miguelgrinberg.com

The Flask Mega-Tutorial, Part XI: Facelift

Posted by Miguel Grinberg *on* December 3, 2023 *under* Flask Python Programming

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.

You are reading the 2024 edition of the Flask Mega-Tutorial. The complete course is also available to order in e-book and paperback formats from Amazon. Thank you for your support!

If you are looking for the 2018 edition of this course, you can find it here.

For your reference, here is the complete list of articles in this series:

- Chapter 1: Hello, World!
- Chapter 2: Templates
- Chapter 3: Web Forms
- Chapter 4: Database
- Chapter 5: User Logins
- Chapter 6: Profile Page and Avatars
- Chapter 7: Error Handling
- Chapter 8: Followers
- Chapter 9: Pagination
- Chapter 10: Email Support
- Chapter 11: Facelift (this article)
- Chapter 12: Dates and Times
- Chapter 13: I18n and L10n
- Chapter 14: Ajax
- Chapter 15: A Better Application Structure
- Chapter 16: Full-Text Search
- Chapter 17: Deployment on Linux
- Chapter 18: Deployment on Heroku
- Chapter 19: Deployment on Docker Containers
- Chapter 20: Some JavaScript Magic
- Chapter 21: User Notifications
- Chapter 22: Background Jobs
- Chapter 23: Application Programming Interfaces (APIs)

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 from 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 great 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](#), [Zip](#), [Diff](#).

CSS Frameworks

While we can argue that coding is hard, our pains are nothing compared to those of web designers, who have to create web pages 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](#). If you want to see the kind of pages that can be designed with this framework, the documentation has some [examples](#).

These are some benefits you get when 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. Then you can start using the general purpose CSS classes it provides, according to the documentation, which is pretty good. You may also want to import the framework's JavaScript code, so that you can also use the most advanced features.

As most open source projects, Bootstrap is constantly evolving. The original version of the Flask Mega-Tutorial was built for Bootstrap 3. The revision that you are currently reading is built for Bootstrap 5.3. The current approach to integrate Bootstrap is fairly generic and can be adapted to newer versions of Bootstrap.

Using Bootstrap

The first step in integrating Bootstrap with Microblog is to add its CSS and JavaScript files to the base template. The Bootstrap's Quick Start page provides a short, yet complete HTML page as an example, which I copy below for your convenience:

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1
    <title>Bootstrap demo</title>
    <link
        href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/css/bo
        rel="stylesheet"
        integrity="sha384-T3c6CoIi6uLrA9TneNEoa7RxnatzjcDSCmG1MXxSR1GA
        crossorigin="anonymous">
  </head>
  <body>
    <h1>Hello, world!</h1>
    <script
        src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/js/boot
        integrity="sha384-C6RzsynM9kWDrMNeT87bh95OGNyZPhcTNXj1NW7RuBCs
        crossorigin="anonymous">
    </script>
  </body>
</html>
```

The approach that I can take to combine this with my *base.html* template is to take the above as my new base template, replacing the `<title>` and `<h1>` tags with the title and body content of the original base template respectively.

The next step is to replace the basic navigation bar with the much nicer one from Bootstrap. Bootstrap's Navbar documentation page shows a

nice example near the top. With this example as guidance, I have created a navigation bar that has the index, explore, profile, log in and log out links from Microblog. For a nice touch, I configured the profile and log in and out links to appear on the far right.

When using Bootstrap, there are some basic layout primitives that are good to be aware of. One of the most important ones is the container, which defines what is the content area of the page. The two main containers are called `container` and `container-fluid`. The former configures the page to use one of a list of five predefined page widths, and centers the content on the browser window. The fluid container, on the other side, gives you access to the entire width of the page. For this application I've decided to use the default container, because it prevents the page from ever going too wide, regardless of screen size. This is the one I'm going to use, so the content portion of the page will be wrapped in one of these containers, as follows:

```
<div class="container">
    ... page contents here ...
</div>
```

The last piece of HTML markup in the *base.html* template that needs to be adapted is the section that displays the flashed messages. Bootstrap's Alert component nicely fits this need.

You can obtain the completely redesigned *base.html* template from the Github repository for this chapter. Below you can see a simplified structure if you want to have an idea of how it looks:

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

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1
    {% if title %}
    <title>{{ title }} - Microblog</title>
    {% else %}
    <title>Welcome to Microblog</title>
    {% endif %}
    <link ... bootstrap CSS ...>
  </head>
  <body>
    <nav>
      ... navigation bar (see complete code on GitHub) ...
    </nav>
    <div class="container mt-3">
      {% with messages = get_flashed_messages() %}
      {% if messages %}
        {% for message in messages %}
        <div class="alert alert-info" role="alert">{{ message }}</div>
        {% endfor %}
      {% endif %}
      {% endwith %}
      {% block content %}{% endblock %}
    </div>
```

```
    <script ... bootstrap JavaScript ...></script>
  </body>
</html>
```

With the redesigned base template in place, there is already a noticeable improvement in how the application looks, without having to change a single line of Python code. If you want to see for yourself, download a copy of *base.html* from the GitHub repository using the links shown at the start of this chapter.

Rendering Bootstrap Forms

An area where Bootstrap does a fantastic job is in rendering of form fields, which have a much nicer and cleaner look than the default fields provided by the browser. The Bootstrap documentation also has a section on Forms. Near the start of this section there is an example of a login form that shows the basic HTML structure.

The HTML code required for each field is somewhat long. Below you can see one of the text fields from the example form in the documentation:

```
  <div class="mb-3">
    <label for="exampleInputPassword1" class="form-label">Password</la
    <input type="password" class="form-control" id="exampleInputPasswo
  </div>
```

But this is too simple for the needs of Microblog, which includes field validation and may need to show validation errors to the user. The documentation page has a section on server-side validation that shows how to style fields with an error message. Here is an example:

```
  <div class="col-md-3">
    <label for="validationServer05" class="form-label">Zip</label>
    <input type="text" class="form-control is-invalid" id="validationS
    <div id="validationServer05Feedback" class="invalid-feedback">
      Please provide a valid zip.
    </div>
  </div>
```

Unfortunately having to type such an amount of boilerplate for every form field in every form is out of the question. It would be too time-consuming and error-prone. One solution is to take advantage of Jinja *macros*, which allow you to define reusable snippets of HTML and then call them from your templates as if they were functions.

For example, a Jinja macro for a text field such as those shown above would be:

```
{% macro form_field(field) %}
  <div class="mb-3">
    {{ field.label(class='form-label') }}
    {{ field(class='form-control' + (' is-invalid' if field.errors els
    {%- for error in field.errors %}
    <div class="invalid-feedback">{{ error }}</div>
    {%- endfor %}
  </div>
{% endmacro %}
```

Note how conditionals are used to selectively add the error style if the field includes one or more error messages.

Having the macro defined in a file called *bootstrap_wtf.html* located in the *templates* directory, when a field needs to be rendered the macro can be called. For example:

```
{% import "bootstrap_wtf.html" as wtf %}
...
{{ wtf.form_field(form.username) }}
```

The field rendering macro can be extended to also support rendering of check boxes, selection drop-downs, submit buttons and other field types. It can also accept a second argument with a boolean indicating if the field should be automatically given focus on the page, which should be done on the first field of the form. For even more convenience, another macro can be created to render an entire form, just by iterating over the form fields and calling the `form_field()` macro for each one.

The complete *bootstrap_wtf.html* file is available on the GitHub repository linked at the start of this chapter. It includes a more complete version of the `form_field()` macro shown above, and a second macro called `quick_form()` which takes a form object and renders all of its fields using the first macro.

