Learning Objectives

- **Use built-in filters in a Django template**
- **Create custom template tags for the posts**
- **Create a custom filter and add it to a template**
- **Render text safe to increase the security of the blog**
- **Pass an argument to a filter**

Clone Blango Repo

Clone Blango Repo

Before we continue, you need to clone the `blango` repo so you have all of your code. You will need the SSH information for your repo.

In the Terminal

- Clone the repo. Your command should look something like this:

```
git clone git@github.com:<your_github_username>/blango.git
```

- You should see a `blango` directory appear in the file tree.

You are now ready for the next assignment.

Custom Filters

Custom Filters

In Django templates, you can pass data through filters before it's rendered. This is done with the `|` (pipe) operator inside a rendering block. For example, to render a value in lowercase, using the `lower` filter, you'd do this:

```
{{ value|lower }}
```

Some filters accept arguments, which can be sent by adding `:` (colon) after the filter name. For example, the `date` filter takes an argument that specifies the output format:

```
{{ post.published_at|date:"M, d Y" }}
```

Django has an extensive set of built-in filters, but if none of them quite do the job you need, you can write your own. These can contain more complex logic, which can be easier than trying to write lots of custom code inside your template.

Before we get into that, let's add a real page to Blango. It will be a page that lists all the blog posts we have.

First, we'll update the `index` view to fetch all the `Post` objects in the system, and send them to the `index.html` template. This should be functionality you've used with Django before. Your view will look something like:

```
def index(request):
    posts =
        Post.objects.filter(published_at__lte=timezone.now())
    return render(request, "blog/index.html", {"posts": posts})
```

Note the use of the `published_at__lte=timezone.now()` filter. This means we'll only load `Post` objects that have been published (have a publication date in the past).

Also, be sure to import `timezone` from `django.utils` and `Post` from `blog.models`, at the start of the file:

```
from django.utils import timezone
from blog.models import Post
```

Next we need to render the post data in the template (`index.html`). We’ll just use a simple loop and render each blog `Post` in a Bootstrap `row`. You can copy and paste this to replace the `content` block in your template:

Open index.html

```
{% block content %}
    <h2>Blog Posts</h2>
    {% for post in posts %}
    <div class="row">
        <div class="col">
            <h3>{{ post.title }}</h3>
            <small>By {{ post.author }} on {{
        post.published_at|date:"M, d Y" }}</small>
            <p>{{ post.summary }}</p>
            <p>
                ({{ post.content|wordcount }} words)
                <a href="#">Read More</a>
            </p>
        </div>
    </div>
    {% endfor %}
{% endblock %}
```

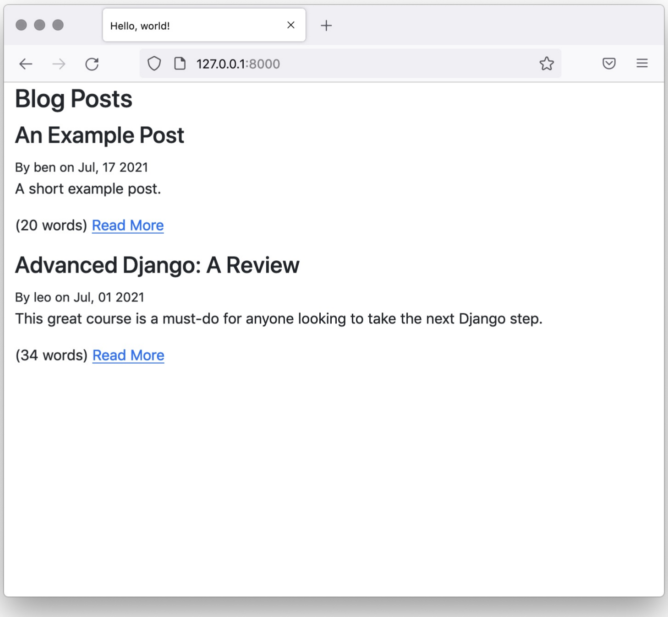
You can see we’re just showing the post’s `title`, `author`, `published_at` date, `summary` and a count of the words in its `content`

You should notice that we’re using two built in filters:
- The `date` filter which outputs the date in *month, day year* format (its argument).
- The `wordcount` filter to count the words in the `Post`’s content. Note that this is not strictly correct as we’re storing raw HTML in the `content` field and thus the count will include the tags and attributes; but for illustrative purposes it is fine.

