Learning how to write better Django code to build more maintainable websites either takes a lot of experience or familiarity with various design patterns. Filled with several idiomatic Django patterns, Django Design Patterns and Best Practices accelerates your journey into the world of web development.

Discover a set of common design problems, each tackling aspects of Django including model design and Views, and learn several recommended solutions with useful code examples and illustrations. You’ll also get to grips with the current best practices in the latest versions of Django and Python. Creating a successful web application involves much more than Django, so advanced topics including REST, testing, debugging, security, and deployment are also explored in detail.

Who this book is written for

If you want to learn how best to utilize commonly found patterns and learn best practices in developing applications with Django, this is the book for you. This book, like Django itself, is accessible to amateur and professional developers alike and assumes little in the way of prior experience.

Although written for Python 3, the majority of the code in this book works in Python 2 or can be easily translated.

What you will learn from this book

- Understand common design patterns that can help you write better Django code
- Discover the latest best practices and idioms in this rapidly evolving framework
- Deep dive into various aspects of Django from Models and Views to testing and deployment
- Get to grips with difficult topics such as dealing with legacy code and debugging
- Familiarize yourself with the various nuances of web development such as browser attacks and databases
- Reduce the maintenance burden with well-tested, cleaner code
- Show your work to the world by learning about hosting, deploying, and securing your Django projects
- Reduce the maintenance burden with well-tested, cleaner code

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 4 'Views and URLs'
- A synopsis of the book’s content
- More information on *Django Design Patterns and Best Practices*

About the Author

*Arun Ravindran* is an avid speaker and blogger who has been tinkering with Django since 2007 for projects ranging from intranet applications to social networks. He is a long-time open source enthusiast and Python developer. His articles and screencasts have been invaluable to the rapidly growing Django community. He is currently a developer member of the Django Software Foundation. Arun is also a movie buff and loves graphic novels and comics.
Django Design Patterns and Best Practices

Django is one of the most popular web frameworks in use today. It powers large websites, such as Pinterest, Instagram, Disqus, and NASA. With a few lines of code, you can rapidly build a functional and secure website that can scale to millions of users.

This book attempts to share solutions to several common design problems faced by Django developers. Sometimes, there are several solutions but we often wonder whether there is a recommended approach. Experienced developers frequently use certain idioms while deliberately avoiding certain others.

This book is a collection of such patterns and insights. It is organized into chapters each covering a key area of the framework, such as Models, or an aspect of web development, such as Debugging. The focus is on building clean, modular, and more maintainable code.

Every attempt has been made to present up-to-date information and use the latest versions. Django 1.7 comes loaded with exciting new features, such as built-in schema migrations and app reloading. Python 3.4 is the bleeding edge of the language with several new modules, such as asyncio. Both, both of which have been used here.

Superheroes are a constant theme throughout the book. Most of the code examples are about building SuperBook—a social network of superheroes. As a novel way to present the challenges of a web development project, an exciting fictional narrative has been woven into each chapter in the form of story boxes.

What This Book Covers

Chapter 1, Django and Patterns, helps us understand Django better by telling us why it was created and how it has evolved over time. Then, it introduces design patterns, its importance, and several popular pattern collections.

Chapter 2, Application Design, guides us through the early stages of an application life cycle, such as gathering requirements and creating mockups. We will also see how to break your project into modular apps through our running project—SuperBook.

Chapter 3, Models, gives us insights into how models can be graphically represented, structured using several kinds of patterns, and later altered using migrations (built into Django 1.7).

Chapter 4, Views and URLs, shows us how function-based views evolved into class-based views with the powerful mixin concept, familiarizes us with useful view patterns, and teaches us how short and meaningful URLs are designed.
Chapter 5, Templates, walks us through the Django template language constructs explaining their design choices, suggests how to organize template files, introduces handy template patterns, and points to several ways in which Bootstrap can be integrated and customized.

Chapter 6, Admin Interface, shows us how to use Django's brilliant out-of-the-box admin interface more effectively, and several ways to customize it, from enhancing the models to improving its default look and feel.

Chapter 7, Forms, illustrates the often confusing forms workflow, different ways of rendering forms, how to improve a form's appearance using crispy forms and various applied form patterns.

Chapter 8, Dealing with Legacy Code, tackles the common issues with legacy Django projects, such as identifying the right version, locating the files, where to start reading a large codebase, and how to enhance legacy code by adding new functionality.

Chapter 9, Testing and Debugging, gives us an overview of various testing and debugging tools and techniques, introduces test-driven development, mocking, logging, and debuggers.

Chapter 10, Security, familiarizes you with various web security threats and their countermeasures, and especially with how Django can protect you. Finally, a handy security checklist reminds you of commonly overlooked areas.

