# Overview

This document provides a step by step guide to perform demos that are directed aligned with the steps described in the Django related eLearning modules (modules 4, 5 and 6) of the **Development Intermediate 1** class in ACI.

In order to facilitate the parallel progress through the eLearning modules, the document is structure in a very specific way.

* **Level 1 headings** represent the **eLearning Skillbuilder's modules**.
  + For example, the Level 1 heading, "Week 4 – Django Overview", matches the " Django Overview" skillbuilder module, which is offered in week 4 of the course.
* **Level 2 headings** represent a **major section** with the eLearning module
  + By major sections I mean the main divisions in an eLearning module, which are seen in the left side navigation bar.
* **Level 3 headings** represent a **heading within a section** in eLearning
  + Each eLearning section generally includes multiple topics, introduced with bold headings.
* **Level 4 headings** represent one **specific demo**
  + Every demo in this document is within a Level 4 heading, with an appropriate title, for easy referral.
  + The position of that level 4 heading, represents the closest location in the eLearning module that lines up with that specific demo. So by looking at the level 3 and level 2 headings, one can get a good approximation of which eLearning section and heading lines up with the demo provide.

## What are these "demos"

I refer to these steps as "demos", because they were primarily written as a guide to demonstrate the steps to learners. Generally, what they are a combination of steps that are described in eLearning as "Practice exercises" to be performed in the Practice Environment.

Most of the demos and code blocks provided come directly from eLearning. Since the demos are placed in the specific section and headings, you should be able to match them with the eLearning section that discusses them. The demos and code blocks are repeated here for the following reasons:

* First of all, not every practice and code block from eLearning is included here as a demo. That's because some of the code blocks in eLearning are shown just as early examples, but they may not actually work in the section they are provided.
* In other cases, some code blocks are shown here in a different order, because they flow better that way.
* I have also added a few extra code blocks, particularly to add more data, so that examples can be more easily understood

**In summary,** the demos I have here, in the order they are provided, represent my recommended set of steps to build the Django applications described in eLearning, particularly the Pet App.

## Disclaimer

This document is part of the AWS Cloud Institute, and it is provided for educational instructional purposes only.

# Week 4 – Django Overview

## Section: Developing a Django Web Application (lesson 6 of 10)

### Heading: Environment setup

#### Demo: Install Django

* Create a "*django*" folder to work on your Django projects
* cd into the newly created *django* folder

mkdir ~/environment/django

cd ~/environment/django

* Set up a virtual environment (not required, but a good practice and eLearning does it)

python -m venv .venv

source .venv/bin/activate

* Install Django in the virtual environment

pip install django

* Verify installation with pip list

pip list --local

### Heading: Create a Django project

#### Demo: Create pets\_project

* Use *django-admin* to create a project in the current folder

django-admin startproject pets\_project .

* This creates a project directory structure in the current directory including:
  + A *pets\_project* folder
  + A *manage.py* file

#### Demo: Create an application within the project

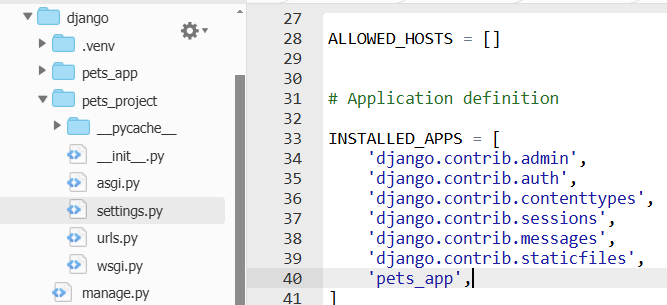
* Use *./manage.py startapp* to create an app under the project

./manage.py startapp pets\_app

* This creates an app directory structure under the folder pets\_app in the current directory

#### Demo: Register the application

* In the *pets\_project* folder, update the *settings.py* file
  + Append *pets\_app* in INSTALLED\_APPS



* Apply migrations:
  + This is **not strictly necessary now**, and will make more sense after we've created models, but it **will avoid some warnings** when we start the server

./manage.py makemigrations

./manage.py migrate

### Heading: Preview your application

#### Demo: Start Django server and preview in Cloud9

Cloud9 requires some special setup to preview Django applications

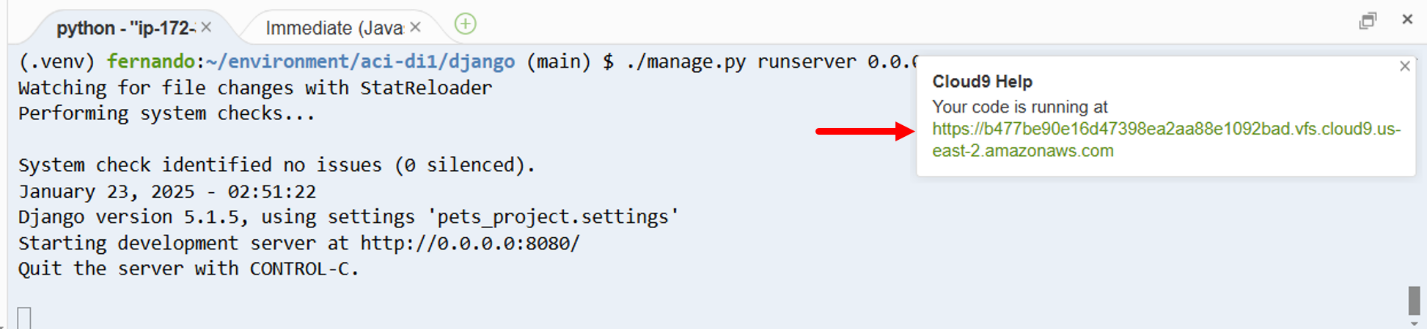
* Update *settings.py*
  + Add '.amazonaws.com' in the ALLOWED\_HOSTS
  + Comment out XFrameOptionsMiddleware under MIDDLEWARE



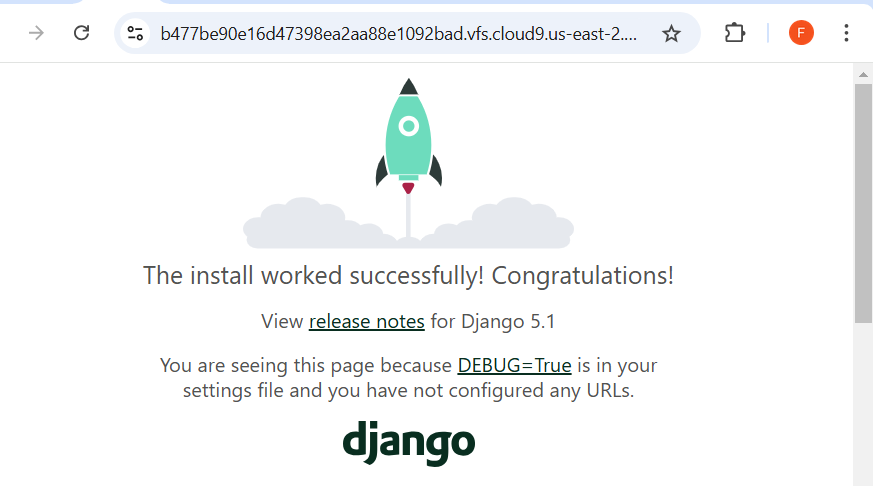
* Run local server using *manage.py*
  + Normally you can start the server with a simple "./*manage.py runserver*" command
  + In Cloud9, we need a slightly modified way to start the server, as seen below

./manage.py runserver 0.0.0.0:8080

* Open up Preview window using recommended URL from Cloud9



* View default Django page, which confirms installation was successful



# Week 4 – Django Models

## Section: Model Creation and Migration (lesson 4 of 10)

### Heading: Design

#### Demo: Review pet\_apps design

* Review the class model
  + Look at class diagram, pointing out different field types
  + Note the text description below

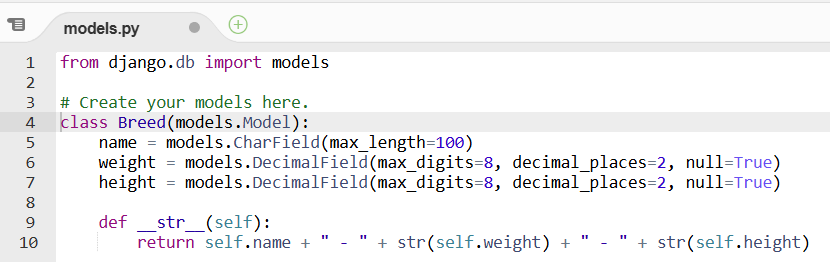
### Heading: Implementation

#### Demo: Create initial models without dependencies

* In the *pets\_app* folder, **open the *models*.py file**

**Breed class**

* Copy the code shown for the ***Breed*** class, and paste it under the *"#Create your models here."* line
* Remove extra import line that was already in the file
* The resulting fill should look like this:



**VaccinationCard class**

* Copy/paste the code for the ***VaccinationCard*** class into **models.py**
* Remove extra import lines that were already in the file, and move new ones to the top

#### Demo: Create models with dependencies

**Gender class**

* Copy/paste the code for the ***Gender*** class into **models.py,** and **rearrange imports**
* Remove extra import lines that were already in the file, and move new ones to the top
* It's **not a model** like the others
  + Note that it **extends "models.TextChoices"**, instead of "models.Model"
  + **TextChoices creates an enum type**, so that we can **use Gender.MALE and Gender.FEMALE** to represent genders
  + "M" and "F" are the values saved, and "Male" and "Female" are used for display
* Note *"from django.utils.translation import gettext\_lazy as \_"*
  + Used for internalization of text

**Pet class**

* Copy/paste the code for the ***Pet*** class into **models.py,** and **rearrange imports**
* Note use of Gender.TextChoices

class Pet(models.Model):

name = models.CharField(max\_length=100)

**gender = models.CharField(max\_length=6, choices=Gender.choices, default=Gender.FEMALE)**

birth = models.DateField(default=None, null=True, blank=True)

owner = models.CharField(max\_length=100)

weight = models.DecimalField(max\_digits=8, decimal\_places=2, null=True)

height = models.DecimalField(max\_digits=8, decimal\_places=2, null=True)

**card = models.OneToOneField(VaccinationCard, on\_delete=models.CASCADE)**

**breed = models.ManyToManyField(Breed, blank=True)**

* Note **relationship** on ***card*** and ***breed***
  + **on\_delete option** means if pet is deleted, the card will also be deleted.

**VetVisit class**

* Copy/paste the code for the ***VetVisit*** class into **models.py,** and **rearrange imports**
* Note **relationship** on ***pet***

class VetVisit(models.Model):

**pet = models.ForeignKey(Pet, on\_delete=models.CASCADE)**

vet = models.CharField(max\_length=100)

date = models.DateField(default=datetime.today().strftime('%Y-%m-%d'))

notes = models.TextField(null=True, blank=True)

* + **on\_delete option** means if pet is deleted, the visit will also be deleted.

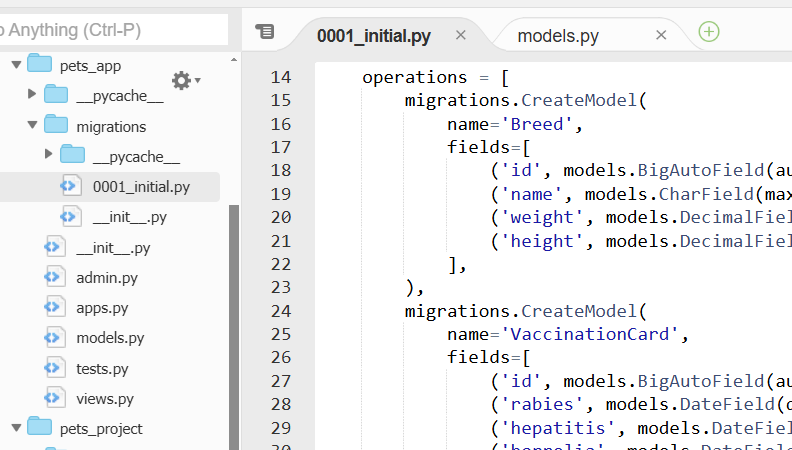
### Heading: Model Migration

#### Demo: Migrate Models

* Execute command to prepare migrations for models created

./manage.py makemigrations

* Open file created under the *pets\_app/migrations* folder, just to briefly explain these commands were created



* Execute migrations

./manage.py migrate

* We'll see the tables created in the next demo

## Section: Data Operations Through a Model (lesson 5 of 10)

### Heading: Database SQL client

#### Demo: Basic SQL operations

* Execute command to list all tables:
  + Highlight the pets\_app\_XXX tables created

sqlite3 -line db.sqlite3 '.tables'

* Execute command retrieve Husky breed:
  + **Expect no results**, because we haven't added any data
  + We will add data next

sqlite3 -line db.sqlite3 'SELECT \* FROM pets\_app\_breed WHERE name="Husky"'

### Heading: Django model data API

#### Demo: Start Django shell

* Start the shell

./manage.py shell

* Subsequent creation steps will depend on each other, so we **need to stay on the shell** until we've created all the objects below

#### Demo: Object creation

**Create Breeds**

* Create a breed for "Husky"
  + ***full\_clean***() validates the values in the model
    - Classes can define a clean() method to perform validations
    - Our example classes didn't have a clean() example, so we won't be doing it each time
  + ***save***() commits the changes to the database.

from pets\_app.models import Breed

husky = Breed(name='Husky', weight=10, height=10)

husky.full\_clean()

husky.save()

* Create breeds for "Golden Retriever" and "German Spitz"

golden = Breed(name='Golden Retriever', weight=80, height=10)

golden.save()

spitz = Breed(name='German Spitz', weight=80, height=10)

spitz.save()

**Create Vaccination Cards**

* Create two vaccination cards

from pets\_app.models import VaccinationCard

from datetime import datetime

card1 = VaccinationCard (rabies = datetime(2021, 7, 19), hepatitis = datetime(2021, 8, 1), distemper = datetime(2021, 2, 9))

card1.save()

card2 = VaccinationCard (rabies = datetime(2024, 8, 16), hepatitis = datetime(2023, 7, 6), distemper = datetime(2023, 12, 9))

card2.save()

**Create Pets**

* Create a few Pets
  + Pets will include references to Breeds and vaccination cards used earlier

from pets\_app.models import Pet, Gender

from datetime import datetime

tucker = Pet(name='Tucker', gender=Gender.MALE, birth=datetime(2020, 5, 19), owner='Fernando', weight=80, height=20, card=card1)

tucker.save()

tucker.breed.add(golden)

tucker.save()

moose = Pet(name='Moose', gender=Gender.MALE, birth=datetime(2022, 7, 12), owner='Fernando', weight=20, height=12, card=card2)

moose.save()

moose.breed.add(spitz)

moose.save()

* At this point we can exit the shell using quit() or Ctrl-D

>>>

now exiting InteractiveConsole...

