The Flask Mega-Tutorial Part XI: Facelift (/post/the-flask-mega-tutorial-part-xi-facelift)

February 13 2018

Posted by Miguel Grinberg (/author/Miguel Grinberg) under Flask (/category/Flask), Python (/category/Python), Programming (/category/Programming).

Tweet Like Share

This is the eleventh installment of the Flask Mega-Tutorial series, in which I'm going to tell you how to replace the basic HTML templates with a new set that is based on the Bootstrap user interface framework.

For your reference, below is a list of the articles in this series.

- Chapter 1: Hello, World! (/post/the-flask-mega-tutorial-part-i-hello-world)
- Chapter 2: Templates (/post/the-flask-mega-tutorial-part-ii-templates)
- Chapter 3: Web Forms (/post/the-flask-mega-tutorial-part-iii-web-forms)
- Chapter 4: Database (/post/the-flask-mega-tutorial-part-iv-database)
- Chapter 5: User Logins (/post/the-flask-mega-tutorial-part-v-user-logins)
- Chapter 6: Profile Page and Avatars (/post/the-flask-mega-tutorial-part-vi-profile-page-and-avatars)
- Chapter 7: Error Handling (/post/the-flask-mega-tutorial-part-vii-error-handling)
- Chapter 8: Followers (/post/the-flask-mega-tutorial-part-viii-followers)
- Chapter 9: Pagination (/post/the-flask-mega-tutorial-part-ix-pagination)
- Chapter 10: Email Support (/post/the-flask-mega-tutorial-part-x-email-support)
- Chapter 11: Facelift (/post/the-flask-mega-tutorial-part-xi-facelift) (this article)
- Chapter 12: Dates and Times (/post/the-flask-mega-tutorial-part-xii-dates-and-times)
- Chapter 13: I18n and L10n (/post/the-flask-mega-tutorial-part-xiii-i18n-and-l10n)
- Chapter 14: Ajax (/post/the-flask-mega-tutorial-part-xiv-ajax)
- Chapter 15: A Better Application Structure (/post/the-flask-mega-tutorial-part-xv-a-better-application-structure)
- Chapter 16: Full-Text Search (/post/the-flask-mega-tutorial-part-xvi-full-text-search)
- Chapter 17: Deployment on Linux (/post/the-flask-mega-tutorial-part-xvii-deployment-on-linux)
- Chapter 18: Deployment on Heroku (/post/the-flask-mega-tutorial-part-xviii-deployment-on-heroku)
- Chapter 19: Deployment on Docker Containers (/post/the-flask-mega-tutorial-part-xix-deployment-on-docker-containers)
- Chapter 20: Some JavaScript Magic (/post/the-flask-mega-tutorial-part-xx-some-javascript-magic)

- Chapter 21: User Notifications (/post/the-flask-mega-tutorial-part-xxi-user-notifications)
- Chapter 22: Background Jobs (/post/the-flask-mega-tutorial-part-xxii-background-jobs)
- Chapter 23: Application Programming Interfaces (APIs) (/post/the-flask-mega-tutorial-part-xxiii-application-programming-interfaces-apis)

Note 1: If you are looking for the legacy version of this tutorial, it's here (/post/the-flask-mega-tutorial-part-i-hello-world-legacy).

Note 2: If you would like to support my work on this blog, or just don't have patience to wait for weekly articles, I am offering the complete version of this tutorial packaged as an ebook or a set of videos. For more information, visit courses.miguelgrinberg.com (https://courses.miguelgrinberg.com).

You have been playing with my Microblog application for a while now, so I'm sure you noticed that I haven't spent too much time making it look good, or better said, I haven't spent any time on that. The templates that I put together are pretty basic, with absolutely no custom styling. It was useful for me to concentrate on the actual logic of the application without having the distraction of also writing good looking HTML and CSS.

But I've focused on the backend part of this application for a while now. So in this chapter I'm taking a break from that and will spend some time showing you what can be done to make the application look a bit more polished and professional.

This chapter is going to be a bit different than previous ones, because I'm not going to be as detailed as I normally am with the Python side, which after all, is the main topic of this tutorial. Creating good looking web pages is a vast topic that is largely unrelated to Python web development, but I will discuss some basic guidelines and ideas on how to approach the task, and you will also have the application with the redesigned looks to study and learn from.

The GitHub links for this chapter are: Browse (https://github.com/miguelgrinberg/microblog/tree/v0.11), Zip (https://github.com/miguelgrinberg/microblog/archive/v0.11.zip), Diff (https://github.com/miguelgrinberg/microblog/compare/v0.10...v0.11).

