tutorial.djangogirls.org

Code editor · Django Girls Tutorial

DjangoGirls

3 minuti

> For readers at home: this chapter is covered in the Installing Python & Code Editor video.

You're about to write your first line of code, so it's time to download a code editor!

> If you're using a Chromebook, skip this chapter and make sure you follow the Chromebook Setup instructions.
>
> **Note** You might have done this earlier in the Installation chapter – if so, you can skip right ahead to the next chapter!

There are a lot of different editors and it largely boils down to personal preference. Most Python programmers use complex but extremely powerful IDEs (Integrated Development Environments), such as PyCharm. As a beginner, however, that's probably less suitable; our recommendations are equally powerful, but a lot simpler.

Our suggestions are below, but feel free to ask your coach what their preferences are – it'll be easier to get help from them.

Gedit

Gedit is an open-source, free editor, available for all operating systems.

Download it here

Sublime Text 3

Sublime Text is a very popular editor with a free evaluation period. It's easy to install and use, and it's available for all operating systems.

Download it here

Atom

Atom is an extremely new code editor created by GitHub. It's free, open-source, easy to install and easy to use. It's available for Windows, OS X and Linux.

Download it here

Why are we installing a code editor?

You might be wondering why we are installing this special code editor software, rather than using something like Word or Notepad.

The first reason is that code needs to be **plain text**, and the problem with programs like Word and Textedit is that they don't actually produce plain text, they produce rich text (with fonts and formatting), using custom formats like RTF (Rich Text Format).

The second reason is that code editors are specialized for editing code, so they can provide helpful features like highlighting code with color according to its meaning, or automatically closing quotes for you.

We'll see all this in action later. Soon, you'll come to think of your trusty old code editor as one of your favorite tools. :)

tutorial.djangogirls.org

CSS – make it pretty · Django Girls Tutorial

DjangoGirls

10-12 minuti

Our blog still looks pretty ugly, right? Time to make it nice! We will use CSS for that.

What is CSS?

Cascading Style Sheets (CSS) is a language used for describing the look and formatting of a website written in a markup language (like HTML). Treat it as make-up for our web page. ;)

But we don't want to start from scratch again, right? Once more, we'll use something that programmers released on the Internet for free. Reinventing the wheel is no fun, you know.

Let's use Bootstrap!

Bootstrap is one of the most popular HTML and CSS frameworks for developing beautiful websites: https://getbootstrap.com/

It was written by programmers who worked for Twitter. Now it's developed by volunteers from all over the world!

Install Bootstrap

To install Bootstrap, you need to add this to your `<head>` in your `.html` file:

blog/templates/blog/post_list.html

```
<link rel="stylesheet"
href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0
/css/bootstrap.min.css">
<link rel="stylesheet"
href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0
/css/bootstrap-theme.min.css">
```

This doesn't add any files to your project. It just points to files that exist on the Internet. Just go ahead, open your website and refresh the page. Here it is!

Looking nicer already!

Static files in Django

Finally we will take a closer look at these things we've been calling **static files**. Static files are all your CSS and images. Their content doesn't depend on the request context and will be the same for every user.

Where to put static files for Django

Django already knows where to find the static files for the built-in "admin" app. Now we just need to add some static files for our

own app, `blog`.

We do that by creating a folder called `static` inside the blog app:

```
djangogirls
├── blog
│   ├── migrations
│   ├── static
│   └── templates
└── mysite
```

Django will automatically find any folders called "static" inside any of your apps' folders. Then it will be able to use their contents as static files.

Your first CSS file!

Let's create a CSS file now, to add your own style to your web page. Create a new directory called `css` inside your `static` directory. Then create a new file called `blog.css` inside this `css` directory. Ready?

```
djangogirls
└── blog
    └── static
        └── css
            └── blog.css
```

Time to write some CSS! Open up the `blog/static/css/blog.css` file in your code editor.

We won't be going too deep into customizing and learning about CSS here. It's pretty easy and you can learn it on your own after this workshop. There is a recommendation for a free course to learn more at the end of this page.

But let's do at least a little. Maybe we could change the color of our header? To understand colors, computers use special codes. These codes start with `#` followed by 6 letters (A–F) and numbers (0–9). For example, the code for blue is `#0000FF`. You can find the color codes for many colors here: http://www.colorpicker.com/. You may also use predefined colors, such as `red` and `green`.

In your `blog/static/css/blog.css` file you should add the following code:

blog/static/css/blog.css

```
h1 a {
    color: #FCA205;
}
```

`h1 a` is a CSS Selector. This means we're applying our styles to any `a` element inside of an `h1` element. So when we have something like `<h1><a href="">link</a></h1>`, the `h1 a` style will apply. In this case, we're telling it to change its color to `#FCA205`, which is orange. Of course, you can put your own color here!

In a CSS file we determine styles for elements in the HTML file. The first way we identify elements is with the element name. You might remember these as tags from the HTML section. Things like `a`, `h1`, and `body` are all examples of element names. We also identify elements by the attribute `class` or the attribute `id`. Class and id are names you give the element by yourself. Classes define groups of elements, and ids point to specific elements. For example, you could identify the following tag by using the tag name `a`, the class `external_link`, or the id `link_to_wiki_page`:

```
<a href="https://en.wikipedia.org/wiki/Django"
class="external_link" id="link_to_wiki_page">
```

You can read more about CSS Selectors at w3schools.

