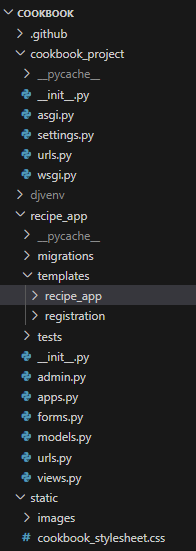
Cookbook App

Stephanie Mayes

CS 3300

# Base Environment Setup

* Install python and create virtual environment
* Activate virtual environment
* Install Django
* Create Django project
  + django-admin startproject <project\_name> (project\_name will be “cookbook\_project”)
* Create directory for the project
  + Should look similar to this, but not all items will be there yet



* Run server
  + python manage.py runserver
* Navigate to http://localhost:8000 to ensure the server is running. Should give generic splash screen for Django

# Baseline App Setup – Homepage

* Activate virtual environment
* Create the app
  + python manage.py startapp <app\_name> (app\_name will be “recipe\_app”)
* Configure project’s settings.py INSTALLED\_APPS list to include app name
* Configure project’s settings.py AUTHENTICATION\_BACKENDS list to include support for user authentication

AUTHENTICATION\_BACKENDS = [

    'django.contrib.auth.backends.ModelBackend'

]

* Update project’s urls.py file to include the built-in admin panel path and the app’s url paths

from django.contrib import admin

from django.urls import path, include

urlpatterns = [

    path('admin/', admin.site.urls),

  #connect path to recipe\_app urls

    path('', include('recipe\_app.urls')),

]

* Create templates folder in app folder
* In templates folder, create recipe\_app folder
* In templates/recipe\_app folder: create base\_template.html file

{% load static %}

<!DOCTYPE html>

<html lang="en">

<head>

    <title>Mayes Family Cookbook</title>

    <meta charset="utf-8" />

    <meta name="viewport" content="width=device-width, initial-scale=1" />

    {% load django\_bootstrap5 %}

    {% bootstrap\_css %}

    {% bootstrap\_javascript %}

    {% bootstrap\_messages %}

    <link rel="stylesheet"  href="/static/cookbook\_stylesheet.css">

    <link rel="icon" href="/static/images/mfc-favicon-color.png">

    <link href='https://fonts.googleapis.com/css?family=Bellota Text' rel='stylesheet'>

</head>

<body>

    <!-- Navbar -->

    <nav id="navbar" class="navbar navbar-expand-sm">

        <div class="container-fluid">

            <a class="navbar-brand" href="#">

                <img src="{% static 'images/mfcLogo\_resized.png' %}" alt="Mayes Family Cookbook logo">

            </a>

            <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target="#navbarNav"

                aria-controls="navbarNav" aria-expanded="false" aria-label="Toggle navigation">

                <span class="navbar-toggler-icon"></span>

            </button>

            <div class="collapse navbar-collapse" id="navbarNav">

                <div class="navbar-nav">

                    <a class="nav-link" href="{% url 'index' %}">Home</a>

                    <a class="nav-link" href="{% url 'recipes' %}">Recipes</a>

                    {% if user.is\_authenticated %}

                        <a class="nav-link" href="{% url 'logout' %}"> Logout </a>

                    {% else %}

                        <a class="nav-link" href="{% url 'login' %}"> Login </a>

                    {% endif %}

                </div>

            </div>

        </div>

    </nav>

    <!-- svg for bootstrap v5 alert icons -->

    <svg xmlns="http://www.w3.org/2000/svg" style="display: none;">

        <!-- triangle exclamation icon -->

        <symbol id="exclamation-triangle-fill" fill="currentColor" viewBox="0 0 16 16">

            <path

                d="M8.982 1.566a1.13 1.13 0 0 0-1.96 0L.165 13.233c-.457.778.091 1.767.98 1.767h13.713c.889 0 1.438-.99.98-1.767L8.982 1.566zM8 5c.535 0 .954.462.9.995l-.35 3.507a.552.552 0 0 1-1.1 0L7.1 5.995A.905.905 0 0 1 8 5zm.002 6a1 1 0 1 1 0 2 1 1 0 0 1 0-2z" />

        </symbol>

    </svg>

    <!-- add block content from html template -->

    {% block content %}

    {% endblock %}

</body>

</html>

* Create index.html file in templates/recipe\_app folder

<!-- inherit from base.html-->

{% extends "recipe\_app/base\_template.html" %}

<!-- Replace block content in base\_template.html -->

{% block content %}

<div id="indexBody" class="container-fluid text-center">

    <hr id="hzIndex">

    <h1 id="indexH1">Mayes Family Cookbook</h1>

    <hr id="hzIndex">

    <p id="indexOverview">

        Welcome to the Mayes Family Cookbook!