#### Demo: Object lookup

Enter Django shell

* Enter the Django shell and import all models we'll need

$ ./manage.py shell

Python 3.10.12 (main, Jan 17 2025, 14:35:34) [GCC 11.4.0] on linux

Type "help", "copyright", "credits" or "license" for more information.

(InteractiveConsole)

>>> from pets\_app.models import Pet, Gender, Breed, VaccinationCard

Retrieve all objects

* **Breed.objects.all()** to retrieve all breeds to a QuerySet
  + Value returned is a QuerySet
  + A QuerySet is a collection of data from a database
  + The value shown is what we defined in the str() method

>>> all\_breeds = Breed.objects.all()

>>> all\_breeds

<QuerySet [<Breed: Husky - 10.00 - 10.00>, <Breed: Golden Retriever - 80.00 - 10.00>, <Breed: German Spitz - 80.00 - 10.00>, <Breed: Cavalier - 10.00 - 10.00>]>

* You can retrieve a specific object from the QuerySet with standard List operations

>>> all\_breeds[0]

<Breed: Husky - 10.00 - 10.00>

* **Breed.objects.all().values()** to retrieves the complete values

>>> Breed.objects.all().values()

<QuerySet [{'id': 1, 'name': 'Husky', 'weight': Decimal('10.00'), 'height': Decimal('10.00')}, {'id': 2, 'name': 'Golden Retriever', 'weight': Decimal('80.00'), 'height': Decimal('10.00')}, {'id': 3, 'name': 'German Spitz', 'weight': Decimal('80.00'), 'height': Decimal('10.00')}, {'id': 4, 'name': 'Cavalier', 'weight': Decimal('10.00'), 'height': Decimal('10.00')}]>

Get a single object

* **Pet.objects.get()** to retrieve a single Pet
  + Condition needs to return a single element
  + Returns an actual object, not a QuerySet

>>> Pet.objects.get(name="Tucker")

<Pet: Tucker - M - 2020-05-19 - Fernando>

Filter multiple objects

* **Pet.objects.filter()** to retrieve all male Pets

>>> Pet.objects.filter(gender=Gender.MALE)

<QuerySet [<Pet: Tucker - M - 2020-05-19 - Fernando>, <Pet: Moose - M - 2022-07-12 - Fernando>]>

#### Demo: Object update

Update a vaccination card

* Retrieve the first vaccination card
* Show that it has no date for the borrelia vaccine

>>> card1 = VaccinationCard.objects.get(id=1)

>>> print(card1.borrelia)

None

* Update the value with today's date and time
* Show that borrelia vaccine is updated

>>> from datetime import datetime

>>> card1.borrelia = datetime.now()

>>> card1.save()

>>> print(card1.borrelia)

2025-01-25 18:48:03.506942

#### Demo: Object deletion

Delete a Breed

* Retrieve the object for Husky breed
* Delete husky
* Show that husky is no longer in the list of all Breeds

>>> husky = Breed.objects.filter(name="Husky")[0]

>>> husky.delete()

(1, {'pets\_app.Breed': 1})

>>> Breed.objects.all()

<QuerySet [<Breed: Golden Retriever - 80.00 - 10.00>, <Breed: German Spitz - 80.00 - 10.00>

### Heading: Django utility

#### Demo: Insert data with a fixture

Create a fixtures folder

* Create ***fixtures*** directory under pets\_app
* Create a ***data.yaml*** file in there
* Place this definition in the file

- model: pets\_app.VaccinationCard

pk: 3

fields:

rabies: 2022-06-01

- model: pets\_app.Breed

pk: 4

fields:

name: Cavalier King Charles Spaniel

- model: pets\_app.Pet

pk: 4

fields:

name: Max

gender: F

birth: 2006-03-24

owner: Ashley

weight: 23

height: 9

card: 3

breed: [4]

- model: pets\_app.VetVisit

pk: 1

fields:

pet: 4

vet: Jane Doe

date: 2023-04-30

notes: healthy puppy

Load the data

* Install Python package to handle YAML
  + Not required if the file was in JSON format, but the example in eLearning uses YAML

pip install pyyaml

* Run command to load data

./manage.py loaddata data.yaml

* Run SQL command to show data was loaded

sqlite3 -line db.sqlite3 'SELECT \* FROM pets\_app\_pet'

# Week 5 – Django Views

## Section: Processing a Request and Producing a Response (lesson 5 of 9)

### Heading: Accessing data in the HTTP request

#### Demo: Creating a simple view

* Use *./manage.py startapp* to create a *hello\_app* in the existing *django* folder

./manage.py startapp hello\_app

* Add *hello\_app* to the existing pets\_project
  + Add *hello\_app* to *settings.py* file to the INSTALLED\_APPS list variable

INSTALLED\_APPS = [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

'pets\_app',

**'hello\_app',**

]

* Create a view function named *sayHello()* to implement the application logic
  + Add this code to the *views.py* file in the *hello\_app* folder

from django.http import HttpResponse

def sayHello(request):

# Get the name parameter from the URL query string.

name = request.GET.get("name")

# Return an HTTPResponse with a "Hello" message that includes the name.

return HttpResponse("Hello, {}!".format(name))

* Define the URL mapping for the view in an application-level *urls.py* file
  + Create a file named *urls.py* in the hello\_app folder
  + Copy and paste the following code into it:

from django.urls import path

from hello\_app import views

urlpatterns = [

path("", views.sayHello)

]

* Open the *urls.py* file in the pets\_project folder
* Add "include" in the import line

from django.urls import path, include

* Add the following path definition to the urlpatterns list:

path("hello/", include("hello\_app.urls"))

* Start the server and preview the app in Cloud9 as seen earlier:
  + Use the link provided in the "Help" popup, or preview running app

./manage.py runserver 0.0.0.0:8080

* After the server starts, we will need to edit the URL
  + The initial page will have an error
* Add "/hello/?name=John" to the end of the URL

https://[...]amazonaws.com/hello/?name=John

* The output should be a simple page with "Hello, John"

### Heading: Producing a response

#### Demo: Creating a simple pet list view

* Edit the ***views.py***file in the ***pets\_app*** folder
* Replace the contents of the file with the code below

from django.http import HttpResponse

from .models import Pet

def listPets(request):

# Retrieve all the pets from the database.

all\_pets = Pet.objects.all()

# Build a page that displays the name, gender, owner, and birthdate of each pet.

response = HttpResponse()

response.write("<!doctype html>")

response.write("<html lang='en'>")

response.write("<head><title>Pets List</title></head>")

response.write("<body>")

response.write("<h1>List of Pets</h1>")

for aPet in all\_pets:

response.write("<p>")

response.write(f"Pet name: {aPet.name} <br>")

response.write(f"Gender: {aPet.gender} <br>")

response.write(f"Owner: {aPet.owner} <br>")

response.write(f"Birthdate: {aPet.birth:%Y-%m-%d} <br>")

response.write("</p>")

response.write("</body>")

response.write("</html>")

# Return the response.

return response

* Create a new ***urls.py*** file in the ***pets\_app*** folder
  + This will define the paths within the pets\_app
* Paste the contents below:

from django.urls import path

from pets\_app import views

urlpatterns = [

path("pets/", views.listPets, name="petsList"),

]

* Open the ***urls.py*** file in the ***pets\_project*** folder
* Add the following path definition to the urlpatterns list:

path("", include("pets\_app.urls")),

* The updated ***urls.py*** file in the ***pets\_project*** folder should now look like this (excluding comments on top)

from django.contrib import admin

from django.urls import path, include

urlpatterns = [

path('admin/', admin.site.urls),

path("hello/", include("hello\_app.urls")),

path("", include("pets\_app.urls")),

]

* Start the server and preview the app in Cloud9 as seen earlier:

./manage.py runserver 0.0.0.0:8080

* After the server starts, we will need to edit the URL
* Add "/pets" to the end of the URL

https://[...]amazonaws.com/pets

* The output should look similar to the one below:



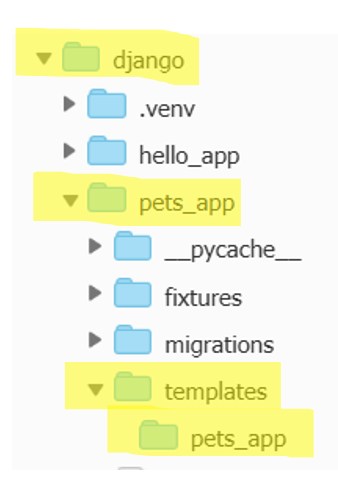
# Week 5 – Django Templates

## Section: Organizing and Referencing Templates (lesson 4 of 12)

### Heading: Folders and files structure

#### Demo: Creating a template folder structure

* Under pets\_app, create a templates folder
* Under the templates folder, create another pets\_app folder
  + This is where we'll place our template files
* The final folder structure for templates will look like this (not all folders in the project are shown):



## Section: Displaying Data from a View (lesson 5 of 12)

### Heading: Passing application data to a template

#### Demo: Creating a view and template to display one pet

**NOTE**: Although some of the files and information are showing in this section, a much of the required detail is in other sections. The steps below will be pulling from those different locations, or adding information as needed.

Views

* Add the following code to the end of **pets\_app/views.py**, and remove any repeated imports:
  + There are two possible versions shown in eLearning, and I'm using the shorter one

from .models import Pet

from django.shortcuts import render

def pet(request, pet\_id):

# 1 - prepare application data in a context dict

context = {'pet': Pet.objects.filter(id=pet\_id).first()}

# 2-3-4 - load template, render it and return the HTTP response

return render(request, "pets\_app/pet.html", context)

Templates

* Under the **pets\_app/templates/pets\_app** folder created earlier, create a new file **pet.html**
* Paste the following content into it from eLearning into it"
  + The file extends base.html, so we'll need to add that next

**{% extends "pets\_app/base.html" %}**

{% block content %}

<div class="pet" style="background-color: lightblue;">

<h2>{{ pet.name }} - WORK IN PROGRESS</h2>

<div class="pet">

Gender: {{ pet.get\_gender\_display }}<br/>

Birth: {{ pet.birth }}<br/>

Owner: {{ pet.owner }}<br/>

<br/>

Weight: {{ pet.weight }} lb<br/>

Height: {{ pet.height }} in<br/>

<br/>

Vaccines: rabies {{ pet.card.rabies }} /

hepatitis {{ pet.card.hepatitis }}<br/>

<br/>

</div>

</div>

{% endblock content %}

* Under the **pets\_app/templates/pets\_app** create a new file **base.html**
  + The file contents are not provided in the module, but I was able to reverse engineer it from a screenshot in a previous "Organizing and Referencing Templates" section

{% load static %}

<!DOCTYPE html>

<html>

<body style="background-color: lightpink">

<div class="banner">

<h1>Pets Application</h1>

<hr/>

</div>

{% block content %}

This is a placeholder for any content

{% endblock content %}

<div class="footer">

<hr/>

<p>&copy; 2024, Amazon Web Services, Inc. or its Affiliates. All rights reserved.</p>

</div>

</body>

</html>

URL routing

* Under the **pets\_app** folder, update the **urls.py** file
  + Add a path for "pet.html" template
  + The updated file should now look like the one below

from django.urls import path

from pets\_app import views

urlpatterns = [

path("pets/", views.listPets, name="petsList"),

**path("pet/<str:pet\_id>", views.pet, name="pet"),**

]

Preview page

* Start the server and preview the app in Cloud9 as seen earlier:

./manage.py runserver 0.0.0.0:8080

* After the server starts, we will need to edit the URL
* Add "/pet/1" to the end of the URL
  + This will display the pet with id 1 (you can enter 2 for the second pet)

https://[...]amazonaws.com/pet/1

* The output should look similar to the one below:



## Section: Template Language Tags and Filters (lesson 6 of 12)

### Heading: Loops tags

#### Demo: Update pets list to use template with loop

Templates

* Under the **pets\_app/templates/pets\_app** folder created earlier, create a new file **pets.html**
  + The earlier example for listing pets did not use a template, so even though this is an update of the pet list, there is not pets.html template
* Paste the following content from eLearning into it:

{% extends "pets\_app/base.html" %}

{% block content %}

<div class="pets" style="background-color: lightblue;">

<h3>List of pets - WORK IN PROGRESS</h3>

<table>

<tr>

<td>Name</td>

<td>Gender</td>

<td>Birth</td>

<td>Weight</td>

<td>Owner</td>

</tr>

{% for p in pets %}

<tr style="background-color: {% cycle 'lightgrey' 'lightgreen' %};">

<td>{{ p.name }}</td>

<td>{{ p.get\_gender\_display }}</td>

<td>{{ p.birth }}</td>

<td>{{ p.weight }}</td>

<td>{{ p.owner }}</td>

</tr>

{% empty %}

<tr><td>No pet yet</td></tr>

{% endfor %}

</table>

</div>

{% endblock content %}

View

* The previous view for listPets was rendering the HTML code directly, so we need to update that to now retrieve the data and then use the template
* **Update** the **listPets** function with the following code, and remove any repeated imports:

def listPets(request):

context = {'pets': Pet.objects.all()}

return render(request, "pets\_app/pets.html", context)

* The **complete views.py** file now should look like this (extra comments are ok):

from django.http import HttpResponse

from django.shortcuts import render

from .models import Pet

def listPets(request):

context = {'pets': Pet.objects.all()}

return render(request, "pets\_app/pets.html", context)

def pet(request, pet\_id):

context = {'pet': Pet.objects.filter(id=pet\_id).first()}

return render(request, "pets\_app/pet.html", context)

Preview page

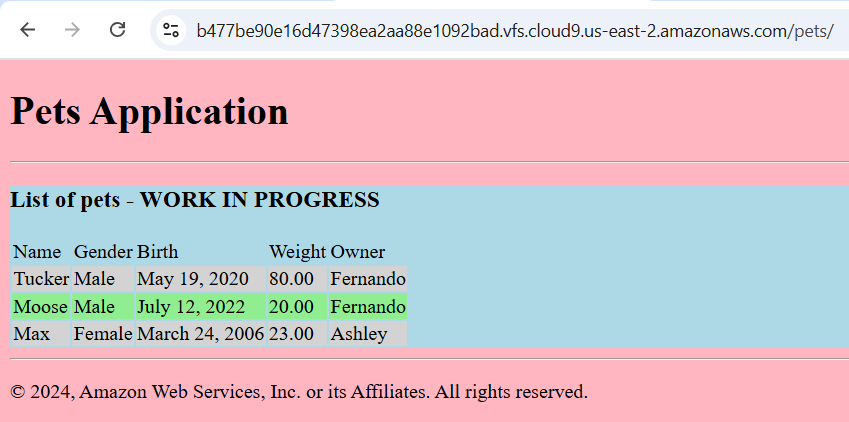
* Start the server and preview the app in Cloud9 as seen earlier:

./manage.py runserver 0.0.0.0:8080

* After the server starts, we will need to edit the URL
* Add "/pets" to the end of the URL

https://[...]amazonaws.com/pets

* The output should look similar to the one below:



### Heading: Choices tags

#### Demo: Update pets list to use {% if} to conditionally values Templates

Templates

* Update the **pets.html** file from the previous demo to include
  + New columns are being added, and we'll use conditions to decide what to display
* Paste the following content from eLearning into it (fully replace contents):

{% extends "pets\_app/base.html" %}

{% block content %}

<div class="pets" style="background-color: lightblue;">

<h3>List of pets - WORK IN PROGRESS</h3>

<table>

<tr>

<td>Name</td>

<td>Gender</td>

<td>Birth</td>

<td>Size</td>

<td>Owner</td>

</tr>

{% for p in pets %}

<tr style="background-color: {% cycle 'lightgrey' 'lightgreen' %};">

<td>{{ p.name }}</td>

<td>{{ p.get\_gender\_display }}</td>

<td>{{ p.birth }}</td>

<td>{% if not p.weight %}

-

{% elif p.weight < 8 %}

Small

{% elif p.weight < 16 %}

Medium

{% else %}

Large

{% endif %}</td>

<td>{{ p.owner }}</td>

</tr>

{% empty %}

<tr><td>No pet yet</td></tr>

{% endfor %}

</table>

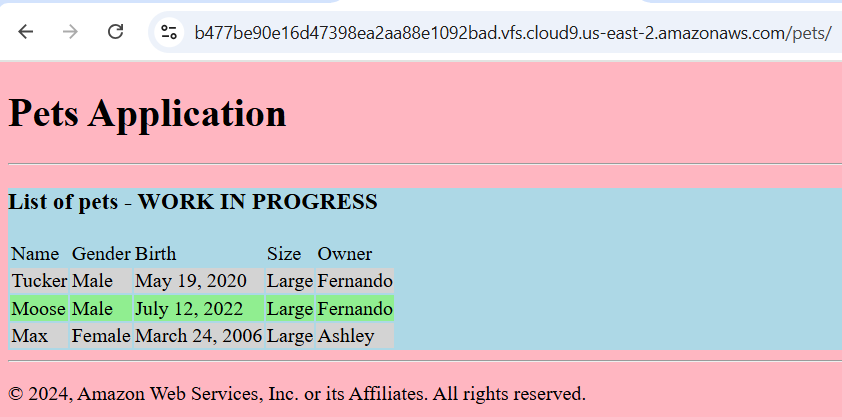
</div>

{% endblock content %}

* We will not have to update **views.py**, because the view is still pointing to the same template, and passing the same data
  + This is a good example of how the clean separation of layers limits the impact of UI changes in an MVT (or MVC) framework

Preview page

* We don't need to restart the server from the previous step, and we can simply refresh the ".../pets" url in the browser
  + If the server had been stopped, simply restart it with the steps from the previous demo
* The output should look like this, with the previous "Weight" column being replaced with a more general "Size" column



### Heading: Date and time tag

#### Demo: Update templates to use date time tags

Templates

* Update the **base.html** file from the previous demo **to use {% now ...} tag** in the footer
  + The line is provided in eLearning text, not a Copy button
* This should be the updated file, with the changed line highlighted in red:

{% load static %}

<!DOCTYPE html>

<html>

<body style="background-color: lightpink">

<div class="banner">

<h1>Pets Application</h1>

<hr/>

</div>

{% block content %}

This is a placeholder for any content

{% endblock content %}

<div class="footer">

<hr/>

**<p>&copy; {% now "Y" %}, Amazon Web Services, Inc. or its Affiliates. All rights reserved.</p>**

</div>

</body>

</html>

* Update the **pets.html** file from the previous demo **to use timesince tag** in the footer
  + The line is provided in eLearning text, not a Copy button
* In the same **pets.html** file, change the birthdate column to use the **timesince** options
  + The line is provided in eLearning text, not a Copy button
  + Change the column heading to "Age" as well
* This should be the updated file, with the changed line highlighted in red:

{% extends "pets\_app/base.html" %}

{% block content %}

<div class="pets" style="background-color: lightblue;">

<h3>List of pets - WORK IN PROGRESS</h3>

<table>

<tr>

<td>Name</td>

<td>Gender</td>

**<td>Age</td>**

<td>Size</td>

<td>Owner</td>

</tr>

{% for p in pets %}

<tr style="background-color: {% cycle 'lightgrey' 'lightgreen' %};">

<td>{{ p.name }}</td>

<td>{{ p.get\_gender\_display }}</td>

**<td>{{ p.birth | timesince }}</td>**

<td>{% if not p.weight %}

-

{% elif p.weight < 8 %}

Small

{% elif p.weight < 16 %}

Medium

{% else %}

Large

{% endif %}</td>

<td>{{ p.owner }}</td>

</tr>

{% empty %}

<tr><td>No pet yet</td></tr>

{% endfor %}

</table>

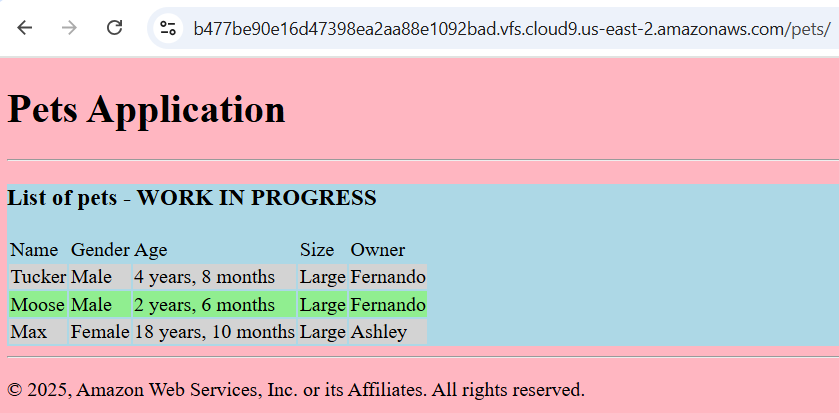
</div>

{% endblock content %}

* We will not have to update **views.py**, because the view is still pointing to the same template, and passing the same data

Preview page

* We don't need to restart the server from the previous step, and we can simply refresh the ".../pets" url in the browser
  + If the server had been stopped, simply restart it with the steps from the previous demo
* The output should look like this, with the column and corresponding values showing the age, not birthdate, and the footer showing the correct year (it was 2024 before).



#### Demo: OPTIONAL> Copy basic version of PetsApp

The next set of demos will significantly change this basic version of the application. If you want to save the simple version to study later, you can make a copy of the entire pets\_app folder with this simple Linux command:

cp -r pets\_app pets\_app\_basic

**Please note** that the code will not be executed from this "*pets\_app\_basic*" folder, since you didn't formally create it as a Django application, and you didn't add "*pets\_app\_basic*" to INSTALLED\_APPS in your project *settings.py*. For it to work you would have to revert it back to be the regular *pets\_app* folder. But it serves as a place to hold all the critical files we created for views, models and templates, in case we want to look at them.

## Section: Model-View-Template Example (lesson 7 of 12)

This section combines elements from the last 3 modules, include concepts from the previews week when Models were covered. It's an end to end example.

### Heading: The models

#### Demo: Update the Pets App Application

Update the model

The model is very similar to what was created in Section 3.1 of this document. It only has a couple small updates.

* Replace the model in *pets\_app/models.py* with the one below
  + One of the main differences, highlighted in red, is the addition of a picture
* **PLEASE NOTE** that this is not the same version of the file provide in eLearning
  + The eLearning version is similar but they use an *ImageField* tag to include the picture
  + Using ImageField will fail because it requires additional libraries, and changes in settings.py
  + We can include those libraries, but the overall topic of using the ImageField tag is not something eLearning discusses, which is why I'm providing a simpler version

from django.db import models

from datetime import datetime

from django.utils.translation import gettext\_lazy as \_

class Breed(models.Model):

name = models.CharField(max\_length=100)

weight = models.DecimalField(max\_digits=8, decimal\_places=2, null=True)

height = models.DecimalField(max\_digits=8, decimal\_places=2, null=True)

def \_\_str\_\_(self):

return self.name + " - " + str(self.weight) + " - " + str(self.height)

class VaccinationCard(models.Model):

rabies = models.DateField(default=datetime.today().strftime('%Y-%m-%d'), null=False, blank=False)

hepatitis = models.DateField(null=True, blank=True)

borrelia = models.DateField(null=True, blank=True)

distemper = models.DateField(null=True, blank=True)

def \_\_str\_\_(self):

return str(self.pet) + " - " + str(self.rabies)

class Gender(models.TextChoices):

FEMALE = "F", \_("Female")

MALE = "M", \_("Male")

class Pet(models.Model):

name = models.CharField(max\_length=100)

gender = models.CharField(max\_length=6, choices=Gender.choices, default=Gender.FEMALE)

birth = models.DateField(default=None, null=True, blank=True)

owner = models.CharField(max\_length=100)

weight = models.DecimalField(max\_digits=8, decimal\_places=2, null=True)

height = models.DecimalField(max\_digits=8, decimal\_places=2, null=True)

card = models.OneToOneField(VaccinationCard, on\_delete=models.CASCADE)

breed = models.ManyToManyField(Breed, blank=True)

**picture = models.CharField(max\_length=50, default="images/default\_dog.jpg")**

def \_\_str\_\_(self):

return self.name + " - " + self.gender + " - " + str(self.birth.year)

class VetVisit(models.Model):

pet = models.ForeignKey(Pet, on\_delete=models.CASCADE)

vet = models.CharField(max\_length=100)

date = models.DateField(default=datetime.today().strftime('%Y-%m-%d'))

notes = models.TextField(null=True, blank=True)

@property

def is\_today(self):

return self.date == datetime.today().date()

def \_\_str\_\_(self):

return self.pet.name + " - " + self.vet + " - " + str(self.date) + " - " + self.notes

* Now we can migrate it to the database as we've done before
  + Since this new model is only adding columns to the previous model, if you already had data from the previous demo steps, they will be maintained

./manage.py makemigrations

./manage.py migrate

OPTIONAL: Update images for existing Pets

If you've been following this example from the beginning, at this point you would arrive have two pets in the database: Tucker and Moose. A default image is added when the model was created, but if we want real images for them, we can update the picture files in the objects.

