Web Programming with Flask

Author: Denis Rinfret

Sending Data to the Web Server Using HTML Forms

Example: Contact Form

- Project: `04_flask_forms` , file `01_contact_form.html`

<!DOCTYPE html>

Contact Form

Your name

Your email address

Your message

Send Clear

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Contact Form</title>
</head>
<body>
<h1>Contact Form</h1>
<!-- The action is usually an endpoint on the server to handle the submitted
        data -->
<form action="/contact_form" method="post">
    <label>Your name <input type="text" name="name"/></label><br/>
    <label>Your email address <input type="email" name="email"/></label><br/>
    <label>Your message <textarea name="message"></textarea></label><br/>
    <button type="submit">Send</button>
    <button type="reset">Clear</button>
</form>
```

```
</body>
</html>
```

The `form` element

- The 2 most important attributes to the `form` element are the `action` and `method` attributes
- The `action` specifies where the form data will be sent
  - in this case, it will send the data to the `/contact_form` endpoint
- The `method` specifies how to send the data
  - in this case, it will send the data using the *POST* method
  - the default method is *GET*, which should **not** normally be used by forms in most cases
  - the `/contact_form` endpoint should be configured to handle POST requests
  - by default, endpoints will only accept GET requests

Labels and Inputs

- It is common to label the input elements with a `label` element containing the input element
- An `input` of type `text` is a simple text box, containing only 1 line of input text
- An `input` of type `email` is a specialised version of the simple text box: it differs only in the way the data will be validated
- A `textarea` is a generalisation of the simple text box: it supports multiple rows of input text
- In order for the data input to be sent to the server, the `name` attributes have to be set to names unique within the form (some exceptions to this rule will be covered later on)
  - without a `name` attribute, the input data will not be sent
- The last 2 form elements are buttons
  - the first button will `submit` the form data to the `action` specified in the `form` element
  - the second button will clear, or `reset` the form data
  - the text shown on the buttons will be, respectively, *Send* and *Clear*

`/` and `/contact_form` endpoints

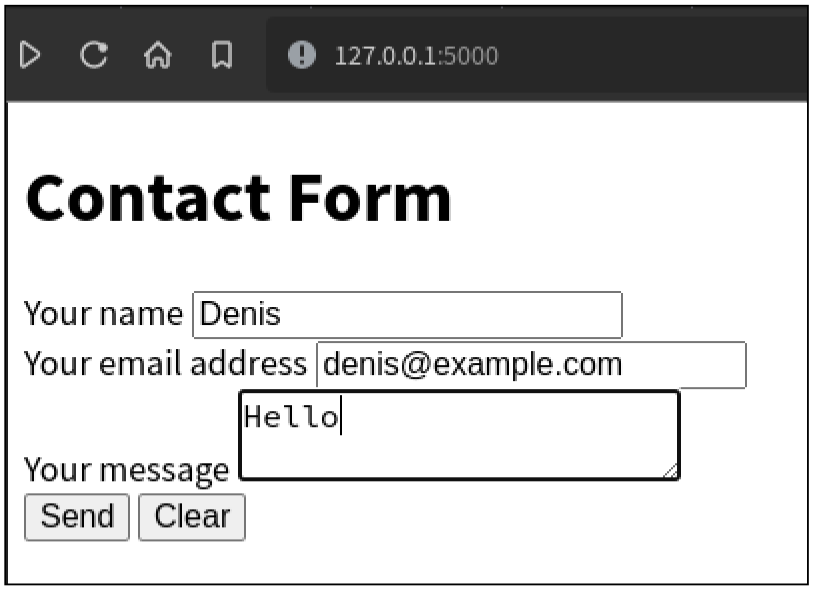
```
@app.route('/')
def contact_form():
    return render_template('01_contact_form.html')


@app.route('/contact_form', methods=['POST'])
def handle_contact_form():
    # we should normally validate the submitted form data before using it
    # by default, the template rendering will be considered unsafe,
    # so the data will be escaped
    return render_template('03_contact_response.html',
                           data=request.form)
```

- The `/` endpoint simply renders the template containing the form, which is just the HTML file shown above
- The `/contact_form` endpoint will handle the POST request coming from the form submission
- The `methods=['POST']` argument to the `route` function specifies that this endpoint will only handle POST requests
  - GET requests will not be accepted
- The only thing we do, for now, is to render the `03_contact_response.html` template with the form data that was submitted
  - to access the submitted form data, simply use `request.form`
  - request must be imported from the `flask` package
  - `request.form` is a special kind of object holding the form data, and its properties correspond to the `name` attributes specified in the form

File: `03_contact_response.html`

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Message Received</title>
</head>
<body>
<h1>Thank you <em>{{ data.name }}</em> for your message!</h1>
<p>We will contact you shortly.</p>
<p>Email: {{ data.email }}</p>
<p>Message: {{ data.message }}</p>
</body>
</html>
```

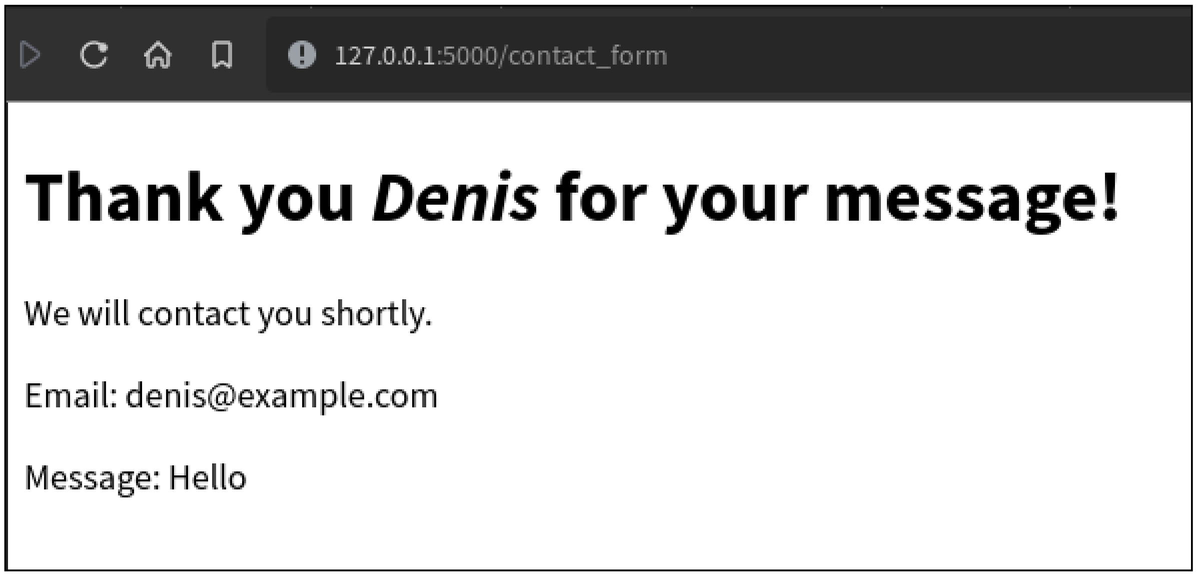

127.0.0.1:5000

Contact Form

Your name Denis

Your email address denis@example.com

Your message Hello


127.0.0.1:5000/contact_form

Thank you *Denis* for your message!

