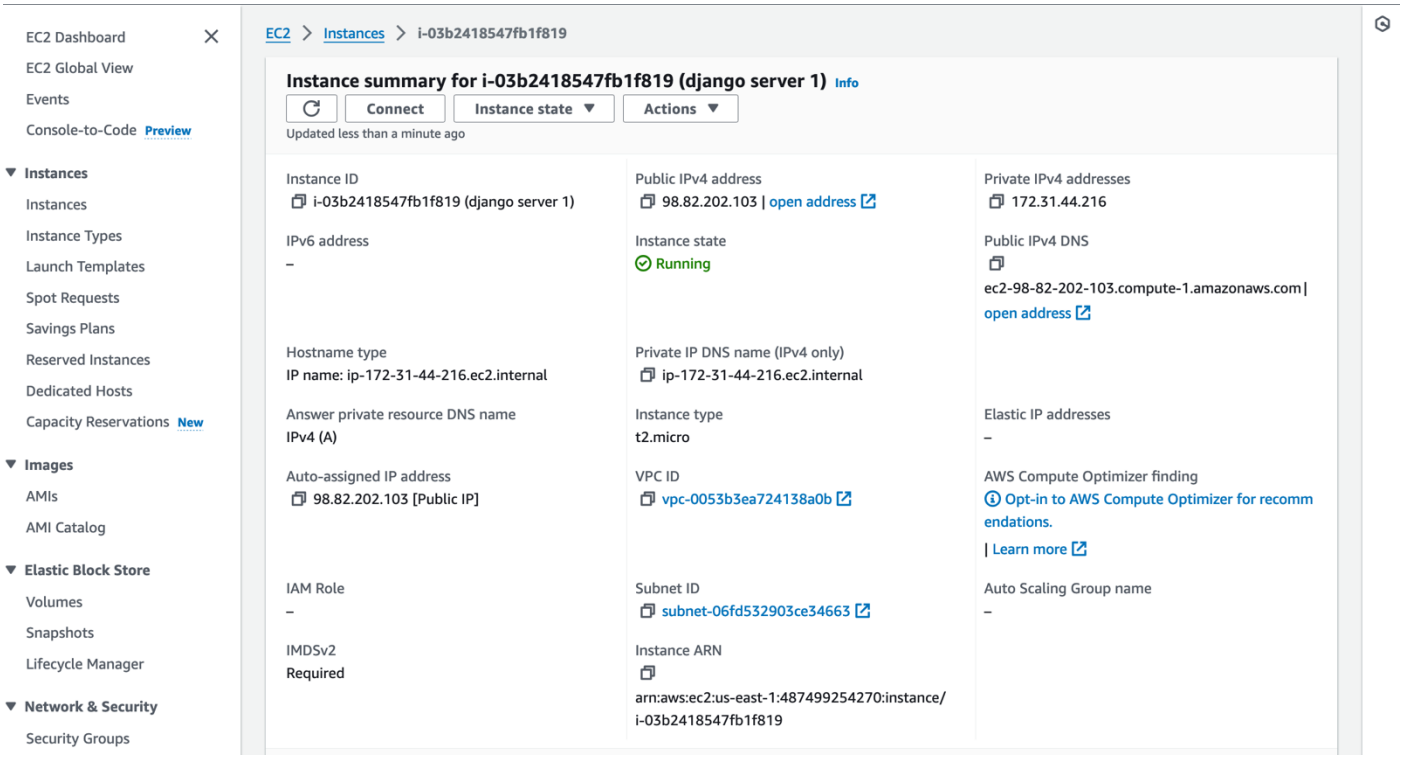
Find "Create environment" yellow button and use it to create a new working environment providing a name: "django server 1" and selecting "secure shell (SSH)" option in Network settings and then clicking on create button.



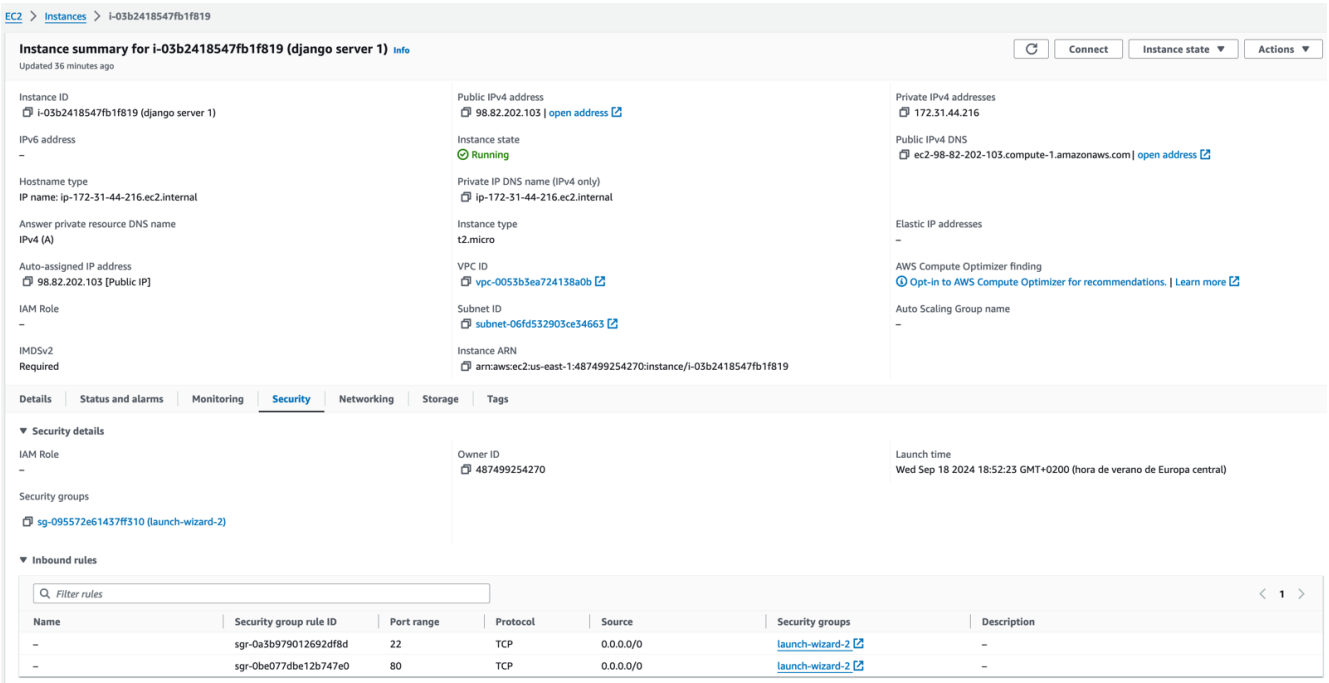
After some minutes of creation and configuration, we will have our new Django server environment. To open it, use "Open in Cloud9" button. A new IDE welcome page will be opened