CSS Frameworks

While we can argue that coding is hard, our pains are nothing compared to those of web designers, who have to write templates that have a nice and consistent look on a list of web browsers. It has gotten better in recent years, but there are still obscure bugs or quirks in some browsers that make the task of designing web pages that look nice everywhere very hard. This is even harder if you also need to target resource and screen limited browsers of tablets and smartphones.

If you, like me, are a developer who just wants to create decent looking web pages, but do not have the time or interest to learn the low level mechanisms to achieve this effectively by writing raw HTML and CSS, then the only practical solution is to use a *CSS framework* to simplify the task. You will be losing some creative freedom by taking this path, but on the other side, your web pages will look good in all browsers without a lot of effort. A CSS framework provides a collection of high-level CSS classes with pre-made styles for common types of user interface elements. Most of these frameworks also provide JavaScript add-ons for things that cannot be done strictly with HTML and CSS.

Introducing Bootstrap

One of the most popular CSS frameworks is Bootstrap (http://getbootstrap.com/), created by Twitter. If you want to see the kind of pages that can be designed with this framework, the documentation has some examples (https://getbootstrap.com/docs/3.3/getting-started/#examples).

These are some of the benefits of using Bootstrap to style your web pages:

- Similar look in all major web browsers
- Handling of desktop, tablet and phone screen sizes
- Customizable layouts
- Nicely styled navigation bars, forms, buttons, alerts, popups, etc.

The most direct way to use Bootstrap is to simply import the *bootstrap.min.css* file in your base template. You can either download a copy of this file and add it to your project, or import it directly from a CDN (https://en.wikipedia.org/wiki/Content_delivery_network). Then you can start using the general purpose CSS classes it provides, according to the documentation (https://getbootstrap.com/docs/3.3/getting-started/), which is pretty good. You may also want to import the *bootstrap.min.js* file containing the framework's JavaScript code, so that you can also use the most advanced features.

Fortunately, there is a Flask extension called Flask-Bootstrap (https://pythonhosted.org/Flask-Bootstrap/) that provides a ready to use base template that has the Bootstrap framework installed. Let's install this extension:

```
(venv) $ pip install flask-bootstrap
```

Using Flask-Bootstrap

Flask-Bootstrap needs to be initialized like most other Flask extensions:

app/__init__.py: Flask-Bootstrap instance.

```
# ...
from flask_bootstrap import Bootstrap

app = Flask(__name__)
# ...
bootstrap = Bootstrap(app)
```

With the extension initialized, a *bootstrap/base.html* template becomes available, and can be referenced from application templates with the `extends` clause.

But as you recall, I'm already using the `extends` clause with my own base template, which allows me to have the common parts of the page in a single place. My *base.html* template defined the navigation bar, which included a few links, and also exported a `content` block . All other templates in my application inherit from the base template and provide the `content` block with the main content of the page.

So how can I fit the Bootstrap base template? The idea is to use a three-level hierarchy instead of just two. The *bootstrap/base.html* template provides the basic structure of the page, which includes the Bootstrap framework files. This template exports a few blocks for derived templates such as `title` , `navbar` and `content` (see the complete list of blocks here (https://pythonhosted.org/Flask-Bootstrap/basic-usage.html#available-blocks)). I'm going to change my *base.html* template to derive from *bootstrap/base.html* and provide implementations for the `title` , `navbar` and `content` blocks. In turn, *base.html* will export its own `app_content` block for its derived templates to define the page content.

Below you can see how the *base.html* looks after I modified it to inherit from the Bootstrap base template. Note that this listing does not include the entire HTML for the navigation bar, but you can see the full implementation on GitHub or by downloading the code for this chapter.

app/templates/base.html: Redesigned base template.

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

{% block title %}
    {% if title %}{{ title }} - Microblog{% else %}Welcome to Microblog{% endif %}
{% endblock %}

{% block navbar %}
    <nav class="navbar navbar-default">
        ... navigation bar here (see complete code on GitHub) ...
    </nav>
{% endblock %}

{% block content %}
    <div class="container">
        {% with messages = get_flashed_messages() %}
        {% if messages %}
            {% for message in messages %}
            <div class="alert alert-info" role="alert">{{ message }}</div>
            {% endfor %}
        {% endif %}
        {% endwith %}