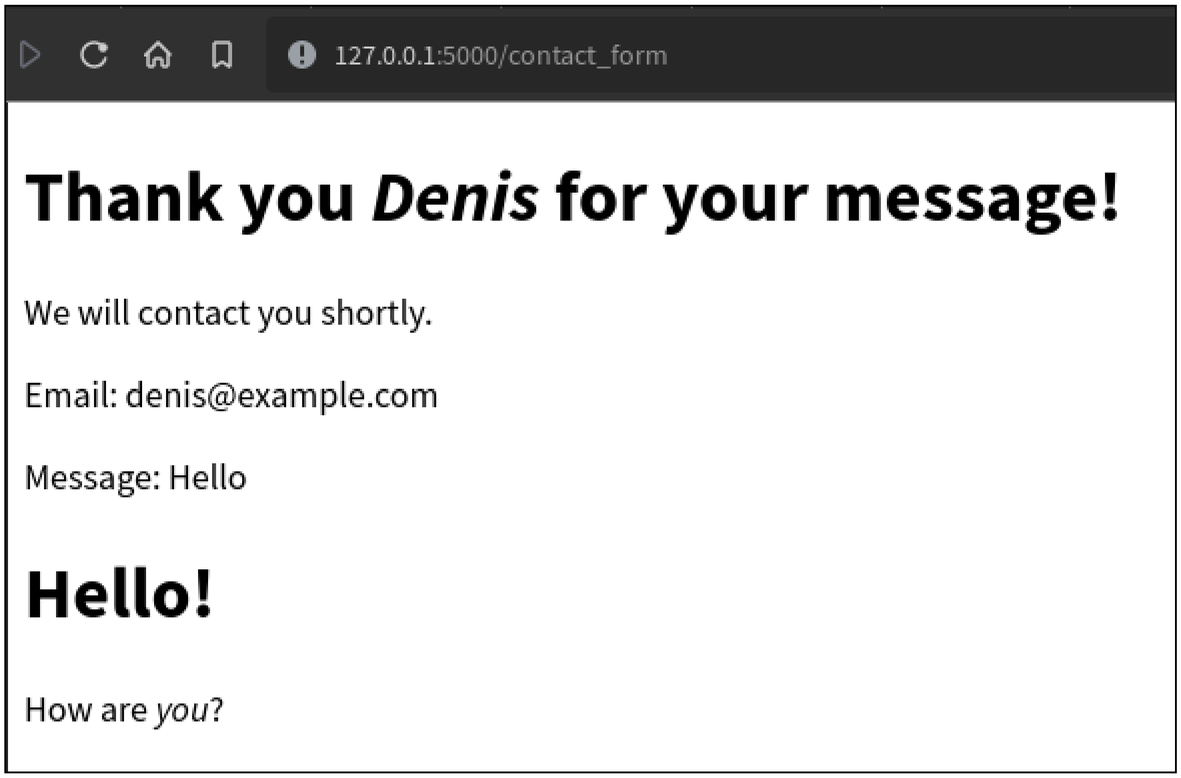
We will contact you shortly.

Email: denis@example.com

Message: Hello

Do Not Trust the Data Sent to the Server!

- By default, Flask and Jinja2 do not trust the data sent to the server
- This is the safe and smart way to deal with form data
- Before inserting the form data in the template, it will *escape* the data
    - the message `Hello <h1>Hello</h1>! How are <em>you</em>?` is not safe, it can mess up the template if not escaped properly



Thank you *Denis* for your message!

We will contact you shortly.

Email: denis@example.com

Message: Hello

Hello!

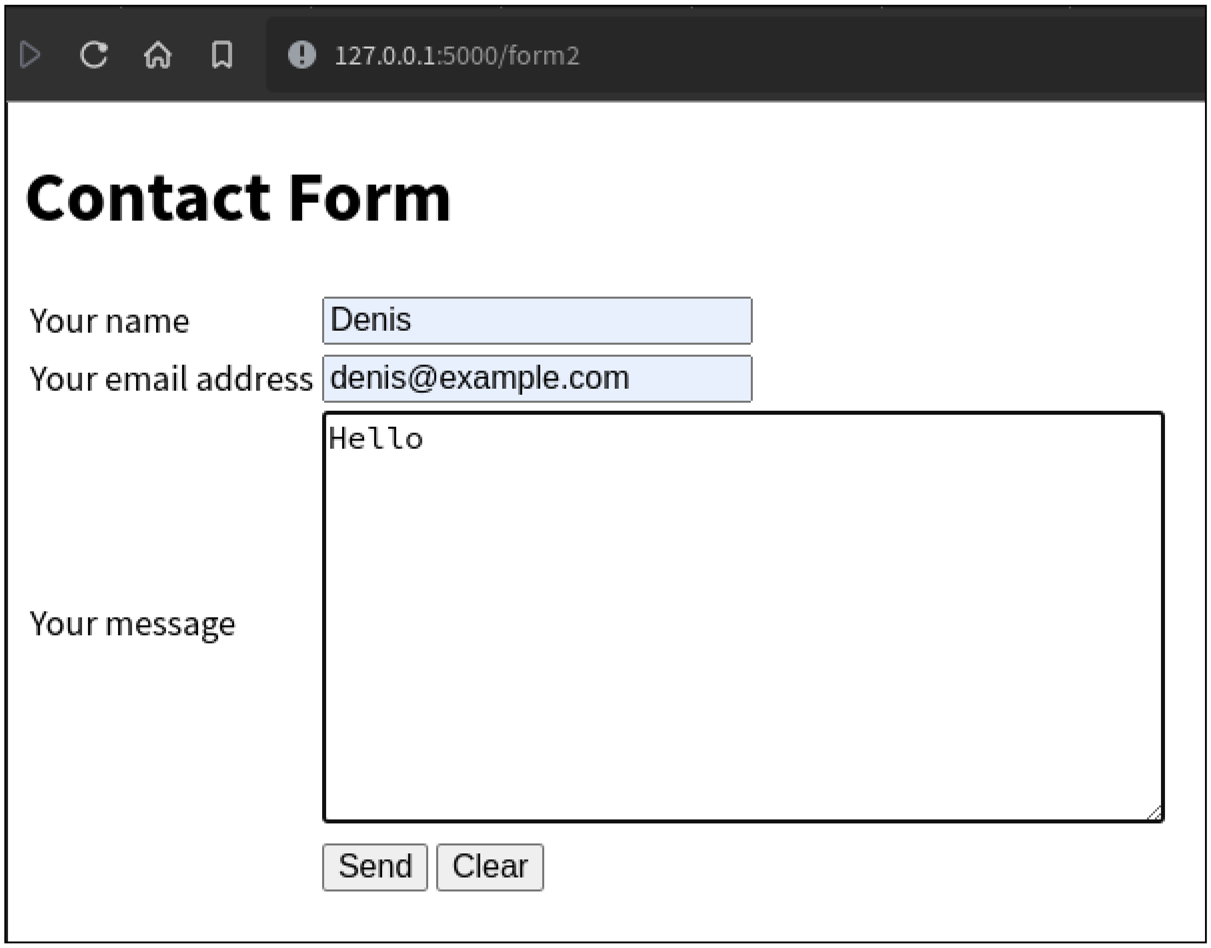
How are *you*?

- Escaping the string `Hello <h1>Hello</h1>! How are <em>you</em>?` will give the string `Hello &lt;h1&gt;Hello&lt;/h1&gt;! How are &lt;em&gt;you&lt;/em&gt;?`
- In this way, the form data will not mess up the HTML in the template
- Instead of a simple `h1` element, someone could try to inject some JavaScript code and try to take advantage of the client and/or server
- If the data to be inserted in the template can be trusted, then we can use the `safe` template decorator in this way:
    - `<p>Message: {{ data.message|safe }}</p>`
    - in this case, `data.message` will not be escaped, and we will get the result shown in the previous screenshot

`/contact_form2` endpoint

- Usually, we should not only echo the form data back to the client, we should process the data in some way
- For our example, we could save the form data in a file or in a database, and/or send some email to the website owner
- Since interacting with a database and sending emails have not been covered yet, we will, for now, save the form data in a CSV file
- The `/form2` endpoint renders the `02_contact_form.html` template
    - it contains a slightly different version of the contact form
    - the form data will be sent to the `/contact_form2` endpoint

