Django Workshop - Pt 1

Tyler Schwehr

RedRoom CTF

What we’re going to cover

- Web Frameworks & MVC Pattern
- How to start a Django project
- Core concepts
- Templates
- The Admin panel
- CRUD implementation

What is a web framework?

- A web framework simply put is either software of a set of conventions that support the development of a website
- Developing websites can be very complex and require a lot of work
- Web frameworks can provide constraints that enforce consistency, which reduces complexity
- They can also reduce workload by providing pre-built solutions for common problems

Pro’s and Con’s of Web Frameworks

- Frameworks can radically improve development time
- Frameworks often reduce complexity with enforced conventions
- You can achieve a much greater level of sophistication, without needing as much knowledge
- They can help reduce errors and security flaws

- Abstraction often obfuscates underlying details, accruing technical debt
- Frameworks will typically provide lower performance than low level sites
- Reduced flexibility in creating a website

What is ~~MVC~~ *MVT*?

- MVC is a design pattern, and it stands for Model-View-Controller
- We're not going to cover MVC in depth because Django doesn't quite use MVC, we're going to refer to their pattern as MVT
- MVT stands for Model-View-Template instead and more closely resembles the Django framework
- In this pattern, the Model refers to the data model and maps 1:1 with the database schema
- The View is the logic controller which facilitates the requests and returns a HTML document
- The Template is the backbone of the HTML document, and the information is pass into it

Why use the MVT pattern

- This design pattern splits the website into distinct and separate components
- It’s a very modular design and decouples the logic, data, and markup from each other
- You can work on the aspects that you specialise in rather than having to know a bit of everything
- This also makes it easier to have multiple developers work on the same project, as each can work on a separate component
- It’s great for data-driven websites, you can simply upload data to the database and the site will dynamically display it
- It is a highly extensible pattern, allowing you to create new features will relative simplicity

Understanding apps

- One of the unique aspects of Django is the use of apps
- Apps are a further abstraction
- Every project should have at least one app, but I don’t think it’s enforced
- Apps are a logical abstraction and can be used however you like
- Apps can be packaged and used for other projects, increasing code re-usability
- Apps offer you a way to separate out the different components of your information system
- They make it easier to organise and manage the logic of your project
- And they further break down the project making it easier to work as a group

Starting your Django Project

We're going to set up your project in 3 steps

1. Set up your Github repository
2. Set up a virtual environment
3. Install Django

- Creating your setup in this way is good practice
- It helps reinforce using Git as a source control, and gives you the ability to rollback if need be
- Venv allows you to isolate and control the packages for your install
- The manage.py script has many different commands for managing your project

Configuring Django settings

- Django keeps it’s core settings in the project root directory, in a file called ‘settings.py’
- This is where we can modify the key details of any given project
- I recommend briefly having a look through the starting details
- You can find many more optional settings in the documentation
- When creating a new app or starting a new project, the first thing we need to do is link our app to the settings
- You can find the listed apps under ‘INSTALLED_APPS’, simply add your app to this list
- Take a moment to update the timezone and the default language

Django Models

- Simply put models map the information from your database to your website
- In our setup we’re using SQLite, Django will manage just about every interaction you would have with your database
- The models use classes to define a new model, and member fields to assign attributes
- Each app has its own set of models kept in the file models.py
- When we update a model, we need to migrate the changes to the database using the manage.py script

Setting up a model

- To make our own model the first thing we need to do is declare a new class in the relevant models.py file
- The new class should inherit from ‘models.Model’
- The primary key field is set by default, and is named ‘id’
- To give the model a new field, we declare a new variable and make a specific assignment
- Every variable assignment should be done using a method from the models class
- An example of this could be wanting to create a character field, which is done using ‘models.CharField()’
- We pass in parameters according to what constraints we would like to assign
- Defining the __str__ function allows us to return a string where referencing the name of the object

Attaching our model

- There are many locations where we might need to reference our model within our website
- To do so we must import the model into any given script
- If it is within the app that it is declared, it is as simple as using 'from .models import <modelName>'
- One of the key areas that we might want to link our model is in our admin panel
- In the admin.py file of your app we want to register our model, we can do this with this line:

  admin.site.register(<modelName>)

- We can pass a second parameter to modify how the information is displayed, the second parameter should be a new class
- This class should include a tuple of strings named for each member field

The admin panel

- One of the nice features of Django is that it comes pre-built with an admin panel
- The admin panel can be used to directly interface with the data in your database
- It can be further modified to perform additional functions
- We access the admin panel using the URL path '/admin/'
- The superuser has access to the admin panel

Running the site locally

- This is a good time to have a look at how we can see our work so far
- We can run Django locally quite easily since we have the Python interpreter installed
- This will only allow you to view the site locally and will not expose it to other devices on your network
- You can view the site using ‘localhost:8000’

What’s a view?