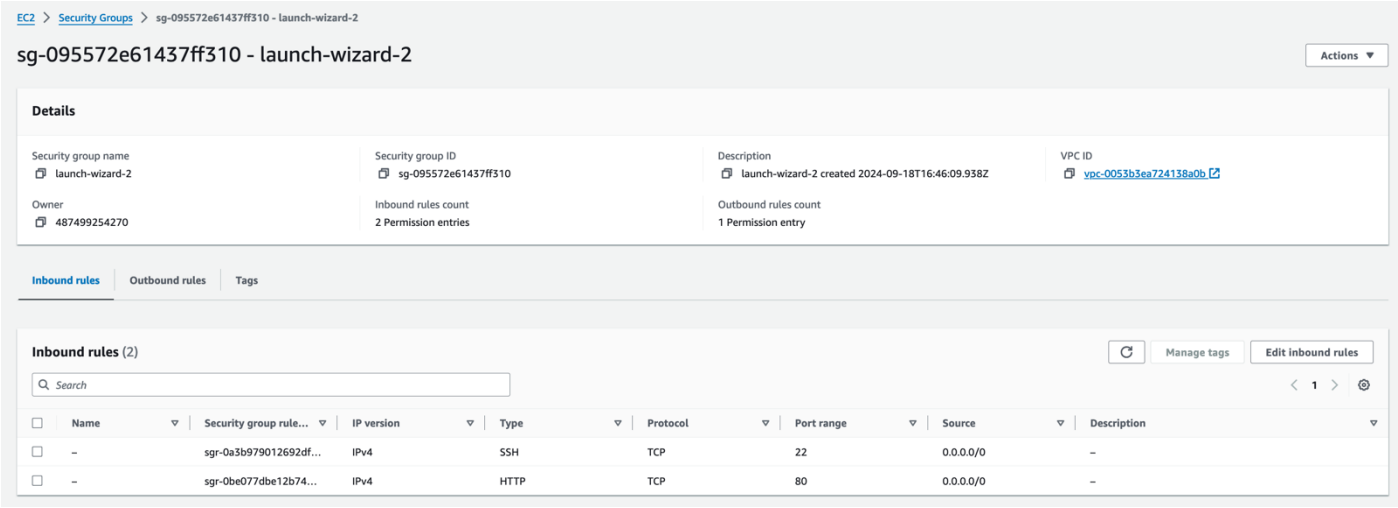
Go back to AWS services console home and search for EC2 instances. You will find a new instance that is running. Please click on instance ID blue link to review its details:



Please write down your public IPv4 address for later use. In our example this address is 98.82.202.103. Now, we need to create an inbound rule to allow web traffic coming in. Click on the **Security** tab, then in **Security groups**