        {# application content needs to be provided in the app_content block #}
        {% block app_content %}{% endblock %}
    </div>
{% endblock %}
```

Here you can see how I make this template derive from *bootstrap/base.html*, followed by the three blocks that implement the page title, navigation bar and page content respectively.

The `title` block needs to define the text that will be used for the page title, with the `<title>` tags. For this block I simply moved the logic that was inside the `<title>` tag in the original base template.

The `navbar` block is an optional block that can be used to define a navigation bar. For this block, I adapted the example in the Bootstrap navigation bar documentation so that it includes a site branding on the left end, followed by the Home and Explore links. I then added the Profile and Login or Logout links aligned with the right border of the page. As I mentioned above, I omitted the HTML in the example above, but you can obtain the full *base.html* template from the download package for this chapter.

Finally, in the `content` block I'm defining a top-level container, and inside it I have the logic that renders flashed messages, which are now going to appear styled as Bootstrap alerts. That is followed with a new `app_content` block that is defined just so that derived templates can define their own content.

The original version of all the page templates defined their content in a block named `content`. As you saw above, the block named `content` is used by Flask-Bootstrap, so I renamed my content block as `app_content`. So all my templates have to be renamed to use `app_content` as their content block. As an example, here how the modified version of the *404.html* template looks like:

app/templates/404.html: Redesigned 404 error template.

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

{% block app_content %}
    <h1>File Not Found</h1>
    <p><a href="{{ url_for('index') }}">Back</a></p>
{% endblock %}
```

Rendering Bootstrap Forms

An area where Flask-Bootstrap does a fantastic job is in rendering of forms. Instead of having to style the form fields one by one, Flask-Bootstrap comes with a macro that accepts a Flask-WTF form object as an argument and renders the complete form using Bootstrap styles.

Below you can see the redesigned *register.html* template as an example:

app/templates/register.html: User registration template.

```
{% extends "base.html" %}
{% import 'bootstrap/wtf.html' as wtf %}

{% block app_content %}
    <h1>Register</h1>
    <div class="row">
        <div class="col-md-4">
            {{ wtf.quick_form(form) }}
        </div>
    </div>
{% endblock %}
```

Isn't this great? The `import` statement near the top works similarly to a Python import on the template side. That adds a `wtf.quick_form()` macro that in a single line of code renders the complete form, including support for display validation errors, and all styled as appropriate for the Bootstrap framework.

Once again, I'm not going to show you all the changes that I've done for the other forms in the application, but these changes are all made in the code that you can download or inspect on GitHub.

Rendering of Blog Posts

The presentation logic that renders a single blog posts was abstracted into a sub-template called *_post.html*. All I need to do with this template is make some minor adjustments so that it looks good under Bootstrap.

app/templates/_post.html: Redesigned post sub-template.

```
<table class="table table-hover">
    <tr>
        <td width="70px">
            <a href="{{ url_for('user', username=post.author.username) }}">
                <img src="{{ post.author.avatar(70) }}" />
            </a>
        </td>
        <td>
            <a href="{{ url_for('user', username=post.author.username) }}">
                {{ post.author.username }}
            </a>
            says:
            <br>
            {{ post.body }}
        </td>
    </tr>
</table>
```

Rendering Pagination Links

Pagination links is another area where Bootstrap provides direct support. For this I just went one more time to the Bootstrap documentation (https://getbootstrap.com/docs/3.3/components/#optional-disabled-state) and adapted one of their examples. Here is how these look in the *index.html* page:

app/templates/index.html: Redesigned pagination links.

```
...
<nav aria-label="...">
    <ul class="pager">
        <li class="previous{% if not prev_url %} disabled{% endif %}">
            <a href="{{ prev_url or '#' }}">
                <span aria-hidden="true">&larr;</span> Newer posts
            </a>
        </li>
        <li class="next{% if not next_url %} disabled{% endif %}">
            <a href="{{ next_url or '#' }}">
                Older posts <span aria-hidden="true">&rarr;</span>
            </a>
        </li>
    </ul>
</nav>
```

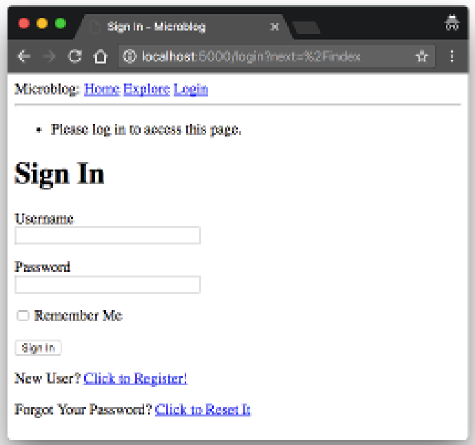
Note that in this implementation, instead of hiding the next or previous link when that direction does not have any more content, I'm applying a disabled state, which will make the link appear grayed out.

