

Handling Forms in Flask with Flask-WTF

👤 Todd 📁 Flask 👁 9 Min Read



- XI. Managing Session Data with Flask-Session & Redis
- X. Handle User Accounts & Authentication in Flask with Flask-Login
- IX. Connect Flask to a Database with Flask-SQLAlchemy
- VIII. Compiling and Serving Frontend Assets in Flask
- VII. Organizing Flask Apps with Blueprints
- VI. Demystifying Flask's Application Factory
- V. Configuring Your Flask App
- IV. The Art of Routing in Flask
- III. Handling Forms in Flask with Flask-WTF
- II. Rendering Pages in Flask Using Jinja
- I. Creating Your First Flask Application

Let's talk about something that everybody hates: forms. The only thing more painful than filling out a form is creating one, much less a functional one with feedback. If the thought

of building functional forms doesn't sicken you, you're probably into some freaky shit. Call me.

If you don't have a pressing need to create forms in Flask, I won't be offended if you decide to ditch this post. Feel free to scroll endlessly through Instagram, but know this: handling form authentication and data submission is the *pinnacle of app development*. This is where frontend meets backend: a dance of data where a person's keystrokes result in meaningful action. He who creates forms is a harbinger of a golden age: a hero who brings us to the pinnacle of Western technology. Then again, there's always Instagram.

Both Flask and Django have relied heavily on a single Python library by the name of [WTForms](#). WTForms has strangely remained the undisputed library for handling forms for years without consequence: the library is easy to use, and nobody has maliciously hijacked the library to steal everybody's personal information or Bitcoin. That's the real difference between the Python and Javascript ecosystems. The flavor of WTForms is actually a Flask plugin called [Flask-WTF](#), which provides a few nice perks for Flask users. Truthfully, Flask-WTF isn't far from the original library at all (this tutorial had actually originally used WTForms instead of Flask-WTF, and the source code somehow remained almost exactly the same).

Laying the Groundwork

I know you're lazy so copy+paste the line below so we can get on with it:

```
$ pip3 install flask flask-wtf
```

Before laying down some code snippets for you to mindlessly copy+paste, it helps to understand what we're about to throw down. Here's a look at our structure before anybody gets lost:

```
/flask-wtforms-tutorial
├ /templates
├ /static
├ app.py
├ config.py
└ forms.py
```

Creating a functioning form has a minimum of three parts:

1. **Routes** for determining what users should be served when visiting a URL within our app. We're going to keep things simple by defining our routes in a single `app.py` file.

2. **Form classes** are Python models that determine the data our forms will capture, as well as the logic for validating whether or not a user has adequately completed a form when they attempt to submit. These classes are going to live in `forms.py`.
3. **Jinja templates** will render the actual HTML forms that users will see. As we'll soon discover, Flask makes bridging the gap between Python models and HTML forms easy.

We'll start off by creating our form logic in `forms.py`.

What The Form

Flask-WTF comes packaged with **WTForms** as a dependency: the two libraries are intended to be used together, as opposed to one obfuscating the other. The core of Flask-WTF is a class called `FlaskForm`, which we extend with the form classes we define ourselves. This is mostly for convenience, as creating classes which extend `FlaskForm` inherently gives us some out-of-the-box logic related to validation, etc.:

```
from flask_wtf import FlaskForm
```

Our form is going to need input fields, as well as a way of validating the content of those fields once submitted. We can import field types directly from the `wtforms` package, and validators from the `wtforms.validators` package:

forms.py

```
from flask_wtf import FlaskForm,
from wtforms import StringField, TextField, SubmitField
from wtforms.validators import DataRequired, Length
```

We're going to create a single Python class per form. We'll start simple by creating a straightforward "contact us" form:

forms.py

```
...

class ContactForm(FlaskForm):
    """Contact form."""
    name = StringField('Name', [
        DataRequired()])
    email = StringField('Email', [
```

```
Email(message=('Not a valid email address.')),
DataRequired())
body = TextField('Message', [
    DataRequired(),
    Length(min=4, message=('Your message is too short.'))]
recaptcha = RecaptchaField()
submit = SubmitField('Submit')
```

We've created a form with 3 input fields (`name` , `email` , `body`) as well as a `submit` field (which is really just a button). Each of our input fields consists of the following pieces:

- **Type of Input:** These are the field types we imported from `wtforms` , where `StringField` is a single-line text field, `TextField` is a multiline textarea, and so forth. WTForms has a strong collection of input types which includes inputs for things like passwords, date-pickers, multi-select drop-downs, and so forth. We'll get more into these later.
- **Label:** The first parameter passed to a "field" object is the field's "label," AKA the human-readable name for each field. Labels will carry through to our end users, so we should name our input fields properly.
- **Validators:** A *validator* is a restriction put on a field that must be met for the user's input to be considered valid. These are restrictions, such as ensuring a password is a minimum of 8 characters long. An input field can have multiple validators. If the user attempts to submit a form where *any* field's validators are not fully met, the form will fail and return an error to the user.
- **Error Message:** Any time a validator is not met, we need to tell the user what went wrong. Thus, every validator has an error message.