Chapter 11, Production-ready, introduces a crash course in deploying a public-facing application beginning with choosing your web stack, understanding hosting options, and walking through a typical deployment process. We go into the details of monitoring and performance at this stage.

Appendix, Python 2 versus Python 3, introduces Python 3 to Python 2 developers. Starting off by showing the most relevant differences, while working in Django, we then move on to the new modules and tools offered in Python 3.
In this chapter, we will discuss the following topics:

- Class-based and function-based views
- Mixins
- Decorators
- Common view patterns
- Designing URLs

A view from the top

In Django, a view is defined as a callable that accepts a request and returns a response. It is usually a function or a class with a special class method such as as_view().

In both cases, we create a normal Python function that takes an HttpRequest as the first argument and returns an HttpResponse. A URLConf can also pass additional arguments to this function. These arguments can be captured from parts of the URL or set to default values.

Here is what a simple view looks like:

```python
# In views.py
from django.http import HttpResponse

def hello_fn(request, name="World"):  
    return HttpResponse("Hello {}!".format(name))
```
Our two-line view function is quite simple to understand. We are currently not doing anything with the request argument. We can examine a request to better understand the context in which the view was called, for example by looking at the GET/POST parameters, URI path, or HTTP headers such as REMOTE_ADDR.

Its corresponding lines in URLConf would be as follows:

```
# In urls.py
url(r'^hello-fn/\w+/$', views.hello_fn),
url(r'^hello-fn/$', views.hello_fn),
```

We are reusing the same view function to support two URL patterns. The first pattern takes a name argument. The second pattern doesn't take any argument from the URL, and the view function will use the default name of World in this case.

**Views got classier**

Class-based views were introduced in Django 1.4. Here is how the previous view looks when rewritten to be a functionally equivalent class-based view:

```python
from django.views.generic import View

class HelloView(View):
    def get(self, request, name="World"):
        return HttpResponse("Hello {}!").format(name)
```

Again, the corresponding URLConf would have two lines, as shown in the following commands:

```
# In urls.py
url(r'^hello-cl/\w+/$', views.HelloView.as_view()),
url(r'^hello-cl/$', views.HelloView.as_view()),
```

There are several interesting differences between this view class and our earlier view function. The most obvious one being that we need to define a class. Next, we explicitly define that we will handle only the GET requests. The previous view function gives the same response for GET, POST, or any other HTTP verb, as shown in the following commands using the test client in Django shell:

```plaintext
>>> from django.test import Client
>>> c = Client()

>>> c.get("http://0.0.0.0:8000/hello-fn/").content
```
b'Hello World!'

>>> c.post("http://0.0.0.0:8000/hello-fn/").content
b'Hello World!'

>>> c.get("http://0.0.0.0:8000/hello-cl/").content
b'Hello World!'

>>> c.post("http://0.0.0.0:8000/hello-cl/").content
b''

Being explicit is good from a security and maintainability point of view.

The advantage of using a class will be clear when you need to customize your view. Say you need to change the greeting and the default name. Then, you can write a general view class for any kind of greeting and derive your specific greeting classes as follows:

```python
class GreetView(View):
    greeting = "Hello {}!"
    default_name = "World"
    def get(self, request, **kwargs):
        name = kwargs.pop("name", self.default_name)
        return HttpResponse(self.greeting.format(name))

class SuperVillainView(GreetView):
    greeting = "We are the future, {}. Not them."
    default_name = "my friend"
```

Now, the URLConf would refer to the derived class:

```python
# In urls.py
    url(r'^hello-su/(?P<name>[^/]+)$', views.SuperVillainView.as_view()),
    url(r'^hello-su/$', views.SuperVillainView.as_view()),
```

While it is not impossible to customize the view function in a similar manner, you would need to add several keyword arguments with default values. This can quickly get unmanageable. This is exactly why generic views migrated from view functions to class-based views.
Django Unchained

After spending 2 weeks hunting for good Django developers, Steve started to think out of the box. Noticing the tremendous success of their recent hackathon, he and Hart organized a Django Unchained contest at S.H.I.M. The rules were simple—build one web application a day. It could be a simple one but you cannot skip a day or break the chain. Whoever creates the longest chain, wins.

The winner—Brad Zanni was a real surprise. Being a traditional designer with hardly any programming background, he had once attended week-long Django training just for kicks. He managed to create an unbroken chain of 21 Django sites, mostly from scratch.

The very next day, Steve scheduled a 10 o’clock meeting with him at his office. Though Brad didn’t know it, it was going to be his recruitment interview. At the scheduled time, there was a soft knock and a lean bearded guy in his late twenties stepped in.

As they talked, Brad made no pretense of the fact that he was not a programmer. In fact, there was no pretense to him at all. Peering through his thick-rimmed glasses with calm blue eyes, he explained that his secret was quite simple—get inspired and then focus.

