# Building a blog

Estimated Time: 2-3 hours

In this assignment you will be extending your base from the previous assignment to actually create the blog engine. By the end of the assignment you should be able to add and view blog entries.

## Creating the model

The first step is to create a SQLAlchemy model which you'll use to store and retrieve blog entries. In *database.py* add the following code:

```
import datetime

from sqlalchemy import Column, Integer, String, Text, DateTime

class Entry(Base):
    __tablename__ = "entries"

    id = Column(Integer, primary_key=True)
    title = Column(String(1024))
    content = Column(Text)
    datetime = Column(DateTime, default=datetime.datetime.now)

Base.metadata.create_all(engine)
```

This should all be familiar from Unit 3. You create a new class which inherits from the declarative base object. Give the model a table name, and add a series of columns. These store a primary key id, the title of the entry, the entry content, and the date and time at which the entry was created. Then use the `Base.metadata.create_all` function to construct the table in the database.

## Adding some example data

It would be nice to have a simple way to add some example entries to the database so you could test the application easily. Add a task to your manager which will generate a series of entries. Add the following code to the *manage.py* file:

```
from blog.database import session, Entry

@manager.command
def seed():
    content = """Lorem ipsum dolor sit amet, consectetur adipisicing e

    for i in range(25):
        entry = Entry(
            title="Test Entry #{}".format(i),
            content=content
        )
        session.add(entry)
    session.commit()
```

Here you create a command called `seed` which will add a series of entries to the database. In the `seed` function you create a string of dummy text for the entry content. You then run a loop 25 times. In the loop you create a new entry and add it to the session. Finally you use the `session.commit` function to synchronize our changes with the database.

Run the command using `python manage.py seed`. You should now have a series of entries in the database which you can use to test our future work.

# Displaying the entries

Now that you have some entries in the database you can start to build the views and templates to allow you to display them.

First, set up a simple view in *views.py* designed so that you can see all of the entries:

```
from flask import render_template

from . import app
from .database import session, Entry

@app.route("/")
def entries():
    entries = session.query(Entry)
    entries = entries.order_by(Entry.datetime.desc())
    entries = entries.all()
    return render_template("entries.html",
        entries=entries
    )
```

Here you construct a query of `Entry` objects. You order the entries by the `datetime` column, getting the most recent ones first. Then you use the `entries.all` method to retrieve all of the results. Finally you render a template called *entries.html*, passing in the list of entries.

Now you have the view sorted let's set up the templates. First, you'll make a base template which uses Bootstrap to provide some basic styling and UI components. In the *templates* folder create a new file called *base.html*:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8" />
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <meta name="viewport" content="width=device-width, initial-sca
        <title>Blogful</title>

        <!-- CSS -->
        <!-- Bootstrap -->
        <link rel="stylesheet" href="//netdna.bootstrapcdn.com/bootstr
        <!-- Blog -->
        <link rel="stylesheet" href="{{ url_for('static', filename='cs

        <!-- JavaScript -->
        <!-- jQuery -->
        <script src="http://code.jquery.com/jquery-2.1.1.min.js"></scr
        <!-- Bootstrap -->
        <script src="//netdna.bootstrapcdn.com/bootstrap/3.1.1/js/boot
    </head>

    <body>
        <div class="navbar navbar-inverse navbar-static-top" role="nav
            <div class="container">
                <div class="navbar-header">
                    <button type="button" class="navbar-toggle" data-t
                        <span class="sr-only">Toggle navigation</span>
                        <span class="icon-bar"></span>
                        <span class="icon-bar"></span>
                        <span class="icon-bar"></span>
                    </button>
                    <a class="navbar-brand" href="{{ url_for('entries'
                </div>
                <div class="collapse navbar-collapse">
                    <ul class="nav navbar-nav navbar-right">
                    </ul>
                </div>
            </div>
        </div>

        <div class="container">
            {% block content %}{% endblock %}
        </div>
    </body>
</html>
```

This is a fairly standard Bootstrap template. You'll have a static navbar at the top, and a content block below that. You can see that there's a css file called *main.css* which is served from the *static/css* directory. Let's quickly add in a css rule to this file which will add some basic styling to our entries:

```
div.metadata {
    margin-top: 20px;
    font-size: 1.5em;
}
```

Next you'll need to render the entries. Because there are a number of places where you might want to render an entry you will separate this code into a Jinja macro. In a file called *macros.html* in the *templates* directory, add the following code:

```
{% macro render_entry(entry) %}
<div class="row">
    <div class="col-md-2 text-right metadata">
        <ul class="list-unstyled">
            <li>
                {{ entry.datetime | dateformat("%d/%m/%y") }}
            </li>
        </ul>
    </div>
    <div class="col-md-10">
        <h1>
            {{ entry.title }}
        </h1>
        {{ entry.content | markdown }}
    </div>
</div>
{% endmacro %}
```

Here you declare a macro called `render_entry` which takes an entry as a single argument. When this is called, you render a narrow metadata column containing the entry date, and a wider body column which contains the entry title and content.

There are a couple of things to note here. You're using the `markdown` filter on the entry content. This allows you to use the Markdown syntax, making it simpler to write well-formatted blog entries. You use the `dateformat` filter to display the datetime in *dd/mm/yy* format. Jinja doesn't include these filters by default, so you need to add this to the *filters.py* file:

```
from . import app
from flask import Markup
import mistune as md

@app.template_filter()
def markdown(text):
    return Markup(md.markdown(text,escape=True))

@app.template_filter()
def dateformat(date, format):
    if not date:
        return None
    return date.strftime(format)
```

Here you're creating a function which takes two arguments: the date which is piped in from the template, and a format string which you provide as an argument. Check to make sure that you have a date object, then you use the `strftime` method to format the date correctly.

The final step towards viewing the entries is to actually add the *entries.html* template:

```
{% import "macros.html" as macros %}
{% extends "base.html" %}
{% block content %}

{% for entry in entries %}
{{ macros.render_entry(entry) }}
{% endfor %}

{% endblock %}
```

First you import the *macros.html* file. Then specify that the template should inherit from *base.html*. Finally in the `content` block you loop through the entries, rendering each one using the `render_entry` macro.

Try running your app using `python manage.py run` , then visit the site. You should see all 25 of your test entries listed, starting with the most recent.

Once you've confirmed that your new feature works correctly, you can merge your changes back into master.

## Adding pagination

Although the view works fine for 25 entries, what about if you have thousands? It is unlikely that anyone will need to see all of these entries at once, and it will make rendering far slower. So try modifying the view slightly to introduce pagination.

```python
PAGINATE_BY = 10

@app.route("/")
@app.route("/page/<int:page>")
def entries(page=1):
    # Zero-indexed page
    page_index = page - 1

    count = session.query(Entry).count()

    start = page_index * PAGINATE_BY
    end = start + PAGINATE_BY

    total_pages = (count - 1) // PAGINATE_BY + 1
    has_next = page_index < total_pages - 1
    has_prev = page_index > 0

    entries = session.query(Entry)
    entries = entries.order_by(Entry.datetime.desc())
    entries = entries[start:end]

    return render_template("entries.html",
        entries=entries,
        has_next=has_next,
        has_prev=has_prev,
        page=page,
        total_pages=total_pages
    )
```

Here you create a new route, `/page/<int:page>` designed to take you to a specific page of content. You add an argument to your route for the page number, `page`, and a module constant `PAGINATE_BY`, which indicates how many items should be on each page. (Python doesn't enforce anything about these, but by convention, an ALL_UPPERCASE_NAME is a constant.) You then use the `count` method of a query object to find out how many entries there are in total. You can use all of this data to calculate a number of pieces of information about the pagination:

- `start` - The index of the first item that you should see
- `end` - The index of the last item that you should see
- `total_pages` - The total number of pages of content
- `has_next` - Whether there is a page after the current one
- `has_prev` - Whether there is a page before the current one

Then change the query so that rather than finding all entries it slices the query so you only find the entries between the start and end indices. Finally you pass all of this information into the template.

Try visiting your app again. You should the most recent 10 items only. If you then visit `/page/2` you should see the next 10 items. But typing in the URL is not a particularly nice user interface. Adjust your *entries.html* template to display some buttons which will direct you to the next and previous pages:

```
{% import "macros.html" as macros %}
{% extends "base.html" %}
{% block content %}

{% for entry in entries %}
{{ macros.render_entry(entry) }}
{% endfor %}
```

```
<ul class="pager">
    {% if has_next %}
    <li class="previous" >
    <a href="{{ url_for('entries', page=page + 1) }}">&larr; Older</a>
    </li>
    {% endif %}
    {% if has_prev %}
    <li class="next">
    <a href="{{ url_for('entries', page=page - 1) }}">&rarr; Newer</a>
    </li>
    {% endif %}
</ul>

{% endblock %}
```

If the `has_next` variable is true you'll add a link to the next page using Flask's `url_for` function to generate the URL for the route. Similarly if the `has_prev` variable is true you'll render a link to the previous page.

## Adding blog entries

Next you can start to think about how to add new entries without needing to use the command line. The most common way to get user input in web apps is through HTML forms. So let's create a form in a new template called `add_entry.html` :

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

{% block content %}
<h1>
    Add Entry
</h1>

<form role="form" method="POST">
    <div class="form-group">
        <label for="title">Title</label>
        <input type="text" class="form-control" id="title" name="title
    </div>
    <div class="form-group">
        <label for="content">Content</label>
        <textarea class="form-control" id="content" name="content" pla
    </div>
    <button type="submit" class="btn btn-default">Submit</button>
</form>
{% endblock %}
```

There are a couple of things to notice here. The form has the `method` attribute set to `POST` . This means that when you submit the form it will send the data via a POST request. As the form element does not have an `action` attribute the request will be sent to the same URL as you used to GET the form.

Inside the form you have two fields, one for the title and one for the content. Notice how both of these fields have the `name` attribute set. You'll use this name to access the contents of the fields. Finally you have a `submit` button which you will use to submit the form.

Now, create a view in *views.py* so that you can display the form:

```
@app.route("/entry/add", methods=["GET"])
def add_entry_get():
    return render_template("add_entry.html")
```

Notice how you use the `methods=["GET"]` parameter in the `route` decorator. This specifies that the route will only be used for GET requests to the page; you will have to add a new view for the POST request which takes place when you submit the form.

Try visiting the `/entry/add` page of your app. You should see the form for adding an entry. If you try to submit the form you should see an error saying that the route does not accept POST requests. Add a new route which will take your form data and create the new entry.

```
from flask import request, redirect, url_for
```

```
@app.route("/entry/add", methods=["POST"])
def add_entry_post():
    entry = Entry(
        title=request.form["title"],
        content=request.form["content"],
    )
    session.add(entry)
    session.commit()
    return redirect(url_for("entries"))
```

Here you create an similar route to your `add_entry_get` view, except this one only accepts POST requests. Inside the function you create a new `Entry` object. You use Flask's `request.form` dictionary to access the data submitted with your form and assign it to the correct fields in the entry.

Next you add the entry to your session, and commit it to the database. Finally you use Flask's `redirect` function to send the user back to the front page once their entry has been created.

Now try visiting the `/entry/add` page again and submitting a new entry. You should see that the entry is added correctly and is now at the top of your page of most recent entries.

---

### How would you generate a link tag which points to a static stylesheet in a Jinja template?

> *You can use the* `url_for` *function to generate the href tag. For example* `<link rel="stylesheet" href="{{ url_for('static', filename='css/main.css') }}">` *would link to the file static/css/main.css.*

### How do we access data which a user has submitted through a form?

> *You can access the data using the* `request.form` *dictionary.*

### What does the `Flask.template_filter` method do?

> *It wraps a function to add a custom filter to Flask's Jinja environment. It is most commonly used as a decorator.*

### What is pagination? What are some advantages and disadvantages of using a pagination scheme?

> *Pagination is the splitting of list of items into multiple pages of content. It allows data to be retrieved in manageable chunks, speeding up page loading times and preventing the user from being overwhelmed with data. In some situations this may lead to a less seamless user experience, with the user constantly having to click between multiple pages of content.*

---

☆ ☆ ☆ ☆ ☆ · Report a typo or other issue

✓ Mark as completed

Previous Next