Learn Python with Django

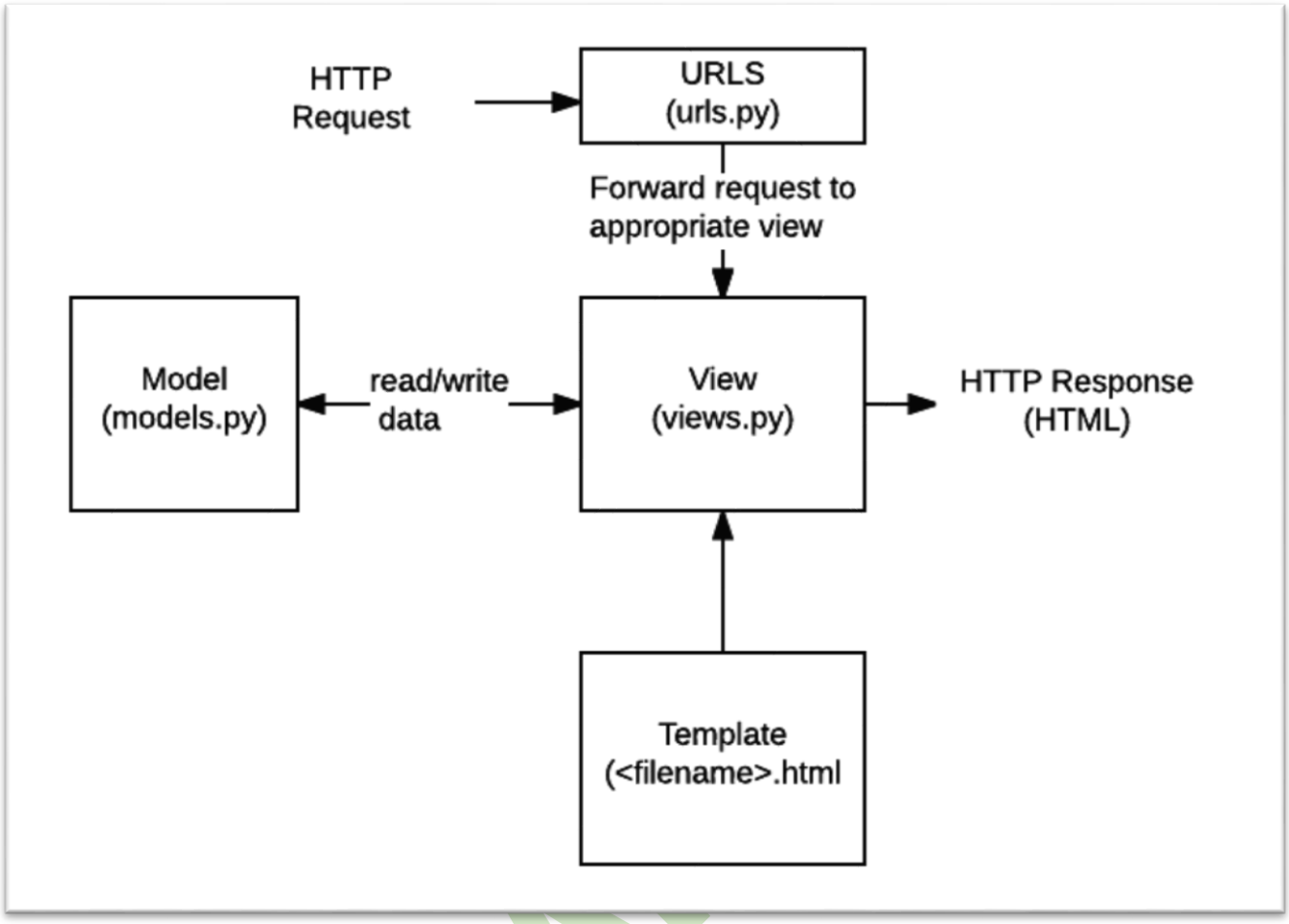
Creating home page

Overview

Now we have defined our models and created some initial library records to work with, it's time to write the code to present that information to users. The first thing we need to do is determine what information we want to be able to display in our pages, and then define appropriate URLs for returning those resources. Then we're going to need to create the URL mapper, views, and templates to display those pages.

