CSX Website Backend Documentation

Release

BHCC CSX

Jun 10, 2020

Contents

1 Development Documentation **1**

1.1 Documentation . . . 1
1.1.1 Building Documentation . . . 1
1.1.2 Documentation Format . . . 1
1.2 Environment Setup . . . 1
1.2.1 Python . . . 2
1.2.2 Git . . . 4
1.2.3 The Editor . . . 5
1.2.4 Postgres . . . 7
1.3 Using Git . . . 10
1.3.1 Cloning a Repository . . . 10
1.3.2 Pulling Changes . . . 10
1.3.3 Making Changes . . . 10
1.4 Writing a Django App . . . 11
1.4.1 Creating the App . . . 11
1.4.2 Models . . . 11
1.4.3 Migrations . . . 12
1.4.4 Views . . . 13
1.4.5 Templates . . . 14
1.4.6 URLs . . . 15
1.4.7 The `admin.py` File . . . 16

2 Deployment Documentation **17**

CHAPTER 1

Development Documentation

1.1 Documentation

That's right, this is documentation on documentation. (*Its meta-documentation*)

The documentation for the CSX backend uses Sphinx to build documentation. When a new commit to specific branches is made (master, primarily), the repository is automatically pulled, docs built and deployed on readthedocs.org.

1.1.1 Building Documentation

If you're developing new documentation, its likely you'll want or need to test build it before committing to your branch. To achieve this you can use Sphinx to build the documentation locally as it appears on Read the Docs.

Activate your virtual environment (if you're using one) and install the documentation requirements found in `docs/requirements.txt` using pip, e.g. `pip install -r docs/requirements.txt`.

You can then build the docs by changing to the `docs/` directory and running `make html` or `make dirhtml`, depending on how the Read the Docs project is configured. Either should work fine for testing. You can now find the output of the build in the `/docs/_build/` directory.

Occasionally you may need to fully rebuild the documents by running `make clean` first, usually when you add or rearrange toctrees.

1.1.2 Documentation Format

CommonMark Markdown is the current preferred format, via recommonmark. reStructuredText is supported if required, or you can execute snippets of reST inside Markdown by using a code block:

```
```eval_rst
reStructuredText here
```
```

Markdown is used elsewhere on Github so it provides the most portability of documentation from Issues and Pull Requests.

1.2 Environment Setup

Getting setup to develop for the CSX website backend is relatively simple. There are just a few basic requirements.

- *Python 3.6+*
- *Git*
- *An IDE/Editor to write code in*
- *Postgres*
  - This can be locally hosted, or can be on Google Cloud SQL. We will include documentation for both.