        <br>

        <p>Can't figure out what to have for dinner? <br>

            Look on the "Recipes" tab for ideas or to add your own!</p>

    </p>

</div>

{% endblock %}

* Update app’s views.py file to create the “home” page (index.html is our home)

from django.shortcuts import render, get\_object\_or\_404, redirect

from django.http import HttpResponse

from .models import \*

from django.views import generic

from django.contrib import messages

from recipe\_app.forms import \*

from django.contrib.auth.forms import UserCreationForm

from django.contrib.auth import authenticate, login, logout

from django.contrib.auth.decorators import login\_required

# Create your views here.

def index(request):

#render html template index.html with data in context var

    return render(request, 'recipe\_app/index.html')

* Create urls.py file in app with index view

from django.urls import path

from . import views

urlpatterns = [

    #path function: defines a url pattern

    #'' is empty to represent based path to app

    # views.index is the function defined in views.py

    # name='index' parameter is to dynamically create url

    # example in html <a href="{% url 'index' %}">Home</a>.

    path('', views.index, name='index'),

* Start the server and navigate to webpage, should display our home page (index)
  + python manage.py runserver
* Add static folder in the main cookbook folder (see file structure above for reference)
* Add images folder in the static folder
* Update project’s settings.py file so that the app can reference the static files

# include import statements at the top of the file:

import os

from pathlib import Path

# . . . more code here . . . #

# Static files (CSS, JavaScript, Images)

# https://docs.djangoproject.com/en/4.2/howto/static-files/

STATIC\_URL = 'static/'

STATICFILES\_DIRS = [

    os.path.join(BASE\_DIR, 'static')

]

MEDIA\_URL = '/images/'

# Understanding and creating Models, Views, and Templates for this app

## Models

* Create the recipe model in the app’s models.py

from django.db import models

from django.urls import reverse

from django import forms

# Create your models here.

#create recipe model

class Recipe(models.Model):

    title = models.CharField(max\_length=200, blank=False)

    createdBy = models.CharField(max\_length=20, blank=False)

    ingredients = models.TextField(blank=False)

    directions = models.TextField(blank=False)

    def \_\_str\_\_(self):

        return self.title

    def get\_absolute\_url(self):

        return reverse('recipe-detail', args=[str(self.id)])

* + Models are define by a class
  + The recipe model has 4 fields: title, createdBy, ingredients, directions
    - Each field has a specific type and different requirements for defining them. See the Django documentation for specifics
    - All 4 fields are required, as indicated by “blank=false”
  + The “\_\_str\_\_” function defines the default string that Recipe object will return, in our case we are returning the Recipe model object’s title.
  + The “get\_absolute\_url” function returns the URL that accesses an instance of the Recipe model
  + In the database, an **instance** of the model (a recipe) corresponds to the **rows**, and the model’s/instance’s **fields** (title, ingredients, etc.) correspond to the **columns**

## Views, Templates

* Create the paths to the views for the app by updating the app’s urls.py file

from django.urls import path

from . import views

urlpatterns = [

    #path function: defines a url pattern

    #'' is empty to represent based path to app

    # views.index is the function defined in views.py

    # name='index' parameter is to dynamically create url

    # example in html <a href="{% url 'index' %}">Home</a>.

    path('', views.index, name='index'),

    #add view paths

    path('recipes/', views.RecipeListView.as\_view(), name='recipes'),

    path('recipe/<int:pk>', views.RecipeDetailView.as\_view(), name='recipe-detail'),

    path('recipes/create\_recipe/', views.createRecipe, name='create\_recipe'),

    path('recipes/delete\_recipe/<int:pk>', views.deleteRecipe, name='delete\_recipe'),

    path('recipes/update\_recipe/<int:pk>', views.updateRecipe, name='update\_recipe'),

    #add registration/login/logout pages

    path('accounts/register/', views.registerPage, name='register\_page'),

    path('accounts/login/', views.loginPage, name='login'),

    path('accounts/logout/', views.logoutUser, name='logout'),

]

* + Django has built-in list view and detail view functions already (generic views)
    - “recipes” and “recipe-detail” are examples of the built-in views
  + See code comment for syntax explanations
* Now create/update the views in the app’s views.py folder

from django.shortcuts import render, get\_object\_or\_404, redirect

from django.http import HttpResponse

from .models import \*

from django.views import generic

from django.contrib import messages

from recipe\_app.forms import \*

from django.contrib.auth.forms import UserCreationForm

from django.contrib.auth import authenticate, login, logout

from django.contrib.auth.decorators import login\_required

# Create your views here.

def index(request):