He used to start each day with a simple wireframe. He would then create an empty Django project with a Twitter bootstrap template. He found Django’s generic class-based views a great way to create views with hardly any code. Sometimes, he would use a mixin or two from Django-braces. He also loved the admin interface for adding data on the go.

His favorite project was Labyrinth—a Honeypot disguised as a baseball forum. He even managed to trap a few surveillance bots hunting for vulnerable sites. When Steve explained about the SuperBook project, he was more than happy to accept the offer. The idea of creating an interstellar social network truly fascinated him.

With a little more digging around, Steve was able to find half a dozen more interesting profiles like Brad within S.H.I.M. He learnt that rather that looking outside he should have searched within the organization in the first place.

Class-based generic views

Class-based generic views are commonly used views implemented in an object-oriented manner (Template method pattern) for better reuse. I hate the term generic views. I would rather call them stock views. Like stock photographs, you can use them for many common needs with a bit of tweaking.
Generic views were created because Django developers felt that they were recreating the same kind of views in every project. Nearly every project needed a page showing a list of objects (ListView), details of an object (DetailView), or a form to create an object (CreateView). In the spirit of DRY, these reusable views were bundled with Django.

A convenient table of generic views in Django 1.7 is given here:

<table>
<thead>
<tr>
<th>Type</th>
<th>Class Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>View</td>
<td>This is the parent of all views. It performs dispatch and sanity checks.</td>
</tr>
<tr>
<td>Base</td>
<td>TemplateView</td>
<td>This renders a template. It exposes the URLConf keywords into context.</td>
</tr>
<tr>
<td>Base</td>
<td>RedirectView</td>
<td>This redirects on any GET request.</td>
</tr>
<tr>
<td>List</td>
<td>ListView</td>
<td>This renders any iterable of items, such as a queryset.</td>
</tr>
<tr>
<td>Detail</td>
<td>DetailView</td>
<td>This renders an item based on pk or slug from URLConf.</td>
</tr>
<tr>
<td>Edit</td>
<td>FormView</td>
<td>This renders and processes a form.</td>
</tr>
<tr>
<td>Edit</td>
<td>CreateView</td>
<td>This renders and processes a form for creating new objects.</td>
</tr>
<tr>
<td>Edit</td>
<td>UpdateView</td>
<td>This renders and processes a form for updating an object.</td>
</tr>
<tr>
<td>Edit</td>
<td>DeleteView</td>
<td>This renders and processes a form for deleting an object.</td>
</tr>
<tr>
<td>Date</td>
<td>ArchiveIndexView</td>
<td>This renders a list of objects with a date field, the latest being the first.</td>
</tr>
<tr>
<td>Date</td>
<td>YearArchiveView</td>
<td>This renders a list of objects on year given by URLConf.</td>
</tr>
<tr>
<td>Date</td>
<td>MonthArchiveView</td>
<td>This renders a list of objects on a year and month.</td>
</tr>
<tr>
<td>Date</td>
<td>WeekArchiveView</td>
<td>This renders a list of objects on a year and week number.</td>
</tr>
<tr>
<td>Date</td>
<td>DayArchiveView</td>
<td>This renders a list of objects on a year, month, and day.</td>
</tr>
<tr>
<td>Date</td>
<td>TodayArchiveView</td>
<td>This renders a list of objects on today's date.</td>
</tr>
<tr>
<td>Date</td>
<td>DateDetailView</td>
<td>This renders an object on a year, month, and day identified by its pk or slug</td>
</tr>
</tbody>
</table>

We have not mentioned base classes such as BaseDetailView or mixins such as SingleObjectMixin here. They are designed to be parent classes. In most cases, you would not use them directly.
Most people confuse class-based views and class-based generic views. Their names are similar but they are not the same things. This has led to some interesting misconceptions as follows:

- **The only generic views are the ones bundled with Django**: Thankfully, this is wrong. There is no special magic in the generic class-based views that are provided.

  You are free to roll your own set of generic class-based views. You can also use a third-party library such as django-vanilla-views (http://django-vanilla-views.org/), which has a simpler implementation of the standard generic views. Remember that using custom generic views might make your code unfamiliar to others.

- **Class-based views must always derive from a generic view**: Again, there is nothing magical about the generic view classes. Though 90 percent of the time, you will find a generic class such as View to be ideal for use as a base class, you are free to implement similar features yourself.

### View mixins

Mixins are the essence of DRY code in class-based views. Like model mixins, a view mixin takes advantage of Python's multiple inheritance to easily reuse chunks of functionality. They are often parent-less classes in Python 3 (or derived from object in Python 2 since they are new-style classes).