1.2.1 Python

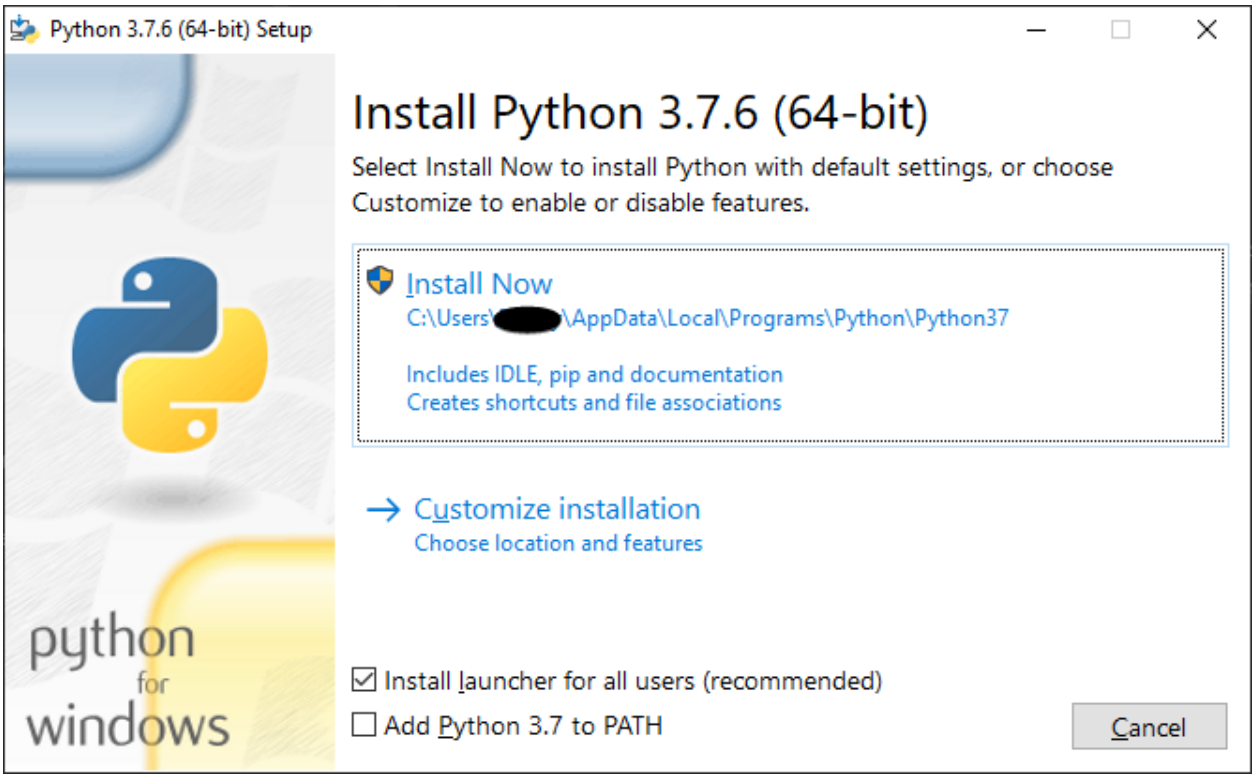
The CSX website backend is a Django project, which means it is written almost exclusively in Python. So in order to be able to contribute to the project you will need to make sure you have Python 3.6 or greater installed.

Installing Python

Windows

To install python on windows you will want to download the installer for any compatible version (anything >3.6) from the python website. (Or, if you are using Windows 10, you can install it from the Windows store, just search for Python)

When you run the installer you should see a window like this:



Note: Keep *Install launcher for all users (recommended)* selected. This will make managing your python versions much easier should you ever need to install other versions.

Just hit `Install Now` and keep any recommended settings on subsequent screens.

Congratulations! Python is now installed on your machine!

To verify the installation, open up a command prompt window and run the command `py` or `py -3`.

Linux

There are many different linux distributions, and many of them handle packages differently. The most popular distribution of linux is by far Ubuntu, so these we will focus on that.

If you are running Ubuntu 18.xx or 19.xx, put your feet up, and relax! You already have a compatible version of python installed. To verify this, open up a terminal window and run the `python3` command. If you are running 19.xx you should see `Python 3.7.x` and if you are running 18.xx you should see `Python 3.6.x` (Release versions may vary slightly depending on exact Ubuntu version)

If you are on any version before 18.04, you will need to check to make sure that the package `python3.7` is available for your version of Ubuntu. To do this run the following command

```
$ apt search ^python3.[0-9]$
```

This command will search the package repositories and return all the python version packages that are available for you to install. If to install one of the available packages just run `sudo apt-get install` followed by the package name. You can then verify the installation by running `python3.x` where `x` represents the compatible minor version that you installed.

Note: Ubuntu 17.10 may have a compatible version of python3 installed by default.

---

macOS

MacOS comes with Python 2 installed, however to avoid any compatability concerns we require a Python version greater than 3.6. You can find a free download for any Python version from the python website.

Double click on your downloaded .pkg file to run the installer and follow the instructions. There is no need to customize the installation.

Congratulations! Python is now installed on your machine!

To verify the installation, open up a command prompt window and run the command `python3`.

Note: On macOS, *python* will run Python 2. To use the correct version of Python for the project you must specify *python3*.

Setting Up a Python Virtual Environment

What is a Virtual Environment?

A virtual environment is an isolated working copy of python, which allows you to work on a specific project without affecting other projects that you might be working on.