#render html template index.html with data in context var

    return render(request, 'recipe\_app/index.html')

#create generic recipe list and detail views

class RecipeListView(generic.ListView):

    model = Recipe

class RecipeDetailView(generic.DetailView):

    model = Recipe

#create view for new recipe using forms

@login\_required(login\_url='login')

def createRecipe(request):

    #create a new recipe form

    form = RecipeForm()

    #form handling

    if request.method == 'POST':

        form = RecipeForm(request.POST)

        #check if the form is valid

        if form.is\_valid():

            #Save the recipe to db

            form.save()

            #redirect back to recipe list view page

            return redirect(reverse('recipes'))

            #return redirect(reverse('recipes'))

    #update dictionary

    context = {'form': form}

    #render the form

    return render(request, 'recipe\_app/recipe\_form.html', context)

#create view to delete recipe

@login\_required(login\_url='login')

def deleteRecipe(request, pk):

    #get recipe to delte

    recipe = get\_object\_or\_404(Recipe, pk=pk)

    #set dictionary vals

    context = {'title': recipe.title}

    #handle deleting

    if request.method == "POST":

        recipe.delete()

        #redirect to recipe list

        return redirect(reverse('recipes'))

    #display delete confirmation

    return render(request, 'recipe\_app/recipe\_delete.html', context)

#create view to update recipe

@login\_required(login\_url='login')

def updateRecipe(request, pk):

    #init dictionary

    context = {}

    #get recipe to update

    recipe = get\_object\_or\_404(Recipe, pk=pk)

    #pass recipe as form instance

    form = RecipeForm(request.POST or None, instance=recipe)

    #save form data and redirect

    if form.is\_valid():

        form.save()

        return redirect(reverse('recipes'))

    #add form to dictionary and render

    context["form"] = form

    return render(request, 'recipe\_app/recipe\_update.html', context)

#create view for creating a user

def registerPage(request):

    if request.user.is\_authenticated:

        return redirect('index')

    else:

        form = CreateUserForm()

        if request.method == 'POST':

            form = CreateUserForm(request.POST)

            if form.is\_valid:

                form.save()

                user = form.cleaned\_data.get('username')

                messages.success(request, 'Account was created for ' + user)

                return redirect('login')

        context = {'form':form}

        return render(request, 'registration/register.html', context)

#create view for login

def loginPage(request):

    if request.user.is\_authenticated:

        return redirect('index')

    else:

        if request.method == 'POST':

            username = request.POST.get('username')

            password = request.POST.get('password')

            user = authenticate(request, username=username, password=password)

            if user is not None:

                login(request, user)

                return redirect('index')

            else:

                messages.info(request, "Username OR password is incorrect!")

        context = {}

        return render(request, 'registration/login.html', context)

#create view for logout

def logoutUser(request):

    logout(request)

    return redirect('login')

* + These functions/views handle the information given by users or information retrieved from the database, then they pass it off to the templates for rendering on your screen
  + Some views have been restricted to users who are logged in, denoted by “@login\_required(…)” before the view’s function declaration. If users who have not successfully logged in attempt to visit those pages, they will be redirected to the login page.
  + The views basically handle the “behind screen” logic for what you will eventually see, and the templates handle the presentation.
* In templates/recipe\_app folder, create the html templates for each view. For this project, we will have the following templates:
  + base\_template.html
  + index.html
  + recipe\_delete.html
  + recipe\_detail.html
  + recipe\_form.html
  + recipe\_list.html
  + recipe\_update.html
* Here is an example of a generic view template (recipe\_list.html):

{% extends 'recipe\_app/base\_template.html' %}

{% block content %}

<h1 id="recCards">Recipe Cards</h1>

<!-- add button to create recipe -->

<a id="newRecipeBtn" class="btn" href="{% url 'create\_recipe' %}" role="button"> New Recipe </a>

<!-- if recipe list exists -->

{% if recipe\_list %}

<ul>

    <!-- for each recipe in the list -->

    {% for recipe in recipe\_list %}

    <li>

        <!-- create list item for each recipe -->

        {{ recipe.title }}

        <a id="viewRecipeBtn" class="btn" href="{{ recipe.get\_absolute\_url }}" role="button"> View Recipe </a>

        <a id="delRecipeBtn" class="btn btn-danger" href="{% url 'delete\_recipe' recipe.pk %}" role="button"> Delete </a>

        <a id="updateRecipeBtn" class="btn btn-warning" href="{% url 'update\_recipe' recipe.pk %}" role="button"> Update </a>

    </li>

    {% endfor %}

</ul>

<!-- if no recipe list exists, give msg -->

{% else %}

<p> There are no recipes in the cookbook. </p>

{% endif %}

{% endblock %}

* And an example of a function based view template (recipe\_delete.html)

<!-- inherit from base template -->

{% extends 'recipe\_app/base\_template.html' %}

<!-- Replace block content in base\_template.html -->

{% block content %}

<h1>

    Are you sure you want to delete {{ title }}?