How does this look when implemented on an actual application form? Below you can see a redesigned *register.html* template as an example:

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

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

{% block content %}
    <h1>Register</h1>
    {{ wtf.quick_form(form) }}
{% 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 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 templates 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 to 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.usern
                <img src="{{ post.author.avatar(70) }}" />
            </a>
        </td>
        <td>
            <a href="{{ url_for('user', username=post.author.usern
                {{ post.author.username }}
            </a>
            says:
            <br>
            {{ post.body }}
        </td>
    </tr>
</table>
```

Rendering Pagination Links

Pagination links is another area where Bootstrap provides support. For this I just went one more time to the Bootstrap documentation 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="Post navigation">
        <ul class="pagination">
            <li class="page-item{% if not prev_url %} disabled{% endif
                <a class="page-link" href="{{ prev_url }}">
                    <span aria-hidden="true">&larr;</span> Newer posts
                </a>
            </li>
            <li class="page-item{% if not next_url %} disabled{% endif
                <a class="page-link" href="{{ next_url }}">
                    Older posts <span aria-hidden="true">&rarr;</span>
                </a>
            </li>
```

```

    </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!
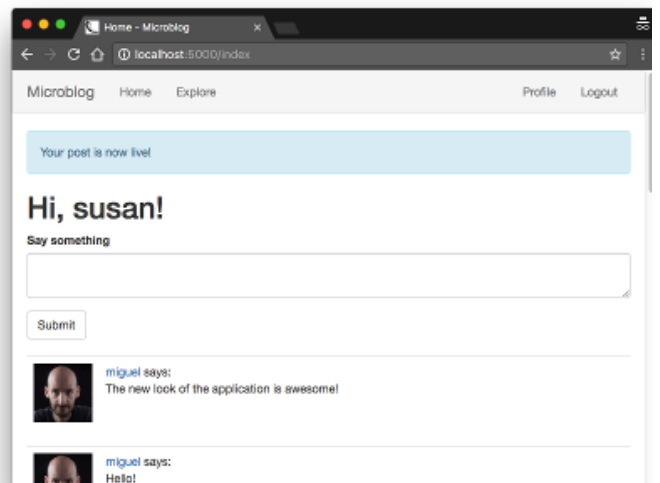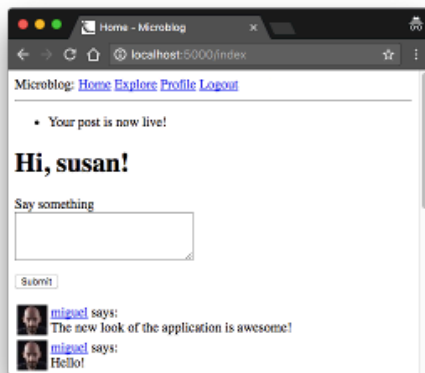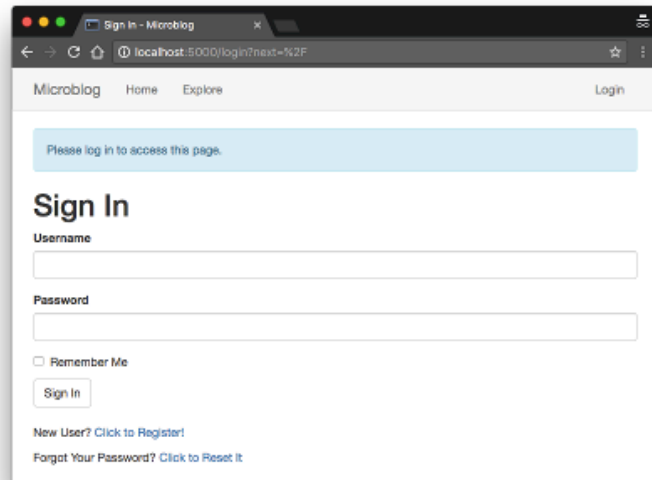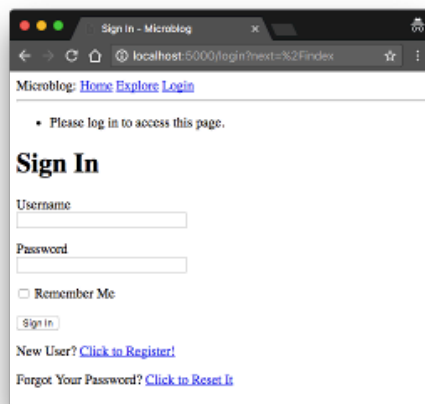
Continue on to the next chapter.

## Become a Patron!

Hello, and thank you for visiting my blog! If you enjoyed this article, please consider supporting my work on this blog on Patreon!

[BECOME A PATRON]

## Share this post:

Hacker News          Reddit          Twitter          LinkedIn          Facebook

E-Mail

17 comments

**#1    Anggoro Dhanumurti**   said a year ago

is your website here based on microblog template?

**#2    Miguel Grinberg**   said a year ago

@Anggoro: No. I created microblog separately from this blog, they are not the same project.

**#3    Andre**   said a year ago

You're adding bootstrap directly to your templates. Which works of course, but did you consider bootstrap-flask [0] instead? You're probably aware of it, so I wonder what your reasons to not use it are.

[0] https://bootstrap-flask.readthedocs.io/en/stable/

**#4    Miguel Grinberg**   said a year ago

@Andre: Bootstrap-Flask is already behind, using Bootstrap 5.1. This is always going to be the case when you depend on a developer updating the Flask extension to add support for new versions of Bootstrap. I prefer to show you how to work with Bootstrap directly, and then you have full control of your upgrades. Basically a teaching you to fish instead of giving you the fish directly.

**#5    Andre**   said a year ago

5.1 is linked from their online docs, but they're actually up to date atm.
But anyway, what I'd like to avoid is to fiddle with jinja templates for bootstrap directly, as that can get rather messy.