We also need to tell our HTML template that we added some CSS. Open the `blog/templates/blog/post_list.html` file and add this line at the very beginning of it:

blog/templates/blog/post_list.html

```
{% load staticfiles %}
```

We're just loading static files here. :) Between the `<head>` and `</head>` tags, after the links to the Bootstrap CSS files, add this line:

blog/templates/blog/post_list.html

```
<link rel="stylesheet" href="{% static
'css/blog.css' %}">
```

The browser reads the files in the order they're given, so we need to make sure this is in the right place. Otherwise the code in our file may be overriden by code in Bootstrap files. We just told our template where our CSS file is located.

Your file should now look like this:

blog/templates/blog/post_list.html

```
{% load staticfiles %}
<html>
    <head>
        <title>Django Girls blog</title>
        <link rel="stylesheet"
href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0
/css/bootstrap.min.css">
        <link rel="stylesheet"
```

```
href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0
/css/bootstrap-theme.min.css">
        <link rel="stylesheet" href="{% static
'css/blog.css' %}">
    </head>
    <body>
        <div>
            <h1><a href="/">Django Girls
Blog</a></h1>
        </div>

        {% for post in posts %}
            <div>
                <p>published: {{
post.published_date }}</p>
                <h1><a href="">{{ post.title
}}</a></h1>
                <p>{{ post.text|linebreaksbr
}}</p>
            </div>
        {% endfor %}
    </body>
</html>
```

OK, save the file and refresh the site!

Nice work! Maybe we would also like to give our website a little air and increase the margin on the left side? Let's try this!

blog/static/css/blog.css

```
body {
    padding-left: 15px;
}
```

Add that to your CSS, save the file and see how it works!

Maybe we can customize the font in our header? Paste this into your `<head>` in `blog/templates/blog/post_list.html` file:

blog/templates/blog/post_list.html

```
<link href="//fonts.googleapis.com
/css?family=Lobster&subset=latin,latin-ext"
rel="stylesheet" type="text/css">
```

As before, check the order and place before the link to `blog/static/css/blog.css`. This line will import a font called *Lobster* from Google Fonts (https://www.google.com

/fonts).

Find the `h1 a` declaration block (the code between braces `{` and `}`) in the CSS file `blog/static/css/blog.css`. Now add the line `font-family: 'Lobster';` between the braces, and refresh the page:

blog/static/css/blog.css

```
h1 a {
    color: #FCA205;
    font-family: 'Lobster';
}
```

Great!

As mentioned above, CSS has a concept of classes. These allow you to name a part of the HTML code and apply styles only to this part, without affecting other parts. This can be super helpful! Maybe you have two divs that are doing something different (like your header and your post). A class can help you make them look different.

Go ahead and name some parts of the HTML code. Add a class called `page-header` to your `div` that contains your header, like this:

blog/templates/blog/post_list.html

```
<div class="page-header">
    <h1><a href="/">Django Girls Blog</a></h1>
</div>
```

And now add a class `post` to your `div` containing a blog post.

blog/templates/blog/post_list.html

```
<div class="post">
    <p>published: {{ post.published_date }}</p>
    <h1><a href="">{{ post.title }}</a></h1>
    <p>{{ post.text|linebreaksbr }}</p>
</div>
```

We will now add declaration blocks to different selectors. Selectors starting with `.` relate to classes. There are many great tutorials and explanations about CSS on the Web that can help you understand the following code. For now, just copy and paste it into your `blog/static/css/blog.css` file:

blog/static/css/blog.css

```
.page-header {
    background-color: #ff9400;
    margin-top: 0;
    padding: 20px 20px 20px 40px;
}

.page-header h1, .page-header h1 a, .page-header
h1 a:visited, .page-header h1 a:active {
    color: #ffffff;
    font-size: 36pt;
    text-decoration: none;
}
```

```
.content {
    margin-left: 40px;
}

h1, h2, h3, h4 {
    font-family: 'Lobster', cursive;
}

.date {
    color: #828282;
}

.save {
    float: right;
}

.post-form textarea, .post-form input {
    width: 100%;
}

.top-menu, .top-menu:hover, .top-menu:visited {
    color: #ffffff;
    float: right;
    font-size: 26pt;
    margin-right: 20px;
}

.post {
    margin-bottom: 70px;
}
```

```
.post h1 a, .post h1 a:visited {
    color: #000000;
}
```

Then surround the HTML code which displays the posts with declarations of classes. Replace this:

blog/templates/blog/post_list.html

```
{% for post in posts %}
    <div class="post">
        <p>published: {{ post.published_date
}}</p>
        <h1><a href="">{{ post.title }}</a></h1>
        <p>{{ post.text|linebreaksbr }}</p>
    </div>
{% endfor %}
```

in the `blog/templates/blog/post_list.html` with this:

blog/templates/blog/post_list.html

```
<div class="content container">
    <div class="row">
        <div class="col-md-8">
            {% for post in posts %}
                <div class="post">
                    <div class="date">
                        <p>published: {{
post.published_date }}</p>
                    </div>
                    <h1><a href="">{{ post.title
}}</a></h1>
                    <p>{{ post.text|linebreaksbr
}}</p>
                </div>
```

```
            {% endfor %}
        </div>
    </div>
</div>
```

Save those files and refresh your website.