The diagram below is provided as a reminder of the main flow of data and things that need to be implemented when handling an HTTP request/response. As we've already created the model, the main things we'll need to create are:

- URL mappers to forward the supported URLs (and any information encoded in the URLs) to the appropriate view functions.
- View functions to get the requested data from the models, create an HTML page displaying the data, and return it to the user to view in the browser.
- Templates used by the views to render the data.



Defining the resource URLs

As this version of *LocalLibrary* is essentially read-only for end users, we just need to provide a landing page for the site (a home page), and pages that *display* list and detail views for books and authors. The URLs that we're going to need for our pages are:

- `catalog/` — The home/index page.
- `catalog/books/` — The list of all books.
- `catalog/authors/` — The list of all authors.
- `catalog/book/`*<id>* — The detail view for the specific book with a field primary key of *<id>* (the default). So for example, `/catalog/book/3`, for the third book added.
- `catalog/author/`*<id>* — The detail view for the specific author with a primary key field named *<id>*. So for example, `/catalog/author/11`, for the 11th author added.

The first three URLs are used to list the index, books, and authors. These don't encode any additional information, and while the results returned will depend on the content in the database, the queries run to get the information will always be the same.

By contrast the final two URLs are used to display detailed information about a specific book or author — these encode the identity of the item to display in the URL (shown as *<id>* above). The URL mapper can extract the encoded information and pass it to the view, which will then dynamically

determine what information to get from the database. By encoding the information in our URL we only need one URL mapping, view, and template to handle every book (or author).

As discussed in the overview, the rest of this article describes how we construct the index page.

Creating the index page

The first page we'll create will be the index page (`catalog/`). This will display a little static HTML, along with some calculated "counts" of different records in the database. To make this work we'll have to create an URL mapping, view and template.

URL mapping

Updated **locallibrary/urls.py** so that whenever an URL starting with `catalog/` is received, the *URLConf* `catalog.urls` is included to process the remainder of the string.

```
from django.contrib import admin
from django.urls import include
from django.urls import path
from django.views.generic import RedirectView
from django.conf import settings
from django.conf.urls.static import static

urlpatterns = [
    path('admin/', admin.site.urls),
    path('catalog/', include('catalog.urls')),
    path('/', RedirectView.as_view(url='/catalog/', permanent=True)),
] + static(settings.STATIC_URL, document_root=settings.STATIC_ROOT)
```

We also update *URLConf*: **/catalog/urls.py**. Open this file and paste in the line shown in bold below.

```
from django.conf.urls import url

from . import views
from django.urls import path

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

]
```

This `path()` function defines a URL pattern (in this case an empty string: `''` — we'll talk a lot more about URL patterns when discussing some of the other views), and a view function that will be called if the pattern is detected (`views.index` — a function named `index()` in **views.py**).

This `path()` function also specifies a `name` parameter, which uniquely identifies *this* particular URL mapping. You can use this name to "reverse" the mapper — to dynamically create a URL pointing to the resource the mapper is designed to handle. For example, with this in place we can now link to our home page from any other page by creating the following link in a template:

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

View (function-based)

A view is a function that processes an HTTP request, fetches data from the database as needed, generates an HTML page by rendering this data using an HTML template, and then returns the HTML in an HTTP response to be shown to the user. The index view follows this model — it fetches information about how many `Book`, `BookInstance`, available `BookInstance` and `Author` records we have in the database, and passes them to a template for display.