Before we create a virtual environment, it is a good idea to create a folder in which to put our virtual environment folders. This can be anywhere you like, and called what ever you like. Just make sure you know what it is called, and that it is in a convenient place for you to access.

Now that we have a folder to hold out virtual environments, lets create one!

1. First, open a terminal window (or Command Prompt for those on Windows), and navigate to the folder that you made to hold your virtual environments.
2. Run the following command `python -m venv NAME` where `NAME` is the name of your virtual environment.
3. To check that the virtual environment was created, you can check to see if there is a folder that now matches what you called the virtual environment you created.

Warning:

- *python* is used to be generic, on windows you may have to use *py* or *py -3* in order to run python commands from the command line. Linux and mac users may need to use *python3*
- In somecases *venv* does not work, you can instead try replacing it with *virtualenv*
- Linux users may need to install *python3-venv* (or *python3.x-venv* if you had to install your python version after the fact)

Using a Python Virtual Environment

To use a venv (virtual environment) you must first activate. When you activate a venv you are telling your terminal to use it for anything python related, rather than your system-wide installation.

Like many things, activating your venv is slightly different on Windows than it is on Linux or macOS. Luckily, once activated, all the commands are the same going forward.

- Windows: Navigate to `PATH\TO\VENV\Scripts\` and run `activate.bat`
- Linux/macOS: Run the command `source /path/to/venv/lib/activate`

You will know you that the venv is active when you see `(NAME)` (where `NAME` is the name fo your venv) at the beginning of every prompt.

While the venv is active, you can run python using the `python` command, regardless of OS or python version.

To deactivate the venv, simply run the command `deactivate`.

Note: ALL other project documentation will assume you are running with your venv active.

1.2.2 Git

This project is developed using Git for version control. Git allows multiple people to work on the same project without getting in each other's way, and makes tracking and reversing changes simple.

Installing Git

Windows

The Git windows installer can be found at git-scm.com. The installation is very simple, just keep all the default settings in the installer, and you'll be good to go.

Once git is installed, you will want to run the following commands either in Git Bash or in Command Prompt to set some global git settings.

```
$ git config --global user.name "FIRST_NAME LAST_NAME"
$ git config --global user.email "MY_NAME@example.com"
```

Linux

Installing git on Ubuntu is a very simple process. All you need to do is open up a terminal window and run the following commands:

```
$ sudo apt-get install git
```

Once installed you will want to run the some commands to set global git settings.

```
$ git config --global user.name "Your Name"
$ git config --global user.email "youremail@yourdomain.com"
```

macOS

Depending on your version of macOS, you might already have git installed. To check, open a terminal window and run the following command:

```
$ git --version
```

If you get a see `git version x.xx.xx` come up, that means that it is already installed. If not, you will have to install it yourself.

Just like on windows you can find the Git Installer for mac at git-scm.com. Just download and run the installer.

Once installed (or after you have verified that it came preinstalled) you will want to run the following commands to set some global settings.

```
$ git config --global user.name "Your Name"
$ git config --global user.email "youremail@yourdomain.com"
```

Using Git

Git is a command line tool for managing git repositories. However, just because it is a command line tool does not mean that you have to use it in the command line. There are many GUI Git clients out there that make working with git very user friendly.

You can find a list of Git GUI clients on the GUI Clients page at git-scm.com.

For more information about using git, checkout our *Git Help Documentation*.

1.2.3 The Editor

The two most common editors you will likely come across for python are VSCode and PyCharm, though you can use anything you want to write python code.

VSCode

VSCode, or Visual Studio Code is an open source, cross platform text editor developed and maintained by Microsoft. Popular among web developers, VSCode has a similar look and feel to Microsoft's Visual Studio line of IDEs which are commonly used for C++ and C# development.

Though the editor itself is fairly bare bones, it has an extensive extension library to add all sorts of functionality as well as modify the look and feel of the application.

For the purposes of this project there are no extensions that you *need* other than the python extension authored by Microsoft.

Configuring VSCode

Coming Soon

PyCharm

PyCharm is a Python IDE made by the same people that make the popular Java IDE IntelliJ. In fact, PyCharm runs on the same code base as IntelliJ, meaning if you are familiar with the layout and workings of IntelliJ, you will be right at home with PyCharm.