Mixins intercept the processing of views at well-defined places. For example, most generic views use `get_context_data` to set the context dictionary. It is a good place to insert an additional context, such as a `feed` variable that points to all posts a user can view, as shown in the following command:

```python
class FeedMixin(object):
    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context["feed"] = models.Post.objects.viewable_posts(self.request.user)
        return context
```

The `get_context_data` method first populates the context by calling its namesake in all the bases classes. Next, it updates the context dictionary with the `feed` variable.
Now, this mixin can be easily used to add the user's feed by including it in the list of base classes. Say, if SuperBook needs a typical social network home page with a form to create a new post followed by your feed, then you can use this mixin as follows:

```python
class MyFeed(FeedMixin, generic.CreateView):
    model = models.Post
    template_name = "myfeed.html"
    success_url = reverse_lazy("my_feed")
```

A well-written mixin imposes very little requirements. It should be flexible to be useful in most situations. In the previous example, `FeedMixin` will overwrite the `feed` context variable in a derived class. If a parent class uses `feed` as a context variable, then it can be affected by the inclusion of this mixin. Hence, it would be more useful to make the context variable customizable (which has been left to you as an exercise).

The ability of mixins to combine with other classes is both their biggest advantage and disadvantage. Using the wrong combination can lead to bizarre results. So, before using a mixin, you need to check the source code of the mixin and other classes to ensure that there are no method or context-variable clashes.

### Order of mixins

You might have come across code with several mixins as follows:

```python
class ComplexView(MyMixin, YourMixin, AccessMixin, DetailView):
```

It can get quite tricky to figure out the order to list the base classes. Like most things in Django, the normal rules of Python apply. Python's **Method Resolution Order** (MRO) determines how they should be arranged.

In a nutshell, mixins come first and base classes come last. The more specialized the parent class is, the more it moves to the left. In practice, this is the only rule you will need to remember.

To understand why this works, consider the following simple example:

```python
class A:
    def do(self):
        print("A")

class B:
```

```python
class ComplexView(MyMixin, YourMixin, AccessMixin, DetailView):
```

```python
class SimpleView(Mixin, BaseView):
```

Now, let's consider the class definitions with a different order of base classes:

```python
class ComplexView(DetailView, AccessMixin, YourMixin, MyMixin):
```

In this case, the order of base classes is different, but the method resolution order (MRO) will still determine the order of method lookup. The `__init__` method of `SimpleView` will still call the `do` method of `A`, even though `SimpleView` comes before `ComplexView` in the class definition.
Views and URLs

def do(self):
    print("B")

class BA(B, A):
    pass

class AB(A, B):
    pass

BA().do()  # Prints B
AB().do()  # Prints A

As you would expect, if B is mentioned before A in the list of base classes, then B's method gets called and vice versa.

Now imagine A is a base class such as CreateView and B is a mixin such as FeedMixin. The mixin is an enhancement over the basic functionality of the base class. Hence, the mixin code should act first and in turn, call the base method if needed. So, the correct order is BA (mixins first, base last).

The order in which base classes are called can be determined by checking the __mro__ attribute of the class:

```python
>>> AB.__mro__
(__main__.AB, __main__.A, __main__.B, object)
```

So, if AB calls super(), first A gets called; then, A's super() will call B, and so on.

```
| Python's MRO usually follows a depth-first, left-to-right order to select a method in the class hierarchy. More details can be found at http://www.python.org/download/releases/2.3/mro/.
```

Decorators

Before class-based views, decorators were the only way to change the behavior of function-based views. Being wrappers around a function, they cannot change the inner working of the view, and thus effectively treat them as black boxes.

A decorator is function that takes a function and returns the decorated function. Confused? There is some syntactic sugar to help you. Use the annotation notation @, as shown in the following login_required decorator example:

```python
@login_required
```
def simple_view(request):
    return HttpResponse()

The following code is exactly same as above:

def simple_view(request):
    return HttpResponse()

    simple_view = login_required(simple_view)

Since login_required wraps around the view, a wrapper function gets the control first. If the user was not logged in, then it redirects to settings.LOGIN_URL. Otherwise, it executes simple_view as if it did not exist.

Decorators are less flexible than mixins. However, they are simpler. You can use both decorators and mixins in Django. In fact, many mixins are implemented with decorators.

View patterns
Let's take a look at some common design patterns seen in designing views.

Pattern – access controlled views
Problem: Pages need to be conditionally accessible based on whether the user was logged in, is a member of staff, or any other condition
Solution: Use mixins or decorators to control access to the view

Problem details
Most websites have pages that can be accessed only if you are logged in. Certain other pages are accessible to anonymous or public visitors. If an anonymous visitor tries to access a page, which needs a logged-in user, they could be routed to the login page. Ideally, after logging in, they should be routed back to the page they wished to see in the first place.