Open **catalog/views.py**, and note that the file already imports the render() shortcut function which generates HTML files using a template and data.

```
from django.shortcuts import render

# Create your views here.
```

Copy the following code at the bottom of the file. The first line imports the model classes that we will use to access data in all our views.

```
from django.shortcuts import render

# Create your views here.
from .models import Book, Author, BookInstance, Genre


def index(request):
    """
    View function for home page of site.
    """
    # Generate counts of some of the main objects
    num_books = Book.objects.all().count()
    num_instances = BookInstance.objects.all().count()
    # Available books (status = 'a')
    num_instances_available = BookInstance.objects.filter(status__exact='a').count()
    num_authors = Author.objects.count()  # The 'all()' is implied by default.

    # Render the HTML template index.html with the data in the context variable
    return render(
        request,
        'index.html',
        context={'num_books': num_books, 'num_instances': num_instances,
                 'num_instances_available': num_instances_available, 'num_authors':
num_authors},
    )
```

The first part of the view function fetches counts of records using the `objects.all()` attribute on the model classes. It also gets a list of `BookInstance` objects that have a status field value of 'a' (Available).

At the end of the function we call the `render()` function to create and return an HTML page as a response (this shortcut function wraps a number of other functions, simplifying this very common use-case). This takes as parameters the original `request` object (an `HttpRequest`), an HTML

template with placeholders for the data, and a `context` variable (a Python dictionary containing the data that will be inserted into those placeholders).

We'll talk more about templates and the context variable in the next section; let's create our template so we can actually display something to the user!

Template

A template is a text file defining the structure or layout of a file (such as an HTML page), with placeholders used to represent actual content. Django will automatically look for templates in a directory named '**templates**' in your application. So for example, in the index view we just added, the `render()` function will expect to be able to find the file **/locallibrary/catalog/templates/*index.html***, and will raise an error if the file cannot be found. You can see this if you save the previous changes and go back to your browser — accessing `127.0.0.1:8000` will now give you a fairly intuitive error message "`TemplateDoesNotExist at /catalog/`", plus other details.

Extending templates

The index template is going to need standard HTML markup for the head and body, along with sections for navigation (to the other pages in the site that we haven't yet created) and for displaying some introductory text and our book data. Much of this text (the HTML and navigation structure) will be the same for every page on our site. Rather than forcing developers to duplicate this "boilerplate" in every page, the Django templating language allows you to declare a base template, and then extend it, replacing just the bits that are different for each specific page.

For example, a base template **base_generic.html** might look like the text below. As you can see, this contains some "common" HTML and sections for title, sidebar and content marked up using named `block` and `endblock` template tags (shown in bold). The blocks can be empty, or contain content that will be used "by default" for derived pages.

```
<!DOCTYPE html>
<html lang="en">
<head>
  {% block title %}<title>Local Library</title>{% endblock %}
</head>

<body>
  {% block sidebar %}<!-- insert default navigation text for every page -->{% endblock %}
  {% block content %}<!-- default content text (typically empty) -->{% endblock %}
</body>
</html>
```

When we want to define a template for a particular view, we first specify the base template (with the `extends` template tag — see the next code listing). If there are any sections that we want to

replace in the template we declare these, using identical `block`/`endblock` sections as used in the base template.

For example, the code fragment below shows how we use the `extends` template tag, and override the `content` block. The final HTML produced would have all the HTML and structure defined in the base template (including the default content you've defined inside the `title` block), but with your new `content` block inserted in place of the default one.

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

{% block content %}
<h1>Local Library Home</h1>
<p>Welcome to <em>LocalLibrary</em>, a very basic Django website.</p>
{% endblock %}
```

The LocalLibrary base template

The base template we plan to use for the *LocalLibrary* website is listed below. As you can see, this contains some HTML and defined blocks for `title`, `sidebar`, and `content`. We have a default title (which we may want to change) and a default sidebar with links to lists of all books and authors (which we will probably not want to change, but we've allowed scope to do so if needed by putting this in a block).

Create a new file — **/locallibrary/catalog/templates/*base_generic.html*** — and give it the following contents:

```
<!DOCTYPE html>
<html lang="en">
<head>

  {% block title %}<title>Local Library</title>{% endblock %}
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css">
  <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.12.4/jquery.min.js"></script>
  <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js"></script>