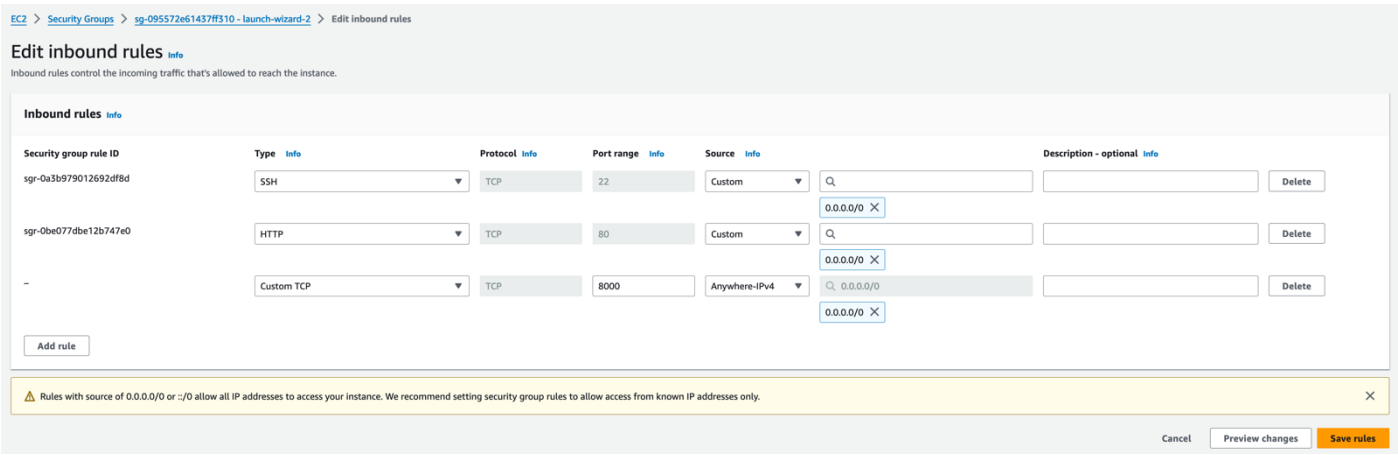


Click on **Edit inbound rules** box



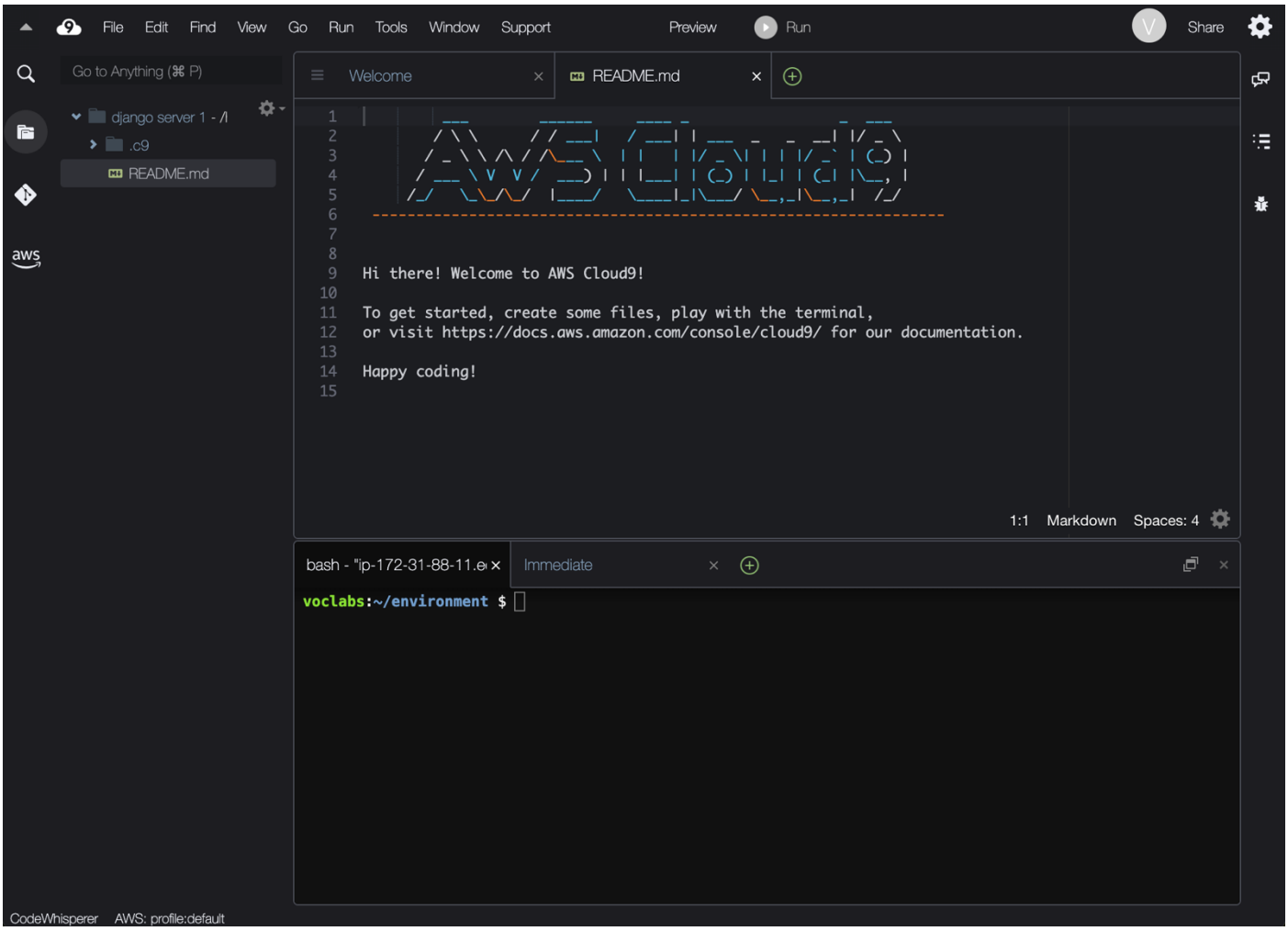
Now **Add a new Custom TCP rule** with the following parameters,
Type: Custom TCP
Port range: 8080
Source: Anywhere-IPv4, 0.0.0.0/0

and then press **Save rules**



3. Django environment creation

Go back to your AWS Cloud 9 development environment tab in your browser to create a new Django application. Check that the browser window has a file navigation, a file viewer and a console window in the bottom



To use Django, we first need to install Python and then install Django latest distribution using pip installer. In our case we are going to use a Cloud9 environment Linux system. Use the bottom terminal and type the commands to create a Django application environment:

```
cd /home/ec2-user/environment
sudo yum install python-is-python3 -y
sudo yum install pip -y
pip install virtualenv

virtualenv polls-env
source polls-env/bin/activate
pip install django==2.2
```

Now, we can start using Django functionalities and start with our web application.