If you don’t already have some test Blog data added, you should log to the Django admin and add a couple of posts, and another user as an extra author. This will come in handy later.

Provided you’ve done this and have some test data, when you start the Django dev server and navigate to the main page you should see something like this:

View Blog



Blog Posts

An Example Post

By ben on Jul, 17 2021

A short example post.

(20 words) Read More

Advanced Django: A Review

By leo on Jul, 01 2021

This great course is a must-do for anyone looking to take the next Django step.

(34 words) Read More

Blog post list

Next we'll look at how to set up the files that contain the template tags and filters.

Template Tag File Set Up

You've probably loaded custom template tag libraries into a template before using the build in `load` template tag. More than likely, you've loaded the `static` template tag library like this:

```
{% load static %}
```

Django looks for template libraries to load inside Python files in the `templatetags` folder inside Django apps. The template library it loads is simply the name of the file.

For example, in Blango we'll create a template tag library with the path `blog/templatetags/blog_extras.py`. We would load it into the template like this:

```
{% load blog_extras %}
```

Note that the convention is to name template tag files in the format * [something]_extras.py*, but this is not mandatory. You just need to be sure that your file won't conflict with any of the built in template libraries or third parties that you might have included. Template tag files aren't namespaced so `blog_extras` could be used by any app in our project.

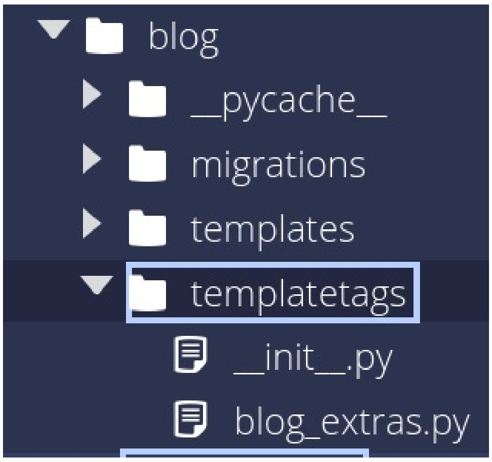
Try It Out

Let's start by setting up the directory structure for the custom template tags. You'll need to create the following things:

- A directory named `templatetags` inside the `blog` app directory.
- An empty `__init__.py` file inside the `templatetags` directory.
- An empty `blog_extras.py` file, also inside the `templatetags` directory.

▼ **Solution**

After adding the new directory and files, your file tree should look like this:



Template Tags Directory Structure

You can leave this file open as we'll be working with it straight away.

A Simple Filter

A Simple Filter

In its simplest form, a filter is just a function that take a single argument and returns a string to be rendered in the template.

In our Blango `Post` list we’re just showing the auther’s username `{{ post.author }}`. However, we want to show some more details if available. That is, the author’s first and/or last name instead. We’ll call this filter `author_details`.

Try It Out

You can try writing the function called `author_details`.
* It should take a single function, the author of the `Post` (which is a `django.contrib.auth.models.User` object).
* If the author has a `first_name` and a `last_name` set, then it should return a string in the format “{first_name} {last_name}”.
* Otherwise it should return the author’s username. It can also be useful to check that a `User` object has been passed to the function and return an empty string if not – this is a failsafe default.

▼ **Solution**

Your `blog_extras.py` file should now look something like this:

```
from django.contrib.auth import get_user_model
user_model = get_user_model()


def author_details(author):
    if not isinstance(author, user_model):
        # return empty string as safe default
        return ""

    if author.first_name and author.last_name:
        name = f"{author.first_name} {author.last_name}"
    else:
        name = f"{author.username}"

    return name
```

Note the use of `isinstance` to check that we're working with a `User` object, just in case someone has passed the wrong type of variable. In our case, it's not likely to happen as the `author` of a `Post` can't be anything but a `User`. But it's a good habit to do checks like this in filters, just in case a variables happens to be the wrong type – perhaps `None` when you don't expect it. An exception raised in filter code will cause an error to be returned to the end user.

Before the filter can be used, it needs to be registered into the template library. This is actually a three step process:

1. Import the `django template` module.
2. Create and instance of the `django.template.Library` class.
3. Register the filter function into the `Library` with its `filter` function.