</h1>

<form action="" method="POST">

    {% csrf\_token %}

    <input class="btn btn-primary" type="submit" value="Yes">

    <button onclick="window.history.back();" class="btn btn-danger" type="button">Cancel</button>

</form>

{% endblock %}

* + In this example, we are displaying a confirmation page for the recipe that the user wants to delete.
  + If the user selects yes, the recipe instance will be deleted.
  + If the user selects no, the recipe instance will remain and the user will return to the page they were previously on.
* The remaining templates will need to be built following those examples, but will not be shown in this document

# CRUD – Model Forms

* This project is built primarily off the use of Django’s Model forms
* In the cookbook/recipe\_app folder (see file structure from beginning for reference), create the forms.py file:

from django.forms import ModelForm, modelformset\_factory

from .models import \*

from django.contrib.auth.forms import \*

from django import forms

from django.contrib.auth.models import User

#create class for the recipe form

class RecipeForm(ModelForm):

    class Meta:

        model = Recipe

        fields = ['title', 'createdBy', 'ingredients', 'directions']

        labels = {'createdBy': "Created by"}

#create class for the user registration form

class CreateUserForm(UserCreationForm):

    class Meta:

        model = User

        fields = ['username', 'email', 'password1', 'password2']

* + The RecipeForm uses Django’s built-in ModelForm template
  + The “meta” class within the forms are essentially overrides of the defaults. This is where you can specify what you would like the form to be based off of, and which fields you would like displayed on the form.
  + For the RecipeForm, we are basing the form off of the Recipe model, displaying all of the fields, and changing the displayed label for the recipe’s creator to be something a little more visually appealing.
  + The RecipeForm will be referenced by the following templates:
    - recipe\_form.html (for creating a new recipe)
    - recipe\_update.html (for updating an existing recipe)
    - recipe\_detail.html (for viewing the details of an existing recipe)
  + The CreateUserForm is utilizing Django’s built-in user authentication model’s UserCreationForm. These fields come built-in, see Django’s documentation for further reading.

# User Creation and Authentication

* Main source I used and followed to create all of my user-related tasks:
  + <https://www.youtube.com/watch?v=tUqUdu0Sjyc&list=PL-51WBLyFTg2vW-_6XBoUpE7vpmoR3ztO&index=15>
* In the templates folder, create a registration folder with the following files:
  + logged\_out.html
    - displays a message that user has been logged out and redirects to the login page
  + login.html
    - gives the login form for user to input their username and password
    - verifies that the user has an account, gives error if user does not
    - displays link to registration page and forgot password page
  + password\_reset\_complete.html
    - displays message that password has been reset
    - gives link to login page
  + password\_reset\_confirm.html
    - checks if password reset link is valid, if not gives error message
    - gives form for resetting password
    - verifies no errors (passwords match, meet requirements, etc.)
  + password\_reset\_done.html
    - displays messages that password reset has been requested and instructions for resetting
  + password\_reset\_email.html
    - template for email sent for password reset
    - *\*\*this doesn’t actually function properly, it just sends the link to the console\*\**
  + password\_reset\_form.html
    - gives form for user to fill out to receive password reset link
  + register.html
    - gives the form for creating user account
      * username
      * email
      * password
      * verify password

# Testing

## Unit testing in Django

* Main source followed for creating tests using Django’s testing functionality:
  + https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Testing
* I created test files for models, forms, and views
* For my model testing (test\_models.py) I created a test recipe, checked that the labels were what was expected, checked that the title’s length didn’t exceed the maximum length requirement, and verified that the URL was referencing the correct recipe and information

from django.test import TestCase

#import models

from recipe\_app.models import Recipe

#test the recipe model

class RecipeModelTest(TestCase):

    @classmethod

    def setUpTestData(cls):