2. Django basic poll application

In the following tutorial, we are going to create a basic poll application that will consist in two main parts:

- a public site to review polls and vote in them
- an administration tool to manage polls for users

2.1 First step: create a new Django project.

The initial step is to create a project with some configuration files and specific settings that we will later modify for our application.

We will use the terminal for most of the operations and we also will need to use a web browser to review the results of some of the operations.

```
django-admin startproject mysite
cd mysite/mysite
```

This will create a mysite directory in your system where your application code will be updated. The structure of the project created is:

```
mysite/
     manage.py
     mysite/
          __init__.py
          settings.py
          urls.py
          wsgi.py
```

These files are:

- The outer **mysite/** root directory is just a container for your project. Its name doesn't matter to Django; you can rename it to anything you like.
- **manage.py**: A command-line utility that lets you interact with this Django project in various ways.
- The inner mysite/ directory is the actual Python package for your project. Its name is the Python package name you'll need to use to import anything inside it (e.g. mysite.urls).
- **mysite/__init__.py**: An empty file that tells Python that this directory should be considered a Python package.
- **mysite/settings.py**: Settings/configuration for this Django project.
- **mysite/urls.py**: The URL declarations for this Django project; a "table of contents" of your Django-powered site.
- **mysite/wsgi.py**: An entry-point for WSGI-compatible web servers to serve your project.

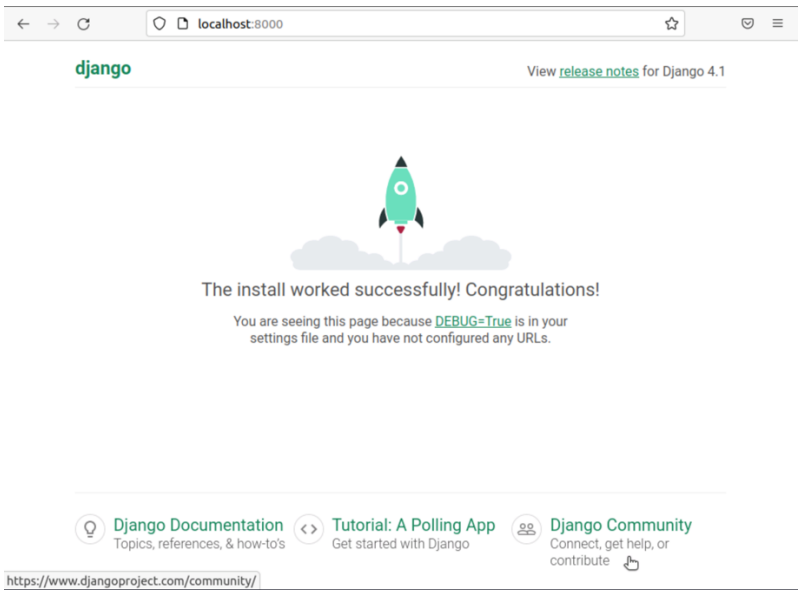
Open mysite/mysite/settings.py file in Cloud9 IDE and edit ALLOWED_HOSTS line so that it contains this configuration:

```
ALLOWED_HOSTS = ['*',]
```

Save the file and now let's verify the Django project activity by starting the internal web server:

```
cd /home/ec2-user/environment/mysite
python manage.py runserver 0:8080
Performing system checks...

System check identified no issues (0 silenced).

You have unapplied migrations; your app may not work properly until
they are applied.
Run 'python manage.py migrate' to apply them.
September 22, 2022 - 22:22:22
Django version 2.2, using settings 'mysite.settings'
Starting development server at http://0:8080/
Quit the server with CONTROL-C.
```

Open a new browser tab and type the IP address that you have saved before:**http://<your ip here>:8080**. You should see this congratulations page with a rocket taking off, that means it is working.



2.2 Creating the Polls app: hello world

Each application you write in Django consists of a Python package that follows a certain convention. Django comes with a utility that automatically generates the basic directory structure of an app, so you can focus on writing code rather than creating directories.

Our application app will be created next to the manage.py file, so we need to create the app in our root mysite folder.

Quit the server with control+C.

Now type the following commands in the terminal of your Cloud9 environment

```
cd /home/alumno/environment/mysite
python manage.py startapp polls
```

which will create the folders and basic files for a polls application (you will see the new polls folder on the left of Cloud9 page)

```
mysite/
     polls/
          __init__.py
          admin.py
          apps.py
          migrations/
          __init__.py
          models.py
          tests.py
          views.py
```

Views, models and templates

Django application development will consist in defining data models, then designing the functionalities that we want to show to the users with views, and specific web designs using html templates.