(Steps 1 and 2 only need to be done once per template tag file).

Let's do this now:

- Add the import to the top of your `blog_extras.py` file:

▼ **Solution**

Import `template` like this:

```
from django import template
```

- Create the `template.Library` instance. Convention is to call this variable `register`, as it makes it more clear what its methods do when you use them. It's a global variable in the file, and should come before the filters.

▼ **Solution**

Instantiate the `register` variable like this:

```
register = template.Library()
```

- Filters must be registered, so that any other non-filter functions in the file don't get picked up by Django. There are couple of different ways to use the `Library.filter` function, which you can read about in the official documentation, but we're going to go with the simplest method: use it as a decorator

▼ **Solution**

Add the decorator to the line before the function definiton like this:

```
@register.filter
def author_details(author):
    # existing function body
```

As you can see, since we called the `Library` instance `register`, it makes it clear that the decorator is *registering* a *filter*.

The name of the filter in the template is automatically made the same as the name of the function, but this can be customized by passing a `name` argument to `register.filter`. For example, `@register.filter(name="author_details")`. We don't need to do this as we're happy with the `author_details` name.

Now let's use it in our template. Open the `index.html` template and load the `blog_extras` template library using the `load` tag.

Open index.html

Load the `blog_extras` template library.

- Add `blog_extras` in `index.html`.

▼ **Solution**

Add `{% load blog_extras %}` after the `extends` template tag in `index.html`.

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

- Update the byline text so it includes the `author_details` filter.

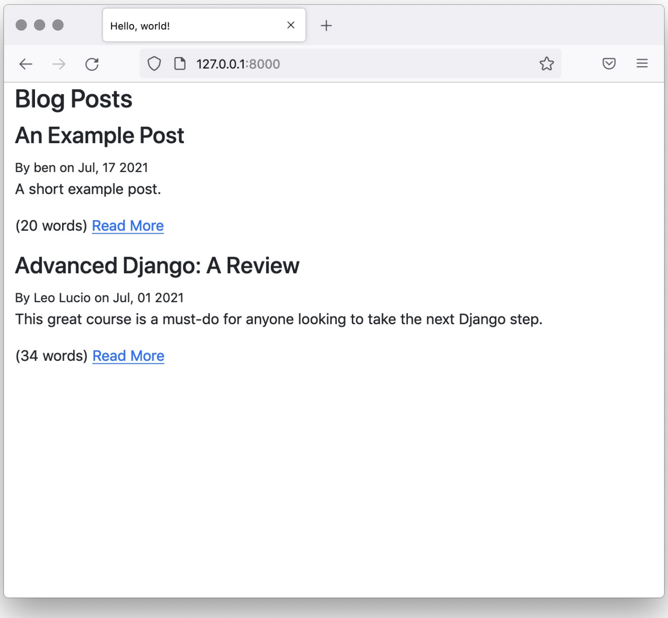
▼ **Solution**

Updating the byline text:

```
<small>By {{ post.author|author_details }}
```

Refresh the page in your browser and you should see the author's first and last name instead of their username, if set. Make sure you've set a first and last name for one of your users in Django admin though!

View Blog



Blog Posts

An Example Post

By ben on Jul, 17 2021

A short example post.

(20 words) Read More

Advanced Django: A Review

By Leo Lucio on Jul, 01 2021

This great course is a must-do for anyone looking to take the next Django step.

(34 words) Read More

Author Name Filter

Next we’ll see how Django handles HTML with filters.

Safe Text

Safe Text

Django is very safe and secure by default. Any variables that are rendered automatically have their HTML entities escaped, to prevent malicious HTML being injected into templates unexpectedly.

For example, the string `<a href="#">Click me!</a>` would be rendered as:

```
&lt;a href=&quot;#&quot;&gt;Click me!&lt;/a&gt;
```

This prevents Cross Site Scripting (XSS) and other types of attacks to your site, in which visitors can be redirected to malware or have the page content changed unexpectedly, due to bad HTML being supplied by users.

What if we need to use HTML entities in our output value? There are a couple of ways. Django has the built in filter `safe` which skips the encoding, if you know a string is safe to output verbatim. Generally you'd only want to use this with your own values, and not mark any user-supplied data as "safe".

The other way is to use a "safe" string. This is a special string class that is not escaped when rendered. We'll look at how to mark normal strings as safe soon.