Woohoo! Looks awesome, right? Look at the code we just pasted to find the places where we added classes in the HTML and used them in the CSS. Where would you make the change if you wanted the date to be turquoise?

Don't be afraid to tinker with this CSS a little bit and try to change some things. Playing with the CSS can help you understand what the different things are doing. If you break something, don't worry – you can always undo it!

We really recommend taking this free online Codeacademy HTML & CSS course. It can help you learn all about making your websites prettier with CSS.

Ready for the next chapter?! :)

tutorial.djangogirls.org

Django admin · Django Girls Tutorial

DjangoGirls

3-4 minuti

To add, edit and delete the posts we've just modeled, we will use Django admin.

Let's open the `blog/admin.py` file and replace its contents with this:

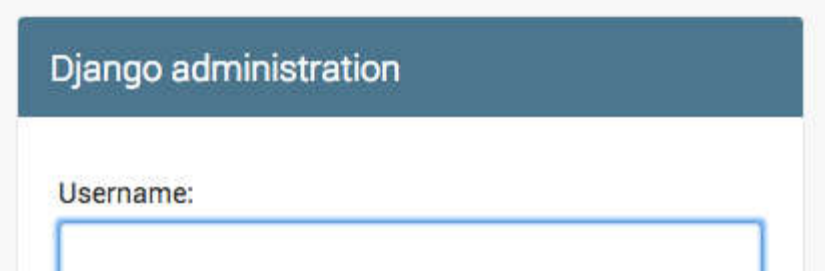
blog/admin.py

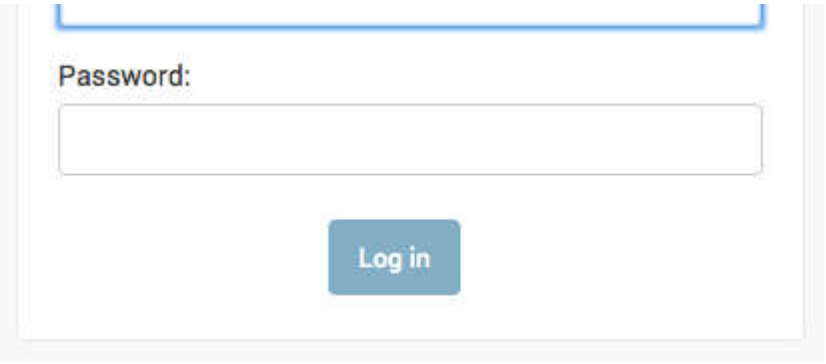
```
from django.contrib import admin
from .models import Post

admin.site.register(Post)
```

As you can see, we import (include) the Post model defined in the previous chapter. To make our model visible on the admin page, we need to register the model with `admin.site.register(Post)`.

OK, time to look at our Post model. Remember to run `python manage.py runserver` in the console to run the web server. Go to your browser and type the address http://127.0.0.1:8000/admin/. You will see a login page like this:





To log in, you need to create a *superuser* - a user account that has control over everything on the site. Go back to the command line, type `python manage.py createsuperuser`, and press enter.

> Remember, to write new commands while the web server is running, open a new terminal window and activate your virtualenv. We reviewed how to write new commands in the **Your first Django project!** chapter, in the **Starting the web server** section.

When prompted, type your username (lowercase, no spaces), email address, and password. Don't worry that you can't see the password you're typing in – that's how it's supposed to be. Just type it in and press `enter` to continue. The output should look like this (where the username and email should be your own ones):

command-line

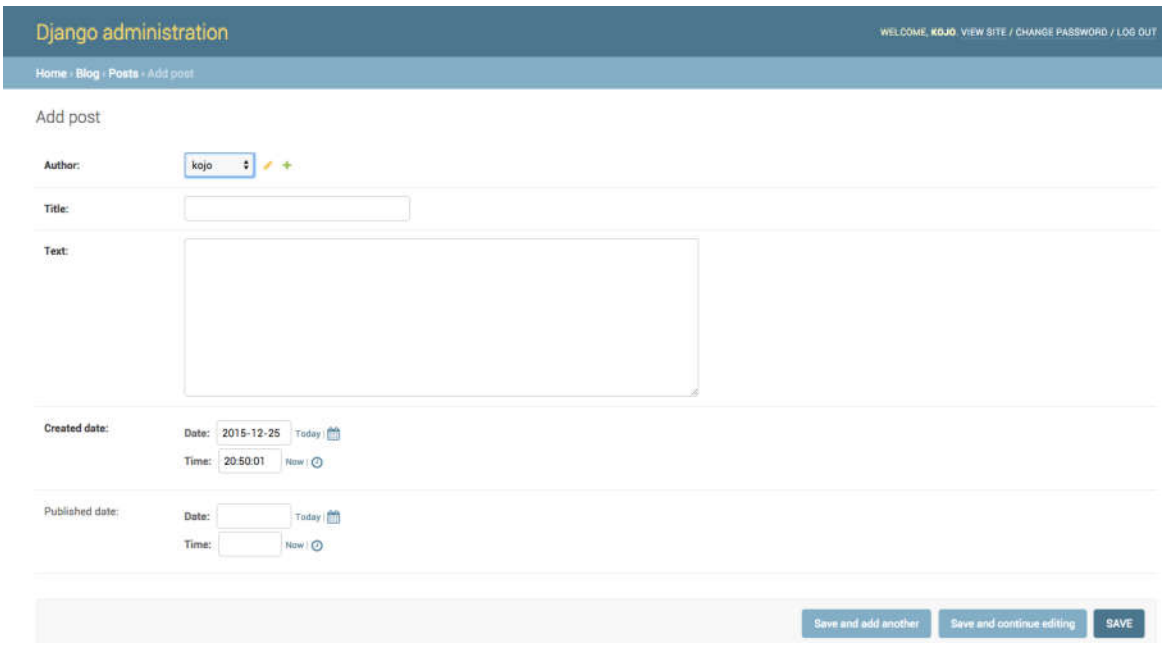
```
(myvenv) ~/djangogirls$ python manage.py
createsuperuser
Username: admin
Email address: admin@admin.com
Password:
Password (again):
Superuser created successfully.
```