There are two versions of PyCharm available. Community and Pro.

The Community edition is free while the Pro version is a paid product.

Feature	Community	Pro
IntelliJ Based Editor	Y	Y
Built in Debugger and Test runner	Y	Y
Intelligent Refactoring and Navigation	Y	Y
Code Inspection	Y	Y
VCS (Git) Support	Y	Y
Scientific Tools	N	Y
Web Dev (WebStorm Features Bundled)	N	Y
Python Web Frameworks	N	Y
Python Performance Profiler	N	Y
Remote Development	N	Y
Database & SQL Support	N	Y

Though PyCharm Pro is a paid product, you can apply for a Student license for free here. If you plan to use PyCharm it is a good idea to get the free license to have access to the extra features that Pro provides, however the community version would work just fine.

PyCharm Project Settings

Once you have setup pycharm, and opened the project, there are some settings that you will want to configure.

Project Interpreter

First, you will want to set the project interpreter. This is what pycharm will use to give you hints when you use libraries. To open the settings window, you can go to `File > Settings` or hit `Ctrl+Alt+S`. Then find `Project: <project-name>` in the sidebar, and expand it to find the `Project Interpreter` settings. To add an interpreter, click the cog (⚙) and select `Add New`.

In the window that appears, check the radio button next to `Existing Environment` and then click on the "..." button (■) and then navigate to your venv folder, and find the python executable. Select the python executable, and click the `OK` button. Hit OK on the new interpreter window, and then select the interpreter that you just added from the dropdown.

Note:

- **On Windows you will be looking for either *python.exe* or *pythonw.exe*, both will work.**
  - On Windows this will be located in the *Scripts* folder of your venv.
- **On Linux and macOS the executable file will just be called *python***
  - On Linux and macOS this will be located in the *lib* folder of your venv.

Django Support

Back in the settings window, you should find `Languages and Frameworks` in the sidebar. When you expand this category, you one of the sub-categories will be `Django`. Select Django and tick the `Enable Django Support` box.

Now click the little folder icon in the text box next to `Django Project Root`, and navigate to the folder in the project folder containing `manage.py`, and hit ok.

Run/Debug Configurations

PyCharm comes with a debugger that you can use to run your applications with, but it needs to be configured.

At the top right of the IDE window you will see a button that says `Add Configuration`. Click this button and you should have a window pop up titled `Run/Debug Configurations`. At the top left of the window you should see a plus (+) sign. Click on the plus sign and select the `Django Server` option from the dropdown.

You should not need to change any of the settings, except for the interpreter as it may or may not be automatically set to the project interpreter. If it is not set to the project interpreter just select the project interpreter from the dropdown.

By default it will start the server on port 8000, you should not need to change this unless you already have something running on that port.

1.2.4 Postgres

Installing Postgres

Windows

Installing postgresql on Windows is pretty simple.

Install instructions and a link to the installer download can be found at the postgres website.

Simply download and run the installer for the desired version of postgresql to install it.

Note: The windows version of postgres comes with the option to install pgAdmin alongside postgres which provides a simple web-ui for managing your postgresql installation.

Note: In order to make use of the psql command in command prompy on windows you will want to add *C:\Program Files\PostgreSQL\XX\bin\psql.exe* to your PATH.

Instructions for adding to your PATH on your version of windows can be found here.

Linux

Like other sections of this documentation, we will assume that you are using Ubuntu.

Depending on the version of Ubuntu that you are running, the versions listed, and some commands may be slightly different.

To install postgres on Ubuntu, you will need to install two packages from apt, `postgresql` and `postgresql-contrib`.

```
$ sudo apt-get install postgresql postgresql-contrib
```

Configuring Postgres

The postgres service may not start automatically, to check if it is running run the folloqing command:

```
$ sudo service postgresql status
```

If is is not running you should see `10/main (port 5432): down`, to turn it on run the following command:

```
$ sudo service postgresql start
```

Postgres should now be running.

macOS

Coming Soon

Creating a User and Database

It is generally understood that running applications with root permissions is a bad idea. This extends to database access too. So to avoid using the postgres equivalent of a root user (which is just postgres... so original) we will create a new user, and the database that we are going to use.