* Use SQL to update the picture for Tucker and Moose
  + We could have also used the Django shell or Admin tool, but I thought an example of SQL would be good
  + Enter these two statements directly in the regular command prompt

sqlite3 -line db.sqlite3 'UPDATE pets\_app\_pet set picture="images/tucker\_pic.jpg" WHERE name="Tucker"'

sqlite3 -line db.sqlite3 'UPDATE pets\_app\_pet set picture="images/moose\_pic.jpg" WHERE name="Moose"'

Update the View

* Replace the model in *pets\_app/views.py* with the one below
  + One of the main differences, highlighted in red, is the addition of a visit view.
  + This is similar to the one in eLearning, but I updated a bit to handle the case where there were no previous vet visits.

from django.shortcuts import render

from .models import Pet, VetVisit

from datetime import datetime

def listPets(request):

context = {'pets': Pet.objects.all()}

return render(request, "pets\_app/pets.html", context)

def pet(request, pet\_id):

context = {'pet': Pet.objects.filter(id=pet\_id).first()}

return render(request, "pets\_app/pet.html", context)

**def visit(request, pet\_id):**

**pet = Pet.objects.filter(id=pet\_id).first()**

**lastvisit = pet.vetvisit\_set.last()**

**# set vet from last visit or default to unkwon**

**vet = "unknown"**

**if lastvisit:**

**vet = pet.vetvisit\_set.last().vet**

**# add a rabies visit today, unless there was visit today already**

**if lastvisit is None or (lastvisit and not lastvisit.is\_today):**

**newvisit = VetVisit(pet=pet, vet=vet, notes="rabies vaccination")**

**newvisit.save()**

**pet.card.rabies = datetime.today().strftime('%Y-%m-%d')**

**pet.card.save()**

**context = {'pet': pet}**

**return render(request, "pets\_app/pet.html", context)**

Create and upload static image files

The updated templated will include static files for logos and icons so we need to create a static folder to store them.

**NOTE**: the various static files included here are available in our course Canvas in the *static.zip* archive

* Create a *static* folder under the *pets\_app* folder
* Upload logo and icon images to the *static* folder
  + The files *"favicon.ico"* and *"logo-white.png"* are shared in the course canvas

Create and upload css file

* Create a *css* folder under the *static* folder created earlier
* Upload the *main.css* file to the *css* folder
  + The *" main.css* *"* file is shared in the course canvas

Create image folder and upload images

* Create a *images* folder under the *static* folder created earlier
* Upload pet images to the *images* folder
  + The files *"default\_dog.jpg", "tucker\_pic.jpg"* and *"moose\_pic.jpg"* are shared in the course canvas

Update the Templates

* Under the **pets\_app/templates/pets\_app** update the file **base.html** with the one below (same as shown in eLearning)

{% load static %}

<!DOCTYPE html>

<html>

<head>

<title>Pets Application</title>

<link rel="icon" href="{% static 'favicon.ico' %}">

<link rel="stylesheet" type="text/css" href="{% static 'css/main.css' %}">

</head>

<body>

<div class="banner">

<table>

<tr>

<td><img src="{% static 'logo-white.png' %}"></td>

<td>Pets Application</td>

</tr>

</table>

</div>

<div class="nav-menu">

<a href="{% url 'index' %}">Home</a>

</div>

{% block content %}

{% endblock content %}

<div class="footer">

<p>&copy; {% now "Y" %}, Amazon Web Services, Inc. or its Affiliates. All rights reserved.</p>

</div>

</body>

</html>

* Under the **pets\_app/templates/pets\_app** update the file **pets.html** with the one below
  + The file below is very similar to the one in eLearning, but the changes in red are needed to load pet images from the static images folder, as opposed to the ImageField tag which had been used earlier

{% extends "pets\_app/base.html" %}

**{% load static %}**

{% block content %}

<div class="pets" id="pets-link">

<h2>Pets</h2>

<table>

<tr>

{% for p in pets %}

{% if forloop.counter0|divisibleby:3 %}

</tr>

<tr>

{% endif %}

<td><a href="{% url 'pet' p.id %}">

**<img src="{% static p.picture %}" width="120" />**

<p>{{ p.name }}</a></p>

<p>{{ p.owner }}</p>

</td>

{% endfor %}

</tr>

</table>

</div>

{% endblock content %}

* Under the **pets\_app/templates/pets\_app** update the file **pet.html** with the one below (same as shown in eLearning)

{% extends "pets\_app/base.html" %}

{% load static %}

{% block content %}

<div class="pet">

<h2>{{ pet.name }}</h2>

<div class="pet">

<img src="{% static pet.picture %}" width="200" /><br/>

Gender: {{ pet.gender }}<br/>

Birth: {{ pet.birth|date:"Y-M-d" }}<br/>

Breed: {% for b in pet.breed.all %}

{% if forloop.last %}

{{ b.name|lower }}

{% else %}

{{ b.name|lower }},

{% endif %}

{% endfor %}<br/>

Owner: {{ pet.owner|title }}<br/>

<br/>

Weight: {{ pet.weight|floatformat }} lb<br/>

Height: {{ pet.height|floatformat }} in<br/>

<br/>

Vaccines: rabies {{ pet.card.rabies }} /

hepatitis {{ pet.card.hepatitis|default\_if\_none:"never" }} /

borrelia {{ pet.card.borrelia|default\_if\_none:"never" }} /

distemper {{ pet.card.distemper|default\_if\_none:"never" }}<br/>

<br/>

Vet visits:

<ul>

{% for vv in pet.vetvisit\_set.all %}

<li>{{ vv.date|date:"Y-M-d" }}: vet {{ vv.vet|title }}

{% if vv.notes %}

wrote <i>{{ vv.notes|striptags }}</i>

{% endif %}

</li>

{% endfor %}

</ul>

<a class="btn btn-info pull-right"

href="{% url 'visit' pet.id %}">

New visit today with rabies vaccine

</a><br/><br/>

</div>

</div>

{% endblock content %}

Update the URL paths

* We need a couple updates in the *pets\_app/urls.py*, which are not shown in eLearning
  + Replace the file with these contents:

from django.urls import path

from pets\_app import views

urlpatterns = [

path("", views.listPets, name="petsList"),

path("index/", views.listPets, name="index"),

path("pet/<str:pet\_id>", views.pet, name="pet"),

path("visit/<str:pet\_id>", views.visit, name="visit"),

]

Preview page

* Start the server and preview the app in Cloud9 as seen earlier:

./manage.py runserver 0.0.0.0:8080

* With the latest update the urls.py, the base URL maps to the list pets page, so we don't need to change the URL.

# Week 6 – Django Administration

## Section: Accessing the Django Admin Site (lesson 3 of 10)

### Heading: Enabling the Django admin site

#### Demo: Create an admin user

* To use the Django admin web interface, we need to create an admin user
  + This was shown earlier in Week 4 but it fits better here
* Use the *createsuperuser* command:
  + Email is not verified, so you can enter anything
  + For this example, I use password: *admin#1234*

$ ./manage.py createsuperuser

Username (leave blank to use 'ubuntu'): admin

Email address: fake\_email@example.com

Password:

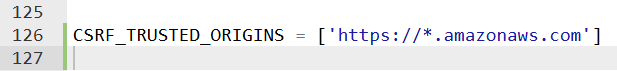
Password (again):

Superuser created successfully.

#### Demo: Django admin web interface

Update settings for the admin application

* Add the following line of code to the end of your *settings.py* file
  + CSRF\_TRUSTED\_ORIGINS = ['https://\*.amazonaws.com']

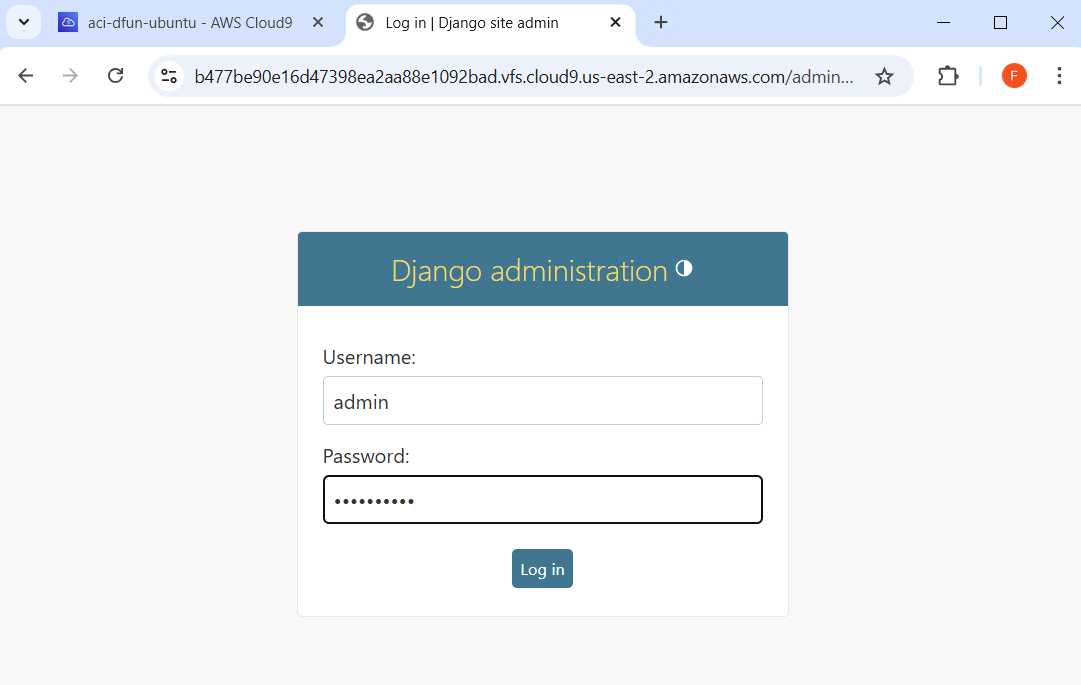


Run the Django server

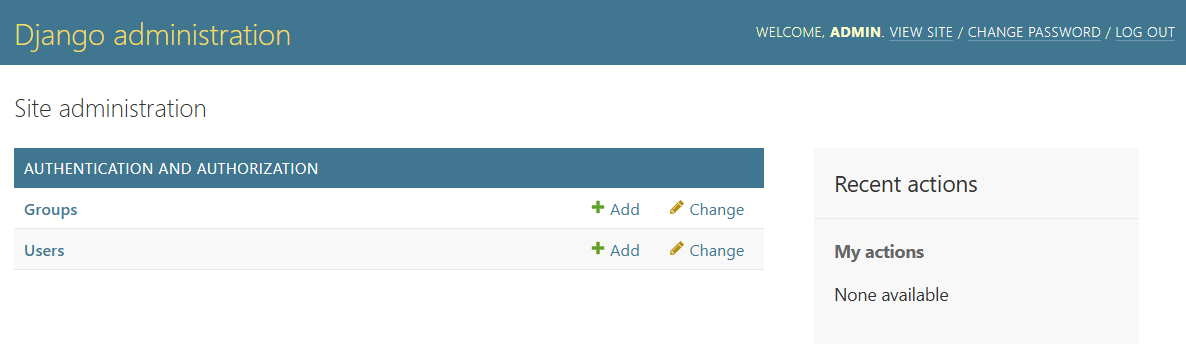
* Start or refresh the Django server in Cloud9, as described in an earlier demo

$ ./manage.py runserver 0.0.0.0:8080

* Add "/admin" to the path int the default URL provided by Cloud9
  + Enter user and password created earlier



* In the admin page,
  + Enter user and password created earlier
* You should see this page
  + By default you have the ability to administer users and groups, but we'll see next how to configure the admin page to support our application



#### Demo: Register models for the Django admin web interface

* Add the models you want to expose your application *admin.py*
  + One line per model
  + Need to import the model classes

from django.contrib import admin

from .models import Breed, VaccinationCard, Pet, VetVisit

# Register your models here.

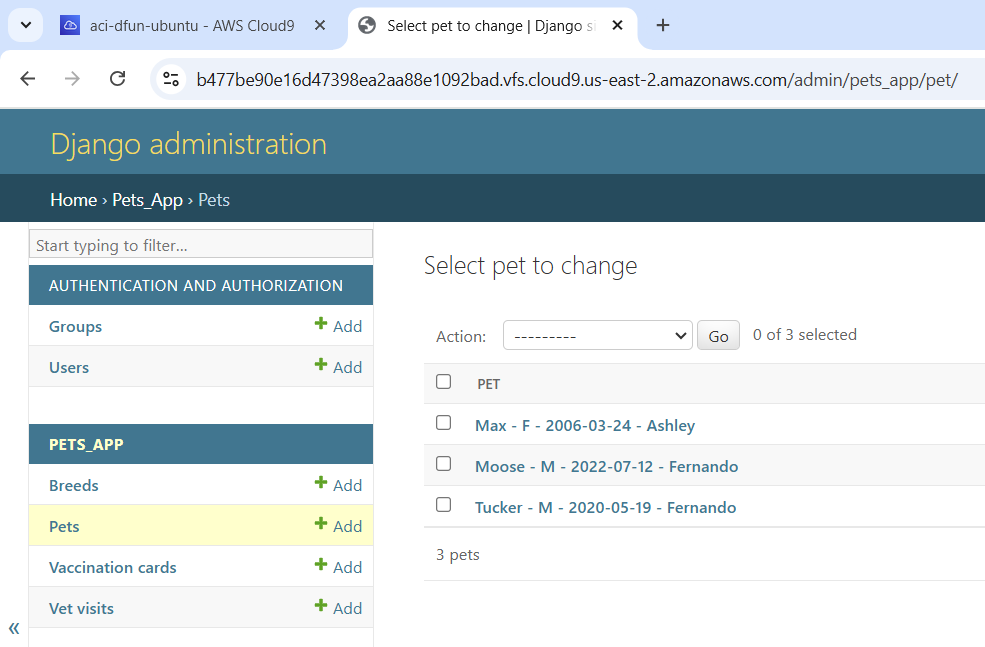
admin.site.register(Breed)

admin.site.register(VaccinationCard)

admin.site.register(Pet)

admin.site.register(VetVisit)

* Refresh the admin web interface if it's still open in the browser
  + If you stopped the server, restart it and navigate to the admin interface as described in the previous demo
* The admin page should now look like this, where you have the ability to manage data in the model we created in the last module.