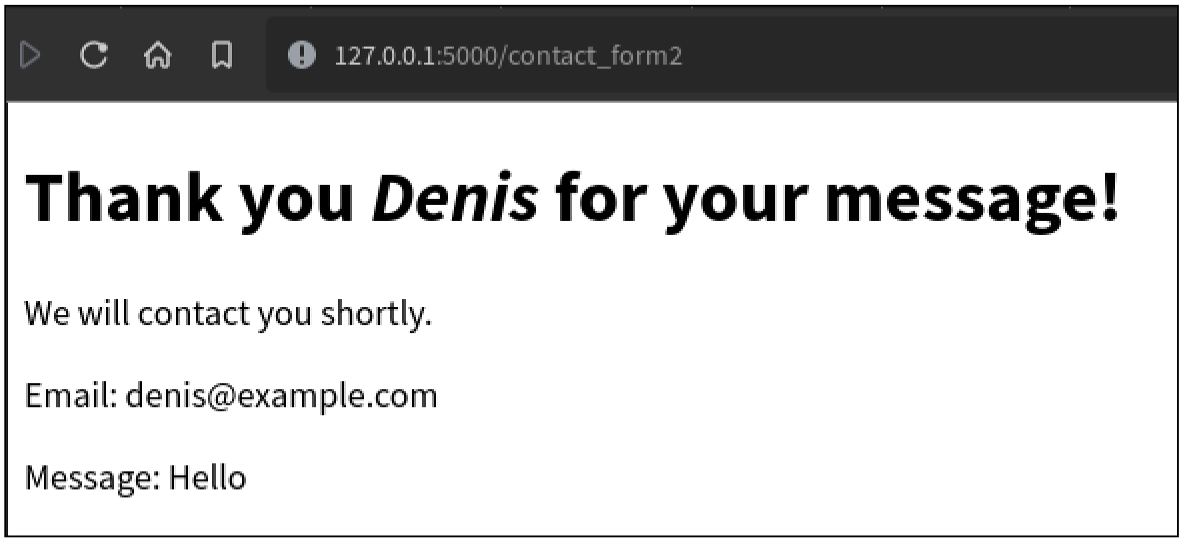
- the result return to the client is the same as before, but information is saved a file before returning
- in practice, it should not be done this way since it will be inefficient and unsafe, especially for many concurrent executions: access to the file is not controlled



127.0.0.1:5000/form2

Contact Form

Your name	Denis
Your email address	denis@example.com
Your message	Hello



127.0.0.1:5000/contact_form2

Thank you *Denis* for your message!

We will contact you shortly.

Email: denis@example.com

Message: Hello

`/contact_form2` Endpoint

```
@app.route('/contact_form2', methods=['POST'])
def handle_contact_form2():
    with open('data/messages.csv', 'a') as f:
        writer = csv.writer(f)
        writer.writerow([request.form['name'],
                         request.form['email'],
                         request.form['message']])
    return render_template('03_contact_response.html',
                           data=request.form)
```

File `messages.csv`

```
Denis,denis@example.com,Hello
Denis,denis@example.com,Hello <h1>Hello</h1>! How are <it>you</it>?
```

Contact Form, File `02_contact_form.html`

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Contact Form</title>
</head>
<body>
<h1>Contact Form</h1>
<!-- The action is usually an endpoint on the server to handle the submitted
        data -->
<form action="/contact_form2" method="post">
    <table>
        <tr>
            <td><label>Your name</label></td>
            <td><input type="text" name="name"/></td>
        </tr>
        <tr>
            <td><label>Your email address </label></td>
            <td><input type="email" name="email"/></td>
        </tr>
        <tr>
            <td><label>Your message </label></td>
            <td><textarea name="message" rows="10" cols="40"></textarea></td>
        </tr>
        <tr>
            <td></td>
            <td>
                <button type="submit">Send</button>
                <button type="reset">Clear</button>
            </td>
        </tr>
    </table>
```

```
</form>
</body>
</html>
```

GET vs. POST Requests

- W3Schools *HTTP Methods* https://www.w3schools.com/tags/ref_httpmethods.asp
- The two most common HTTP methods are: GET and POST.

What is HTTP?

- The Hypertext Transfer Protocol (HTTP) is designed to enable communications between clients and servers.
- HTTP works as a request-response protocol between a client and server.
- A web browser may be the client, and an application on a computer that hosts a web site may be the server.
- Example:
    - A client (browser) submits an HTTP request to the server
    - then the server returns a response to the client
    - the response contains status information about the request and may also contain the requested content.

W3Schools

HTTP Methods

- GET
- POST
- PUT
- HEAD
- DELETE
- PATCH
- OPTIONS

W3Schools

The GET Method

- GET is used to request data from a specified resource.
- GET is one of the most common HTTP methods.
- Note that the query string (name/value pairs) is sent in the URL of a GET request:
  - `test/demo_form.php?name1=value1&name2=value2`
- Some other notes on GET requests:
  - GET requests can be cached
  - GET requests remain in the browser history
  - GET requests can be bookmarked
  - GET requests should never be used when dealing with sensitive data
  - GET requests have length restrictions
  - GET requests is only used to request data (not modify)

W3Schools

The POST Method

- POST is used to send data to a server to create/update a resource.
- The data sent to the server with POST is stored in the request body of the HTTP request:

```
POST /test/demo_form.php HTTP/1.1 Host: w3schools.com name1=value1&name2=value2
```

- POST is one of the most common HTTP methods.
- Some other notes on POST requests:
  - POST requests are never cached
  - POST requests do not remain in the browser history
  - POST requests cannot be bookmarked
  - POST requests have no restrictions on data length

W3Schools

GET vs. POST

	GET	POST
BACK button/Reload	Harmless	Data will be re-submitted (the browser should alert the user that the data are about to be re-submitted)
Bookmarked	Can be bookmarked	Cannot be bookmarked

Cached	Can be cached	Not cached
Encoding type	application/x-www-form-urlencoded	application/x-www-form-urlencoded or multipart/form-data. Use multipart encoding for binary data
History	Parameters remain in browser history	Parameters are not saved in browser history

W3Schools

GET vs. POST

	GET	POST
Restrictions on data length	Yes, when sending data, the GET method adds the data to the URL; and the length of a URL is limited (maximum URL length is 2048 characters)	No restrictions
Restrictions on data type	Only ASCII characters allowed	No restrictions. Binary data is also allowed
Security	GET is less secure compared to POST because data sent is part of the URL Never use GET when sending passwords or other sensitive information!	POST is a little safer than GET because the parameters are not stored in browser history or in web server logs
Visibility	Data is visible to everyone in the URL	Data is not displayed in the URL

W3Schools

Reloading a Page With POSTed Data

- When reloading a page having POSTed data associated to it, depending on which browser you use,
  - it will ask you to confirm if you want to resubmit the data to the server
    - if yes, it will resubmit the data
  - or it will resubmit the data without asking for your confirmation
- In both cases, you may end up with duplicated data on the server, depending on how the form submission is handled
- When no precautions are implemented (like in the previous examples), you will end up with duplicated data
- When forms are handled correctly (like in the following examples), no duplicated data will end up on the server (at least not because of form resubmission through a reload)

Proper Form Submission and Handling

With `wtforms` and `flask_wtf`

- *WTForms* are used to simplify and standardize HTML forms creation and handling
- `flask_wtf` is a Flask wrapper around `wtforms`
- The first step is to define a subclass of `FlaskForm`
  - in our example we define a `ContactForm` class
- We need to import these classes in order to be able to define our contact form

```
from flask_wtf import FlaskForm
from wtforms import StringField, TextAreaField
from wtforms.fields.html5 import EmailField
from wtforms.validators import InputRequired, Email
```

`ContactForm` Class, File `02_app.py`