But, what if you need to output a mix of safe HTML that you've defined, and unsafe data from a user? This discussion is getting a bit abstract and theoretical, so let's look at a real problem in the context of Blango.

We want to have the author's name be a clickable link to email them, if they have an email address in their profile. For reference, this is a `mailto` link, like this:

```html
<a href="mailto:ben@example.com">ben</a>
```

To achieve this we need to update the `author_details` function to work with a variety of combinations of user data. We should wrap the user's name (either their username or full name) in an `<a>` tag only if they have an email address. The output string will be composed of both safe data that we can control (namely the HTML that we write) and unsafe data that the user has provided (their name and email address). We'll be building up this functionality a step at a time to cement the concepts.

This time we'll be building up the clickable link to email. First let's take a naïve approach at updating the `author_details` function just to return the HTML. While this won't give us a clickable link, it's good to see Django's safe-by-default approach.

▼ **Email Addresses**

Login to the Django admin page to verify that users have created an email address for their profile. If not add one. Without an email address, you will not be able to see the changes being made to your blog.

Try It Out

Modify the `author_details` function so that the author name `ben` becomes:

```html
<a href="mailto:ben@example.com">ben</a>
```

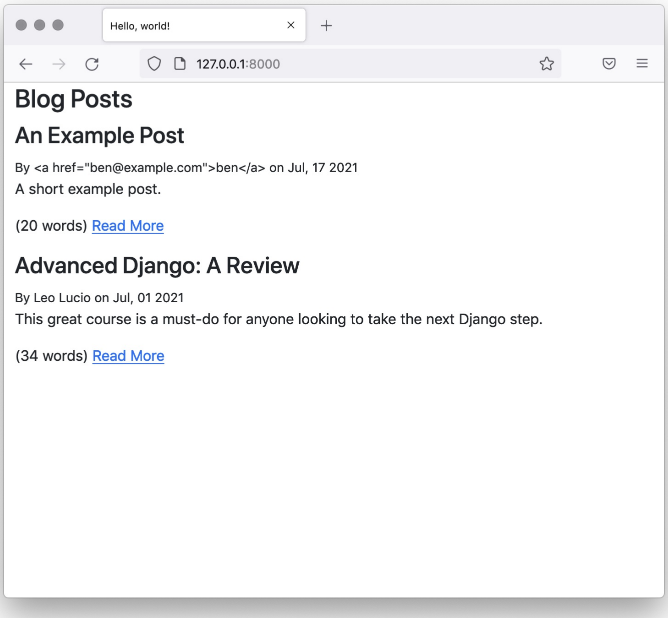
▼ **Solution**

Even the code you wrote works, you should copy and paste this to replace it as we'll be adding to it piece by piece. Replace your `author_details` function with this code:

```python
@register.filter
def author_details(author):
    if not isinstance(author, user_model):
        # return empty string as safe default
        return ""

    if author.first_name and author.last_name:
        name = f"{author.first_name} {author.last_name}"
    else:
        name = f"{author.username}"

    if author.email:
        email = author.email
        prefix = f'<a href="mailto:{email}">'
        suffix = "</a>"
    else:
        prefix = ""
        suffix = ""

    return f"{prefix}{name}{suffix}"
```

Start the dev server; in your browser you'll see something like this:

View Blog



Blog Posts

An Example Post

By <a href="ben@example.com">ben</a> on Jul, 17 2021
A short example post.

(20 words) Read More

Advanced Django: A Review

By Leo Lucio on Jul, 01 2021
This great course is a must-do for anyone looking to take the next Django step.

(34 words) Read More

Escaped Email Link

One way to “fix” this might be to use the `safe` filter in our template, doing something like `{{ post.author|author_details|safe }}`. This would give us a clickable link, but is definitely **NOT** safe! Because we’re marking the whole string as safe, we’d also be outputting any malicious HTML that could have been injected into the user’s name or email address.

To have good output that’s *really* safe, we need to first escape the dangerous parts of the text that we’re dealing with. Then, mark the whole string as safe so that only the HTML we trust is output verbatim.

Escaping strings is done with the `django.utils.html.escape` function. This will encode the HTML entities in a string. Once we have escaped only the dangerous values, we can mark a string as safe with the `django.utils.safestring.mark_safe` function. Once a string has been marked as safe, Django won’t escape it again when outputting in a template, so use with caution.