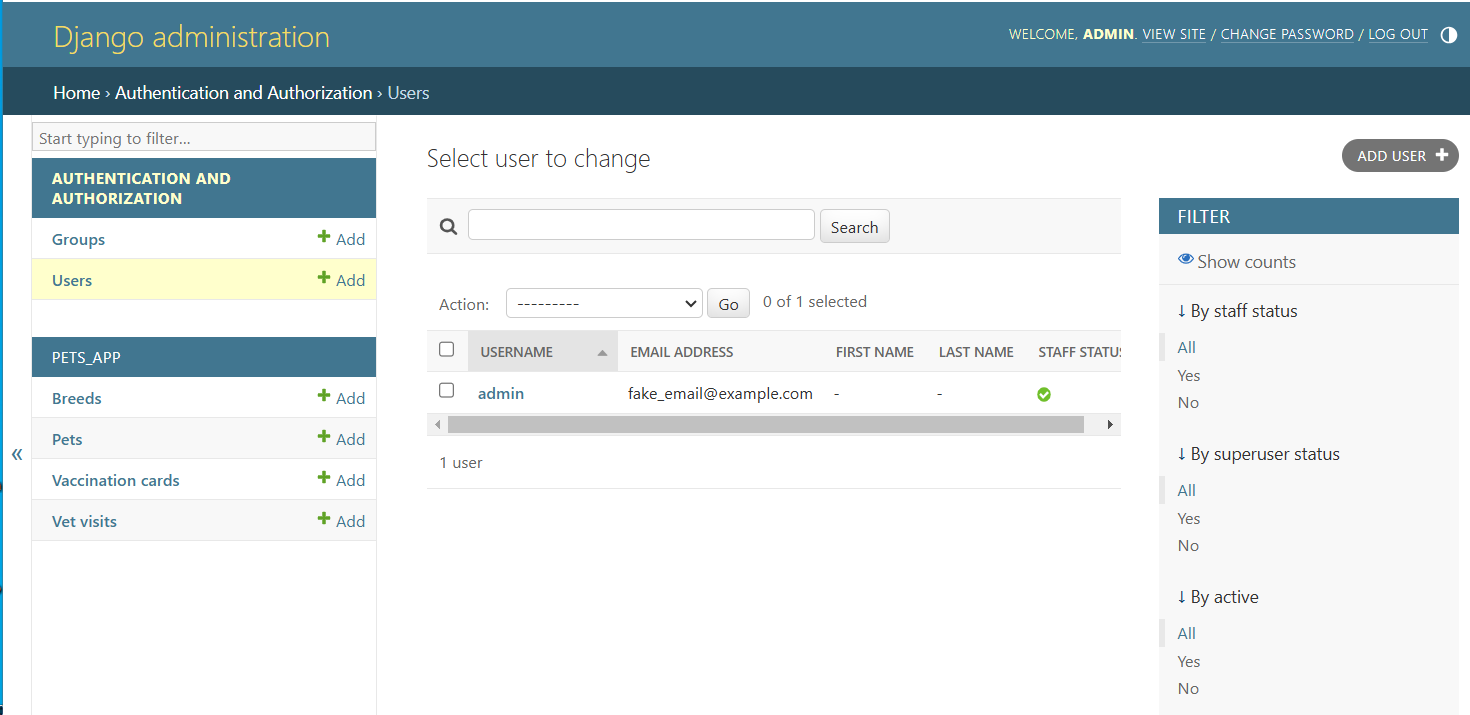
## Section: Managing Admin Users (lesson 5 of 10)

### Heading: Django authentication system

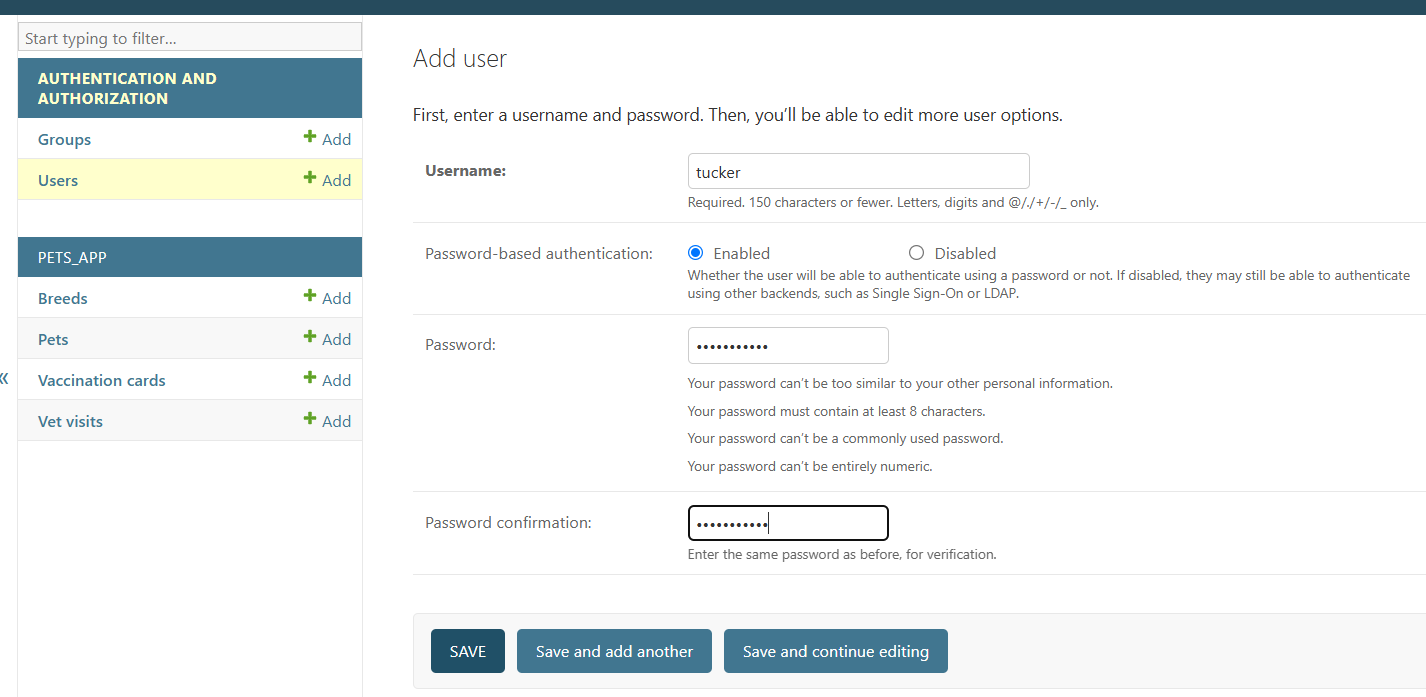
#### Demo: Creating a User in the admin portal

We can use the admin portal to create additional users, with specific set of permissions. In this demo, we will add a new user "tucker", who will not have Django admin permissions, but will have permissions to view and update data on the Pet Apps model.

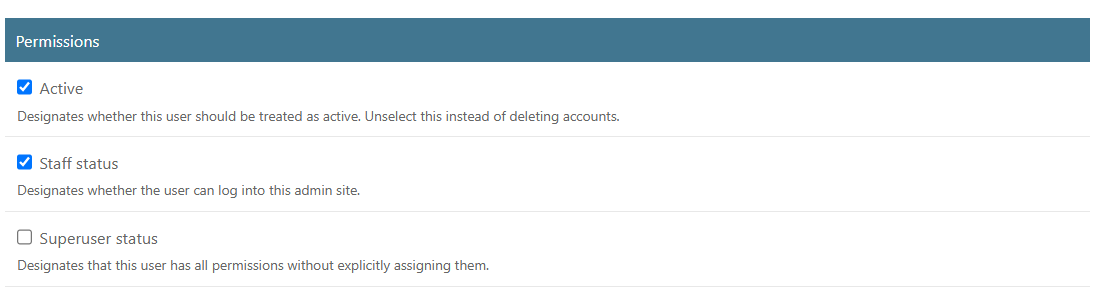
* Login as "admin" as shown earlier
  + If you just did the previous demo, you'll be logged on already
* Click Users on left, and this should open up the page below:



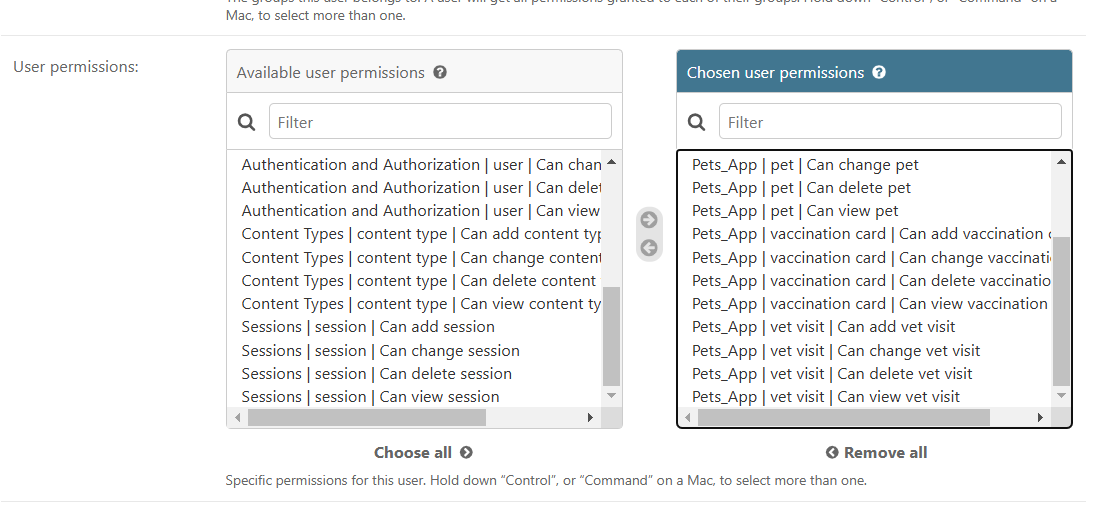
* Click the "ADD USER +" button on the page above, and this should take you to the following page to create a user. Enter the following (or you can change to your own user name):
  + Username: tucker
  + Password: aci#1234



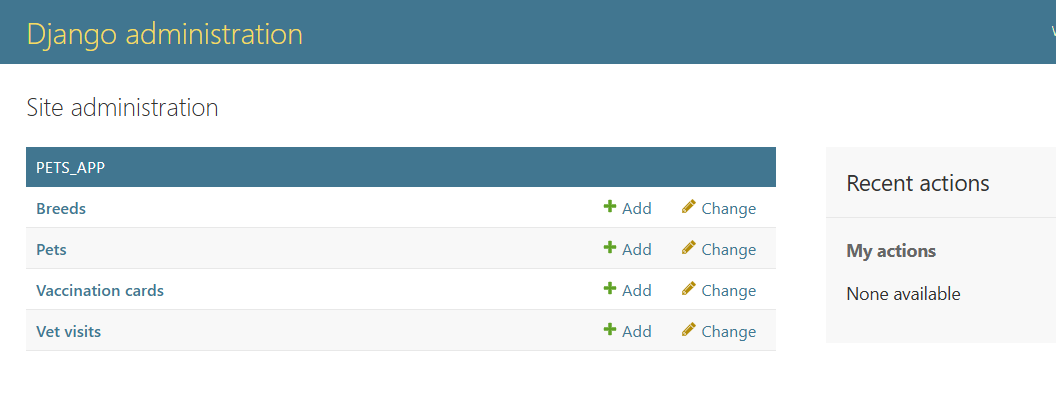
* Click the "Save and continue editing" button on the page above to set permissions for the user.
* On the next page, scroll down to the Permissions section, and check "Staff status" so the user will be able to login to the admin portal



* Scroll further down add all permissions related to the "Pet\_App" application



* Scroll to the bottom and Click "Save" to save the new user
* Logout using the link on the top right of the page
* Now Log back in as "tucker", using the credentials you set earlier
* When the admin page opens, the user will be able to access the Pet\_App model, but not the "Groups" and "Users" section we used earlier.



## Section: Maintaining Model Data (lesson 6 of 10)

### Heading: Listing objects

#### Demo: Listing objects and adjusting display with \_\_str\_\_()

This particular series of examples show the different ways to display data in admin site.