```
class ContactForm(FlaskForm):
    name = StringField('Name', validators=[InputRequired()])
    email = EmailField('Email', validators=[InputRequired(), Email()])
    message = TextAreaField('Message', validators=[InputRequired()])
```

- `name` , `email` and `message` are *class variables* (similar to *static fields* in Java)
- They are initialized to `Field` objects of the correct types
  - the first argument to each field constructor is the field name
  - the second is a list of *validators*
    - in our form, all the fields are *required*, meaning they cannot be empty (`InputRequired` validator)

        - additionally, our email field must be a valid email address (`Email` validator)
        - note the use of parentheses `()` after the validators

Some Other Available Fields

https://wtforms.readthedocs.io/en/stable/fields.html

- `BooleanField`
- `DateField`
- `DecimalField`
- `IntegerField`
- `RadioField`
- `SelectField`
- `EmailField`

Some Other Built-In Validators

https://wtforms.readthedocs.io/en/stable/validators.html

- `EqualTo` : useful to compare 2 form fields for equality
- `Length` : validate a string's length
- `NumberRange` : check if a number is within a given range
- `RegExp` : check if the field matches a given regular expression
- `AnyOf` : check if the field value is in a list of values (*white-listing*)
- `NoneOf` : check if the field value is *not* in a list of values (*black-listing*)

Rendering the Form and Handling the Form Data

- The best solution is to have a single endpoint for both
    - if the request method was GET, or if the submitted data doesn't validate, then render the form
    - if the request method was POST, and the submitted data validates, then process the form data

```
@app.route('/contact_form4', methods=['GET', 'POST'])
def handle_contact_form4():
    form = ContactForm()
    if form.validate_on_submit():
        with open('data/messages.csv', 'a') as f:
            writer = csv.writer(f)
            writer.writerow([form.name.data, form.email.data,
                             form.message.data])
        return redirect(url_for('contact_response',
                                    name=form.name.data))
    return render_template('04_contact_form.html', form=form)
```

1. create a form by instantiating the `ContactForm` class
    - this will automatically use the form data available in `request.form`
2. `form.validate_on_submit()` will first check if there was data submitted, and if the submitted data is valid
3. if it is valid, then process the form data as before,
    - except that the data is taken from `form` , not `request.form`
    - and that on success, we redirect to a response page
        - this way, reloading the response page will not resubmit the data since it will not have any form data associated to it
4. if we don't have valid submitted data, then render the form template

- The first time a user gets to this endpoint should be through a GET request
    - no data was submitted (the submit method is always POST), so the contact form in variable `form` will have empty data
    - `form.validate_on_submit()` will be false, so the form template will be rendered with empty form data
- If there was data submitted (through the POST method)
    - the `form` object will contain that data
    - if the data doesn't validate, then the form template will be rendered with the submitted form data, to avoid requiring the user to fill the form again
    - if the form data validates, then it will be processed as described earlier

File `04_contact_form.html`

`<form>` element

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Contact Form</title>
</head>
<body>
<h1>Contact Form</h1>
{% if form.errors %}
<ul class="errors">
    {% for field_name, field_errors in form.errors|dictsort if field_errors %}
    {% for error in field_errors %}
    <li>{{ form[field_name].label }}: {{ error }}</li>
    {% endfor %}
    {% endfor %}
</ul>
{% endif %}
<!-- The action is usually an endpoint on the server to handle the submitted
        data -->
<form action="/contact_form4" method="post">
    {{ form.csrf_token }}
    <table>
        <tr>
            <td>{{ form.name.label }}</td>
            <td>{{ form.name }}</td>
        </tr>
        <tr>
            <td>{{ form.email.label }}</td>
            <td>{{ form.email }}</td>
        </tr>
        <tr>
            <td>{{ form.message.label }}</td>
            <td>{{ form.message }}</td>
        </tr>
        <tr>
            <td></td>
            <td>
                <button type="submit">Send</button>
                <button type="reset">Clear</button>
            </td>
        </tr>
    </table>


</form>
</body>
</html>
```

- The layout of this form is similar to the previous example, it is using a `table` to align the fields.
- To get a field's label, we need to use, for example for the email field, `form.email.label`
- For the field itself, we need to use `form.email`
  - this will be translated to `<input id="email" name="email" required="" type="email" value="">`
  - in this example, we created the email field with `email = EmailField('Email', validators=[InputRequired(), Email()])`
  - `'Email'` is actually the label for this field, and the `id` and the `name` attributes will be equal to the class variable name used when defining the `ContactForm` (in this case,

    the class variable name is `email`)
  - if we want a different id, we could create the field in this way `email = EmailField('Your email address', id='someotherid', validators=[InputRequired(), Email()])`
- the `form.csrf_token` is for security purpose, to help prevent *Cross-Site Request Forgery* attacks
  - in the Python file, we also need to set the `app.secret_key`
    - to make more secure, we should use something better than `'allo'`
  - more on this later on

Error Reporting

- Some error reporting is done automatically through some kind of tooltips or pop-ups
  - this type of error reporting is handled by the browser, so the way it's done may vary
- The browsers don't catch everything, they only catch the validation rules that can be specified in HTML
- The other validation errors are caught when we create the `ContactForm` object and call `form.validate_on_submit()`
  - the errors are associated with the fields inside the form object
  - the contact form template needs to check if there are any errors and include them in the HTML if necessary
  - the easiest way to do it is with an if statement and 2 for loops before the form element
  - another way to do it will be shown later on

```
{% if form.errors %}
<ul class="errors">
    {% for field_name, field_errors in
    form.errors|dictsort if field_errors %}
    {% for error in field_errors %}
    <li>{{ form[field_name].label }}: {{ error }}</li>
    {% endfor %}
    {% endfor %}
</ul>
{% endif %}
```

Post/Redirect/Get Pattern

From Book: Flask Web Development 2e, Miguel Grinberg

- When the last request sent is a POST request with form data, a refresh would cause a duplicate form submission, which in almost all cases is not the desired action.

- For that reason, the browser asks for confirmation from the user.
- Many users do not understand this warning from the browser.
- *Consequently, it is considered good practice for web applications to never leave a POST request as the last request sent by the browser.*
- This is achieved by responding to POST requests with a redirect instead of a normal response.
- A redirect is a special type of response that contains a URL instead of a string with HTML code.
- When the browser receives a redirect response, it issues a GET request for the redirect URL, and that is the page that it displays.
- Now the last request is a GET, so the refresh command works as expected.
- **This trick is known as the Post/Redirect/Get pattern.**
- In our example, the line

  `return redirect(url_for('contact_response', name=form.name.data))`

  will redirect to the contact response using the GET method
- That endpoint doesn't have access to the form data, so we pass the name to it to have a personalized message
- Don't pass `form.name` to the redirect because it will pass the whole input field to the redirect
- `form.name.data` will pass only the value of that field
- The contact response endpoint simply takes the name given as argument and renders a template

```
@app.route('/contact_response/<name>')
def contact_response(name):
    return render_template('05_contact_response.html', name=name)
```

- The template is also quite simple

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Message Received</title>
</head>
<body>
<h1>Thank you <em>{{ name }}</em> for your message!</h1>
<p>We will contact you shortly.</p>
</body>
</html>
```

Bootstrap Forms

Files `03_app.py` and `forms.py`

- Since in this section we will have more forms and longer forms, it is more appropriate to create a separate file, `forms.py` , and put all the form definitions there
- In `03_app.py` , we need to import these forms with