Admittedly, form classes aren't always easy to decipher at first glance due to their complexity. Here's a quick blueprint of what each input field is comprised of:

```
[VARIABLE] = [FIELD TYPE]('[LABEL]', [
    validators=[VALIDATOR TYPE](message=('[ERROR MESSAGE']))
])
```

Serving Forms In Flask Routes

So we have a form class, but we haven't even created a Flask app yet. Let's take care of that, and let's create a route for our contact form while we're at it:

app.py

```
from flask import Flask, url_for, render_template, redirect
```

```
from forms import ContactForm

app = Flask(__name__, instance_relative_config=False)
app.config.from_object('config.Config')

@app.route('/', methods=('GET', 'POST'))
def contact():
    form = ContactForm()
    if form.validate_on_submit():
        return redirect(url_for('success'))
    return render_template('index.html', form=form)
```

It seems simple enough, but there's more happening here than it seems at first glance. Our `contact` route accepts both `GET` and `POST` requests; normally we handle routes like this by using a conditional on Flask's `request` object (such as `if request.method == 'POST'`), but we haven't even imported the `request` object at all! Because we used Flask-WTF's `FlaskForm` base class to create `ContactForm()`, we're able to simplify routes containing form logic into two scenarios: one where the user submitted a valid, and one for *everything* else. We're able to do this because our form now has a built-in method called `validate_on_submit()`, which detects if a request is *both* a `POST` request and a valid request.

When we do serve our form, we're doing so by passing our form variable into an `index.html` Jinja template.

Building Forms in Jinja

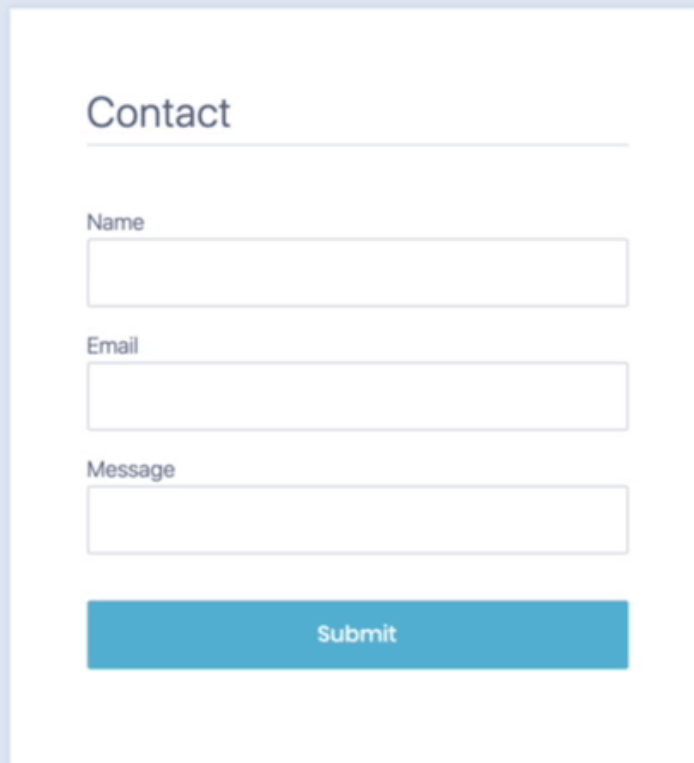
Writing anything resembling HTML is obviously the shittiest part of our lives. Jinja luckily reduces the pain a bit by making form handling easy:

contact.html

```
{% extends 'layout.html' %}

{% block content %}
<div class="formwrapper">
  <h2 class="title">Contact</h2>
  <form method="POST" action="/">
    <div class="form-field">{{ form.name.label }} {{ form.name(size=20) }}</div>
    <div class="form-field">{{ form.email.label }} {{ form.email }}</div>
    <div class="form-field">{{ form.body.label }} {{ form.body }}</div>
    {{ form.submit }}
  </form>
</div>
{% endblock %}
```

This is truly as easy as it looks. We passed our form (named `form`) to this template, thus `{{ form.name }}` is our form's *name* field, `{{ form.email }}` is our form's *email* field, and so forth. Let's see how it looks:

A screenshot of a web form titled "Contact". The form is centered on a light blue background. It contains three text input fields labeled "Name", "Email", and "Message". Below these fields is a blue "Submit" button. The form is currently empty, and there are no error messages displayed.

Submitting an invalid form.