Default listing

* The intent of the "Default listing" shown in eLearning was to have a VaccinationCard class with no added functions, however in the previous example we had defined a \_\_str\_\_() for it
  + We will replace the VaccinationCard we currently have with the one below
  + This is based on the one in eLearning, but some extra lines are removed

# model definition in models.py

class VaccinationCard(models.Model):

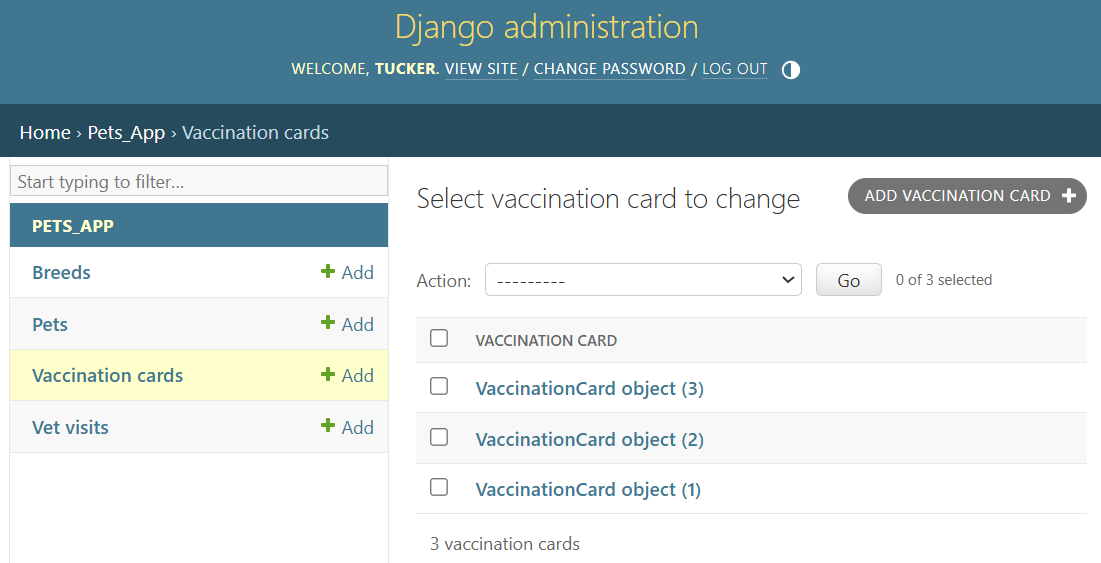
rabies = models.DateField(default=datetime.today().strftime('%Y-%m-%d'), null=False, blank=False)

hepatitis = models.DateField(null=True, blank=True)

borrelia = models.DateField(null=True, blank=True)

distemper = models.DateField(null=True, blank=True)

* If the admin page is not up from the previous demo, restart it now and go to the admin portal
* Click on "Vaccination cards" on the left, and see the default display:



Define the \_\_str\_\_() method

* Add the a \_\_str\_\_() method to the class
  + eLearning has a couple of classes in there, but you should only modify the VaccinationCard class:

class VaccinationCard(models.Model):

rabies = models.DateField(default=datetime.today().strftime('%Y-%m-%d'), null=False, blank=False)

hepatitis = models.DateField(null=True, blank=True)

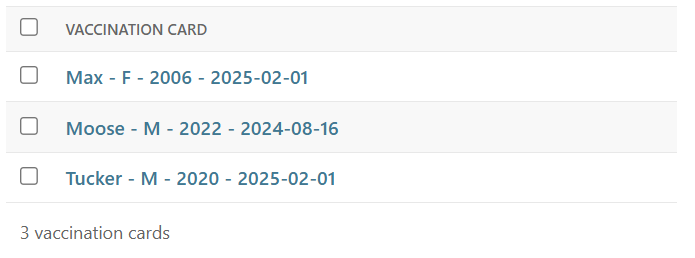
borrelia = models.DateField(null=True, blank=True)

distemper = models.DateField(null=True, blank=True)

def \_\_str\_\_(self):

return str(self.pet) + " - " + str(self.rabies)

* Refresh the admin page and see the updated list displayed:



### Heading: Configuring the listing with admin.ModelAdmin

#### Demo: Subclassing admin.ModelAdmin

* Update the *pets\_app/admin.py* file with the code below
  + The code is provided on eLearning, but it doesn't explicitly specify where to place it:

from django.contrib import admin

from .models import Breed, VaccinationCard, Pet, VetVisit

admin.site.register(Breed)

admin.site.register(Pet)

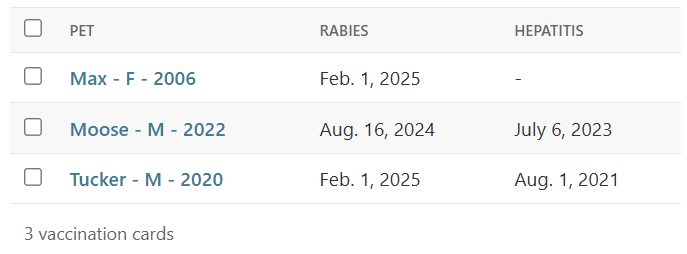
admin.site.register(VetVisit)

@admin.register(VaccinationCard)

class VaccinationCardAdmin(admin.ModelAdmin):

list\_display = ["pet", "rabies", "hepatitis"]

* Refresh the admin page and see the updated list displayed:



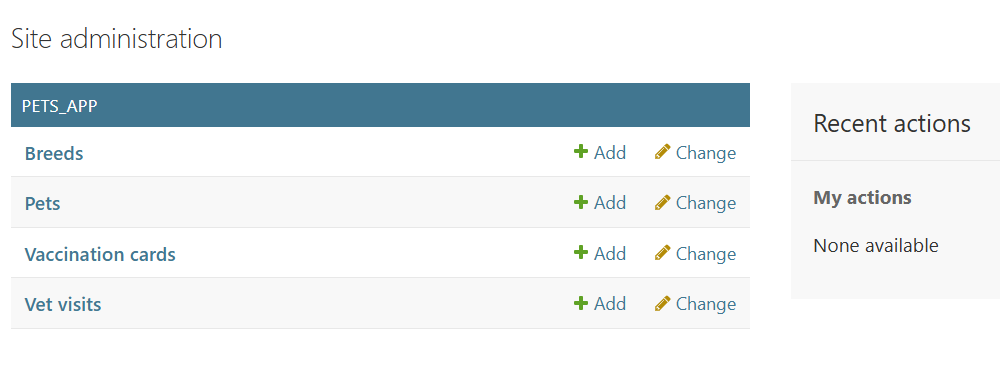
* There are other examples for customizing the display on eLearning, and you can try each of them replacing the contents on admin.py and refreshing the screen.

### Heading: Editing an object

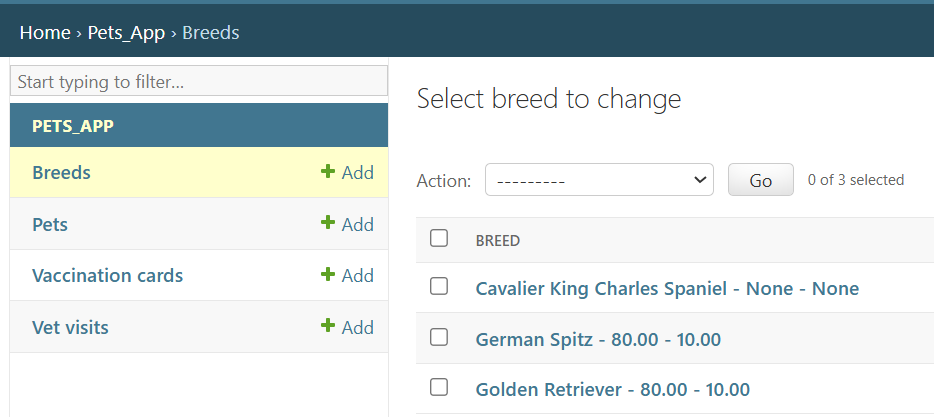
#### Demo: Editing objects in the admin portal

The admin portal can be used to update your model data, including adding, updating, and deleting objects.

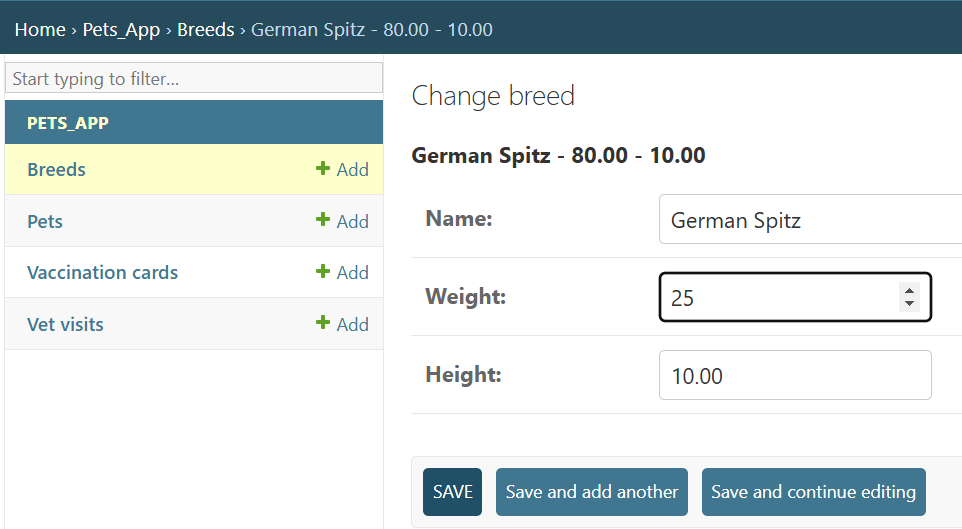
* eLearning includes an example of code for the Pet class, but you **shouldn't copy that**
  + The example is actually behind the latest version we should have at this point, because it doesn't include the image
* Editing the objects is pretty self-explanatory, so feel free to try some yourself.
* As an example, I'll use the interface to fix the average weight for a "German Spitz" breed
  + Click Breeds on the left



* + Click on the "German Spitz" link



* + Update the weight to 25, and click SAVE



# Week 7 – Deploying to AWS Elastic Beanstalk

**PLEASE NOTE: This section cannot be run in the Practice Environment, because it requires AWS resources to be allocated, include S3, RDS, and others**. I've adapted the steps provided in the eLearning so that they can be performed in the Lab environment used for the Bicycle shop application. This may require some awkward steps in places, but at least you'll be able to follow the pets project till the end if you're motivated to do so. Since I'm borrowing pieces of the official lab to create a new demo/lab, I'm calling this exercise, "**Frankendemo**"

## Section: Moving from Development to Production (lesson 2 of 11)

### Heading: Path to a full deployment

#### Demo: Special Setup for the Lab Environment

This is not directly associated with the section and heading above, although they fall in the overall topic of setting up a deployment environment. These are special steps required to use the Lab environment for the Pets Project.

Open Lab Environment

* **Start Week 7 Lab**, "*Deploying a Web App on AWS Elastic Beanstalk*"
* Open Cloud9, but **DO NOT FOLLOW LAB INSTRUCTIONS UNLESS OTHERWISE NOTED**

Setup Pets Project

* **Follow instructions in Appendix 5** for "*Restoring your environment if you have a pets\_project.zip backup*"
  + If you do not have a *pets\_project.zip* backup file, you can upload the latest version shared in Canvas
* Once you've confirmed that the Pets Project is working as expected in the Lab environment, you can proceed to the next steps

## Section: Using Amazon S3 for Media Files (lesson 3 of 11)

### Heading: Django application files

#### Demo: Create and configure an S3 bucket for static contents

Create an S3 bucket

* Switch to the **S3 Dashboard** on the Lab AWS console
* **Create an S3 bucket** with a name like "***bicycle-project****-pets-<account-id>"*
  + **PLEASE NOTE**, the name has to start with " *bicycle-project",* because there are specific S3 permissions that are only open for buckets that start with that.
  + The <account-id> is available in the left side bar

Configure CORS for your bucket

* Choose the **link for the "*pets-project-< account-id>"*** bucket you just created.
* Click on the **Permissions tab**, scroll to the Cross-origin resource sharing (CORS) section, and then choose Edit.
* Copy the following JSON, and paste it into the editor:

[

{

"AllowedOrigins": ["http://\*.elasticbeanstalk.com", "http://127.0.0.1", "http://localhost"],

"AllowedHeaders": ["\*"],

"AllowedMethods": ["GET", "PUT", "POST", "DELETE"],

"MaxAgeSeconds": 9000,

"ExposeHeaders": ["x-amz-server-side-encryption"]

}

]

* Choose Save changes .

#### Demo: Copy the static files in Django project to the S3 bucket

* Inside the project folder in your Cloud9 environment, execute the following command:
  + Replace REPLACE\_WITH\_BUCKET\_NAME with the name of the bucket you created

aws s3 cp pets\_app/static s3://REPLACE\_WITH\_BUCKET\_NAME --recursive

* In S3, confirm that the objects were uploaded

### Heading: Storage configuration for Amazon S3

#### Demo: Change project configuration to use S3 storage

Install additional python libraries

* Install additional packages:

pip install "pyyaml<5.4"

pip install django-storages boto3

pip install Pillow

Change images to ImageField

* To be consistent with eLearning we need to **switch** **the** **model back to use an *ImageField***
* Open *pets\_app/models.py*, and change the picture object to an *ImageField* called *image\_url*
* The updated Pet class will look like this:

class Pet(models.Model):

name = models.CharField(max\_length=100)

gender = models.CharField(max\_length=6, choices=Gender.choices, default=Gender.FEMALE)

birth = models.DateField(default=None, null=True, blank=True)

owner = models.CharField(max\_length=100)

weight = models.DecimalField(max\_digits=8, decimal\_places=2, null=True)

height = models.DecimalField(max\_digits=8, decimal\_places=2, null=True)

card = models.OneToOneField(VaccinationCard, on\_delete=models.CASCADE)

breed = models.ManyToManyField(Breed, blank=True)

**image\_url = models.ImageField(max\_length=255, null=True)**

def \_\_str\_\_(self):

return self.name + " - " + self.gender + " - " + str(self.birth.year)

* Now we can migrate it to the database as we've done before

./manage.py makemigrations

./manage.py migrate

Update pets.html and pet.html to use new variable

* Open *pets.html*, and change the line that references the pet image.
  + Change image line from ...

...

<img src="{% static p.picture %}" width="120" />

...

* + ... to

...

<img src="{{ p.image\_url.url}}" width="120" />

...

* Open *pet.html*, and change the line that references the pet image.
  + Change image line from ...

...

<img src="{% static pet.picture %}" width="200" /><br/>

...

* + ... to

...

<img src="{{ pet.image\_url.url }}" width="200" /><br/>

...

* Now our pictures will no longer be dealt with as static files

Update image names for existing Pets

The pictures we had saved earlier for our pets were in the *picture* field, so now we need to update the database to have them in the new *image\_url* field.