To start with the first view, open the file polls/views.py and ADD the following python code at the end of the file. KEEP THE EXISTING LINES OF THE **VIEWS.PY** FILE, DO NOT REMOVE THEM.

```
from django.http import HttpResponse

def index(request):
    return HttpResponse("Hello, world. You are at the polls index.")
```

To open this view to web users, we need to map the view to a URL address of our web app. For that, we will need a URLconf. In our project, we have **to create a new urls.py** file in our mysite/polls directory. Use New File option from your Cloud9 IDE.

```
from django.urls import path
from . import views

urlpatterns = [
    # ex: /polls/
    path('', views.index, name='index'),
]
```

The next step is to point the Django root URLconf at the **mysite/mysite/urls.py** module. We need to manage what will our web server do when it receives a request to access at polls web app. Add the bold lines to your urls.py file:

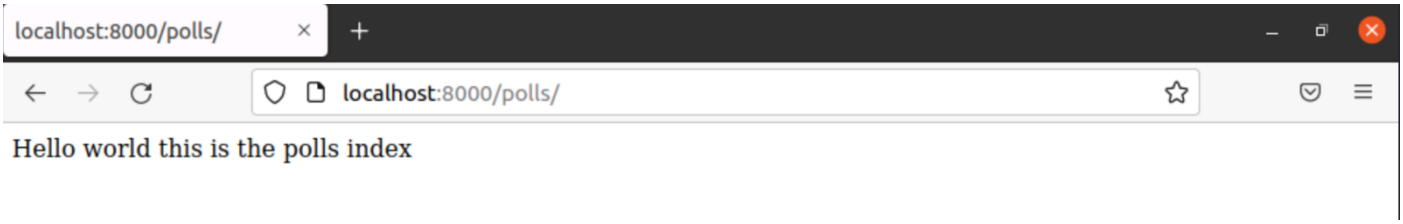
```
from django.contrib import admin
from django.urls import include, path

urlpatterns = [
    path('polls/', include('polls.urls')),
    path('admin/', admin.site.urls),
]
```

The include() function allows referencing other URLconfs. Whenever Django encounters include(), it chops off whatever part of the URL matched up to that point and sends the remaining string to the included URLconf for further processing.

Now we have created a index view of our polls application and we can check that it is working starting up our we server and pointing our web browser to the polls application: **http://<your ip>:8080/polls**

```
python manage.py runserver 0:8080
```

If everything is right, we should see our polls app welcome message:



2.3 Defining data models and webapp administration management

Now that we have our project working, we are going to focus on the data management of Django web apps.

We will start with the database setup.

First, we need to create the tables in our local database before we use them. To create the databases for our application we use the migrate command:

```
python manage.py migrate
```

The migrate command creates any necessary database tables according to the database settings in your mysite/settings.py file.

Now let's focus on our application model creation.

A model is the single, definitive source of truth about your data. It contains the essential fields and behaviors of the data the application is storing. Django follows the Don't Repeat Yourself (DRY) Principle. The goal is to define your data model in one place and automatically derive things from it.

In our simple poll app, we'll create two models: Question and Choice. A Question has a question definition and a publication date. A Choice has two fields: the text of the choice and some votes. Each Choice is associated with a Question.

These concepts are represented by simple Python classes. Edit the *polls/models.py* file to **create** both question and choice models.

```
import datetime
from django.db import models
from django.utils import timezone

class Question(models.Model):
    question_text = models.CharField(max_length=200)
    pub_date = models.DateTimeField('date published')

    def __str__(self):
        return self.question_text

class Choice(models.Model):
    question = models.ForeignKey(Question, on_delete=models.CASCADE)
    choice_text = models.CharField(max_length=200)
    votes = models.IntegerField(default=0)

    def __str__(self):
        return self.choice_text

    def was_published_recently(self):
```

```
        return self.pub_date >= timezone.now() - datetime.timedelta(days=1)
```

The code is straightforward. Each model is represented by a class that subclasses *django.db.models.Model*. Each model has a number of class variables, each of which represents a database field in the model.

Each field is represented by an instance of a Field class – e.g., CharField for character fields and DateTimeField for datetimes. This tells Django what type of data each field holds.

The name of each Field instance (like *question_text* or *pub_date*) is the field's name, in machine-friendly format. You'll use this value in your Python code, and your database will use it as the column name.

Some Field classes have required arguments. CharField, for example, requires that you give it a max_length. That's used not only in the database schema, but in validation.

A Field can also have various optional arguments; in this case, we've set the default value of votes to 0.

Note the addition of import datetime and from *django.utils import timezone*, to reference Python's standard datetime module and Django's time-zone-related utilities in *django.utils.timezone*, respectively.

models.py file gives Django a lot of information. With it, Django is able to:

- Create a database schema (CREATE TABLE statements) for this app.
- Create a Python database-access API for accessing Question and Choice objects.

But first we need to tell our project that the polls app is installed.

To include the app in our project, we need to add a reference to its configuration class in the INSTALLED_APPS settings. The PollsConfig class is in the *polls/apps.py* file, so its dotted path is 'polls.apps.PollsConfig'. Edit the *mysite/mysite/settings.py* file and add that dotted path to the INSTALLED_APPS setting.

```
INSTALLED_APPS = [
    'polls.apps.PollsConfig',
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]
```

Now Django knows to include the polls app. Let's run the *makemigrations* command:

```
python manage.py makemigrations polls
```

```
Migrations for 'polls':
polls/migrations/0001_initial.py:
- Create model Choice
- Create model Question
```

By running *makemigrations*, you're telling Django that you've made some changes to your models (in this case, you've made new ones) and that you'd like the changes to be stored as a migration.

Migrations are how Django stores changes to your models (and thus your database schema) - they're just files on disk.