Accessing the Postgres CLI

Windows

Assuming that you have added `psql.exe` to your path, the following command should be all you need to access the postgres CLI:

```
$ psql -U postgres -W
```

You will be prompted to enter the password that you set when you ran the installer. If you set no password, you can leave off the `-W`.

Linux

To access postgres from the command line you will have to sudo the postgres user, as postgres by default uses system authentication. There are two ways to do this.

The first option is to use `sudo su` to switch your user to the postgres user. Then after switching to the postgres user, you can run the `psql` command to enter the postgres cli.

```
$ sudo su postgres
$ psql
```

The second option is to just use sudo to run the psql command as the postgres user.

```
$ sudo -u postgres psql
```

You will know you are in the postgres CLI rather than bash when you see the following:

```
psql (10.9 (Ubuntu 10.9-0ubuntu0.18.04.1))
Type "help" for help.

dbname=#
```

Where dbname is the database that you are currently connected to. Which for the case of the above commands will be `postgres`.

macOS

Coming Soon

Creating a User

Once you are connected connected to the database CLI, SQL commands are pretty standard. Though if you are used to MySQL or some other SQL server, there are some slight differences.

The following is the command to create a new user.

```
CREATE USER user_name WITH PASSWORD 'password';
```

Where `user_name` is the user name you wish to give to the user, and `'password'` is the password that you would like to give the user (*Thus must be kept inside single quotes!*).

Creating a Database

The following is the command to create a new database.

```
CREATE DATABASE db_name;
```

Now, you have to give the user the ability to work on the database. To do this use the following command.

```
GRANT ALL PRIVILEGES ON DATABASE db_name TO user_name;
```

Once you have created your database and user, as well as assigned permissions to the user on the database, you can exit the postgres CLI by entering the `\q` command.

1.3 Using Git

Git is a command line tool for managing git repositories. However, just because it is a command line tool does not mean that you have to use it in the command line. There are many GUI Git clients out there that make working with git very user friendly.

You can find a list of Git GUI clients on the GUI Clients page at git-scm.com.

1.3.1 Cloning a Repository

Cloning (downloading) a repository (also called a repo), can be done in the command line, by first navigating to the directory where you want the repository's files to go. Then running the following command.

```
$ git clone REPO_URL
```

Where `REPO_URL` is the url of the repository that you wish to clone. Once cloned, you will have an exact copy of the repository as it was the moment you cloned it.

1.3.2 Pulling Changes

So, now you have a copy of the repository, but some one has made a change, and you don't see it on your computer. To get these changes synced to your computer you run the `git pull` command, which is like the clone command, except it only downloads the changes that have been made since the last time you pulled (or cloned).

The Fetch Command

In using GUI clients or googling around, you will likely come across mentions of the `git fetch` command. On the surface, the fetch command seems to do the same thing that the pull command does. However, they are slightly different.

The fetch command will get information from the remote repository about changes that have happened since the last time you pulled/fetched/cloned but it will not actually download the changes, so will not affect your working files.

1.3.3 Making Changes

This is the fun part. We get to write code, and share it with everyone!

Committing Your Changes

Coming Soon

Pushing Your Changes

Coming Soon

1.4 Writing a Django App

Note: This section focuses on traditional Django apps. Django Rest Framework specific documentation can be found in the next section.

Now that you have setup your development environment, you are ready to start writing code to contribute to the backend. One way to do this is to write an app for the backend!

This part of the documentation will guide you through the very basics of doing this.

1.4.1 Creating the App

First, you will want to make sure that you have cloned the project from git, activated your venv, and installed all the project dependencies.

Assuming you have done this, all you need to do to start creating a new app is to run the following command in your terminal from the `mysite` directory (the the one that contains `manage.py`):

```
$ django-admin startapp AppName
```

Now you should see a folder in the `mysite` directory that has the same name as the app name that you used in the command. Congrats you just created a django application. Now lets talk about how to make it actually do something.