  ```
  from forms import ContactForm, ExamplesForm
  ```

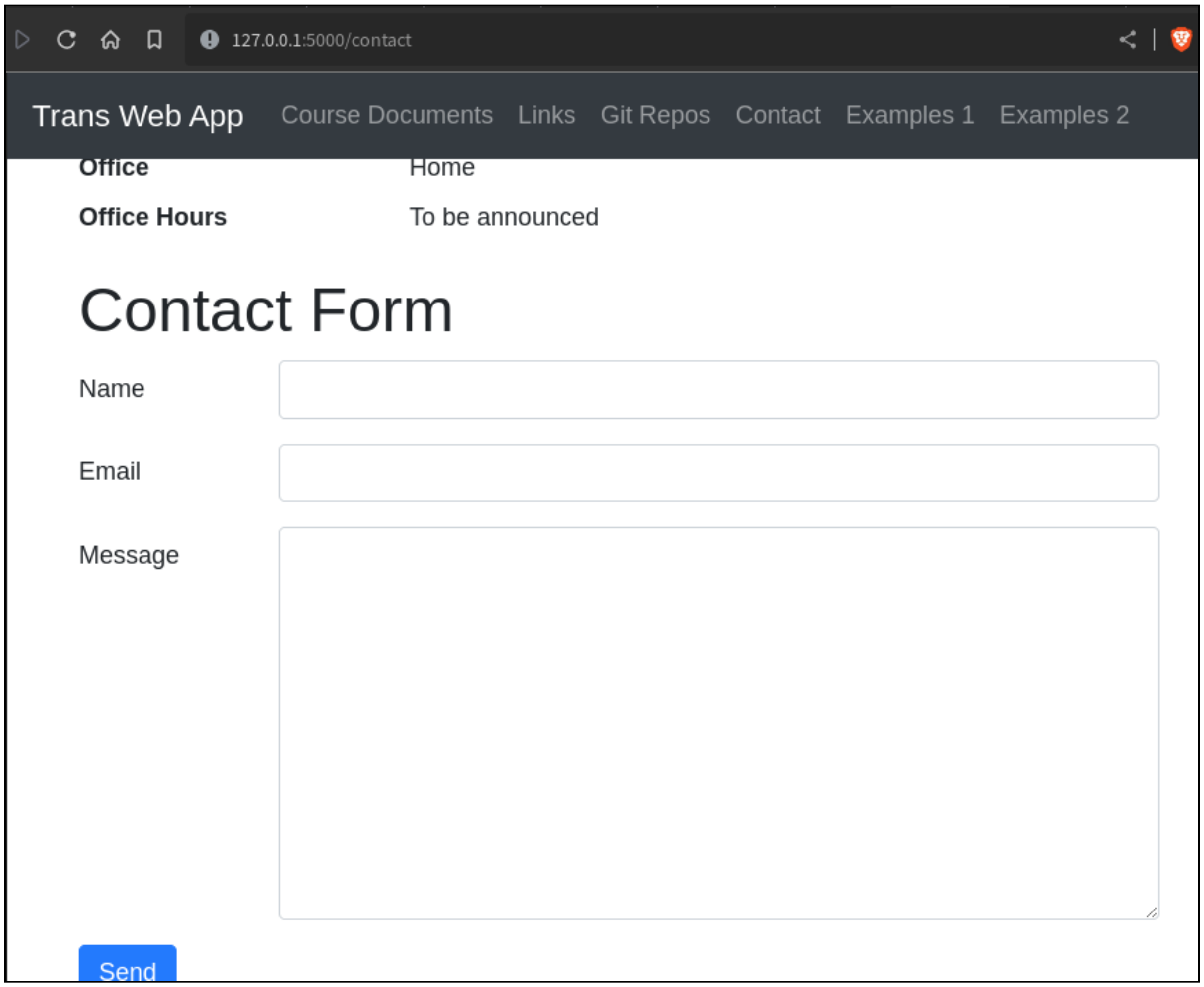
- Starting from the Bootstrap templates created earlier, we will start by adding the contact form to the contact page
- Then more form fields will be introduced in 2 different formats:
  - plain HTML
  - wtforms + Bootstrap

ContactForm

- The Python code for the contact form is almost identical compared to the previous section
  - the endpoints code has been slightly modified to change the routes and the template files names to render
  - The `ContactForm` class has been moved to `forms.py` with 2 small changes
    - to change the number of rows in the message text area, an extra argument as been given to the field: `render_kw={'rows': 10}`
    - a `submit` field has been added directly in the form

```
class ContactForm(FlaskForm):
    name = StringField('Name', validators=[InputRequired()])
```

```
    email = EmailField('Email', validators=[InputRequired(), Email()])
    message = TextAreaField('Message', validators=[InputRequired()],
                            render_kw={'rows': 10})
    submit = SubmitField('Send')
```



Office Home

Office Hours To be announced

Contact Form

Name

Email

Message

Contact Form

`/contact` and `/contact_response` endpoints

```
@app.route('/contact', methods=['GET', 'POST'])
def contact():
    form = ContactForm()
    if form.validate_on_submit():
        with open('data/messages.csv', 'a') as f:
            writer = csv.writer(f)
            writer.writerow([form.name.data, form.email.data,
                             form.message.data])
        return redirect(url_for('contact_response',
                                name=form.name.data))
    return render_template('contact.html', form=form)
```

```
@app.route('/contact_response/<name>')
def contact_response(name):
    return render_template('contact_response.html', name=name)
```

`contact.html`

- The contact template is quite simple because it is calling a *macro* (similar to a function) defined in the base template
- It is calling the `quick_form` macro taking the form object as an argument along with the action attribute for the form

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

{% block title %}{{ super() }} - Contact{% endblock %}

{% block content %}
<div class="container">
    {# This div container is the same as before #}
    <h1>Contact</h1>
    <dl class="row">
        <dt class="col-sm-3">Professor</dt>
        <dd class="col-sm-9">Denis Rinfret</dd>

        <dt class="col-sm-3">Email</dt>
        <dd class="col-sm-9"><a
         href="mailto:denis@example.com">denis@example.com</a>
        </dd>

        <dt class="col-sm-3">Office</dt>
        <dd class="col-sm-9">Home</dd>

        <dt class="col-sm-3">Office Hours</dt>
        <dd class="col-sm-9">To be announced</dd>
    </dl>
</div>

<div class="container">
    <h1>Contact Form</h1>
    {{ quick_form(form, '/contact') }}
</div>
{% endblock %} <!-- content -->
```

`base.html` macros

- There are actually 2 macros in the base template
    - `quick_form` , called by the contact template to quickly render a form

    - `form_group` , called by the other macro, to render 1 form field
- `quick_form` basically loops on all the form fields, and calls `form_group` on each field (with some exceptions)

```
{% macro quick_form(form, action) -%}
    <form action="{{ action }}" class="form" method="POST">
        {% for field in form %}
            {% if field.type in ('HiddenField', 'CSRFTokenField') %}
                {{ field() }}
            {% elif field.type == 'SubmitField' %}
                {{ field(class_="btn btn-primary") }}
            {% else %}
                {{ form_group(field) }}
            {% endif %}
        {% endfor %}
    </form>
{% endmacro %}
```

- `HiddenField` , `CSRFTokenField` and `SubmitField` are special fields
    - they don't need any label
    - they won't have any error messages associated to them
    - so render them directly

`form_group` macro

- This macro renders a form field
- Depending on the field type, it will be rendered differently
- It starts by putting the field in a *form-group row*
- And then renders the field label with the proper Bootstrap classes
- The following step depends on the field type:
    - if it's a checkbox group (`MultiCheckboxField`) or a radio button group (`RadioField`), it will loop over all the choices to include them in proper `div` elements with proper Bootstrap classes
        - if there are errors, it will use the `is-invalid` class
        - if not, it will use the `col-sm-6` class
        - it will render all the choices in a single row (class `form-check-inline`)

```
{% macro form_group(field) -%}
    <div class="form-group row required">
        {{ field.label(class_="col-sm-2 col-form-label") }}
        {% if field.type in ['MultiCheckboxField', 'RadioField'] %}
            {% if field.errors %}
```

```
                <div class="form-control col-sm-8 is-invalid">
            {% else %}
                <div class="form-control col-sm-8">
            {% endif %}
            {% for choice in field %}
                <div class="form-check form-check-inline">
                    {{ choice(class_="form-check-input") }}
                    {{ choice.label(class_="form-check-label") }}
                </div>
            {% endfor %}
            </div>
```