        #set up non-modded objs used by all test methods

        Recipe.objects.create(title="TestRecipe")

    def test\_title\_label(self):

        recipe = Recipe.objects.get(id=1)

        field\_label = recipe.\_meta.get\_field('title').verbose\_name

        self.assertEqual(field\_label, 'title')

    def test\_title\_max\_length(self):

        recipe = Recipe.objects.get(id=1)

        max\_length = recipe.\_meta.get\_field('title').max\_length

        self.assertEqual(max\_length, 200)

    def test\_ingredients\_label(self):

        recipe = Recipe.objects.get(id=1)

        field\_label = recipe.\_meta.get\_field('ingredients').verbose\_name

        self.assertEqual(field\_label, 'ingredients')

    def test\_directions\_label(self):

        recipe = Recipe.objects.get(id=1)

        field\_label = recipe.\_meta.get\_field('directions').verbose\_name

        self.assertEqual(field\_label, 'directions')

    def test\_get\_abs\_url(self):

        recipe = Recipe.objects.get(id=1)

        self.assertEqual(recipe.get\_absolute\_url(), '/recipe/1')

* For forms testing (test\_forms.py) I just verified that the password labels were displaying correctly. If they were wrong and showed “username” but expected the password, the user would be confused and would likely get locked out of their account (if I had that set up)

from django.test import TestCase

from recipe\_app.forms import CreateUserForm

#create tests for user registration form

class CreateUserFormTest(TestCase):

    def test\_create\_user\_form\_password1\_field(self):

        form = CreateUserForm()

        self.assertTrue(form.fields['password1'].label is None or form.fields['password1'].label == 'Password')

* For my view tests (test\_views.py) I registered a test user, created a test recipe while the user was logged in, and then tested if a user could create a recipe if they were not logged in.
  + Note: I had to delete the user after every test because I did not set it up to have it destroy the test user

from django.test import TestCase

from django.contrib.auth import get\_user\_model

from recipe\_app.models import Recipe

from django.urls import reverse

#create test for restricted recipe creation view

User = get\_user\_model()

class CreateRecipeViewTest(TestCase):

    #create a test user

    def setUp(self):

        user1 = User.objects.create(

            username="testUser1",

            email="user1@test.com",

            password="testPW123",

            #password2="testPW123"

            )

        user2 = User.objects.create(

            username="testUser2",

            email="user2@test.com",

            password="testPW123",

            #password2="testPW123"

            )

        user1.save()

        user2.save()

        #create a recipe

        recipe = Recipe.objects.create(

            title="testRecipe",

            ingredients="ingr1, ingr2",

            directions="step1, step2"

        )

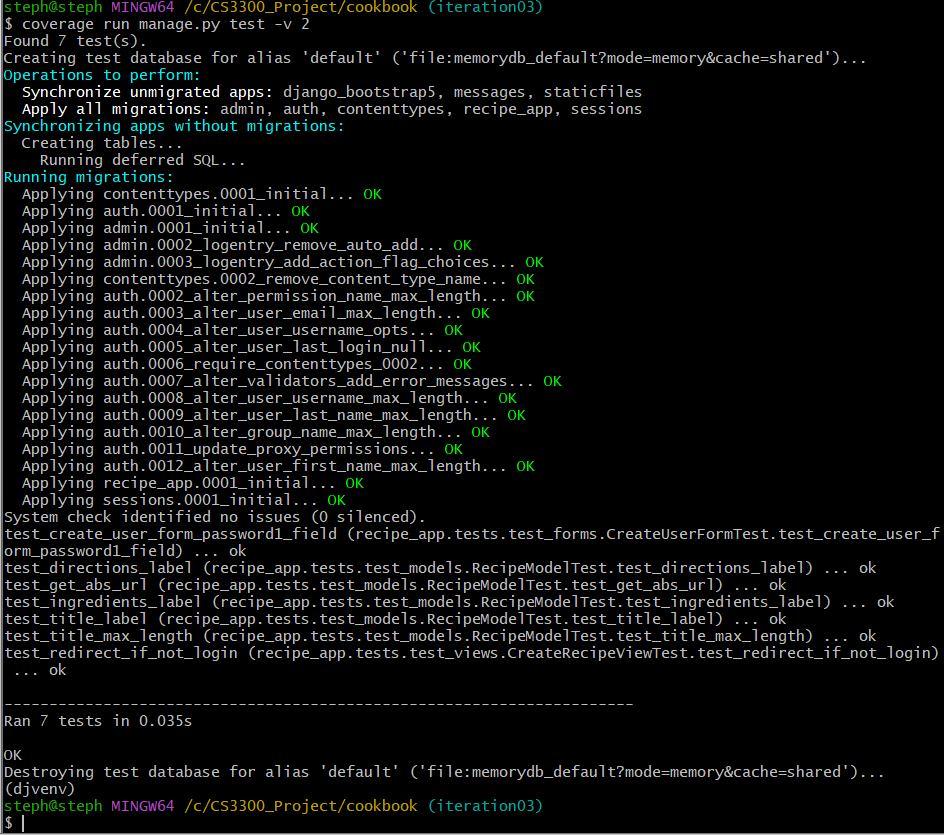
        recipe.save()

    def test\_redirect\_if\_not\_login(self):