1.4.2 Models

Django comes packaged with an extremely powerful ORM, or Object-Relational-Mapper, which is a tool that handles all your database needs in the most efficient way possible. This means you never have to write a single line of SQL if you don't want to. (*You shouldn't want to unless you really know what you are doing.*)

In order for the ORM to be able to handle your data, you have to tell it how you want your data stored. To do this, we create **Models**. A model is a representation of your data in code.

In our case this is in the form of a `class` that extends `models.Model`. Unlike most python classes we do not create constructors (`__init__` methods) for our django model classes. These classes just consist of member variables, or **fields**.

Model Fields

Fields essentially define the database colums that you want your model to have.

Certain fields are created for you, so you dont have to worry about them. For example, if you do not designate any field as a primary key field, then the ORM will automatically create a standard `id` column in the database.

Note: For more information on what fields exist and how to use them, check out the Model Field Reference on the Django documentation, https://docs.djangoproject.com/en/3.0/ref/models/fields/

The `Meta` Class

The `Meta` class is a class that lives within the model class. This class determines various things about the model class. This class can be used to set unique constraints across fields, set the default ordering of results, define permissions, and more.

Note: For more information on how to use the Meta class, check out the Django Documentation https://docs.djangoproject.com/en/3.0/ref/models/options/

Model Permissions

Model permissions are a django tool that allows you to define what permissions get associated with a model and how they are to be used. This concept may take some playing around with before you become comfortable with it.

This concept is best explained with a hypothetical.

Say we have a blog, and this blog has many users, but not all users should be able to create a blog post. In this case we might define a `create` permission when we create the `BlogPost` model. We can then check to see if the user has that permission before allowing them to attempt to author a blog post.

Example Model

In this example we will create a very simple BlogPost model to give you an idea of what a django model might look like.

```
from django.db import models
from django.contrib.auth.models import User

class BlogPost(models.Model):
    title = models.CharField(max_length=250, null=False)
    created = models.DateTimeField(auto_created=True)
    published = models.BooleanField(default=False)
    author = models.ForeignKey(User, on_delete=models.CASCADE)

    def __str__(self):
        return f"<Blog Post: {self.title} by {self.author.username} ({'T' if self.
↪published else 'F'})>"

    class Meta:
        default_permissions = ('create',)
```

Warning: This example code is for demonstration purposes ONLY.

1.4.3 Migrations

A migration is code that is generated auto-magically by django based on the information in your models. This code is used to ensure that the database is up to date and has all the necessary fields and data that you are expecting it to.

Migrations are arguably one of the **most important** features in django. Any time you make a change to the fields or `Meta` options of a model, you should make sure that you make the migrations to go with the change.

When ever you install a django plugin, pull new code from the repository, or make new migrations, you should always make sure to run those migrations.

To make new migrations:

```
$ python manage.py makemigrations
```

To run migrations:

```
$ python manage.py migrate
```

When uninstalling something, you can clean your database by reversing the migrations. However, be very careful when doing this as you will lose any data related to the app that you are uninstalling.

To reverse migrations (migrate zero):

```
$ python manage.py migrate APP_NAME zero
```

1.4.4 Views

So, now that we know how Django is dealing with our data, how do we make pages that the user can see? How do we use the data from our models?

Well, I am glad you asked! That is what the view is for. In django, the View is the code that processes the request from the user, and returns something for the user to see.

In Django there are two ways to write views, you can write them as classes, or you can write them as functions. Both of these approaches have their pros and cons and neither is necessarily better than the other.

Class Based Views

Class based views can be very powerful tools, especially if you want to respond to multiple request types (i.e GET, POST, HEAD, etc.) at the same url. This approach also allows you to more easily organize other data and/or functions that might go with the view.

Class based views are also very powerful if you have many views that have to the the same basic things. In such cases you can easily create reusable views that can be readily subclassed and extended.

Django provides several generic views that can be subclassed and extended, however this is where the main con of the Class Based view comes in. Especially when using generic views it is easy to fall into the common "copy-paste magic code" trap. Unless you read the code and/or documentation for the superclass (which everyone should be doing anyways) you may at the very least waste a lot of time using the wrong superclass, or at worst accidentally break something or leak data you didn't mean to.