Try It Out

Let’s update the `author_details` function to make its output both clickable and safe. See if you can make the changes before checking the provided code:

- Import the `django.utils.html.escape` function and use it to escape the dangerous user supplied data (name, username and email).

▼ Solution

Here’s the solution; first add these imports at the start of the file:

```
from django.utils.html import escape
from django.utils.safestring import mark_safe
```

- Import the `django.utils.safestring.mark_safe` function and mark the final string safe.

▼ Solution

Then update your filter function:

```
@register.filter
def author_details(author):
    if not isinstance(author, user_model):
        # return empty string as safe default
        return ""

    if author.first_name and author.last_name:
        name = escape(f"{author.first_name} {author.last_name}")
    else:
        name = escape(f"{author.username}")

    if author.email:
        email = escape(author.email)
        prefix = f'<a href="mailto:{email}">'
        suffix = "</a>"
    else:
        prefix = ""
        suffix = ""

    return mark_safe(f"{prefix}{name}{suffix}")
```

Now we know the whole string is safe to be marked as safe, because it’s composed only of HTML that we’ve defined, and all user supplied data is escaped.

Try it out by refreshing the post list. Any users with email addresses should now be clickable.

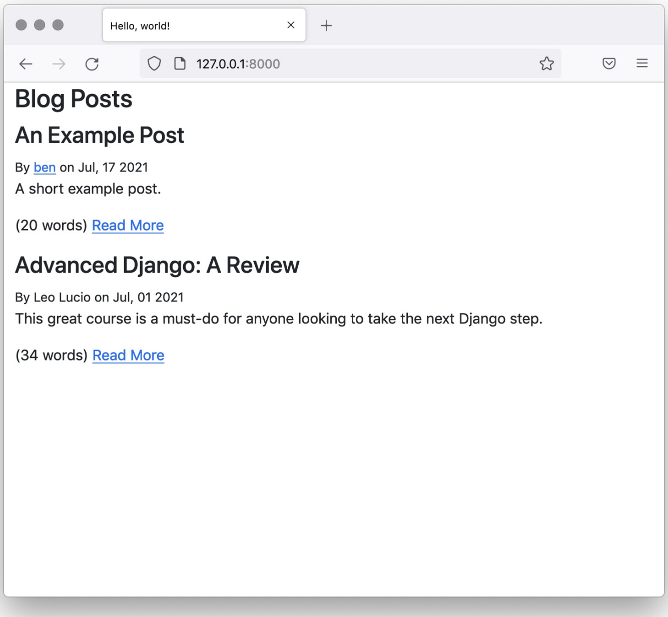
▼ Refresh the Website

Click on the blue, circular arrows to refresh the website.

blue, circular arrows

If you closed those tabs, you can start the dev server again.

View Blog



Blog Posts

An Example Post

By ben on Jul, 17 2021

A short example post.

(20 words) Read More

Advanced Django: A Review

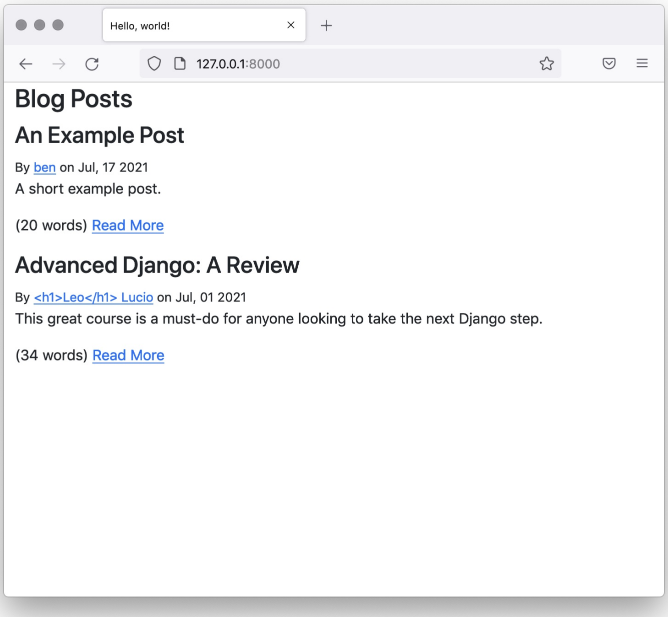
By Leo Lucio on Jul, 01 2021

This great course is a must-do for anyone looking to take the next Django step.