Do you know of a nice set of such templates?
bootstrap-flask is just one option, but in the end it doesn't even have to be a flask extension.

**#6**   **Miguel Grinberg**   said a year ago

@Andre: Flask is not opinionated. If anything that I show in this tutorial is not something you like, you are more than welcome to use your own approach. I'm not trying to force anyone to follow what I do, my goal is to give you the basic knowledge so that you can then build a solution that works for you.

**#7**   **Andre**   said a year ago

Sorry, that probably came out wrong. The not-opinionated fact is what I like about flask. And the fact that I comment here really means that I like how you're writing about and teaching flask ;)

I'm not actually convinced yet that bootstrap-flask is the right fit for what I'm tying to do (it's likely not now that I took a closer look). But similar to your bootstrap_wtf macros it abstracts away some of the messier parts one has to deal with. So I was merely trying to ask if you know about a ready to use set of jinja templates specifically for bootstrap to take that pain away.

I any case, thanks alot for your work and time, much appreciated!

**#8**   **Miguel Grinberg**   said a year ago

@Andre: Any attempts to wrap Bootstrap that existed in the past got out of date pretty quickly, so I do not know of any projects that are current, except maybe Bootstrap-Flask, though it seemed it is already a bit out of date. For that reason I have decided to implement the smallest possible interface to render a form. The more you rely on dependencies that hardest it is to keep your project up to date. My approach is to try to limit dependencies to the strictly necessary.

**#9**   **TheBlack**   said 9 months ago

Is there anything in further tutorial chapters relying on changes made in this chapter? I wondering if it is good idea to skip for now all bootstrap cosmetics and keep focus on learning

backend, postponing all the job on beauty to the very end of this project.

**#10**    **Miguel Grinberg**    said 9 months ago

@TheBlack: up to you, really. All HTML improvements from now on are based on Bootstrap. You can just copy the HTML and keep going if you have no interest in learning this topic.

**#11**    **Ashish M**    said 8 months ago

Due to gmail constraints, the email chapter was challenging to fully implement and test.

I really appreciate how you provide the links for a "reset" at the start of every chapter! It is a huge help, and very thoughtful.

**#12**    **Evan S**    said 6 months ago

Thank you greatly for the tutorial, it's layout and structure has been greatly intuitive, and I've started reaching that stage where I'm comfortable diverging in little way to make minute adjustments.

In the interest of avoiding code repetition where possible, would be best practice to have a "_pagination.html" file that we can include in index.html, user.html, and any other potential sites the application might use it? It seemed intuitive to me to have that feature in a discrete location, especially if we're planning for the future and would like to easily adjust the pagination functionality/design.

**#13**    **Miguel Grinberg**    said 6 months ago

@Evan: Sure, creating a pagination partial template should be possible. I would probably do it as a Jinja macro, however, so that arguments can be passed. You'll probably want some level of customization, such as providing the text to show in the next/prev buttons.