Example of a Class Based View:

```
from django.views.generic import View
from django.shortcuts import render
from .models import Book

class BookView(View):
    template_name = "book.html"

    def get(self, request, book_id):
        book = Book.objects.get(pk=book_id)
        return render(request, template_name=self.template_name, context={"book":␣
↪book})
```

```
    def post(self, request, book_id):
        # do something when we get a post request!
        pass
```

Function Based Views

The main benefit of function based views is that they are simple to implement and are very explicit. Whereas class based views may have functions that are run in the background before the code that you wrote gets called, with a function based view you know exactly will happen at every stage of the process.

Function based views can still do practically everything you might want to do with a class based view with the only exception being (some what obviously) the ability to subclass another view and/or extend another similar view. With function based views we get little in the way of re-usability.

Example of a function based view:

```
from django.shortcuts import render
from .models import Book

def book_view(request, book_id):
    # we can do stuff here too
    if request.method == "POST":
        # do something if we have a post request
    else:
        # otherwise do something else

    book = Book.objects.get(pk=book_id)
    return render(request, "book.html", context={"book": book})
```

1.4.5 Templates

Ok, I might have lied a little before when I said that views are used for making pages that users can see.

Like all websites, we need some HTML code for the browser to render. This is what templates are. Django ships with a powerful Jinja2-like templating engine, which allows you to embed logic in your HTML to make, well... templates.

Template Language Syntax Cheat Sheet

- `{{ ... }}` - Inserts data from a variable.
- `{% ... %}` - Defines a tag, most tags have a corresponding ending tag (ex. `endif` for the `if` tag.)
- `{# ... #}` - Contains comments. Can only be used for single line comments. The `comment` tag should be used for multiline comments.
- `...|...` - The pipe (|) is used to apply a filter to the data to its left.
  - For example: `{{ name|lower }}` would apply the `lower` filter to `name`.

Template Inheritance

One of the features that makes templates so powerful is the concept of inheritance, just like we have with normal classes.

To make use of this feature, you must first build your base template. This should be the most basic layout for your site. You can have as many of these as you want. For example, you might have a home page and a blog page that share the same basic layout, but your user dashboard and its related pages may have a drastically different base layout.

In these base layouts you define different blocks that can be filled by templates that `extend` the base template.

Here is a simple example of a base template:

```
<html>
<head>
    <title> {% block title %} {% endblock %} - SITE NAME</title>
</head>
<body>
    <h1>HEADER (will be on every page)</h1>
    <p>{% block content %} {% endblock %}</p>
    <h1>FOOTER (will be on every page)</h1>
</body>
</html>
```

Now a simple template that extends the base template (assuming its called `base.html`):

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

{% block title %}
    PAGE TITLE
{% endblock %}

{% block content %}
    <strong>Some</strong> content!
{% endblock %}
```

Context

Context is used to pass data to templates that does not already exist in the request. Context consists of a dict, where values are accessed by using their keys just like a variable in the template.

For example, our views used a model called Book, and passed a book object to the template in the context. If we wanted to insert the title of the book into the template somewhere, we would add `{{ book.title }}` where ever in the template we wanted to show the title of the book.

"book" in this example refers to the key used in the example views, not the variable by the same name. In other words if the context dict looked like this: `{"stuff": book}` then we would use `{{ stuff.title }}` in the template.

Note: For a more in-depth look at the Django Template engine, please refer to its documentation here: https://docs.djangoproject.com/en/3.0/ref/templates/

1.4.6 URLs

Now, we need to tell Django to take our URL and figure out what to do with it. This is where the URLs file comes into play. Every application will define its specific URL paths, and which view to use for them.

Each `urls.py` contains a list called `urlpatterns`, this list is used by django as a reference to find which view to run when a request comes in to a url.

Example `urls.py`:

```
from django.urls import path

urlpatterns = [
    path('', ),
    path('', ),
    path('', ),
    path('', ),
    path('', ),
]
```

1.4.7 The `admin.py` File

Brief Description

CHAPTER 2

Deployment Documentation

cough CHAD **cough**