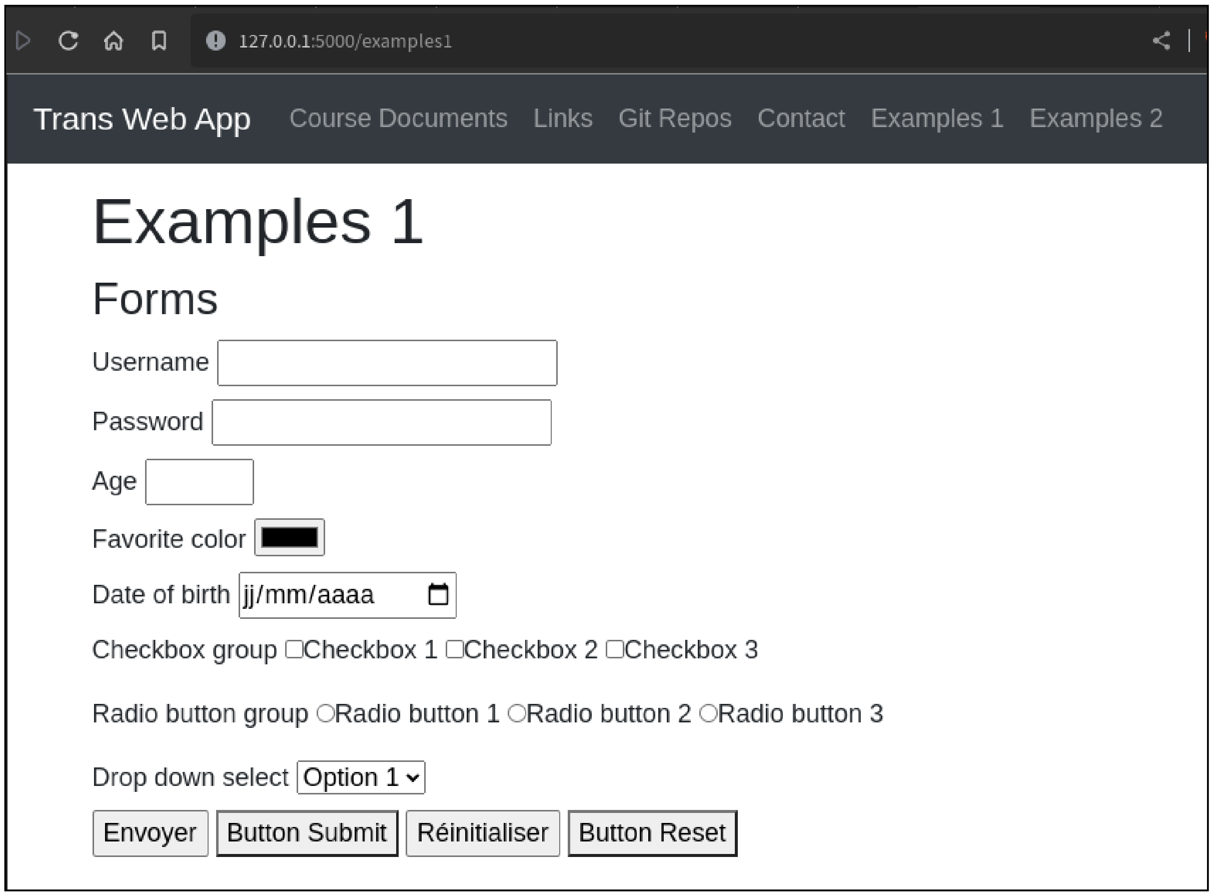
- if the field is not a group, it will also check if there are errors or not, and use either the `is-invalid` or the `col-sm-6` classes
- if there are errors, if will render what the errors are (and not just change the style to invalid)

```
        {% else %}
            {% if field.errors %}
                {{ field(class_="form-control col-sm-8 is-invalid") }}
            {% else %}
                {{ field(class_="form-control col-sm-8") }}
            {% endif %}
        {% endif %}
        {% if field.errors %}
            {%  for error in field.errors %}
                <div class="invalid-feedback">
                    {{ error }}
                </div>
            {% endfor %}
        {% endif %}
    </div>
{% endmacro %}
```

More Form Field Examples

- The `examples1` endpoint shows how to use a few more different kinds of form fields
- It simply renders a template containing a form with many field types
- A more complete example with a `FlaskForm` will follow

```
@app.route('/examples1')
def examples():
    return render_template('examples1.html')
```


127.0.0.1:5000/examples1

Examples 1

Forms

Username

Password

Age

Favorite color

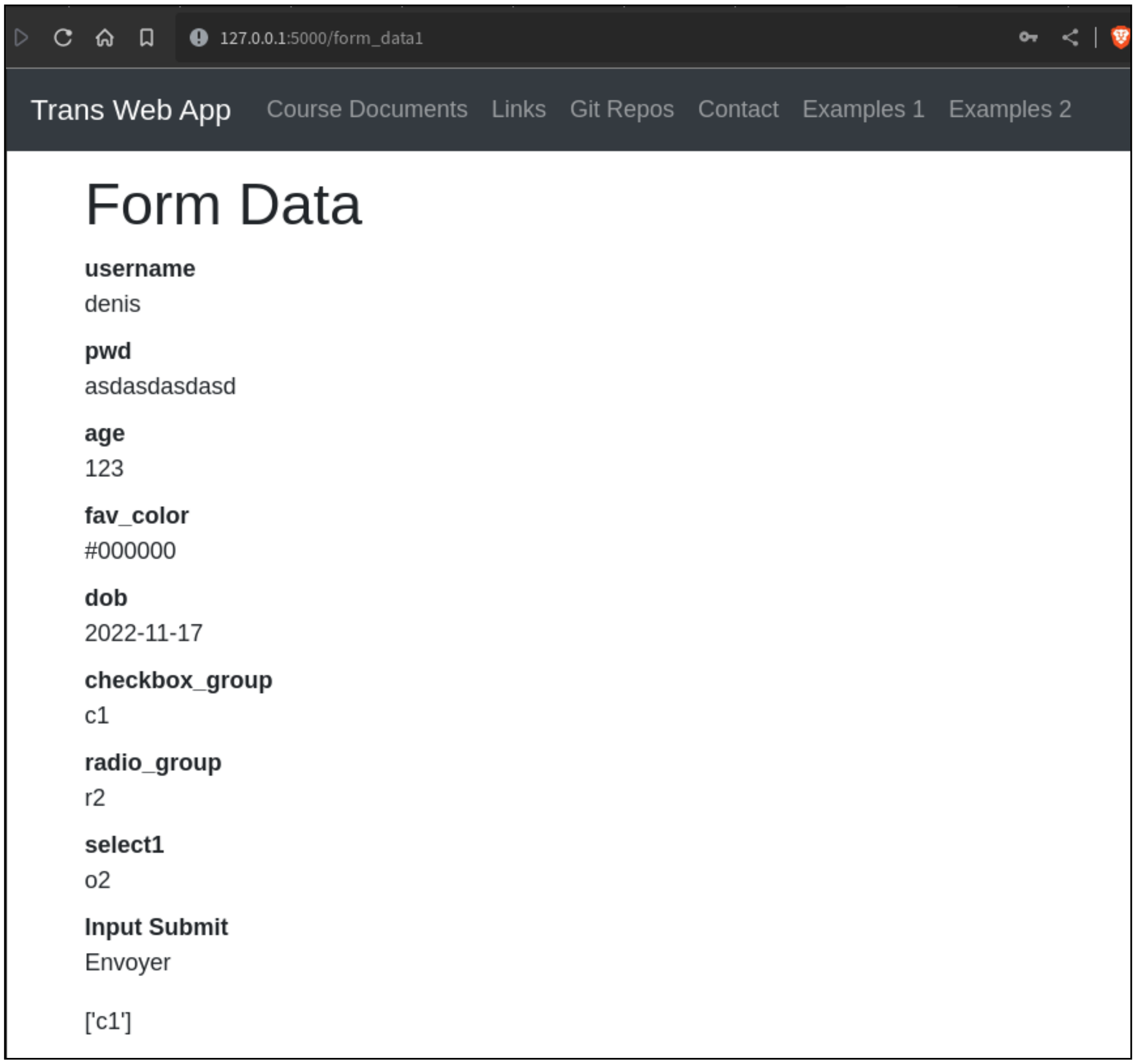
Date of birth jj/mm/aaaa

Checkbox group ☐Checkbox 1 ☐Checkbox 2 ☐Checkbox 3

Radio button group ○Radio button 1 ○Radio button 2 ○Radio button 3

Drop down select Option 1

Contact Form



Form Data

username
denis

pwd
asdasdasdasd

age
123

fav_color
#000000

dob
2022-11-17

checkbox_group
c1

radio_group
r2

select1
o2

Input Submit
Envoyer

['c1']