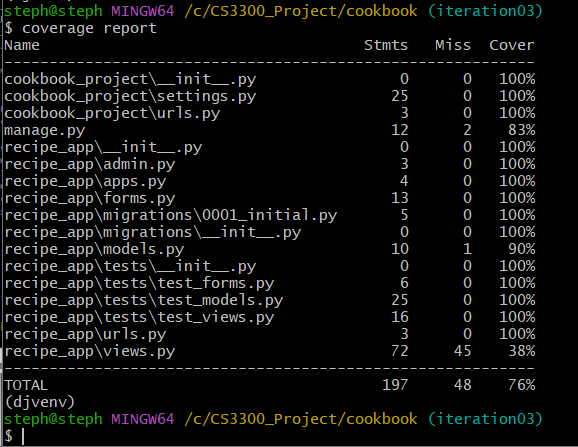
        response = self.client.get(reverse('create\_recipe'))

        self.assertRedirects(response, '/accounts/login/?next=/recipes/create\_recipe/')

* I used “coverage” to run my unit tests. Link to coverage documentation
  + <https://coverage.readthedocs.io/en/7.3.2/>
* Output from running the test:

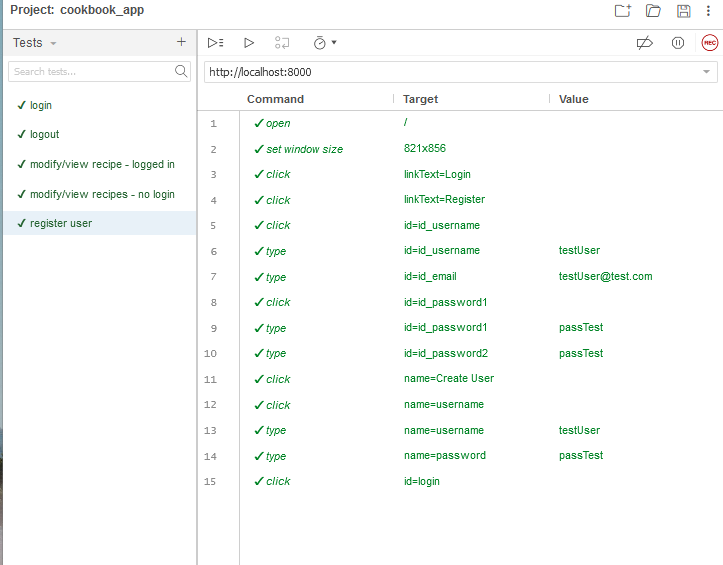


* To get my test coverage report for unit tests, I used coverage again:



## Testing with Selenium

* I couldn’t quite get the syntax down for creating the selenium test file outright, so I utilized the Selenium IDE browser extension to create my tests. Installation/getting started link:
  + <https://www.selenium.dev/selenium-ide/docs/en/introduction/getting-started>
* Once the extension is installed, it records your interactions with the webpage as test items/actions—uses the exact same clicks/keyboard input/movements you do for its automated testing.
* Once you have your test(s) created, you can run them. It will tell you if the test passed/failed. Sample output after running the test using the extension:



* In order to create actual testing files with selenium, I just exported the test suite to a python file. Example of the exported test suite code:

# Generated by Selenium IDE

import pytest

import time

import json

from selenium import webdriver

from selenium.webdriver.common.by import By

from selenium.webdriver.common.action\_chains import ActionChains

from selenium.webdriver.support import expected\_conditions

from selenium.webdriver.support.wait import WebDriverWait

from selenium.webdriver.common.keys import Keys

from selenium.webdriver.common.desired\_capabilities import DesiredCapabilities

class TestTestSet():

  def setup\_method(self, method):

    self.driver = webdriver.Firefox()

    self.vars = {}

  def teardown\_method(self, method):

    self.driver.quit()

  def test\_login(self):

    self.driver.get("http://localhost:8000/")

    self.driver.set\_window\_size(821, 856)

    self.driver.find\_element(By.LINK\_TEXT, "Login").click()

    self.driver.find\_element(By.NAME, "username").click()

    self.driver.find\_element(By.NAME, "username").send\_keys("tester")

    self.driver.find\_element(By.NAME, "password").send\_keys("dummy123")

    self.driver.find\_element(By.ID, "login").click()

  def test\_logout(self):

    self.driver.get("http://localhost:8000/")