Return to your browser. Log in with the superuser's credentials

you chose; you should see the Django admin dashboard.

Go to Posts and experiment a little bit with it. Add five or six blog posts. Don't worry about the content – you can simply copy-paste some text from this tutorial to save time. :)

Make sure that at least two or three posts (but not all) have the publish date set. It will be helpful later.



If you want to know more about Django admin, you should check Django's documentation: https://docs.djangoproject.com/en/1.11/ref/contrib/admin/

This is probably a good moment to grab a coffee (or tea) or something to eat to re-energize yourself. You created your first

Django model – you deserve a little break!

tutorial.djangogirls.org

Django Forms · Django Girls Tutorial

DjangoGirls

15-19 minuti

The final thing we want to do on our website is create a nice way to add and edit blog posts. Django's `admin` is cool, but it is rather hard to customize and make pretty. With `forms` we will have absolute power over our interface – we can do almost anything we can imagine!

The nice thing about Django forms is that we can either define one from scratch or create a `ModelForm` which will save the result of the form to the model.

This is exactly what we want to do: we will create a form for our `Post` model.

Like every important part of Django, forms have their own file: `forms.py`.

We need to create a file with this name in the `blog` directory.

```
blog
    └── forms.py
```

OK, let's open it and type the following code:

blog/forms.py

```
from django import forms

from .models import Post
```

```
class PostForm(forms.ModelForm):

    class Meta:
        model = Post
        fields = ('title', 'text',)
```

We need to import Django forms first (`from django import forms`) and, obviously, our `Post` model (`from .models import Post`).

`PostForm`, as you probably suspect, is the name of our form. We need to tell Django that this form is a `ModelForm` (so Django will do some magic for us) – `forms.ModelForm` is responsible for that.

Next, we have `class Meta`, where we tell Django which model should be used to create this form (`model = Post`).

Finally, we can say which field(s) should end up in our form. In this scenario we want only `title` and `text` to be exposed – `author` should be the person who is currently logged in (you!) and `created_date` should be automatically set when we create a post (i.e. in the code), right?

And that's it! All we need to do now is use the form in a *view* and display it in a template.

So once again we will create a link to the page, a URL, a view and a template.

Link to a page with the form

It's time to open `blog/templates/blog/base.html`. We will add a link in `div` named `page-header`:

blog/templates/blog/base.html

```
<a href="{% url 'post_new' %}" class="top-
menu"><span class="glyphicon glyphicon-plus">
</span></a>
```

Note that we want to call our new view `post_new`. The class `"glyphicon glyphicon-plus"` is provided by the bootstrap theme we are using, and will display a plus sign for us.

After adding the line, your HTML file should now look like this:

blog/templates/blog/base.html

```
{% load staticfiles %}
<html>
    <head>
        <title>Django Girls blog</title>
        <link rel="stylesheet"
href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0
/css/bootstrap.min.css">
        <link rel="stylesheet"
href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0
/css/bootstrap-theme.min.css">
        <link href='//fonts.googleapis.com
/css?family=Lobster&subset=latin,latin-ext'
rel='stylesheet' type='text/css'>
        <link rel="stylesheet" href="{% static
'css/blog.css' %}">
    </head>
    <body>
        <div class="page-header">
            <a href="{% url 'post_new' %}"
class="top-menu"><span class="glyphicon
glyphicon-plus"></span></a>
```

```
            <h1><a href="/">Django Girls
Blog</a></h1>
        </div>
        <div class="content container">
            <div class="row">
                <div class="col-md-8">
                    {% block content %}
                    {% endblock %}
                </div>
            </div>
        </div>
    </body>
</html>
```

After saving and refreshing the page http://127.0.0.1:8000 you will obviously see a familiar `NoReverseMatch` error, right?