Oh no... we haven't built in a way to display errors yet! Without any visual feedback, users have no idea *why* the form is failing, or if it's even attempting to validate at all.

Form fields can handle having multiple validators, which is to say that a single field has several criteria it must meet before the provided data is considered valid. Since a field can have multiple validators, it could also potentially throw multiple errors if said validators are not satisfied. To handle this, we can use Jinja to loop through all errors for each of our fields (if they exist) and display all of them. Here's an example with how we'd handle errors on the **email** field alone:

contact.html

```
...  
<div class="form-field">{{ form.email.label }} {{ form.email }}  
  {% if form.email.errors %}  
    <ul class="errors">  
      {% for error in form.email.errors %}  
        <li>{{ error }}</li>  
      {% endfor %}
```

```

    </ul>
    {% endif %}
</div>
...

```

When a form in Flask fails, it executes the form action with metadata of the request provided. In our case, the action of our form reloads the page it lives on by default but sends along whichever form error messages were attached to the request. We can access form errors per field using `form.email.errors`, and since we know there might be multiple errors, we use `{% for error in form.email.errors %}` to loop through all errors for this single field.

Let's see how the full form looks now with proper error feedback handling:

contact.html

```

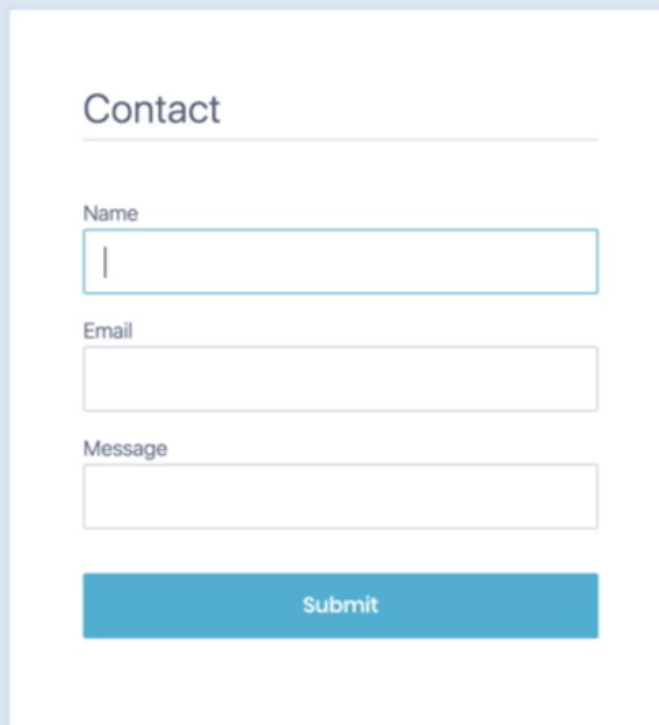
{% extends 'layout.html' %}

{% block content %}
<div class="formwrapper">
  <h2 class="title">Contact</h2>
  <form method="POST" action="/">
    <div class="form-field">{{ form.name.label }} {{ form.name }}
    {% if form.name.errors %}
      <ul class="errors">
        {% for error in form.name.errors %}
          <li>{{ error }}</li>
        {% endfor %}
      </ul>
    {% endif %}
  </div>
  <div class="form-field">{{ form.email.label }} {{ form.email }}
  {% if form.email.errors %}
    <ul class="errors">
      {% for error in form.email.errors %}
        <li>{{ error }}</li>
      {% endfor %}
    </ul>
  {% endif %}
  <div class="form-field">{{ form.body.label }} {{ form.body }}
  {% if form.body.errors %}
    <ul class="errors">
      {% for error in form.body.errors %}
        <li>{{ error }}</li>
      {% endfor %}
    </ul>
  {% endif %}
  </div>

```

```
    {{ form.submit }}  
  </form>  
</div>  
{% endblock %}
```

Our simple form suddenly feels more complex, perhaps cumbersome, as we deal with the reality of form logic handling. We should not let this translate into complexity for our end user. On the contrary, the more robust we make our forms for catching edge-cases, the more usable a form will appear to be to our users, as long as we communicate errors in an effective visual manner:



Fields are now throwing error messages!

We can now see a proper error message being displayed when the user attempts to submit with a bogus email! You'll also notice the prompt which occurs when the user attempts to submit the form without filling the required fields: this prompt is actually built-in, and will occur for any field which contains the `DataRequired()` validator.