Contact Form

File `example1.html`

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

{% block title %}{{ super() }} - Examples 1{% endblock %}

{% block content %}
<div class="container">
    <h1>Examples 1</h1>
    <h3>Forms</h3>
    <form id="form01" action="/form_data1" method="post">
        <label>Username <input type="text" name="username"/></label><br/>
        <label>Password <input type="password" name="pwd"></label><br/>
        <label>Age <input type="number" name="age" min="1"
                  max="125"/></label><br/>
        <label>Favorite color <input type="color"
                                   name="fav_color"/></label><br/>
        <label>Date of birth <input type="date" name="dob"/></label><br/>
        <label>Checkbox group
            <label><input type="checkbox" name="checkbox_group"
 value="c1"/>Checkbox
                1</label>
            <label><input type="checkbox" name="checkbox_group"
 value="c2"/>Checkbox
                2</label>
            <label><input type="checkbox" name="checkbox_group"
 value="c3"/>Checkbox
                3</label>
        </label><br/>
        <label>Radio button group
            <label><input type="radio" name="radio_group" value="r1"/>Radio
                button 1</label>
            <label><input type="radio" name="radio_group" value="r2"/>Radio
                button 2</label>
            <label><input type="radio" name="radio_group" value="r3"/>Radio
                button 3</label>
        </label><br/>
        <label>Drop down select <select name="select1">
            <option value="o1">Option 1</option>
            <option value="o2">Option 2</option>
            <option value="o3">Option 3</option>
        </select></label><br/>
        <input type="submit" name="Input Submit"/>
        <button type="submit">Button Submit</button>
        <input type="reset" name="Input Reset"/>
        <button type="reset">Button Reset</button>
    </form>
</div>
{% endblock %} <!-- content -->
```

- The form’s action is `/form_data1` , which renders a template that simply loops through all the form fields and displays them in a `dl` element

```
@app.route('/form_data1', methods=['POST'])
def form_data1():
    return render_template('form_data1.html', form=request.form,
                           checkboxes=request.form.getlist('checkbox_group'))
```

File `form_data1.html`

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

{% block title %}{{ super() }} - Form Data{% endblock %}
{% block content %}
<div class="container">
    <h1>Form Data</h1>
    <dl>
        {% for key in form %}
        <dt>{{ key }}</dt>
        <dd>{{ form[key] }}</dd>
        {% endfor %}
    </dl>
    {{ checkboxes }}
</div>
{% endblock %} <!-- content -->
```

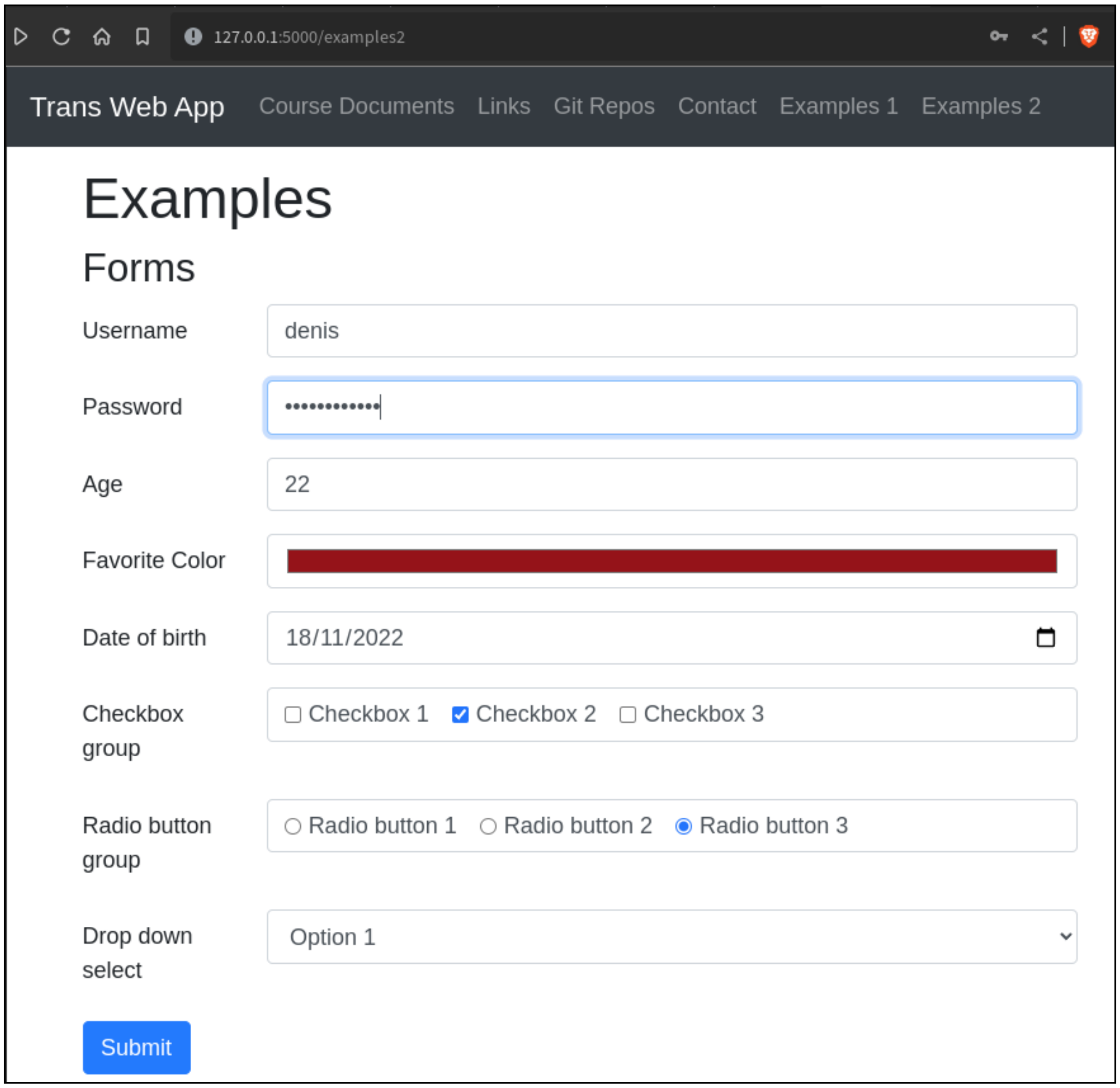
Checkboxes

- There only one tricky part in this example: handling checkboxes
- Checkboxes are different from other fields since there might be many checkboxes checked at any given time
- We need to get all the checked checkboxes from a checkbox group with `request.form.getlist('checkbox_group')`
- In this example, this list of checked checkboxes is passed separately to the template, which inserts it directly in the page with `{{ checkboxes }}`
- We could also loop on `checkboxes` to get a fancier output

`/examples2` endpoint

- This example is similar to the previous one, except that
  - it defines a subclass of `FlaskForm` to facilitate form validation and handling
  - it uses Bootstrap classes for rendering
- For simplicity, it is reusing the `form_data1.html` template from the previous example to display form data if the form is valid

```
@app.route('/examples2', methods=['GET', 'POST'])
def examples2():
    form = ExamplesForm()
    if form.validate_on_submit():
        return render_template('form_data1.html', form=request.form,
                               checkboxes=request.form.getlist(
                                   'checkbox_group'))
    return render_template('examples2.html', form=form)
```