    self.driver.set\_window\_size(821, 856)

    self.driver.find\_element(By.LINK\_TEXT, "Logout").click()

  def test\_modifyviewrecipeloggedin(self):

    self.driver.get("http://localhost:8000/")

    self.driver.set\_window\_size(821, 856)

    self.driver.find\_element(By.LINK\_TEXT, "Login").click()

    self.driver.find\_element(By.NAME, "username").click()

    self.driver.find\_element(By.NAME, "username").send\_keys("tester123")

    self.driver.find\_element(By.NAME, "username").click()

    self.driver.find\_element(By.NAME, "username").send\_keys("tester")

    self.driver.find\_element(By.NAME, "password").send\_keys("dummy123")

    self.driver.find\_element(By.ID, "login").click()

    self.driver.find\_element(By.LINK\_TEXT, "Recipes").click()

    self.driver.find\_element(By.LINK\_TEXT, "View Recipe").click()

    self.driver.find\_element(By.LINK\_TEXT, "Recipes").click()

    self.driver.find\_element(By.LINK\_TEXT, "Delete").click()

    self.driver.find\_element(By.CSS\_SELECTOR, ".btn-primary").click()

    self.driver.find\_element(By.LINK\_TEXT, "Delete").click()

    self.driver.find\_element(By.CSS\_SELECTOR, ".btn-danger").click()

    self.driver.find\_element(By.LINK\_TEXT, "Update").click()

    self.driver.find\_element(By.ID, "id\_directions").click()

    self.driver.find\_element(By.ID, "id\_directions").click()

    element = self.driver.find\_element(By.ID, "id\_directions")

    actions = ActionChains(self.driver)

    actions.double\_click(element).perform()

    self.driver.find\_element(By.ID, "id\_directions").click()

    self.driver.find\_element(By.ID, "id\_directions").click()

    self.driver.find\_element(By.ID, "id\_directions").click()

    self.driver.find\_element(By.ID, "id\_directions").send\_keys("selenium view/edit recipes test")

    self.driver.find\_element(By.NAME, "Submit").click()

    self.driver.find\_element(By.LINK\_TEXT, "View Recipe").click()

    self.driver.find\_element(By.LINK\_TEXT, "Recipes").click()

    self.driver.find\_element(By.LINK\_TEXT, "New Recipe").click()

    self.driver.find\_element(By.ID, "id\_title").click()

    self.driver.find\_element(By.ID, "id\_title").send\_keys("new recipe")

    self.driver.find\_element(By.ID, "id\_ingredients").send\_keys("a new recipe")

    self.driver.find\_element(By.ID, "id\_directions").click()

    self.driver.find\_element(By.ID, "id\_directions").send\_keys("a new recipe test")

    self.driver.find\_element(By.NAME, "submit").click()

    self.driver.find\_element(By.CSS\_SELECTOR, "li:nth-child(2) > .btn-primary").click()

    self.driver.find\_element(By.LINK\_TEXT, "Home").click()

    self.driver.find\_element(By.LINK\_TEXT, "Logout").click()

  def test\_modifyviewrecipesnologin(self):

    self.driver.get("http://localhost:8000/")

    self.driver.set\_window\_size(821, 856)

    self.driver.find\_element(By.LINK\_TEXT, "Recipes").click()

    self.driver.find\_element(By.LINK\_TEXT, "View Recipe").click()

    self.driver.find\_element(By.LINK\_TEXT, "Recipes").click()

    self.driver.find\_element(By.LINK\_TEXT, "Delete").click()

    self.driver.find\_element(By.LINK\_TEXT, "Recipes").click()

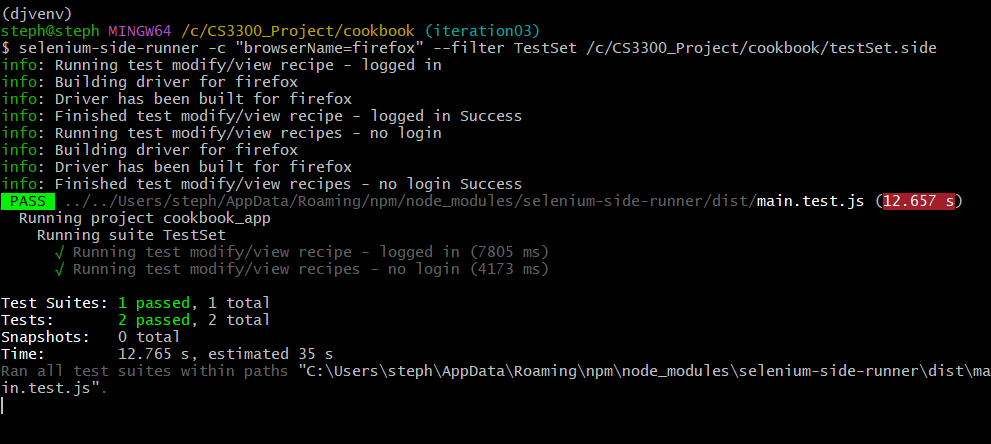
    self.driver.find\_element(By.LINK\_TEXT, "Update").click()