(34 words) Read More

Clickable Email Link

If you really want to be sure that the output is safe, try changing one of the user’s first names to a small piece of HTML, and refreshing the post list. You should see that the HTML is escaped and thus readable, rather than being rendered verbatim and affecting the page’s markup.



Blog Posts

An Example Post

By ben on Jul, 17 2021

A short example post.

(20 words) Read More

Advanced Django: A Review

By <h1>Leo</h1> Lucio on Jul, 01 2021

This great course is a must-do for anyone looking to take the next Django step.

(34 words) Read More

Clickable Escaped Email Link

Now that we've had a look at what's happening at a low level, we can introduce a helper function that adds more security. The `django.utils.html.format_html` is the preferred way of building HTML strings as it escapes values and marks strings safe in a single step. For example, when we build the opening `<a>` tag for the email link, we are escaping the email address on the line before and then putting it in the string. If we forgot to do the escape step, it could be dangerous.

So instead of creating the prefix like this:

```
email = escape(author.email)
prefix = f'<a href="mailto:{email}">'
```

We can do it in a single step:

```
prefix = format_html('<a href="mailto:{}">', author.email)
```

`format_html` works similarly to the built in `str.format` method, except each argument is automatically escaped before being interpolated.

Let's update our `author_details` function to use it now. First we can remove our `escape` and `mark_safe` imports as `format_html` does this for us. Then, we can import `format_html instead`:

```
from django.utils.html import format_html
```

Then we'll use it in our filter function:

```python
@register.filter
def author_details(author):
    if not isinstance(author, user_model):
        # return empty string as safe default
        return ""

    if author.first_name and author.last_name:
        name = f"{author.first_name} {author.last_name}"
    else:
        name = f"{author.username}"

    if author.email:
        prefix = format_html('<a href="mailto:{}">',
        author.email)
        suffix = format_html("</a>")
    else:
        prefix = ""
        suffix = ""

    return format_html('{}{}{}', prefix, name, suffix)
```

info

Check Your Code

To check the clickable email link, you must first open your blog in a new tab. Click the blue arrow (shown below). Then click on the author of the blog post. Your computer should start the email process.

blue, circular arrows

If you closed those tabs, you can start the dev server again.

View Blog

`format_html` can be called with just one argument (like creating the closing `</a>`) tag, in which case the string is just marked as safe. At the end of the function, we're passing in already escaped and safe parameters, `prefix` and `suffix`. Since these are already marked as safe, they won't be re-escaped.

To summarize this section, as the security aspect is very important:

- By default, Django will escape all output in templates before rendering, so it's secure against XSS and other injection attacks automatically.
- Don't use the `safe` filter or mark a string as safe (with `mark_safe`) unless you've constructed all of that string yourself. A string that is composed of any form of use input should be considered unsafe.
- Strings should only be marked safe if any parts that have come from a user have been passed through the `escape` function.
- To be safer and use less code, use the `format_html` function, which escapes strings parameters before interpolation.

We're going to close this section by looking at filters with arguments.

Filters with Arguments

Filters with Arguments

Filter functions can also take a single argument. We've seen this with the `date` filter, which takes a format string. Adding an argument to a filter is simple: just add an argument to the filter function, and then pass in the argument in the template, with the `:` operator.

Let's update `author_details` to return the string `<strong>me</strong>` if a `Post` is authored by the currently logged in user.

Try It Out

You should have all the knowledge to complete this yourself, hint: the current user is accessible in the template with the `request.user` variable. Give it a try yourself, and if you get stuck, come back here.

Here's how you do it, first the update to the `author_details` function.

```
@register.filter
def author_details(author, current_user):
    if not isinstance(author, user_model):
        # return empty string as safe default
        return ""

    if author == current_user:
        return format_html("<strong>me</strong>")

    if author.first_name and author.last_name:
        name = f"{author.first_name} {author.last_name}"
    else:
        name = f"{author.username}"

    if author.email:
        prefix = format_html('<a href="mailto:{}">',
author.email)
        suffix = format_html("</a>")
    else:
        prefix = ""
        suffix = ""

    return format_html('{}{}{}', prefix, name, suffix)
```