I'm not going to show it here, but a similar change needs to be applied to *user.html*. The download package for this chapter includes these changes.

Before And After

To update your application with these changes, please download the zip file for this chapter and update your templates accordingly.

Below you can see a few before and after pictures to see the transformation. Keep in mind that this change was achieved without changing a single line of application logic!



Microblog: Home Explore Login

- Please log in to access this page.

Sign In

Username

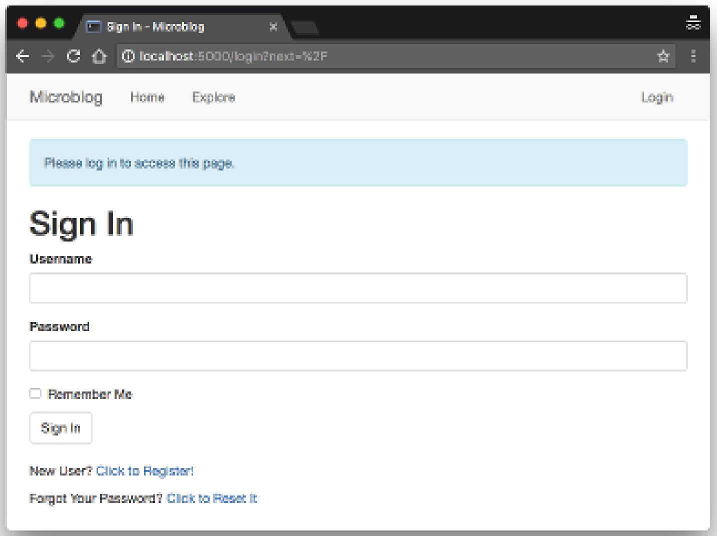
Password

Remember Me

Sign In

New User? Click to Register!

Forgot Your Password? Click to Reset It



Microblog Home Explore Login

Please log in to access this page.

Sign In

Username

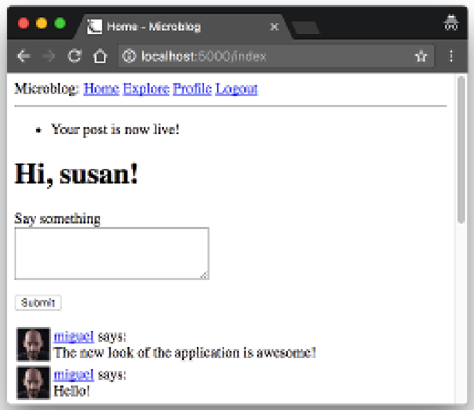
Password

Remember Me

Sign In

New User? Click to Register!

Forgot Your Password? Click to Reset It



Microblog: Home Explore Profile Logout

- Your post is now live!

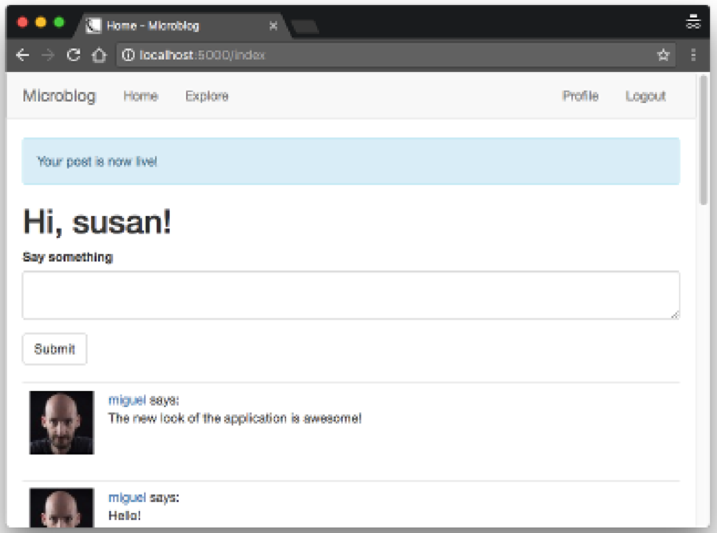
Hi, susan!

Say something

Submit

miguel says:
The new look of the application is awesome!

miguel says:
Hello!



Microblog Home Explore Profile Logout

Your post is now live!

Hi, susan!

Say something

Submit

miguel says:
The new look of the application is awesome!

miguel says:
Hello!

Hello, and thank you for visiting my blog! If you enjoyed this article, please consider supporting my work on this blog on Patreon (https://patreon.com/miguelgrinberg)!

BECOME A PATRON *(https://patreon.com/miguelgrinberg)*

Tweet Like Share