URL

We open `blog/urls.py` and add a line:

blog/urls.py

```
url(r'^post/new/$', views.post_new,
name='post_new'),
```

And the final code will look like this:

blog/urls.py

```
from django.conf.urls import url
from . import views

urlpatterns = [
    url(r'^$', views.post_list,
name='post_list'),
```

```
    url(r'^post/(?P<pk>\d+)/$',
views.post_detail, name='post_detail'),
    url(r'^post/new/$', views.post_new,
name='post_new'),
]
```

After refreshing the site, we see an `AttributeError`, since we don't have the `post_new` view implemented. Let's add it right now.

post_new view

Time to open the `blog/views.py` file and add the following lines with the rest of the `from` rows:

blog/views.py

```
from .forms import PostForm
```

And then our *view*:

blog/views.py

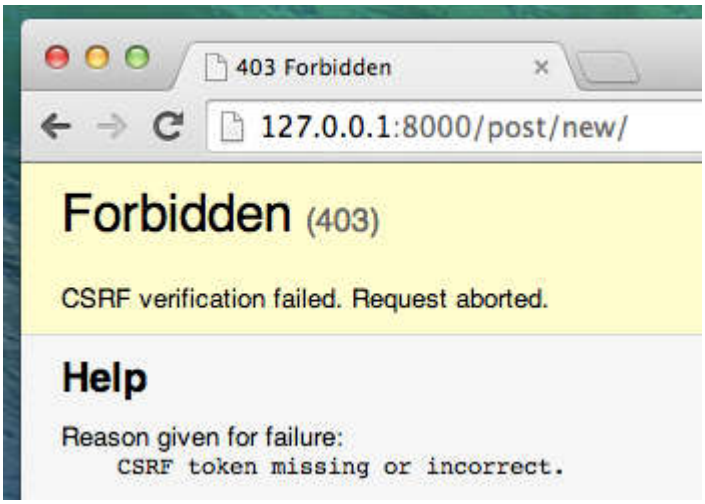
```
def post_new(request):
    form = PostForm()
    return render(request,
'blog/post_edit.html', {'form': form})
```

To create a new `Post` form, we need to call `PostForm()` and pass it to the template. We will go back to this *view*, but for now, let's quickly create a template for the form.

Template

We need to create a file `post_edit.html` in the `blog/templates/blog` directory. To make a form work we need several things:

- We have to display the form. We can do that with (for example) a simple `{{ form.as_p }}`.
- The line above needs to be wrapped with an HTML form tag: `<form method="POST">...</form>`.
- We need a `Save` button. We do that with an HTML button: `<button type="submit">Save</button>`.
- And finally, just after the opening `<form ...>` tag we need to add `{% csrf_token %}`. This is very important, since it makes your forms secure! If you forget about this bit, Django will complain when you try to save the form:



OK, so let's see how the HTML in `post_edit.html` should look:

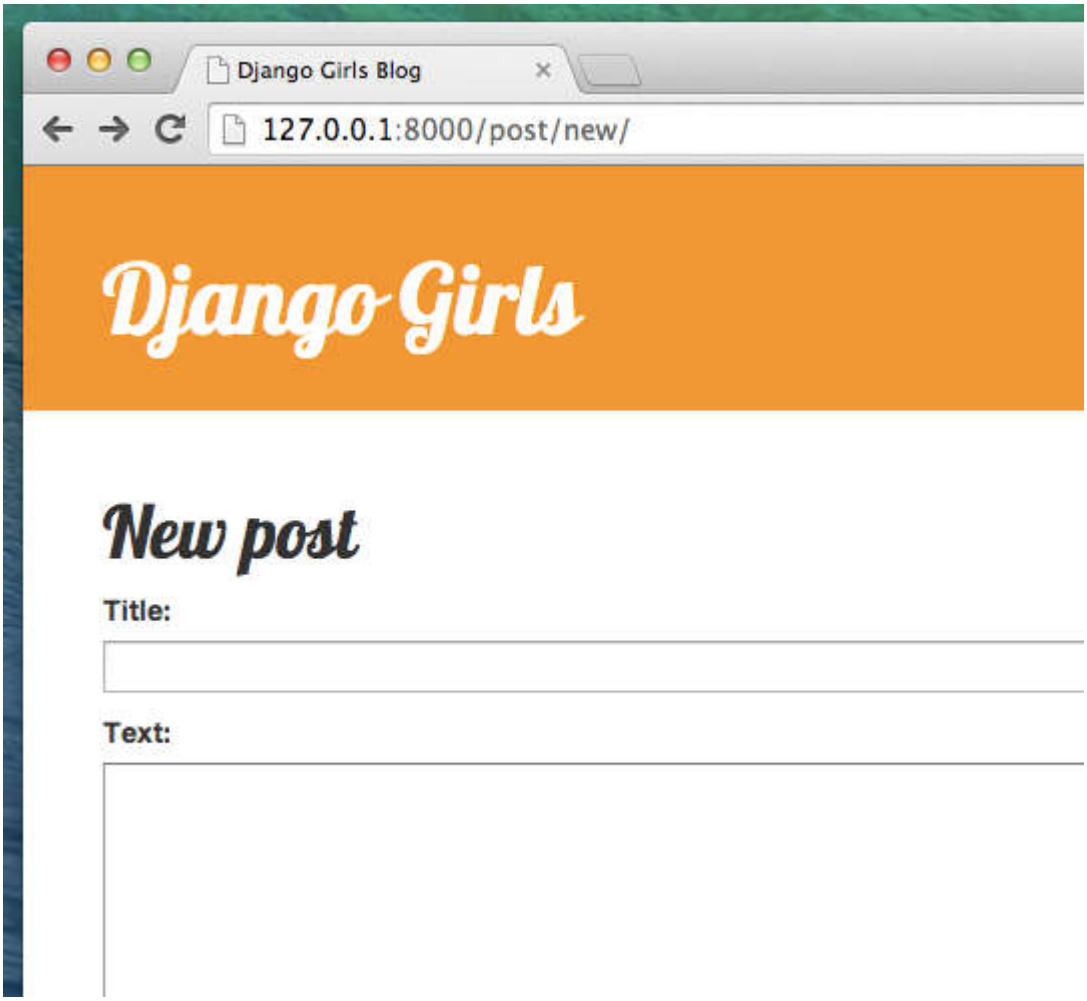
blog/templates/blog/post_edit.html

```
{% extends 'blog/base.html' %}

{% block content %}
    <h1>New post</h1>
    <form method="POST" class="post-form">{%
csrf_token %}
        {{ form.as_p }}
        <button type="submit" class="save btn
```

```
btn-default">Save</button>
    </form>
{% endblock %}
```

