Theory: Django MVC

16 minutes 0 / 4 problems solved

Skip this topic Start practicing

2076 users solved this topic. Latest completion was about 6 hours ago.

§1. MVC Paradigm

Complex software products have their own architecture. Though each example is unique, usually they all contain common design patterns. Patterns are repeatable and rather language-independent structures, so getting familiar with just one of them means understanding a whole bunch of applications that share it. It is basically a general language to express ideas, and Model-View-Controller (**MVC**) is one particularly useful pattern to learn, since many popular frameworks like Django (Python), Spring (Java), and Ruby On Rails (Ruby) are using it.

The main idea of MVC is dividing responsibilities between three components. **The Model** part contains business objects, **the View** represents the application and **the Controller** manages data flow between two of them. The advantage of this design pattern is that we can create different views from the same models, so we write less code by reusing the existing parts. Distinguishing one component from another is the main principle to be guided by.

Each component has its associated files in the default Django layout.

Let's say that we are creating an online magazine. The folder "magazine" is the root of your site and the inner folder "blog" is one of the applications. You can link the components with those files:

```
magazine/
├── blog
    ├── ...
    ├── models.py    # Model
    └── views.py     # Controller
├── magazine
    ├── ...
    └── urls.py      # Controller
└── templates        # View
```

Let's take a closer look at the files that you will need in order to make a web service.

§2. Starting a New Project

For this example, we will use "**magazine**" for a project and "**blog**" for an application. If you want to get creative and use other names, feel free to change the given ones in the code.

The Django project is the root of all code you are writing for the service, and it should have at least one application. To isolate units of code with different business logic, you can create more of them. For example, the ***magazine*** project can have the ***blog, authors, forum***, and ***support*** applications.

With time the codebase becomes large, dividing it into applications helps to control the complexity.

The django-admin utility helps organize the layout of your project. You may create all these files manually, but using django-admin will be a good guide to the common structure of a project for you. Note that different versions of Django have different default layouts, and whichever version you use, try to stick to its provided structure: it will make your code easier to maintain.

To create a project and the first application, run the following in the command line:

Current topic:

Django MVC Stage 1

Topic depends on:

✓ The concept of patterns Stage 1

✗ Introduction to Django Stage 1

Topic is required for:

Launching web server Stage 1

Django ORM

Table of contents:

↑ Django MVC

§1. MVC Paradigm

§2. Starting a New Project

§3. Model

§4. View

§5. Controller

§6. Request to the Web Site

Feedback & Comments

```
django-admin startproject magazine
cd magazine
django-admin startapp blog
```

Having executed these commands, you'll get a whole file tree for the project with a piece of code. Now let's get down to using our MVC pattern.

§3. Model

```
magazine/
└── blog
    └── models.py
```

It's nice to have the same content for all users, but if we want to customize it a bit, we need tools from Python interface.

The Model component includes all the database operations with the business objects in your project. A **business object** is an entity with custom attributes; it reflects a structured piece of data from your application which you want to store persistently or temporarily. For example, in a shop application, it can be a customer, a product and a purchase; in a blog, business objects can be authors, posts and comments.

To keep your code clear, you should implement all operations with the business objects in the "*models.py*" module. The bigger the codebase gets, the harder it is to maintain everything in one file, but it's a good starting point.

You may use `User` and `Group` models from `django.contrib.auth.models`: Django provides them right out of the box. The `User` is a registered person in your web service and the `Group` is a collection of `User`s. We'll create some of those when we attach a database to your project.

§4. View

```
magazine/
├── blog
    └── templates
        └── blog
            └── index.html
└── templates
    └── base.html
```

No one will know what the service does unless it has some form of visual rendition. The View is a representation of your web service. Simply put, it is what the user sees.

The View component is stored in templates. **Templates** are files that support Django/Jinja2 template languages. Besides, they can include content with HTML, CSS, and JavaScript. Template language utilizes the ability to use similar constructs you use in Python. It has a different syntax but the same function words.

To create "templates" directory for the project and for the application, run:

```
# Unix
mkdir templates
mkdir -p blog/templates/blog

# Windows
mkdir templates
mkdir blog\templates\blog
```

In the project folder you keep base files for all other templates, applications folder contains only application-specific templates.

When you create "templates" directory for your application, you should name it *"<application name>/templates/<application name>"*. This redundancy is obligatory. If you use the same file name without the second repetition of *<application name>*, like "blog/templates/index.html" and "news/templates/index.html", then in both cases Django template loader will return the first file it found, which isn't always what the user actually needs.

§5. Controller

```
magazine/
├── blog
│   └── views.py
└── magazine
     └── urls.py
```

Views and models are good instruments, but something should manage how they work together, which is where we turn to the Controller part.

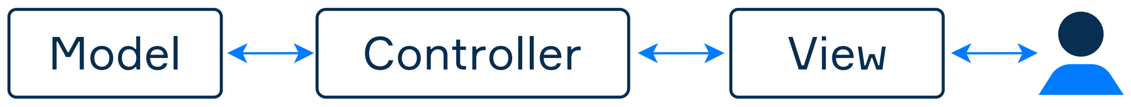
The Controller consists of two types of files: "*views.py*" and "*urls.py*". In "*urls.py*", you define the routing for your service. **Routing** is a process of matching request links with appropriate view handlers. You can include routes from "*urls.py*" files one into another, but the main will be "*<project_name>/<project_name>/urls.py*". If you want to create routing files for each application, do it and include them in the main one.

In "*views.py*" you define view handlers, which play a mediator role between the Model and the View. A **view handler** is a function or a class that responds to requests. Since communication between client and server is an implementation of the HTTP protocol, handler answers with a status code. If the request was successful, the server typically responds with 200 code. In case the requested page is not found, it will be 404; if the server is down, it's 500. Other examples of codes can be found at https://httpstatuses.com/.

§6. Request to the Web Site

Now that you know how functions are divided between the components, let's see how they interact when there's a web site request.

1. A user sees a link or a button in a View, presses it and creates a request.
2. The Controller receives the request.
3. It passes the request to the appropriate handler.
4. The handler calls Model methods to retrieve objects from data storage.
5. It chooses the View template to render a response.
6. A user sees a response.



Each part has its own methods and base classes in the Django package. You should separate the work with each component as Django developers do: this way other developers will understand your code and you will understand theirs.

Report a typo

171 users liked this theory. **22** didn't like it. **What about you?**

Start practicing

Comments (15) Hints (0) Useful links (0) Show discussion