Create a Signup Form

Let's explore Flask-WTF a little more by creating a more complicated form, such as an account sign-up form:

forms.py

```

from flask_wtf import FlaskForm, RecaptchaField
from wtforms import (StringField,
                     TextAreaField,
                     SubmitField,
                     PasswordField,
                     DateField,
                     SelectField)

from wtforms.validators import (DataRequired,
                               Email,
                               EqualTo,
                               Length,
                               URL)

...

class SignupForm(FlaskForm):
    """Sign up for a user account."""
    email = StringField('Email', [
        Email(message='Not a valid email address.'),
        DataRequired()])
    password = PasswordField('Password', [
        DataRequired(message="Please enter a password."),
    ])
    confirmPassword = PasswordField('Repeat Password', [
        EqualTo(password, message='Passwords must match.')
    ])
    title = SelectField('Title', [DataRequired()],
                       choices=[('Farmer', 'farmer'),
                                ('Corrupt Politician', 'politician'),
                                ('No-nonsense City Cop', 'cop'),
                                ('Professional Rocket League Player', 'rocket'),
                                ('Lonely Guy At A Diner', 'lonely'),
                                ('Pokemon Trainer', 'pokemon')])
    website = StringField('Website', validators=[URL()])
    birthday = DateField('Your Birthday')
    recaptcha = RecaptchaField()
    submit = SubmitField('Submit')

```

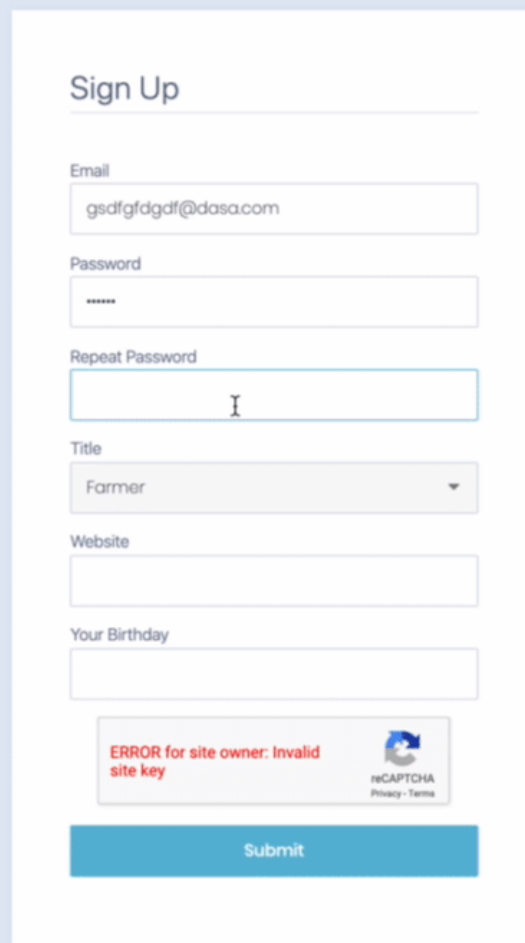
We've added a few things here:

- **RecaptchaField** is a field type specific to Flask-WTF (hence why we import it from `flask_wtf` instead of `wtforms`). As you may expect, this allows us to add a captcha to our form to prevent bots from submitting forms. A valid Recaptcha field requires us to supply two configuration variables to work: `RECAPTCHA_PUBLIC_KEY` and `RECAPTCHA_PRIVATE_KEY`, both of which can be acquired by following the [Google API docs](#).
- **PasswordField** asks new users to set their passwords, which of course have hidden input on the frontend side. We leverage the `EqualTo` validator to create a second

"confirm password" field, where the second field must match the first in order to be considered valid.

- **SelectField** is a dropdown menu where each "choice" is a tuple of a display name and an id.
- **URL** is a built-in validator for string fields, which ensures that the contents of a field match a URL. We're using this for the `website` field, which is our first optional field: in this case, an empty field would be accepted as valid, but a non-URL would not.
- **DateField** is our last field, which will only accept a date as input. This form is also optional.

And here's our new form:




Our new signup form.

What Happens Next?


If this were a real signup form, we'd handle user creation and database interaction here as well. As great as that sounds, I'll save your time as I know you still have an Instagram to check. Hats off to anybody who has made it through this rambling nonsense - you deserve it.


There's only so much a human being can consume from reading a 2000-word tutorial, so we've gone ahead and posted the source for this tutorial on Github for your convenience:



HANDLING FORMS IN FLASK USING FLASK-WTFORMS

hackersandslackers/flask-wtform-tutorial

 Tutorial to implement forms in your Flask app. Contribute to hackersandslackers/flask-wtform-tutorial development by creating...

 hackersandslackers • GitHub

Flask

Python

Software Development



Todd
Birchard's
avatar

Todd Birchard

 118 Posts Website Twitter

Engineer with an ongoing identity crisis. Breaks everything before learning best practices.
Completely normal and emotionally stable.

Faiyaz Hasan

0 points
49 days ago



Learning a lot here. Great resource! Keep up the good work eh bub!

Amaury Van Espen

0 points
3 months ago