Similarly, there are pages that can only be seen by certain groups of users. For example, Django's admin interface is only accessible to the staff. If a non-staff user tries to access the admin pages, they would be routed to the login page.

Finally, there are pages that grant access only if certain conditions are met. For example, the ability to edit a post should be only accessible to the creator of the post. Anyone else accessing this page should see a Permission Denied error.
Solution details

There are two ways to control access to a view:

1. By using a decorator on a function-based view or class-based view:
   ```python
   @login_required(MyView.as_view())
   ```

2. By overriding the `dispatch` method of a class-based view through a mixin:
   ```python
   from django.utils.decorators import method_decorator
   class LoginRequiredMixin:
       @method_decorator(login_required)
       def dispatch(self, request, *args, **kwargs):
           return super().dispatch(request, *args, **kwargs)
   ```

   We really don't need the decorator here. The more explicit form recommended is as follows:
   ```python
   class LoginRequiredMixin:
       def dispatch(self, request, *args, **kwargs):
           if not request.user.is_authenticated():
               raise PermissionDenied
           return super().dispatch(request, *args, **kwargs)
   ```

   When the `PermissionDenied` exception is raised, Django shows the 403.html template in your root directory or, in its absence, a standard "403 Forbidden" page.

   Of course, you would need a more robust and customizable set of mixins for real projects. The `django-braces` package (https://github.com/brack3t/django-braces) has an excellent set of mixins, especially for controlling access to views.

   Here are examples of using them to control access to the logged-in and anonymous views:
   ```python
   from braces.views import LoginRequiredMixin, AnonymousRequiredMixin

   class UserProfileView(LoginRequiredMixin, DetailView):
       # This view will be seen only if you are logged-in
       pass

   class LoginFormView(AnonymousRequiredMixin, FormView):
       # This view will NOT be seen if you are logged in
       authenticated_redirect_url = "/feed"
   ```
Staff members in Django are users with the is_staff flag set in the user model. Again, you can use a django-braces mixin called UserPassesTestMixin, as follows:

```python
from braces.views import UserPassesTestMixin

class SomeStaffView(UserPassesTestMixin, TemplateView):
    def test_func(self, user):
        return user.is_staff
```

You can also create mixins to perform specific checks, such as if the object is being edited by its author or not (by comparing it with the logged-in user):

```python
class CheckOwnerMixin:
    # To be used with classes derived from SingleObjectMixin
    def get_object(self, queryset=None):
        obj = super().get_object(queryset)
        if not obj.owner == self.request.user:
            raise PermissionDenied
        return obj
```

**Pattern – context enhancers**

**Problem:** Several views based on generic views need the same context variable

**Solution:** Create a mixin that sets the shared context variable

**Problem details**

Django templates can only show variables that are present in its context dictionary. However, sites need the same information in several pages. For instance, a sidebar showing the recent posts in your feed might be needed in several views.

However, if we use a generic class-based view, we would typically have a limited set of context variables related to a specific model. Setting the same context variable in each view is not DRY.

**Solution details**

Most generic class-based views are derived from ContextMixin. It provides the get_context_data method, which most classes override, to add their own context variables. While overriding this method, as a best practice, you will need to call get_context_data of the superclass first and then add or override your context variables.
We can abstract this in the form of a mixin, as we have seen before:

class FeedMixin(object):
    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context["feed"] = models.Post.objects.viewable_posts(self.request.user)
        return context

We can add this mixin to our views and use the added context variables in our templates. Notice that we are using the model manager defined in Chapter 3, Models, to filter the posts.

A more general solution is to use StaticContextMixin from django-braces for static-context variables. For example, we can add an additional context variable latest_profile that contains the latest user to join the site:

class CtxView(StaticContextMixin, generic.TemplateView):
    template_name = "ctx.html"
    static_context = {"latest_profile": Profile.objects.latest('pk')}  

Here, static context means anything that is unchanged from a request to request. In that sense, you can mention QuerySets as well. However, our feed context variable needs self.request.user to retrieve the user’s viewable posts. Hence, it cannot be included as a static context here.

**Pattern – services**

**Problem:** Information from your website is often scraped and processed by other applications.

**Solution:** Create lightweight services that return data in machine-friendly formats, such as JSON or XML.

**Problem details**

We often forget that websites are not just used by humans. A significant percentage of web traffic comes from other programs like crawlers, bots, or scrapers. Sometimes, you will need to write such programs yourself to extract information from another website.

Generally, pages designed for human consumption are cumbersome for mechanical extraction. HTML pages have information surrounded by markup, requiring extensive cleanup. Sometimes, information will be scattered, needing extensive data collation and transformation.
A machine interface would be ideal in such situations. You can not only reduce the hassle of extracting information but also enable the creation of mashups. The longevity of an application would be greatly increased if its functionality is exposed in a machine-friendly manner.