* Use SQL to update the picture for Tucker, Moose, and Max
  + Enter these two statements directly in the regular command prompt

sqlite3 -line db.sqlite3 'UPDATE pets\_app\_pet set image\_url="images/tucker\_pic.jpg" WHERE name="Tucker"'

sqlite3 -line db.sqlite3 'UPDATE pets\_app\_pet set image\_url="images/moose\_pic.jpg" WHERE name="Moose"'

sqlite3 -line db.sqlite3 'UPDATE pets\_app\_pet set image\_url="images/default\_dog.jpg" WHERE name="Max"'

Update settings.py to point to use S3

* Under from pathlib import Path, enter the following line:

import os

* At the end of the INSTALLED\_APPS list, enter the following line:
  + We can optionally also remove *hello\_app* at this point, as we won't need it anymore

'storages',

* Under STATIC\_URL = ‘static/’, enter the following line:
  + This defines relative location where static files fill be placed in the folder structure

STATIC\_ROOT = os.path.join(BASE\_DIR, 'static/')

* At the end of the file, enter the following lines:
  + Replace the YOUR\_BUCKET and YOUR\_REGION values below with the bucket name and region of your bucket (in the lab region is likely us-west-2)

STORAGES = {

"default": {

"BACKEND": "storages.backends.s3boto3.S3Boto3Storage",

},

"staticfiles": {

"BACKEND": "django.contrib.staticfiles.storage.StaticFilesStorage",

},

}

AWS\_STORAGE\_BUCKET\_NAME = **'YOUR\_BUCKET'**

AWS\_S3\_REGION\_NAME = **'YOUR\_REGION'**

AWS\_S3\_FILE\_OVERWRITE = False

AWS\_DEFAULT\_ACL = None

Test

* Run or refresh the local server to confirm images are begin taken from S3

./manage.py runserver 0.0.0.0:8080

* On the browser window, use the browser feature to view the source code, and confirm the S3 address is being used.
  + The line should look like this, but with your bucket name

<img src="https://aci-fttg-pets-project.s3.amazonaws.com/images/tucker\_pic.jpg?X-Amz-Algorithm=... width="120" />

## Section: Using ElastiCache for Session Storage (lesson 4 of 11)

**Note:** I will not be demoing ElastiCache, because we do not have that available in the Lab environment. That is not a major problem though. Caching is an important strategy which can be used to improve performance on an enterprise application, but it's not something every application must use. There is some great information in the eLearning if someone wants to try on their own.

## Section: Switching to an Amazon RDS database (lesson 5 of 11)

The Lab environment does not have permissions for you to create a new RDS database from scratch. That's not a big issue, because there is one created already for the bicycle app, and we'll be able to reuse for the Pets project. In fact most of the important configuration steps are left undone, so we'll be able to practice here.

### Heading: Configuring Amazon RDS as a Django database

#### Demo: Configure RDS MySQL for Django access

Analyze Lab Database and collect key information

* Follow **step 32 in the Lab** to locate RDS database information
  + For subsequent steps, we will **need the RDS endpoint**, which is available from RDS database, or left side of instructions
  + Note also the DB name of "*bicycles*" in the "Configuration" section. That won't work for us, so we'll be creating our own later.

Install and configure PyMySQL

* Run this command to install Python MySQL client library:

pip install pymysql

* Run this command to import the library and make it available for the project:
  + This adds two lines to the *pets\_project/\_\_init\_\_.py* file

echo "import pymysql" >> /home/ubuntu/environment/django/pets\_project/\_\_init\_\_.py

echo "pymysql.install\_as\_MySQLdb()" >> /home/ubuntu/environment/django/pets\_project/\_\_init\_\_.py

Configure RDS database network access

* Follow **steps 35 to 41 in the Lab** to configure network access to the database:
  + You will need to **add permissions for two security groups** in these steps
    - One will allow our application to connect to the database after deployment
    - The other to allow us to connect to it during development inside Cloud9

Connect to MySQL

* Connect to database with the following command:
  + **Replace** the **RDS\_ENDPOINT** values with the actual values of the Lab RDS database
  + The root user was created by the Lab
  + For the password, **use the *DBPassword* value on the left side** of the Lab

mysql -h **RDS\_ENDPOINT** -P 3306 -u root -p

Create database for pets project and create user for django

* Still in the "mysql>" prompt, run the following commands to create a new *pets* database for our Pets Project, then switch to it

CREATE DATABASE pets;

use pets

* Create a *django* database user for the application

CREATE USER 'django'@'%' IDENTIFIED BY 'django#1234';

GRANT ALL PRIVILEGES ON pets.\* To 'django'@'%';

exit;

* For subsequent steps, we will need the following credentials we created above
  + RDS user name: django
  + RDS password: django#1234
  + RDS database: pets

#### Demo: Update settings in Django to use the MySQL database

Backing up your current data

After we migrate to use RDS, we won't have all the data we've inserted into our default SQLite database. If we want to save that, we can export here before making changes, and then import it back later.

* Run the following command to back up your data, excluding some stuff you don't need
  + The file *savedata.json* is created inside your project folder

./manage.py dumpdata --exclude contenttypes --exclude auth.permission --exclude sessions --indent 2 > savedata.json

* Keep that file around for later use if you want to import the data

Update settings.py

* Open the *pets\_project/settings.py* file
* Look for the entry for the DATABASES variable
  + By default, it should be pointing to the internal sqllite3 database
* Comment out the current entry (in case we want to change back to it)

# DATABASES = {

# 'default': {

# 'ENGINE': 'django.db.backends.sqlite3',

# 'NAME': BASE\_DIR / 'db.sqlite3',

# }

# }

* Now add the following entry to point to our RDS MySQL database:
  + **Replace** the **RDS\_ENDPOINT** values with the actual values of your RDS database from earlier steps

DATABASES = {

'default': {

'ENGINE':'django.db.backends.mysql',

'NAME':'pets',

'USER':'django',

'PASSWORD':'django#1234',

'HOST':**'RDS\_ENDPOINT'**,

'PORT':'3306',

'OPTIONS': {

'init\_command': "SET sql\_mode='STRICT\_TRANS\_TABLES'"

}

}

}

Apply migrations:

* Apply migrations to create the tables in the new RDS database

./manage.py makemigrations

./manage.py migrate

Verify that tables were created

* Login to MySQL as you did before
  + **Replace** the **RDS\_ENDPOINT** values with the actual values of your RDS database and password from earlier steps
  + **PLEASE NOTE**: **Using the password directly like this is not a security best practice**. Passwords should be maintained in environment variables, or better yet, secure storage. I'm only doing this here because it's demo on a development environment, and the security aspects are beyond the scope of this discussion.

mysql -h **RDS\_ENDPOINT** -P 3306 -u django -pdjango#1234

* In the MySQL prompt, type the following commands, and confirm you see the pets\_app tables created

use pets

show tables;

* Check to see if we have any pets in our new database with the following SQL query

select \* from pets\_app\_pet;

* Unfortunately the query will return an empty set, because although we've created a database in RDS, we have not migrated the data
  + At this point, we can add new pets using the same process we saw earlier, or we can try to import existing data as shown below
* Enter "quit" to exit the MySQL client

Restore data from the previous version

* If you backed up the data from your previous SQlite database as described earlier, we'll use it now
* Run the following command to load the data into our new tables

./manage.py loaddata savedata.json

* Confirm that data was loaded using the following SQL command
  + **Replace** the **RDS\_ENDPOINT** values with the actual values of your RDS database and password from earlier steps
  + In this example I'm showing you can include a select statement on the command line, using the "-e" option

mysql -h **RDS\_ENDPOINT** \

-P 3306 -u django -pdjango#1234 \

-e "use pets;select \* from pets\_app\_pet"

Preview the updated application

* We should be fully converted now, but to make sure of that when we test, **rename** the "**db.sqlite3**" file (the default SQLite database file) **to "\_db.sqlite3"**
* Now run or refresh the local server using *manage.py,* and use the provide URL as seen before

./manage.py runserver 0.0.0.0:8080

* The output should have our pets, with the same pictures and contents

## Section: Deployment to Elastic Beanstalk (lesson 6 of 11)

### Heading: Preparing your Django project

#### Demo: Preparing Django project for Deployment

Network Setup

The Lab environment already has a security group created with the correct ports open, so we'll just examine it.

* **Under the EC2 Dashboard**, select "Security Groups" on the left, and look for the security group with name *ebSecurityGroup*
  + Note that we have inbound rules opening the HTTP (80) and HTTPS (443) ports
* **Execute command** bellow to **set shell variables for VPC, subnet and security group**
  + The Lab has a specific VPC and subnet that have been opened for public access
  + **Replace the variables inside the quotes** with the corresponding values provided in the left side of the Lab (but keep the replaced values within quotes)

vpcId="VPC\_ID"

subnetId="SUBNET\_ID"

ebSecGroupId="SG\_ID"

Install additional libraries

* **Install *gunicorn*** WSGI server to support execution in an external environment such as Elastic Beanstalk:
  + Up to now we didn't need it because the Cloud9 preview feature performed the equivalent function of interfacing between the HTTP server and the django application

pip install gunicorn

Update settings.py

* **Add elasticbeanstalk.com** as an allowed origin in theexisting **CSRF\_TRUSTED\_ORIGINS** list
  + You can replace the variable with the contents below:

CSRF\_TRUSTED\_ORIGINS = ['https://\*.amazonaws.com', 'http://127.0.0.1', 'http://localhost', 'http://\*.elasticbeanstalk.com']

* **Change ALLOWED\_HOSTS to any**, so it will support the Elastic Beanstalk environment.
  + You can replace the variable with the contents below:

ALLOWED\_HOSTS = ['\*']

Preparing static files

* Prepare the project static files for deployment with the command:
  + If asked to confirm overwrite, answer "yes"

./manage.py collectstatic

Create Elastic Beanstalk config file

Elastic Beanstalk requires a specific folder and some configuration files added.

* If you are using Cloud9, make sure you **change the settings** in your file tree **to show hidden files**
* **Create a .ebextensions folder** under the project root folder (the main django folder in this example:

mkdir .ebextensions

* **Create a *django.config*** file under the *.ebextensions* folder with following contents:
  + The file is in YAML format, but could also be JSON
  + You can have multiple configuration files for different settings, but they all need to end with ".config"
  + In this case, we are indicating a WSGI server should be used, and setting the location of static files

option\_settings:

aws:elasticbeanstalk:container:python:

WSGIPath: pets\_project.wsgi:application

aws:elasticbeanstalk:environment:proxy:staticfiles:

/static: static

aws:autoscaling:launchconfiguration:

DisableIMDSv1: true

Freeze required libraries

We have installed a number of libraries in the course of creating this application. They will be needed to run the application in a different environment.

* Use the **pip freeze command** to generate to **generate a requirements.txt file**
  + *requirements.txt* is a text file that specifies the Python packages and specific versions required for an application
  + It should be placed in the project root folder (the *django* folder in our examples)

pip freeze > requirements.txt

* The requirements.txt should look like this:

asgiref==3.8.1

boto3==1.36.21

botocore==1.36.21

Django==5.1.6

django-storages==1.14.5

gunicorn==23.0.0

jmespath==1.0.1

packaging==24.2

pillow==11.1.0

PyMySQL==1.1.1

python-dateutil==2.9.0.post0

PyYAML==5.3.1

s3transfer==0.11.2

six==1.17.0

sqlparse==0.5.3

typing\_extensions==4.12.2

urllib3==2.3.0

Note IAM role provided with permissions for Elastic Beanstalk

Elastic Beanstalk will require permissions to execute commands on your behalf to manage your application. We would normally need to create an IAM role to support that, but the Lab already has one created.

* In the **IAM Dashboard**, locate an IAM role called *ebInstanceRole*
  + You could use a different name, but this is the name used in the Lab
* Note these basic Elastic Beanstalk AWS Managed policies in it:
  + *AWSElasticBeanstalkEnhancedHealth*
  + *AWSElasticBeanstalkService*
  + *AWSElasticBeanstalkWebTier*
  + *AWSElasticBeanstalkWorkerTier*
  + *AmazonS3FullAccess*

**IMPORTANT NOTE**: **Access permissions** are something **very specific to each use case and organization**, so the values above are just my example. The full discussion of what IAM policies might apply to your use case is beyond the scope here.

Setting DEBUG flag to False

eLearning has an additional step to update the DEBUG flag to False in *settings.py* before deploying, and **they are correct**. That's what we should do in a production deployment. However, I will not do this here, since we're experimenting, and we might move back and forth between our local environment and Elastic Beanstalk.

### Heading: Creating an Elastic Beanstalk application and environment

#### Demo: Installing the Elastic Beanstalk CLI

* **Install the Elastic Beanstalk CLI** to interact with Elastic Beanstalk from our local environment
  + You can ignore errors related to botocore dependencies

pip install awsebcli

* Now we should be able to use *eb* commands in subsequent steps to create and manage our application.