  <!-- Add additional CSS in static file -->
  {% load static %}
  <link rel="stylesheet" href="{% static 'css/styles.css' %}">
</head>

<body>

{% load static %}
<img src="{% static 'images/local_library_model_uml.png' %}" alt="My image"
style="width:1000px;height:350px" />

  <div class="container-fluid">

    <div class="row">
      <div class="col-sm-2">
      {% block sidebar %}
      <ul class="sidebar-nav">
```

```
            <li><a href="{% url 'index' %}">Home</a></li>
            <li><a href="">All books</a></li>
            <li><a href="">All authors</a></li>
        </ul>
       {% endblock %}
        </div>
        <div class="col-sm-10 ">
        {% block content %}{% endblock %}
        </div>
      </div>

  </div>
</body>
</html>
```

The template uses (and includes) JavaScript and CSS from Bootstrap to improve the layout and presentation of the HTML page. Using Bootstrap or another client-side web framework is a quick way to create an attractive page that can scale well on different browser sizes, and it also allows us to deal with the page presentation without having to get into any of the details — we just want to focus on the server-side code here!

The base template also references a local css file (**styles.css**) that provides a little additional styling. Create **/locallibrary/catalog/static/css/styles.css** and give it the following content:

```
.sidebar-nav {
    margin-top: 20px;
    padding: 0;
    list-style: none;
}
```

The index template

Create the HTML file **/locallibrary/catalog/templates/*index.html*** and give it the content shown below. As you can see we extend our base template in the first line, and then replace the default `content` block with a new one for this template.

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

{% block content %}
<h1>Local Library Home</h1>

  <p>Welcome to <em>LocalLibrary</em>.</p>

<h2>Dynamic content</h2>

  <p>The library has the following record counts:</p>
  <ul>
    <li><strong>Books:</strong> {{ num_books }}</li>
    <li><strong>Copies:</strong> {{ num_instances }}</li>
    <li><strong>Copies available:</strong> {{ num_instances_available }}</li>
    <li><strong>Authors:</strong> {{ num_authors }}</li>
  </ul>

{% endblock %}
```

In the *Dynamic content* section we've declared placeholders (*template variables*) for the information we wanted to include from the view. The variables are marked using the "double brace" or "handlebars" syntax (see in bold above).

The important thing to note here is that these variables are named with the *keys* that we passed into the `context` dictionary in our view's `render()` function (see below); these will be replaced by their associated *values* when the template is rendered.

```
return render(
    request,
    'index.html',

context={'num_books':num_books,'num_instances':num_instances,'num_instances_available':
num_instances_available,'num_authors':num_authors},
)
```

Referencing static files in templates

Your project is likely to use static resources, including JavaScript, CSS, and images. Because the location of these files might not be known (or might change), Django allows you to specify the location of these files in your templates relative to the `STATIC_URL` global setting (the default skeleton website sets the value of `STATIC_URL` to '`/static/`', but you might choose to host these on a content delivery network or elsewhere).

Within the template you first call the `load` template tag specifying "static" to add this template library (as shown below). After static is loaded, you can then use the `static` template tag, specifying the relative URL to the file of interest.

```
<!-- Add additional CSS in static file -->
{% load static %}
<link rel="stylesheet" href="{% static 'css/styles.css' %}">
```

You could, if desired, add an image into the page in the same sort of fashion. For example:

```
{% load static %}
<img src="{% static 'images/local_library_model_uml.png' %}" alt="My image"
style="width:1000px;height:350px" />
```

Linking to URLs

The base template above introduced the `url` template tag.

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

This tag takes the name of a `path()` function called in your **urls.py** and values for any arguments the associated view will receive from that function, and returns a URL that you can use to link to the resource.

What does it look like?

At this point we should have created everything needed to display the index page. Run the server (`python3 manage.py runserver`) and open your browser to http://127.0.0.1:8000/catalog/ . If everything is set up correctly, your site should look something like the following screenshot.