**#14**    **Elizabeth**    said 2 months ago

Hi Miguel!!! I'm using your blog and github as a starting point for my project, thank you so much for creating them. Do you have any examples of Insert and Update forms inside a modal with bootstrap? And how can server-site validation be managed there? I'm having trouble collecting server-side error validation and displaying them in modal without redirecting the page. Best

**#15**   **Miguel Grinberg**   said 2 months ago

@Elizabeth: updating a modal window with validation results is tricky, it can only be done through JavaScript. The server will have to post the form using an asynchronous request. You can still process the form through Flask-WTF, but the resulting list of validation errors will have to be returned to the client as a JSON response. The client will have to use JavaScript logic to insert the error messages in the proper places of the form.

One "thinking outside of the box" option is to use my Turbo-Flask extension and the turbo.js library on the client, which basically convert all synchronous requests issued by the browser into asynchronous.

**#16**   **Sahaj**   said 2 months ago

Hi Miguel Thanks for updating the tutorial,
I am looking at chapter 11 Facelift in the github repo in bootstrap_wtf.html file.

In macro quick_form you are passing a form field with autofocus and rendering field in form_field macro with 1 same class in all the fields,
I have a question here, what if I want to have different class for my fields some might use more than 1 class in some I just want to add my-2
And how can I pass placeholder attribute in those fields?

**#17**   **Miguel Grinberg**   said 2 months ago

@Sahaj: the quick_form macro is a convenience macro that makes a lot of assumptions. If you need to render your fields in a different way, you can always use the render_field macro to render each field individually.

| «« | « | » | »» |

## Leave a Comment

Name

Email

Comment

Captcha

I'm not a robot

reCAPTCHA
Privacy - Terms

Submit

## The Flask Mega-Tutorial          **New 2024 Edition!**



If you would you like to support my work on my Flask Mega-Tutorial series on this blog and as a reward have access to the complete tutorial nicely structured as a book and/or a set of videos, you can now order it from my Courses site or from Amazon.

Click here to get the Book!
Click here to get the Video Course!

## About Miguel

Welcome to my blog!

I'm a software engineer and technical writer, currently living in Drogheda, Ireland.

You can also find me on Twitter, Mastodon, Github, LinkedIn, YouTube, Facebook and Patreon.

Thank you for visiting!

## Categories

- AI *2*
- AWS *1*
- Arduino *7*
- Authentication *10*
- Blog *1*
- C++ *5*
- CSS *1*
- Cloud *10*
- Database *22*
- Docker *5*
- Filmmaking *6*
- Flask *126*
- Games *1*
- Heroku *1*
- IoT *8*
- JavaScript *35*
- MicroPython *9*
- Microdot *1*
- Microservices *2*
- Movie Reviews *5*
- OpenStack *1*
- Personal *3*
- Photography *7*
- Product Reviews *2*
- Programming *187*
- Project Management *1*
- Python *169*
- REST *7*
- Rackspace *1*
- Raspberry Pi *8*
- React *18*
- Robotics *6*
- Security *12*
- Video *22*
- WebSocket *2*
- Webcast *3*
- Windows *1*