Time to refresh! Yay! Your form is displayed!



But, wait a minute! When you type something in the `title` and `text` fields and try to save it, what will happen?

Nothing! We are once again on the same page and our text is gone… and no new post is added. So what went wrong?

The answer is: nothing. We need to do a little bit more work in our *view*.

Saving the form

Open `blog/views.py` once again. Currently all we have in the `post_new` view is the following:

blog/views.py

```
def post_new(request):
    form = PostForm()
    return render(request,
'blog/post_edit.html', {'form': form})
```

When we submit the form, we are brought back to the same view, but this time we have some more data in `request`, more specifically in `request.POST` (the naming has nothing to do with a blog "post"; it's to do with the fact that we're "posting" data). Remember how in the HTML file, our `<form>` definition had the variable `method="POST"`? All the fields from the form are now in `request.POST`. You should not rename `POST` to anything else (the only other valid value for `method` is `GET`, but we have no time to explain what the difference is).

So in our *view* we have two separate situations to handle: first, when we access the page for the first time and we want a blank form, and second, when we go back to the *view* with all form data we just typed. So we need to add a condition (we will use `if` for that):

blog/views.py

```
if request.method == "POST":
    [...]
else:
    form = PostForm()
```

It's time to fill in the dots `[...]`. If `method` is `POST` then we want to construct the `PostForm` with data from the form, right? We will do that as follows:

blog/views.py

```
form = PostForm(request.POST)
```

Easy! The next thing is to check if the form is correct (all required fields are set and no incorrect values have been submitted). We do that with `form.is_valid()`.

We check if the form is valid and if so, we can save it!

blog/views.py

```
if form.is_valid():
    post = form.save(commit=False)
    post.author = request.user
    post.published_date = timezone.now()
    post.save()
```

Basically, we have two things here: we save the form with `form.save` and we add an author (since there was no `author` field in the `PostForm` and this field is required). `commit=False` means that we don't want to save the `Post` model yet – we want to add the author first. Most of the time you will use `form.save()` without `commit=False`, but in this case, we need to supply it. `post.save()` will preserve changes (adding the author) and a new blog post is created!

Finally, it would be awesome if we could immediately go to the `post_detail` page for our newly created blog post, right? To do that we need one more import:

blog/views.py

```
from django.shortcuts import redirect
```

Add it at the very beginning of your file. And now we can say, "go to the `post_detail` page for the newly created post":

blog/views.py

```
return redirect('post_detail', pk=post.pk)
```

`post_detail` is the name of the view we want to go to. Remember that this *view* requires a `pk` variable? To pass it to the views, we use `pk=post.pk`, where `post` is the newly created blog post!

OK, we've talked a lot, but we probably want to see what the whole *view* looks like now, right?

blog/views.py

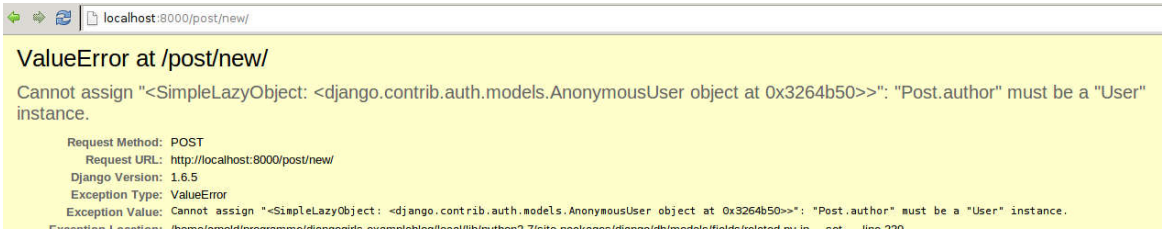
```
def post_new(request):
    if request.method == "POST":
        form = PostForm(request.POST)
        if form.is_valid():
            post = form.save(commit=False)
            post.author = request.user
            post.published_date = timezone.now()
            post.save()
            return redirect('post_detail',
pk=post.pk)
    else:
        form = PostForm()
    return render(request,
'blog/post_edit.html', {'form': form})
```

Let's see if it works. Go to the page http://127.0.0.1:8000/post/new/, add a `title` and `text`, save it... and voilà! The new blog post is added and we are redirected to the `post_detail` page!