Now, run *migrate* to create those model tables in your database:

```
python manage.py migrate

Operations to perform:
Apply all migrations: admin, auth, contenttypes, polls, sessions
Running migrations:
Rendering model states... DONE
Applying polls.0001_initial... OK
```

The migrate command takes all the migrations that haven't been applied (Django tracks which ones are applied using a special table in your database called django_migrations) and runs them against your database - essentially, synchronizing the changes you made to your models with the schema in the database

Finally, we summarize the three-step guide to making model changes:

1. Change your models (in models.py).
2. Run python manage.py makemigrations to apply model migrations for those changes
3. Run python manage.py migrate to apply those changes to the database.

2.4. Django administration

Django was written in a newsroom environment, with a very clear separation between "content publishers" and the "public" site. Site managers use the system to add news stories, events, sports scores, etc., and that content is displayed on the public site. Django solves the problem of creating a unified interface for site administrators to edit content.

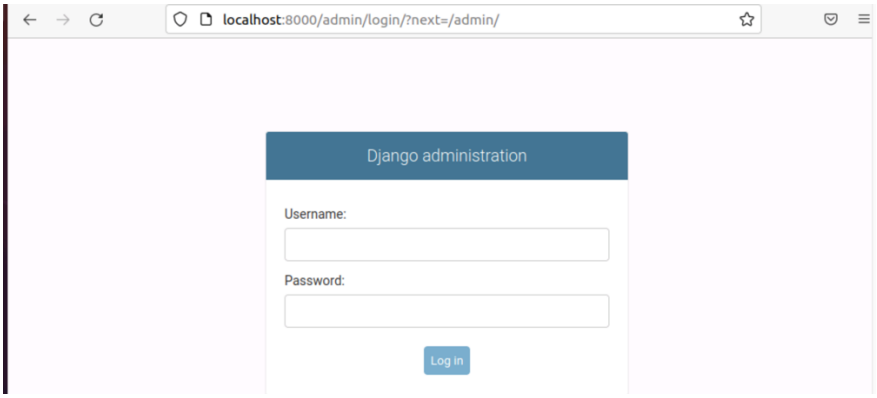
```
python manage.py createsuperuser

Username: admin
Email address: admin@example.com
Password:*****
Password (again): *****
Superuser created successfully.
```

ANY USERNAME AND PASSWORD WILL WORK, just remember it for later.

Start the development server again:

```
cd /home/alumno/mysite/
python manage.py runserver 0:8080
```

And open a web browser with the address: *http://<your ip>:8080/admin*, you should see the login page. Use the admin user that you have created previously to enter the site



As you will see, our poll web app is not visible in the admin index page, we need to tell the admin system that Question and Choice models have an admin interface. Edit *polls/admin.py* file to add the following lines:

```
from django.contrib import admin
from .models import Question, Choice

admin.site.register(Question)
admin.site.register(Choice)
```

Now restart the server and go back to the admin page: *http://<your ip>:8080/admin.* We should see both models available for administration purposes: questions and choices

Please add a couple of questions and choices so that we can use the application in the next section.

2.5. Creating the public interface to our data: views

A view is a "type" of Web page in your Django application that generally shows a specific function of the web app and has a specific template. In our poll application we want to have the following views:

- Question "index" page – displays the latest few questions.
- Question "detail" page – displays a question text, with no results but with a form to vote.
- Question "results" page – displays results for a particular question.
- Vote action – handles voting for a particular choice in a particular question.

In Django, web pages and other content are delivered by views. Each view is represented by a simple Python function (or method, in the case of class-based views). Django will choose a view by examining the URL that's requested (to be precise, the part of the URL after the domain name).

A URL pattern is simply the general form of a URL - for example: /newsarchive/<year>/<month>/.

To go from a URL to a view, Django uses what are known as '*URLconfs*'. A *URLconf* maps URL patterns to views.

We start by adding more views to our application. Some of them need an argument as input. Add the following functions to the existing *polls/views.py.* You will need to overwrite existing index(request) function

```
from django.http import HttpResponse

from .models import Question

def index(request):
    latest_question_list = Question.objects.order_by('-pub_date')[:5]
    output = ', '.join([q.question_text for q in latest_question_list])
    return HttpResponse(output)

def detail(request, question_id):
    return HttpResponse("You're looking at question %s." % question_id)

def results(request, question_id):
    response = "You're looking at the results of question %s."
    return HttpResponse(response % question_id)

def vote(request, question_id):
    return HttpResponse("You're voting on question %s." % question_id)
```

Then, we need to connect these new views to the *polls/urls.py* module so that Django is able to connect URL paths to specific views

```
from django.urls import path
from . import views

urlpatterns = [
    # ex: /polls/
    path('', views.index, name='index'),
    # ex: /polls/5/
    path('<int:question_id>/', views.detail, name='detail'),
    # ex: /polls/5/results/
    path('<int:question_id>/results/', views.results, name='results'),
    # ex: /polls/5/vote/
    path('<int:question_id>/vote/', views.vote, name='vote'),
]
```

Restart the server and open your browser to the addresses:

- http://<your ip>:8080/polls/1/
- http://<your ip>:8080/polls/1/results/
- http://<your ip>:8080/polls/1/vote/

Django will run the detail() method and display whatever ID you provide in the URL.