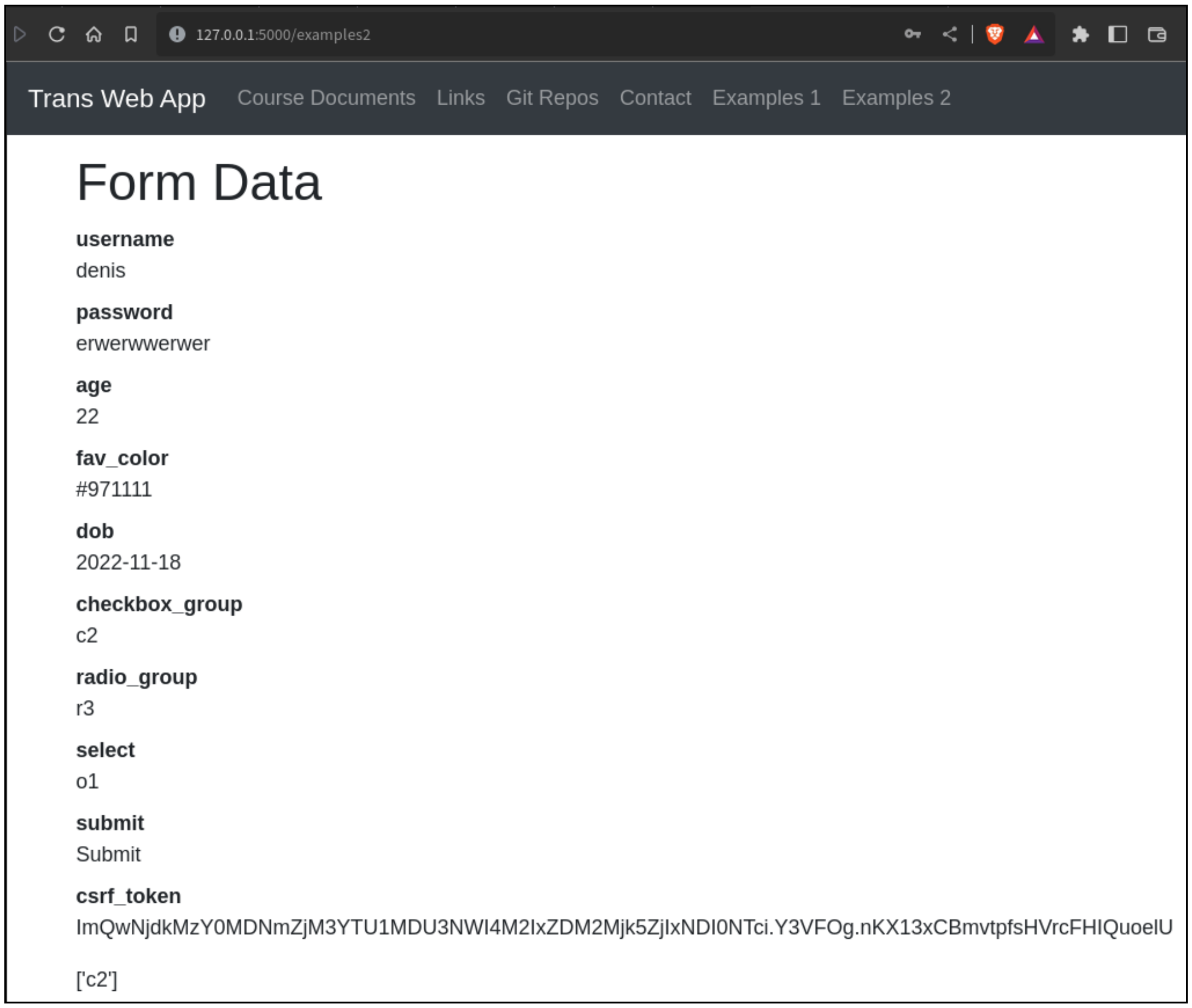
127.0.0.1:5000/examples2

Examples

Forms

Username	denis
Password	•••••••••••••
Age	22
Favorite Color	🟥
Date of birth	18/11/2022
Checkbox group	☐ Checkbox 1 ☑ Checkbox 2 ☐ Checkbox 3
Radio button group	○ Radio button 1 ○ Radio button 2 ● Radio button 3
Drop down select	Option 1

Contact Form



Form Data

username
denis

password
erwerwwerwer

age
22

fav_color
#971111

dob
2022-11-18

checkbox_group
c2

radio_group
r3

select
o1

submit
Submit

csrf_token
ImQwNjdkMzY0MDNmZjM3YTU1MDU3NWI4M2IxZDM2Mjk5ZjIxNDI0NTci.Y3VFOg.nKX13xCBmvtpfsHVrcFHIQuoelU

['c2']

Contact Form

`ExamplesForm`

```python
class ExamplesForm(FlaskForm):
    username = StringField('Username',
                           validators=[InputRequired(), Length(4, 64),
                                       Regexp('^[A-Za-z][A-Za-z0-9_.]*$', 0,
                                              'Usernames must start with a
      letter and must have only letters, numbers, dots or underscores')])
    password = PasswordField('Password',
                             validators=[InputRequired(), Length(8)])
    age = IntegerField('Age',
                       validators=[InputRequired(), NumberRange(1, 125)])
    fav_color = ColorField('Favorite Color',
                           validators=[InputRequired()])
    dob = DateField('Date of birth', validators=[InputRequired()])
```

New validators

- `Length(4, 64)` and `Length(8)` : the string length must be between 4 and 64 characters (inclusive), and the length must be at least 8 characters long
- `NumberRange(1, 125)` : a number must be between 1 and 125 (inclusive)
- `Regexp('^[A-Za-z][A-Za-z0-9_.]*$', 0, 'Error Message')` : the string must match the given regular expression to be valid (refer to the section on regular expressions)
    - in this case, usernames must start with a letter and must have only letters, numbers, dots or underscores

```
checkbox_group = MultiCheckboxField('Checkbox group',
                                    choices=[('c1', 'Checkbox 1'),
                                             ('c2', 'Checkbox 2'),
                                             ('c3', 'Checkbox 3')])
radio_group = RadioField('Radio button group',
                         validators=[InputRequired()],
                         choices=[('r1', 'Radio button 1'),
                                  ('r2', 'Radio button 2'),
                                  ('r3', 'Radio button 3')],
                         render_kw={'required': True})
select = SelectField('Drop down select',
                     validators=[InputRequired()],
                     choices=[('o1', 'Option 1'), ('o2', 'Option 2'),
                              ('o3', 'Option 3')])
submit = SubmitField('Submit')
```

- The field type `MultiCheckboxField` is actually not a standard field type
    - it is defined as a subclass of `SelectMultipleField` to provide checkboxes as default widgets
    - often, checkboxes are not used in a checkgroup, they are added in forms separately from other fields
        - in this case use a `BooleanField`
    - but if you need a checkbox group, you can do it this way
- One problem with checkbox and radio button groups is that the `InputRequired` validator doesn't seem to have any effect in many Flask versions (probably a bug)
    - this is why the parameter `render_kw={'required': True}` has been added to force it to be required
- `choices` is a list of (obviously) choices
    - each choice is a tuple with 2 strings: its value followed by its label

```
class MultiCheckboxField(SelectMultipleField):
    widget = ListWidget(prefix_label=False)
    option_widget = CheckboxInput()
```

- The last part of this `ExamplesForm` is a custom validator
- Built-in validators cannot cover all possible cases
- A custom validator is a function with a name starting with `validate_`
- The remainder of the function name must correspond to a field name (in this case `password`)
- It will automatically be called if the name is set correctly, no need to add it to `validators`

```
    def validate_password(self, field):
        with open('data/common_passwords.txt') as f:
            for line in f.readlines():
                if field.data == line.strip():
                    raise ValidationError('Your password is too common.')
```

- This custom validator checks if a password is too common
- If yes, it will raise a `ValidationError` and the field will be marked as invalid
- We assume here that the file `data/common_passwords.txt` contains 1 password per line, and that it contains the passwords we want to avoid
- In our simplified example, the password file contains only 3 passwords

```
12345678
11111111
password
```

`examples2.html`

- This template is quite simple since we are using the same `quick_form` macro to render the form
- This is not perfect, the layout of some fields is not optimal, but that's why it's qualified as *quick*
  - we could improve the macro to get a better layout
  - or we could skip the macro and fine tune the layout directly in the `examples2.html` template

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

{% block title %}{{ super() }} - Examples 1{% endblock %}

{% block content %}
<div class="container">
    <h1>Examples</h1>
    <h3>Forms</h3>
    {{ quick_form(form, '/examples2') }}
</div>

{% endblock %}
```