You might have noticed that we are setting the publish date before saving the post. Later on, we will introduce a *publish button* in **Django Girls Tutorial: Extensions**.

That is awesome!

> As we have recently used the Django admin interface, the system currently thinks we are still logged in. There are a few situations that could lead to us being logged out (closing the browser, restarting the DB, etc.). If, when creating a post, you find that you are getting errors referring to the lack of a logged-in user, head to the admin page http://127.0.0.1:8000/admin and log in again. This will fix the issue temporarily. There is a permanent fix awaiting you in the **Homework: add security to your website!** chapter after the main tutorial.



Form validation

Now, we will show you how cool Django forms are. A blog post needs to have `title` and `text` fields. In our `Post` model we did not say that these fields (as opposed to `published_date`) are not required, so Django, by default, expects them to be set.

Try to save the form without `title` and `text`. Guess what will happen!

Django is taking care to validate that all the fields in our form are correct. Isn't it awesome?

Edit form

Now we know how to add a new form. But what if we want to edit an existing one? This is very similar to what we just did. Let's create some important things quickly. (If you don't understand something, you should ask your coach or look at the previous chapters, since we covered all these steps already.)

Open `blog/templates/blog/post_detail.html` and add the line

blog/templates/blog/post_detail.html

```
<a class="btn btn-default" href="{% url 'post_edit' pk=post.pk %}"><span class="glyphicon glyphicon-pencil"></span></a>
```

so that the template will look like this:

blog/templates/blog/post_detail.html

```
{% extends 'blog/base.html' %}

{% block content %}
    <div class="post">
```

```
        {% if post.published_date %}
            <div class="date">
                {{ post.published_date }}
            </div>
        {% endif %}
        <a class="btn btn-default" href="{% url
'post_edit' pk=post.pk %}"><span
class="glyphicon glyphicon-pencil"></span></a>
        <h1>{{ post.title }}</h1>
        <p>{{ post.text|linebreaksbr }}</p>
    </div>
{% endblock %}
```

In `blog/urls.py` we add this line:

blog/urls.py

```
    url(r'^post/(?P<pk>\d+)/edit/$',
views.post_edit, name='post_edit'),
```

We will reuse the template `blog/templates/blog/post_edit.html`, so the last missing thing is a *view*.

Let's open `blog/views.py` and add this at the very end of the file:

blog/views.py

```
def post_edit(request, pk):
    post = get_object_or_404(Post, pk=pk)
    if request.method == "POST":
        form = PostForm(request.POST,
instance=post)
        if form.is_valid():
            post = form.save(commit=False)
            post.author = request.user
```

```
            post.published_date = timezone.now()
            post.save()
            return redirect('post_detail',
pk=post.pk)
    else:
        form = PostForm(instance=post)
    return render(request,
'blog/post_edit.html', {'form': form})
```

This looks almost exactly the same as our `post_new` view, right? But not entirely. For one, we pass an extra `pk` parameter from urls. Next, we get the `Post` model we want to edit with `get_object_or_404(Post, pk=pk)` and then, when we create a form, we pass this post as an `instance`, both when we save the form…

blog/views.py

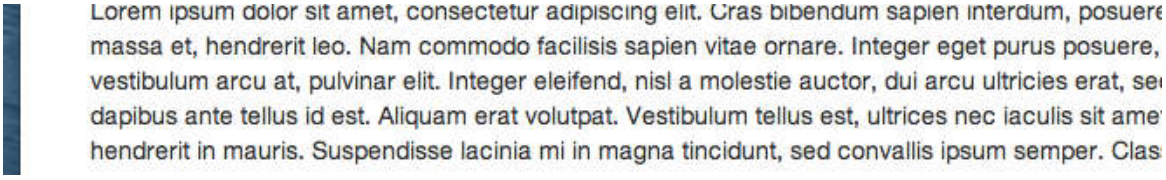
```
form = PostForm(request.POST, instance=post)
```

…and when we've just opened a form with this post to edit:

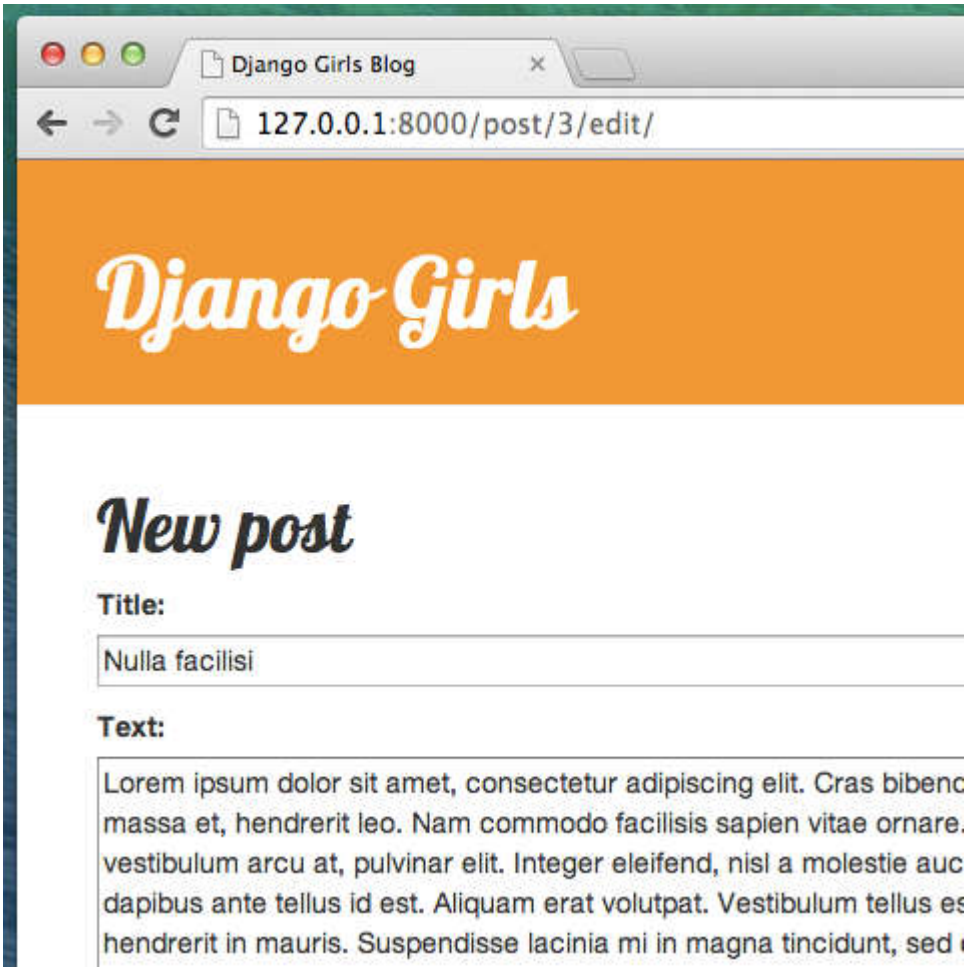
blog/views.py

```
form = PostForm(instance=post)
```

OK, let's test if it works! Let's go to the `post_detail` page. There should be an edit button in the top-right corner:



When you click it you will see the form with our blog post:



Feel free to change the title or the text and save the changes!

Congratulations! Your application is getting more and more complete!

If you need more information about Django forms, you should read the documentation: https://docs.djangoproject.com/en/1.11/topics/forms/

Security

Being able to create new posts just by clicking a link is awesome! But right now, anyone who visits your site will be able to make a new blog post, and that's probably not something you want. Let's

make it so the button shows up for you but not for anyone else.

In `blog/templates/blog/base.html`, find our `page-header div` and the anchor tag you put in there earlier. It should look like this:

blog/templates/blog/base.html

```
<a href="{% url 'post_new' %}" class="top-menu"><span class="glyphicon glyphicon-plus"></span></a>
```

We're going to add another `{% if %}` tag to this, which will make the link show up only for users who are logged into the admin. Right now, that's just you! Change the `<a>` tag to look like this:

blog/templates/blog/base.html

```
{% if user.is_authenticated %}
    <a href="{% url 'post_new' %}" class="top-menu"><span class="glyphicon glyphicon-plus"></span></a>
{% endif %}
```

This `{% if %}` will cause the link to be sent to the browser only if the user requesting the page is logged in. This doesn't protect the creation of new posts completely, but it's a good first step. We'll cover more security in the extension lessons.

Remember the edit icon we just added to our detail page? We also want to add the same change there, so other people won't be able to edit existing posts.

Open `blog/templates/blog/post_detail.html` and find this line:

blog/templates/blog/post_detail.html

```
<a class="btn btn-default" href="{% url
'post_edit' pk=post.pk %}"><span
class="glyphicon glyphicon-pencil"></span></a>
```

Change it to this:

blog/templates/blog/post_detail.html

```
{% if user.is_authenticated %}
    <a class="btn btn-default" href="{% url
'post_edit' pk=post.pk %}"><span
class="glyphicon glyphicon-pencil"></span></a>
{% endif %}
```

Since you're likely logged in, if you refresh the page, you won't see anything different. Load the page in a different browser or an incognito window (called "InPrivate" in Windows Edge), though, and you'll see that the link doesn't show up, and the icon doesn't display either!

One more thing: deploy time!

Let's see if all this works on PythonAnywhere. Time for another deploy!

- First, commit your new code, and push it up to Github:

  command-line

  ```
  $ git status
  $ git add --all .
  $ git status
  $ git commit -m "Added views to create/edit blog
  post inside the site."
  $ git push
  ```

- Then, in a PythonAnywhere Bash console:

command-line

```
$ cd my-first-blog
$ git pull
[...]
```

- Finally, hop on over to the Web tab and hit **Reload**.

  And that should be it! Congrats :)