- A view is the intermediary between the end user and your data
- A user will send a http request to your server, the view accepts this request and performs some logic, and returns the http response
- The view decides what data should be sent back in the response, it can decide this based on a range of factors including the user’s authentication status
- Each app has its own views kept in the views.py file

Building our views

- To build a view, we declare a function in the views.py file
- The function should accept a request as a parameter
- We always return a ‘render’ object
- This object is constructed with three parameters, the request, a template, and the data context
- The context is simply a dictionary
- You declare the key as a string, and the value is the data that you would like to pass in
- You can pass in as much or as little data as you would like

URL routing

- Now that we have some views, the user will need a way to access them
- We can define our URL routes so that when the site receives a specific http request, it passes the request to the right view
- We can also use routes to implicitly pass information about the request, such as including the id of an object in the request
- URLs are written into the urls.py file, this can be in either the root directory of the app directory

Building our routes

- For a simple route, we simply add a new path object to our urlpatterns
- The path object typically takes three parameters, the path pattern, the associated view, and a name for the path
- Here is an example:

```
path("products/", views.index, name = "index")
```

- We can include variable data by using a variable input like this:

```
path("products/<int:product_id>/", ...)
```

- To use URL routing from each app, we need to assign a specific path to that app
- We can use the include keyword to achieve this, here's an example:

```
path("shop/", include("shop.urls"))
```

What is a template?

- For the most part the template is simply a HTML document
- The key difference between straight HTML and a template is that the template is processed by an interpreter on the server, modified and then sent to the user
- The interpreter we’re using is the Django template engine, but we can choose to use others if we so wished
- This interpreter takes the template and the context that we passed in, it replaces our data calls in the template with the data and compiles it into HTML

Setting up our templates

- The first thing we need to do is create a folder in the app called “templates”
- This is a special folder name, and forms part of the system that prevents directory traversal vulnerabilities
- Within the templates folder, we then create another folder that matches our app name
- Within that final folder we can create as many HTML files as we would like
- For now, use Bootstrap for CSS, since I’m having trouble figuring out why my static CSS isn’t working
- You can write these HTML files as you would normally

Adding data to our template

- The context that we declared in our views can be accessed in our template
- To reference the context in the template we use the double curly brackets {{ object }}
- With these we can use a range of functions, we use the percentage symbol for functions {% function %}
- Typically our context includes a list of objects, so the first thing we want to do is iterate over that list
- We can use this loop to iterate

  {% for object in object_list%}

  {% endfor %}
- Within the loop we access the information

  {{ object.name }}
- Line up your loop carefully, otherwise you can encounter some adverse behaviour

Layering templates

- We can layer templates to reduce re-writing the same HTML
- A great way to use this is to create a single header and navigation, and layer that onto every other template
- We can achieve this by creating a base HTML document, and within this document declare where inserted content should be placed
- Use {% block body %} and {% endblock %}
- In the template that we insert into that we start by declaring that this document extends another

{% extends '<appName>/<fileName>' %}

- Then we encapsulate the body content in our block body and endblock content
- We reference the template with the body content that we want, the engine will do the rest

Handling forms

- Django does a lot to handle forms for you
- We need to map form fields to a model for Django to do its work
- That form can be displayed in a template, using the POST method we can manage the form and extract the data

Creating our form

- To create our form, we write a new class in the forms.py file of the given app
- This class should inherit from the ‘forms.ModelForm’ class
- The we declare another class called ‘Meta’
- Within that class we use a field called ‘model’ to assign our model
- We also need a list name ‘fields’
- This list should include a set of strings that match the field names that we want to reference in our model

Using the form in a template

- Whenever we use a form, we need to include {% csrf_token %} within our form tags
- We also need to include a method attribute in our form, is should be ‘POST’
- Now we can loop over our form to grab each field, and display it
- We can also access the ‘name’ property in each field
- When we’re done, we need a submit button to complete the submission

Our creation view

- We’re going to take that form data and insert it into our database
- Create a view as you would typically
- Use a variable which we’ll call ‘form’ and assign it an instantiated object of your form
- When instantiating the object pass this statement as a parameter:

request.POST or None

- The form has a in built method called is_valid() which checks if it is valid
- If it is, we can call the method save()
- The form is now saved in the database
- We might redirect the user by returning a different type of render called redirect()
- You only need to pass in a string with the ‘appName:routeName’

Our update view

- To perform an update is quite similar to create with two key differences
- Firstly we need to get the object which we want to edit
- Your view should accept the id of the object we plan on editing
- Using that we'll retrieve the object from the database using the following method

```
modelName.objects.get(pk = item_id)
```

- Assign this object to a variable
- Secondly when we instantiate our form object, we need to pass in a second parameter
- This parameter is the object that we found in our database
- It should be assigned to the parameter 'instance'

Our delete view

- Deleting is similar to updating, we obtain the object id and get the object from the database
- We perform a check to ensure that the ‘request.method’ is POST
- Then we simply call the delete() method on the object

Thank you

Tyler Schwehr

RedRoom CTF