**Solution details**

Service-oriented architecture (SOA) has popularized the concept of a service. A service is a distinct piece of functionality exposed to other applications as a service. For example, Twitter provides a service that returns the most recent public statuses.

A service has to follow certain basic principles:

- **Statelessness**: This avoids the internal state by externalizing state information
- **Loosely coupled**: This has fewer dependencies and a minimum of assumptions
- **Composable**: This should be easy to reuse and combine with other services

In Django, you can create a basic service without any third-party packages. Instead of returning HTML, you can return the serialized data in the JSON format. This form of a service is usually called a web Application Programming Interface (API).

For example, we can create a simple service that returns five recent public posts from SuperBook as follows:

```python
from django.views.generic import View

class PublicPostJSONView(View):
    def get(self, request, *args, **kwargs):
        msgs = models.Post.objects.public_posts().values(  
            "posted_by_id", "message"")[:5]
        return HttpResponse(list(msgs), content_type="application/json")
```

For a more reusable implementation, you can use the `JSONResponseMixin` class from django-braces to return JSON using its `render_json_response` method:

```python
from django.views.generic import View
from braces.views import JSONResponseMixin

class PublicPostJSONView(JSONResponseMixin, View):
    def get(self, request, *args, **kwargs):
        msgs = models.Post.objects.public_posts().values(  
```

```python
            "posted_by_id", "message"")[:5]
        return self.render_json_response(list(msgs), content_type="application/json")
```
Views and URLs

    "posted_by_id", "message")[:5]
    return self.render_json_response(list(msgs))

If we try to retrieve this view, we will get a JSON string rather than an HTML response:

    >>> from django.test import Client
    >>> Client().get("http://0.0.0.0:8000/public/").content
    b'[
    {"posted_by_id": 23, "message": "Hello!"},
    {"posted_by_id": 13, "message": "Feeling happy"},
    ...

Note that we cannot pass the `QuerySet` method directly to render the JSON response. It has to be a list, dictionary, or any other basic Python built-in data type recognized by the JSON serializer.

Of course, you will need to use a package such as Django REST framework if you need to build anything more complex than this simple API. Django REST framework takes care of serializing (and deserializing) `QuerySets`, authentication, generating a web-browsable API, and many other features essential to create a robust and full-fledged API.

Designing URLs

Django has one of the most flexible URL schemes among web frameworks. Basically, there is no implied URL scheme. You can explicitly define any URL scheme you like using appropriate regular expressions.

However, as superheroes love to say — "With great power comes great responsibility." You cannot get away with a sloppy URL design any more.

URLs used to be ugly because they were considered to be ignored by users. Back in the 90s when portals used to be popular, the common assumption was that your users will come through the front door, that is, the home page. They will navigate to the other pages of the site by clicking on links.

Search engines have changed all that. According to a 2013 research report, nearly half (47 percent) of all visits originate from a search engine. This means that any page in your website, depending on the search relevance and popularity can be the first page your user sees. Any URL can be the front door.

More importantly, Browsing 101 taught us security. Don’t click on a blue link in the wild, we warn beginners. Read the URL first. Is it really your bank’s URL or a site trying to phish your login details?
Today, URLs have become part of the user interface. They are seen, copied, shared, and even edited. Make them look good and understandable from a glance. No more eye sores such as:


Short and meaningful URLs are not only appreciated by users but also by search engines. URLs that are long and have less relevance to the content adversely affect your site's search engine rankings.

Finally, as implied by the maxim "Cool URIs don't change," you should try to maintain your URL structure over time. Even if your website is completely redesigned, your old links should still work. Django makes it easy to ensure that this is so.

Before we delve into the details of designing URLs, we need to understand the structure of a URL.

**URL anatomy**

Technically, URLs belong to a more general family of identifiers called Uniform Resource Identifiers (URIs). Hence, a URL has the same structure as a URI.

A URI is composed of several parts:

\[
\text{URI} = \text{Scheme} + \text{Net Location} + \text{Path} + \text{Query} + \text{Fragment}
\]

For example, a URI (http://dev.example.com:80/gallery/videos?id=217#comments) can be deconstructed in Python using the `urlparse` function:

```python
>>> from urllib.parse import urlparse
>>> urlparse("http://dev.example.com:80/gallery/videos?id=217#comments")
ParseResult(scheme='http', netloc='dev.example.com:80', path='/gallery/videos', params='', query='id=217', fragment='comments')
```

The URI parts can be depicted graphically as follows:
Even though Django documentation prefers to use the term URLs, it might more technically correct to say that you are working with URIs most of the time. We will use the terms interchangeably in this book.

Django URL patterns are mostly concerned about the 'Path' part of the URI. All other parts are tucked away.