#### Demo: Create an Elastic Beanstalk application, and deploy to an environment

Initialize the Elastic Beanstalk application

* **Execute the *eb init* command** to create an Elastic Beanstalk application
  + **Replace YOUR\_REGION** with the AWS region where the lab is running (likely us-west-2).
  + *pets-project-ebapp* is our Elastic Beanstalk application name
  + If these options had not been provided, the *eb init* command would have prompted you for them

eb init --region **YOUR\_REGION** --platform python-3.11 pets-project-ebapp

* **Verify** the **application** was **created** in the AWS Console **Elastic Beanstalk dashboard**
  + The *pets-project-ebapp* application should appear in the *Applications* list
  + Remember that the application will not be deployed until we have created an Environment for it
* Run the following command to **create/update an *.ebignore* file** with a line in it so that Elastic Beanstalk does not try include your Python virtual environment in the deployment environment
  + You can include other folders in the file that you don't want to deploy

touch .ebignore

echo ".venv" >> .ebignore

Create an environment and deploy initial version of the application

* **Execute *eb create*** to create an Elastic Beanstalk environment and make a first deployment of the application
  + **Replace YOUR\_REGION** with the AWS region where the lab is running (likely us-west-2)
  + Remember, the values for $vpcId, $subnetId and $ebSecGroupId were set earlier

eb create pets-project-env \

--single --instance\_type t3.micro \

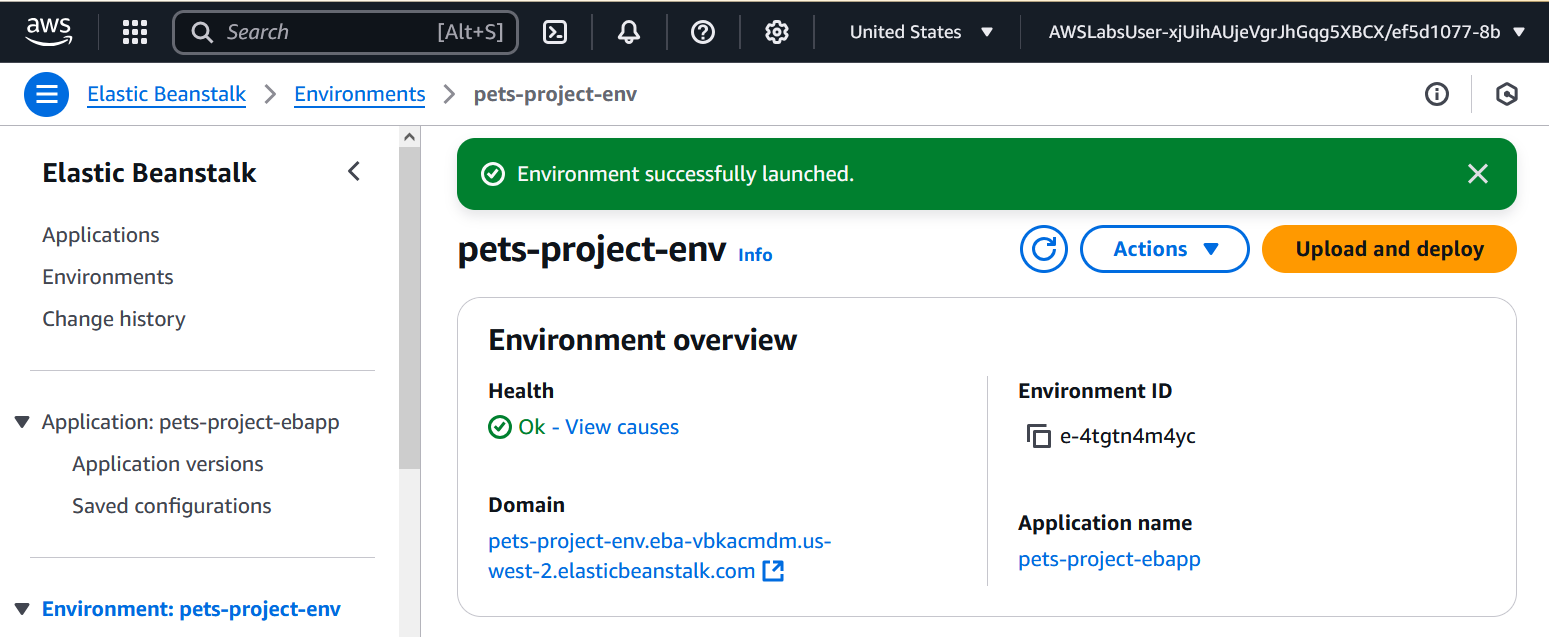
--instance\_profile ebInstanceProfile \

--service-role ebInstanceRole \

--vpc.id $vpcId --vpc.ec2subnets $subnetId --vpc.securitygroups $ebSecGroupId \

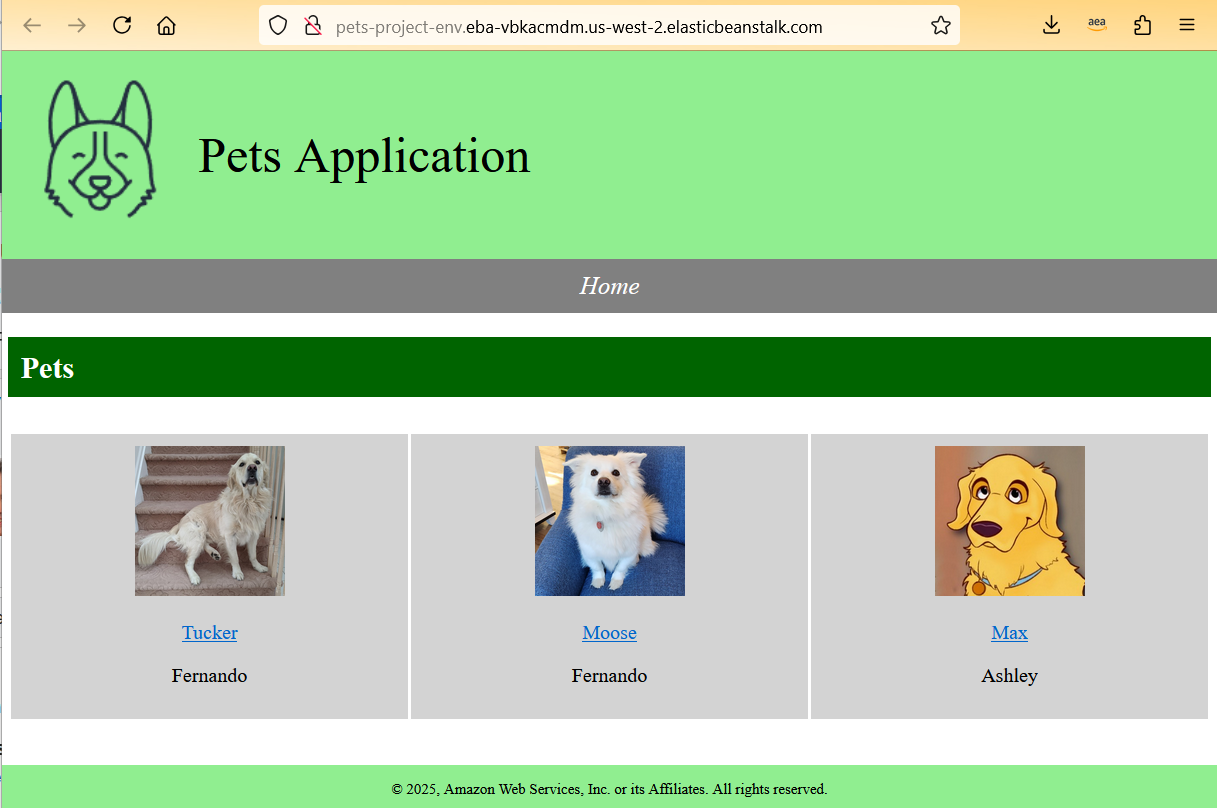
--region **YOUR\_REGION**

* You can follow the progress of the deployment in the command line, or in the Elastic Beanstalk dashboard in the console
  + Select Environments on the left, and click on the "pets-project-env" link
* If the deployment is successful, you should see something like the following in the dashboard



Test the deployed application

* Click on the link under "Domain" in the Environment page above to launch the application:



1. Common Steps
2. Activating Virtual Environment in a new Terminal

* cd into the newly created *django* folder, and activate the virtual environment:

cd django

source .venv/bin/activate

* Note the changed prompt with (.venv) in the beginning:

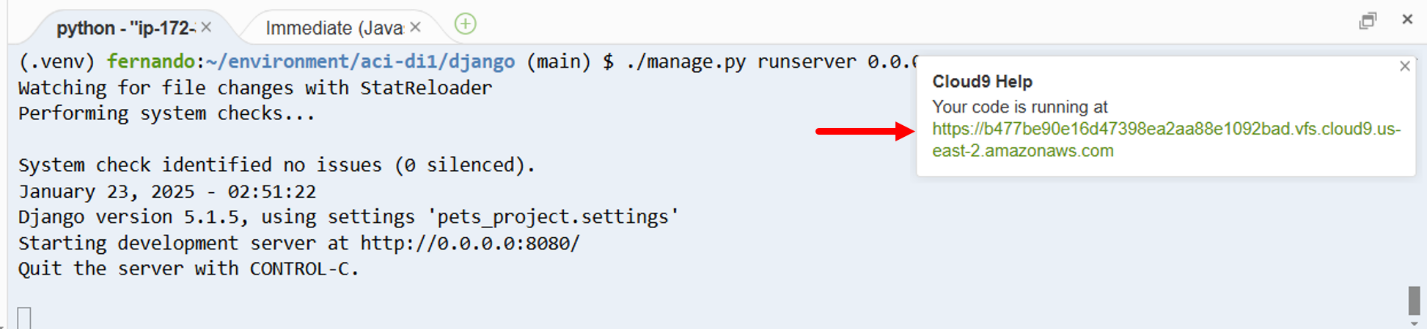


1. Previewing application in Cloud9

* Run local server using *manage.py*

./manage.py runserver 0.0.0.0:8080

* Open up Preview window using recommended URL shown under "Cloud9 Help"



* Alternatively, once the server is running, select *Preview 🡪 Preview Running Application*

1. Setting up Django, Pet Project and Pet Application in a fresh environment

This step should only be needed if your Practice Environment gets terminated. This is essentially a summary of the first demos in this document. There is more detail provided in the steps there. In here, I'm just trying to make it fast to copy/paste.

Setup project environment

mkdir django

cd django

python -m venv .venv

source .venv/bin/activate

Install Django

pip install django

Create pets\_project

django-admin startproject pets\_project .

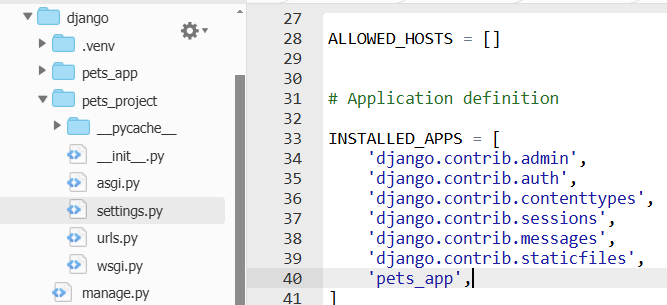
Create an application within the project

./manage.py startapp pets\_app

Updates to Django settings.py

NOTE: If you have a saved copy of settings.py, you can restore the file from that copy instead of the manual updates in this subsection.

* In the *pets\_project* folder, update the *settings.py* file
  + Append *pets\_app* in INSTALLED\_APPS



* Cloud9 requires some special setup to preview Django applications
  + Add '.amazonaws.com' in the ALLOWED\_HOSTS
  + Comment out XFrameOptionsMiddleware under MIDDLEWARE



Apply migrations:

./manage.py makemigrations

./manage.py migrate

Recover code and data from backup:

If you backed up you project earlier, you should be able to upload and replace most of the critical files, such as:

* *settings.py*
* *models.py*
* *views.py*
* *urls.py*
* *admin.py*
* template files
* fixtures
* And a few others I'm probably forgetting

If you had already inserted data, you might be able to recover it as well. You

* **db.sqlite3** – This file locate on the root of your project folder should contain all your data, so if you had downloaded it earlier, you should be able to upload it to the newly created project, replacing the initial one created during installation

If you included a new model and/or, make sure you repeat the "Apply migrations" step:

./manage.py makemigrations

./manage.py migrate

1. Backing up critical project files into a zip archive

As I mentioned earlier, it's very important to backup your practice environment files regularly in cases something goes wrong. However, downloading the entire project folder may take a lot of time and space, because it includes a lot of internal installation files. So the following command generates a zip file of the project folder excluding a lot of the internal files.

* The commands assume that your project was created in a "*django*" folder, under your standard Cloud9 "*environment*" folder (that's the folder you always start at in Cloud9):
  + The last line ("cd –") returns you back to wherever directory you were on before running the commands

cd ~/environment

zip -r pets\_project.zip django -x django/.venv/\\* django/\\*/\_\_pycache\_\_/\\*

cd -

* The command will create a *pets\_project.zip* file under the Cloud9 *~/environment* folder
* You can download that ZIP file to your local environment using the standard Cloud9 download feature

1. Restoring your environment if you have a pets\_project.zip backup

If you have a backup as describe on the previous step, you can restore your whole environment with these steps:

Setup virtual environment and Django project

* Copy/paste these commands in your default Cloud9 environment
  + You can paste them all together

mkdir django

cd django

python -m venv .venv

source .venv/bin/activate

pip install django

django-admin startproject pets\_project .

./manage.py startapp pets\_app

Upload the pets\_project.zip file in Cloud9

* Upload *pets\_project.zip* to the Cloud9 environment using the "*File 🡪 Upload Local Files ...*" menu option
  + The file should be uploaded to the default "environment" folder, so **confirm you see it there**
  + **NOTE**: If you changed the name of the *pets\_project.zip* file to something else (I add different version numbers to mine), rename it back to *pets\_project.zip* for the next step.

Unzip and copy files from your backup into the newly created pets project

* Copy/paste the commands below, which will respectively:
  + cd you to the ~/environment folder where the *pets\_project.zip* should be
  + extract the contents into a "backup" folder
  + copy the contents from the backup folder on top of the base pets project

cd ~/environment

unzip pets\_project.zip -d backup

cp -Rf backup/django .

cd -

* You should be all set now!
* Confirm the application works previewing as usual:

./manage.py runserver 0.0.0.0:8080