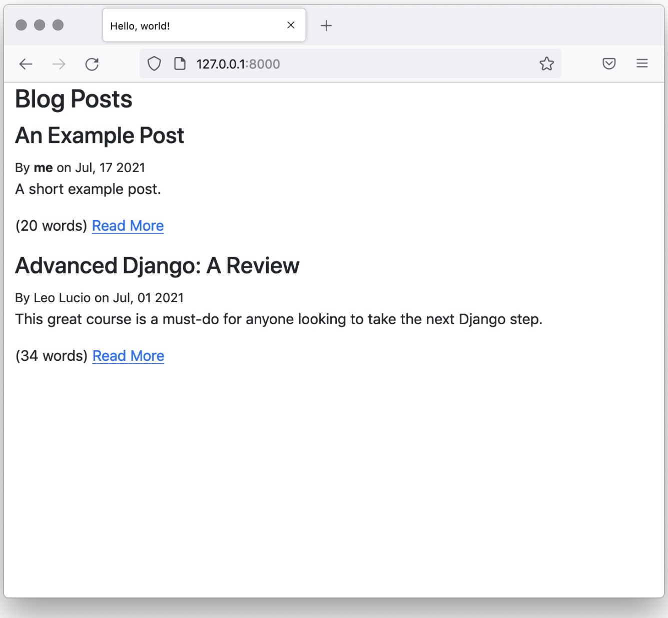
Remember to mark the return string as safe using the `format_html` function, so the `<strong>` element is rendered properly in HTML.

Then update your `index.html` file to pass `request.user` to the filter:

```
<small>By {{ post.author|author_details:request.user }} on {{
        post.published_at|date:"M, d Y" }}</small>
```

Refresh the post list and you page should look like this (provided you're logged in as one of the authors, which you can do through the Django admin section. The login to the admin page carries through to the rest of the site):

View Blog



Blog Posts

An Example Post

By **me** on Jul, 17 2021

A short example post.

(20 words) Read More

Advanced Django: A Review

By Leo Lucio on Jul, 01 2021

This great course is a must-do for anyone looking to take the next Django step.

(34 words) Read More

Current User is Author

Note that you could also provided a default value to `current_user` in the function definition, e.g:

```
def author_details(author, current_user=None):
```

Then, your filter would be callable with our without any arguments. You could use this to provide default values to a filter but still make its behavior customizable.

A Post Script on HTML in Template Tags

One of the key features of Django is its template system. From the start, it has been designed to allow multiple people with different skills to contribute to a Django project. Those who have HTML, CSS and other UX skills can work on the templates in HTML, while back-end Python developers can build the views and models. When you start generating HTML in template filter, you remove the ability for this type of separation of concerns.

Another thing to consider is that it can be harder to locate where HTML is being generated, if it's not in a template and instead is being generated with a template tag.

That's not to say you shouldn't do it, but keep these drawbacks in mind before you do. One advantage is that it can cutdown on many lines of code and logic in a template, which can arguably be harder do read. For comparison, our `author_details` code in a template would look something like this:

```
<small>By
    {% if post.author == request.user %}
        <strong>me</strong>
    {% else %}
        {% if post.author.email %}
            <a href="mailto:{{ post.author.email }}">
        {% endif %}
            {% if post.author.first_name and
        post.author.last_name %}
                {{ post.author.first_name }} {{
        post.author.last_name }}
            {% else %}
                {{ post.author.username }}
            {% endif %}
        {% if post.author.email %}
            </a>
        {% endif %}
    {% endif %}
    on {{ post.published_at|date:"M, d Y" }}
</small>
```

It's a few less lines of code but harder to read. The piece that particularly stands out to the author is the use of `{% if post.author.email %}` twice, once for the opening and once for the closing `<a>` tag. If you have more HTML in between these `if` blocks it can get hard to keep track of the opening and closing tags when they're always wrapped up in conditionals.

As with so many things in programming though, it depends on your particular situation, personal prefererence and company/individual needs. Those were just some points to consider when making your decision.

Next we’re going to look at custom template tags, which can be used when filters aren’t quite flexible enough to get the job done for you.

Pushing to GitHub

Pushing to GitHub

Before continuing, you must push your work to GitHub. In the terminal:

- Commit your changes:

```
git add .
git commit -m "Finish Introduction and Django Admin"
```

- Push to GitHub:

```
git push
```