**What happens in urls.py?**

It is often helpful to consider `urls.py` as the entry point of your project. It is usually the first file I open when I study a Django project. Essentially, `urls.py` contains the root URL configuration or URLConf of the entire project.

It would be a Python list returned from `patterns` assigned to a global variable called `urlpatterns`. Each incoming URL is matched with each pattern from top to bottom in a sequence. In the first match, the search stops, and the request is sent to the corresponding view.

Here, in considerably simplified form, is an excerpt of `urls.py` from Python.org, which was recently rewritten in Django:

```python
urlpatterns = patterns(
    '',
    # Homepage
    url(r'^$', views.IndexView.as_view(), name='home'),
    # About
    url(r'^about/$','
        TemplateView.as_view(template_name="python/about.html"),
        name='about'),
    # Blog URLs
    url(r'^blogs/',' include(\'blogs.urls\', namespace='blog')),
    # Job archive
    url(r'^jobs/(?P<pk>[0-9]+)/$','
        views.JobArchive.as_view(),
        name='job_archive'),
    # Admin
    url(r'^admin/',' include(admin.site.urls)),
)```

Some interesting things to note here are as follows:

- The first argument of the `patterns` function is the prefix. It is usually blank for the root `URLConf`. The remaining arguments are all URL patterns.
- Each URL pattern is created using the `url` function, which takes five arguments. Most patterns have three arguments: the regular expression pattern, view callable, and name of the view.
- The `about` pattern defines the view by directly instantiating `TemplateView`. Some hate this style since it mentions the implementation, thereby violating separation of concerns.
- Blog URLs are mentioned elsewhere, specifically in `urls.py` inside the `blogs` app. In general, separating an app's URL pattern into its own file is good practice.
- The `jobs` pattern is the only example here of a named regular expression.

In future versions of Django, `urlpatterns` should be a plain list of URL pattern objects rather than arguments to the `patterns` function. This is great for sites with lots of patterns, since `urlpatterns` being a function can accept only a maximum of 255 arguments.

If you are new to Python regular expressions, you might find the pattern syntax to be slightly cryptic. Let's try to demystify it.

**The URL pattern syntax**

URL regular expression patterns can sometimes look like a confusing mass of punctuation marks. However, like most things in Django, it is just regular Python.

It can be easily understood by knowing that URL patterns serve two functions: to match URLs appearing in a certain form, and to extract the interesting bits from a URL.

The first part is easy. If you need to match a path such as `/jobs/1234`, then just use the `"^jobs/\d+"` pattern (here \d stands for a single digit from 0 to 9). Ignore the leading slash, as it gets eaten up.

The second part is interesting because, in our example, there are two ways of extracting the job ID (that is, 1234), which is required by the view.
Views and URLs

The simplest way is to put a parenthesis around every group of values to be captured. Each of the values will be passed as a positional argument to the view. For example, the "^jobs/\(\d+)" pattern will send the value "1234" as the second argument (the first being the request) to the view.

The problem with positional arguments is that it is very easy to mix up the order. Hence, we have name-based arguments, where each captured value can be named. Our example will now look like "^jobs/(?P<pk>\d+)/". This means that the view will be called with a keyword argument pk being equal to "1234".

If you have a class-based view, you can access your positional arguments in self.args and name-based arguments in self.kwargs. Many generic views expect their arguments solely as name-based arguments, for example, self.kwargs["slug"].

Mnemonic — parents question pink action-figures

I admit that the syntax for name-based arguments is quite difficult to remember. Often, I use a simple mnemonic as a memory aid. The phrase "Parents Question Pink Action-figures" stands for the first letters of Parenthesis, Question mark, (the letter) P, and Angle brackets.

Put them together and you get (?P< . You can enter the name of the pattern and figure out the rest yourself.

It is a handy trick and really easy to remember. Just imagine a furious parent holding a pink-colored hulk action figure.

Another tip is to use an online regular expression generator such as https://pythex.org/ or https://www.debuggex.com/ to craft and test your regular expressions.

Names and namespaces

Always name your patterns. It helps in decoupling your code from the exact URL paths. For instance, in the previous URLConf, if you want to redirect to the about page, it might be tempting to use redirect("/about"). Instead, use redirect("about"), as it uses the name rather than the path.

Here are some more examples of reverse lookups:

```python
>>> from django.core.urlresolvers import reverse
>>> print(reverse("home"))
"/

>>> print(reverse("job_archive", kwargs={"pk":"1234"}))
"jobs/1234/"
```
Names must be unique. If two patterns have the same name, they will not work. So, some Django packages used to add prefixes to the pattern name. For example, an application named blog might have to call its edit view as 'blog-edit' since 'edit' is a common name and might cause conflict with another application.

Namespaces were created to solve such problems. Pattern names used in a namespace have to be only unique within that namespace and not the entire project. It is recommended that you give every app its own namespace. For example, we can create a 'blog' namespace with only the blog's URLs by including this line in the root URLconf:

```python
url(r'^blog/', include('blog.urls', namespace='blog')),
```

Now the blog app can use pattern names, such as 'edit' or anything else as long as they are unique within that app. While referring to a name within a namespace, you will need to mention the namespace, followed by a ':' before the name. It would be "blog:edit" in our example.

As Zen of Python says—"Namespaces are one honking great idea—let's do more of those." You can create nested namespaces if it makes your pattern names cleaner, such as "blog:comment:edit". I highly recommend that you use namespaces in your projects.

**Pattern order**

Order your patterns to take advantage of how Django processes them, that is, top-down. A good rule of thumb is to keep all the special cases at the top. Broader patterns can be mentioned further down. The broadest—a catch-all—if present, can go at the very end.

For example, the path to your blog posts might be any valid set of characters, but you might want to handle the About page separately. The right sequence of patterns should be as follows:

```python
urlpatterns = patterns(
    '',
    url(r'^about/$', AboutView.as_view(), name='about'),
    url(r'^(?P<slug>\w+)/$', ArticleView.as_view(), name='article'),
)
```

If we reverse the order, then the special case, the AboutView, will never get called.
URL pattern styles
Designing URLs of a site consistently can be easily overlooked. Well-designed URLs can not only logically organize your site but also make it easy for users to guess paths. Poorly designed ones can even be a security risk: say, using a database ID (which occurs in a monotonic increasing sequence of integers) in a URL pattern can increase the risk of information theft or site ripping.

Let’s examine some common styles followed in designing URLs.

Departmental store URLs
Some sites are laid out like Departmental stores. There is a section for Food, inside which there would be an aisle for Fruits, within which a section with different varieties of Apples would be arranged together.

In the case of URLs, this means that you will find these pages arranged hierarchically as follows:

http://site.com/ <section> / <sub-section> / <item>

The beauty of this layout is that it is so easy to climb up to the parent section. Once you remove the tail end after the slash, you are one level up.

For example, you can create a similar structure for the articles section, as shown here:

```python
# project's main urls.py
urlpatterns = patterns(
    '',
    url(r'^articles/$', include(articles.urls), namespace="articles"),
)

# articles/urls.py
urlpatterns = patterns(
    '',
    url(r'^$', ArticlesIndex.as_view(), name='index'),
    url(r'^(?P<slug>\w+)/$', ArticleView.as_view(), name='article'),
)
```

Notice the 'index' pattern that will show an article index in case a user climbs up from a particular article.
RESTful URLs

In 2000, Roy Fielding introduced the term **Representational state transfer (REST)** in his doctoral dissertation. Reading his thesis ([http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm](http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm)) is highly recommended to better understand the architecture of the web itself. It can help you write better web applications that do not violate the core constraints of the architecture.

One of the key insights is that a URI is an identifier to a resource. A resource can be anything, such as an article, a user, or a collection of resources, such as events. Generally speaking, resources are nouns.

The web provides you with some fundamental HTTP verbs to manipulate resources: GET, POST, PUT, PATCH, and DELETE. Note that these are not part of the URL itself. Hence, if you use a verb in the URL to manipulate a resource, it is a bad practice.

For example, the following URL is considered bad:

```
http://site.com/articles/submit/
```

Instead, you should remove the verb and use the POST action to this URL:

```
http://site.com/articles/
```

![Best Practice]

Keep verbs out of your URLs if HTTP verbs can be used instead.

Note that it is not wrong to use verbs in a URL. The search URL for your site can have the verb 'search' as follows, since it is not associated with one resource as per REST:

```
http://site.com/search/?q=needle
```

RESTful URLs are very useful for designing CRUD interfaces. There is almost a one-to-one mapping between the Create, Read, Update, and Delete database operations and the HTTP verbs.

Note that the RESTful URL style is complimentary to the departmental store URL style. Most sites mix both the styles. They are separated for clarity and better understanding.
Download the example code

You can download the example code files for all Packt books you have purchased from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you. Pull requests and bug reports to the SuperBook project can be sent to https://github.com/DjangoPatternsBook/superbook

Summary

Views are an extremely powerful part of the MVC architecture in Django. Over time, class-based views have proven to be more flexible and reusable compared to traditional function-based views. Mixins are the best examples of this reusability.

Django has an extremely flexible URL dispatch system. Crafting good URLs takes into account several aspects. Well-designed URLs are appreciated by users too.

In the next chapter, we will take a look at Django's templating language and how best to leverage it.
Where to buy this book

You can buy Django Design Patterns and Best Practices from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.