When somebody requests a page from your website – say, "/polls/1/", Django will load the mysite.urls Python module because it's pointed to by the ROOT_URLCONF setting. It finds the variable named urlpatterns and traverses the patterns in order. After finding the match at 'polls/', it strips off the matching text ("polls/") and sends the remaining text – "1/" – to the 'polls.urls' URLconf for further processing. There it matches '<int:question_id>/ ', resulting in a call to the detail() view like so:

```
detail(request=<HttpRequest object>, question_id=34)
```

The question_id=1 part comes from <*int:question_id*>. Using angle brackets "captures" part of the URL and sends it as a keyword argument to the view function. The *:question_id*> part of the string defines the name that will be used to identify the matched pattern, and the <*int:* part is a converter that determines what patterns should match this part of the URL path.

Improving web app views

Each view is responsible for doing one of two things: returning an HttpResponse object containing the content for the requested page or raising an exception such as Http404.

Your view can read records from a database. It can generate a PDF file, output XML, create a ZIP file on the fly, anything you want, using whatever Python libraries you want.

All Django wants is a *HttpResponse*. Or an exception.

If we consider our current view.py implementation, we have a problem:

```
def index(request):

    latest_question_list = Question.objects.order_by('-pub_date')[:5]
    output = ', '.join([q.question_text for q in latest_question_list])
    return HttpResponse(output)
```

Output variable is building the design of the page. That is, the page design is hard coded in the view. If we need to change the way the page looks, we must edit this Python code.

To separate design from functionality, Django uses templates to define how the web application will look apart from our Python code. Now we are going to create a templates directory and we will create some html templates for each application view.

```
mkdir polls/templates
mkdir polls/templates/polls
```

Create new file *polls/templates/polls/index.html* file with the following content:

```
{% if latest_question_list %}
    <ul>
    {% for question in latest_question_list %}
        <li><a href="/polls/{{ question.id }}/">{{ question.question_text }}</a></li>
    {% endfor %}
    </ul>
{% else %}
    <p>No polls are available.</p>
{% endif %}
```

After we have created our new index.html, we need to update *polls/views.py* to make a reference to the template. The new views.py code will load the template called polls/index.html and pass it a context. The context is a dictionary mapping template variable names to Python objects. The render() function takes the request object as its first argument, a template name as its second argument and a dictionary as its optional third argument. It returns an HttpResponse object of the given template rendered with the given context.

```
from django.http import HttpResponse
from django.shortcuts import render

from .models import Question

def index(request):
    latest_question_list = Question.objects.order_by('-pub_date')[:5]
    context = {
```

```
        'latest_question_list': latest_question_list,
    }
    return render(request, 'polls/index.html', context)

def detail(request, question_id):
    question = Question.objects.get(pk=question_id)
    return render(request, 'polls/detail.html', {'question':
question})

def results(request, question_id):
    question = Question.objects.get(pk=question_id)
    return render(request, 'polls/results.html', {'question':
question})
```

Restart the server and open your browser to the address: http://<your ip>:8080/polls/ to see a list of questions from our web app.

Create two more templates in our polls application: *mysite/polls/templates/polls/detail.html*

```
<h1>{{ question.question_text }}</h1>
<ul>
{% for choice in question.choice_set.all %}
   <li>{{ choice.choice_text }}</li>
{% endfor %}
</ul>
```

and *mysite/polls/templates/polls/results.html*

```
<h1>{{ question.question_text }}</h1>

<ul>
{% for choice in question.choice_set.all %}
   <li>{{ choice.choice_text }} -- {{ choice.votes }} vote{{ choice.votes|pluralize }}</li>
{% endfor %}
</ul>

<a href="/polls/{{question.id}}/">Vote again?</a>
```

Last modification of *polls/urls.py* to create a polls application namespace:

```
from django.urls import path
from . import views

app_name = 'polls'
urlpatterns = [
```

```
    path('', views.index, name='index'),
    path('<int:question_id>/', views.detail, name='detail'),
    path('<int:question_id>/results/', views.results, name='results'),
    path('<int:question_id>/vote/', views.vote, name='vote'),
]
```

2.5. Interacting with the public interface to our data: simple forms

The last part of our tutorial will be dedicated to write a simple form to update our polls results. First, we need to update the *mysite/polls/templates/polls/detail.html* template to build an html form.

The template creates a radio button for each question choice. The value of each radio button is the associated question choice's ID. The name of each radio button is "choice". That means, when somebody selects one of the radio buttons and submits the form, it'll send the POST data choice=# where # is the ID of the selected choice.

We set the form's action to *{% url 'polls:vote' question.id %}*, and we set *method="post"*. Using method="post" (as opposed to method="get") is very important, because the act of submitting this form will alter data server-side.

```
<h1>{{ question.question_text }}</h1>

{% if error_message %}<p><strong>{{ error_message }}</strong></p>{% endif %}

<form action="{% url 'polls:vote' question.id %}" method="post">
{% csrf_token %}
{% for choice in question.choice_set.all %}
    <input type="radio" name="choice" id="choice{{ forloop.counter }}" value="{{ choice.id }}">
    <label for="choice{{ forloop.counter }}">{{ choice.choice_text }}</label><br>
{% endfor %}
<input type="submit" value="Vote">
</form>
```

Now, let's modify polls/views.py view that handles the submitted data and does something with it.

request.POST is a dictionary-like object that lets you access submitted data by key name. In this case, request.POST['choice'] returns the ID of the selected choice, as a string. request.POST values are always strings.