    self.driver.find\_element(By.LINK\_TEXT, "Recipes").click()

    self.driver.find\_element(By.LINK\_TEXT, "New Recipe").click()

    self.driver.find\_element(By.LINK\_TEXT, "Recipes").click()

* Then to run my tests via command line, I used the Selenium command line runner command and specified my browser type, and filtered out which tests I wanted to use.
* Link to help with specifying browser type:
  + https://www.selenium.dev/selenium-ide/docs/en/introduction/command-line-runner
* Link to help with filtering tests:
  + <https://www.qafox.com/new-selenium-ide-filtering-tests-using-command-line-runner/>
* Output from command line runner:



# Exploration Area

* For my exploration area, I chose to focus on creating a more visually appealing/professional app by overriding Bootstrap.
* I mainly focused on using CSS to override Bootstrap. This turned out to be extremely tedious since my app was initially built on the Bootstrap foundation. In order to style my app properly, I had to go in and find which pieces I wanted to modify, give them the correct HTML attributes, and then adjust the styling within the CSS stylesheet.
* Due to the nature of styling, and myself not being proficient in anything graphics-related, the vast majority of my exploration time was spent in a cycle of adjusting one thing in the style sheet and refreshing the webpage, repeating until I made it “good enough”, and then moving onto the next piece to do it all over again.
* Since tweaking each piece took a significant amount of time, I was not able to get to each aspect like I had hoped, and my project is not as polished as I would have liked.
* I did not keep a record of each change, but will highlight some portions that made the final cut and how I created them.
* Sources I relied on for CSS and overriding Bootstrap:
  + <https://www.w3schools.com/cssref/index.php>
  + <https://www.w3schools.com/cssref/css_selectors.php>
  + <https://blog.hubspot.com/website/bootstrap-navbar>
  + <https://www.freecodecamp.org/news/how-to-center-anything-with-css-align-a-div-text-and-more/>
  + <https://developer.mozilla.org/en-US/docs/Learn/CSS/First_steps/How_CSS_is_structured>
* I created a stylesheet in my static folder and linked the reference to it in the base template.
* Within my stylesheet, I created a rule for the items with id=”newRecipeBtn”

#newRecipeBtn {

    background: #a58777;

    color:#4f5561;

    font-weight:bold;

    display:flex;

    align-items: center;

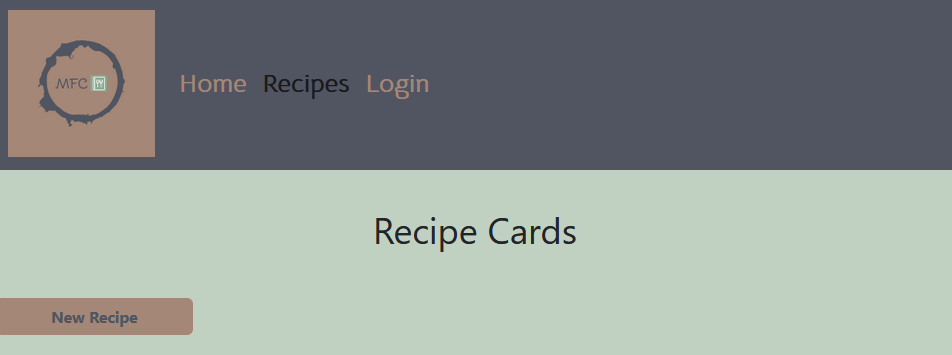
    justify-content: center;

    width: 200px;

    margin-bottom: 20px;

}

* + Background: sets the background color
  + Color: sets the text color
  + Display: sets the type of rendering box
  + Align items: sets the alignment for the items inside the container
  + Justify content: sets the alignment for the items inside the container if all the space around the item isn’t full
  + Width: sets the size of the box
  + Margin bottom: sets the bottom margin of the box
* How the new recipe button looks on the webpage:





* What the “newRecipeBtn” corresponds to in my code:
  + Located in “templates/recipe\_app/recipe\_list.html”

<!-- add button to create recipe -->

<a id="newRecipeBtn" class="btn" href="{% url 'create\_recipe' %}" role="button"> New Recipe </a>