After incrementing the choice count, the code returns an *HttpResponseRedirect* rather than a normal *HttpResponse*. *HttpResponseRedirect* takes a single argument: the URL to which the user will be redirected.

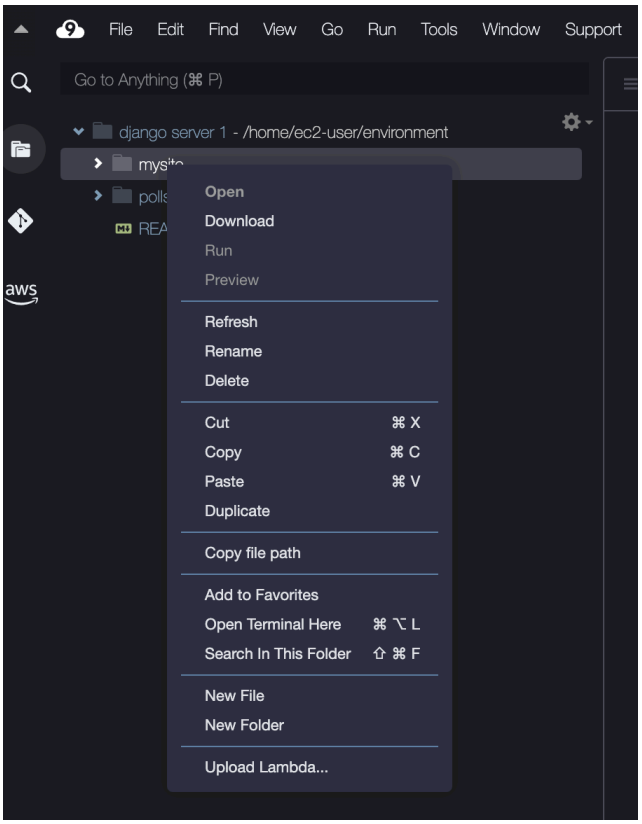
```
from django.http import HttpResponse, HttpResponseRedirect
from django.shortcuts import get_object_or_404, render
from django.urls import reverse

from .models import Choice, Question

def index(request):
    latest_question_list = Question.objects.order_by('-pub_date')[:5]
    context = {'latest_question_list' : latest_question_list }
    return render(request, 'polls/index.html', context)

def detail(request, question_id):
    question = Question.objects.get(pk=question_id)
    return render(request, 'polls/detail.html', {'question': question})

def results(request, question_id):
    question = Question.objects.get(pk=question_id)
    return render(request, 'polls/results.html', {'question': question})

def vote(request, question_id):
    question = Question.objects.get(pk=question_id)
    try:
       selected_choice = question.choice_set.get(pk=request.POST['choice'])
    except (KeyError, Choice.DoesNotExist):
       # Redisplay the question voting form.
       return render(request, 'polls/detail.html', {
           'question': question,
           'error_message': "You didn't select a choice.",
        })
    else:
        selected_choice.votes += 1
        selected_choice.save()
        # Always return an HttpResponseRedirect after successfully dealing
        # with POST data. This prevents data from being posted twice if a
        # user hits the Back button.
        return HttpResponseRedirect(reverse('polls:results', args=(question.id,)))
```

You can now restart the server and open your browser to the addresses to check the fully implemented votation form

- http://<your ip>:8080/polls/1/
- http://<your ip>:8080/polls/1/results/
- http://<your ip>:8080/polls/1/vote/

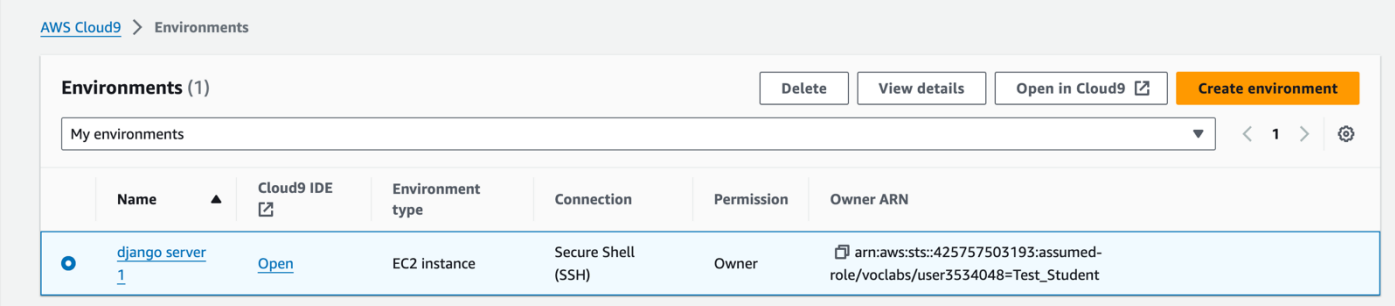
3. Saving your working files for the next session

Remember that you can download a copy of your working files at any time in the folder section of the browser by clicking with the right button in the mysite folder and then Download



4. Closing the AWS Working environment

Please remember to close your AWS Cloud9 environment before you finish. Select your environment in the AWS services console and press the Delete option



Then, type **Delete** and click on the delete option to remove the environment.

Now go back to Launch AWS Academy Learner Lab. Now, you need to press **End Lab** in Sandbox web application and logout of AWS Academy portal.