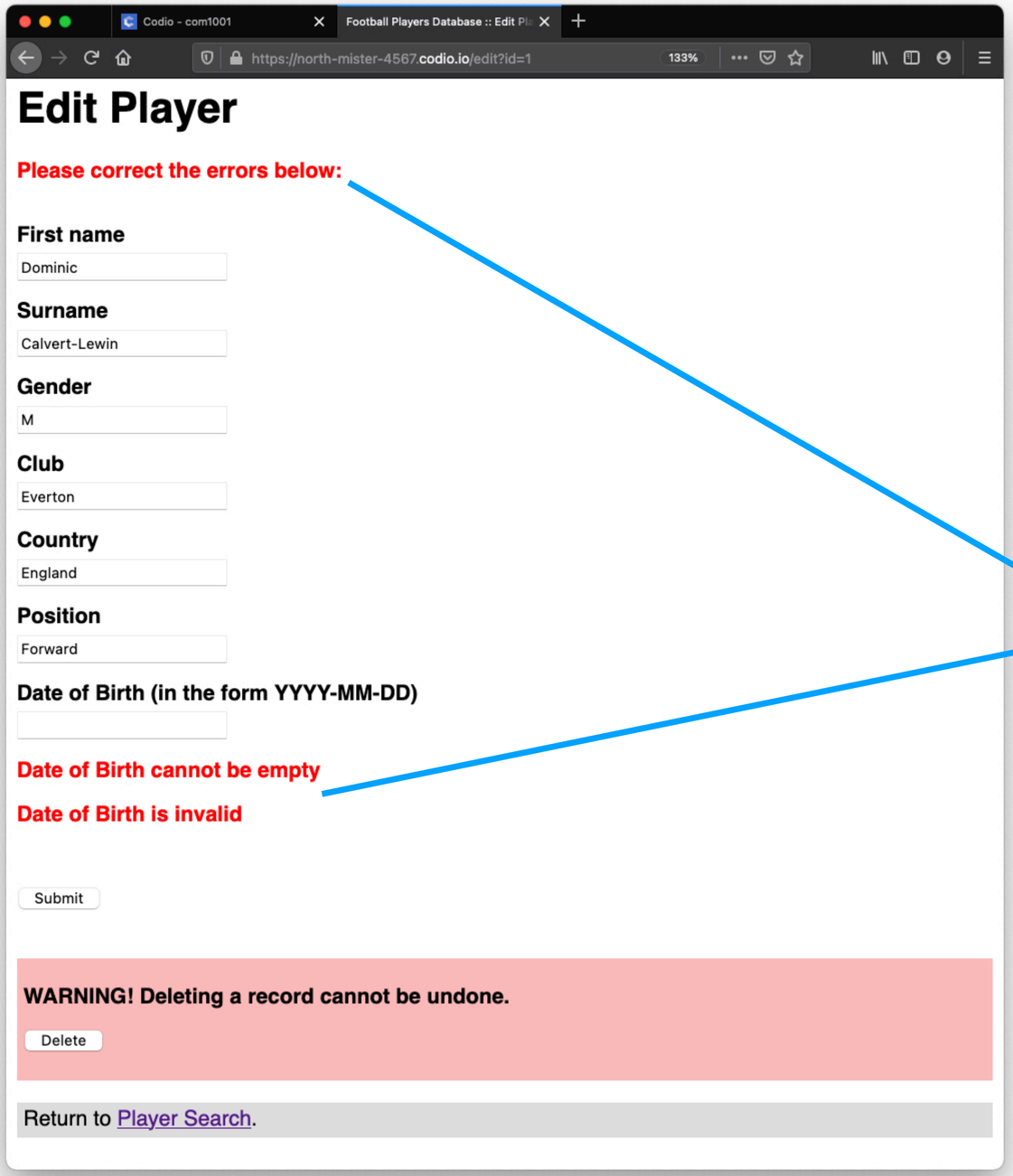
COM 1001

Introduction to Software Engineering

Professor Phil McMinn

Forms: Sanitising and Validating User Inputs



Validation and Sanitisation

Validation means ensuring the user has entered valid values before we use them in our application or insert them into the database.

If a user has done something wrong, we should let them know with error message(s).

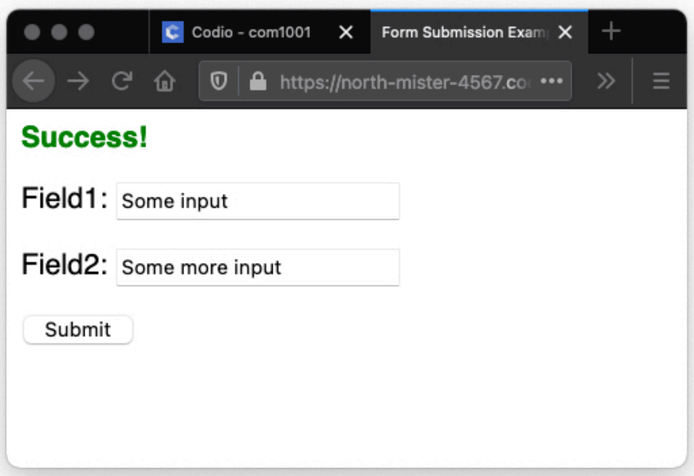
Sanitisation means cleaning user inputs before using them. Sometimes a user has entered a correct input, but has added some trailing spaces or used some characters that may interfere with the operation of our application. These need to be removed or escaped.

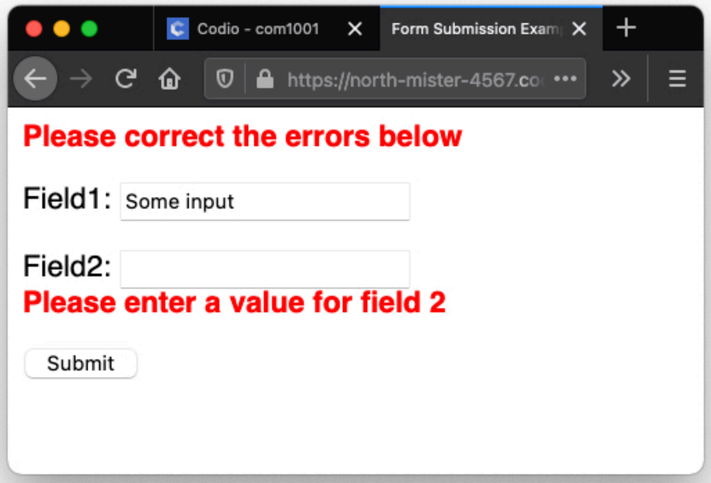
Client-Side or Server-Side?

While it's true that the client (your browser) can check for valid inputs using JavaScript, it's not safe to rely on this method alone, as the **user may turn off scripting in their browser** – leaving your application open to potential attack.

Client-side scripting should always enhance the user experience rather than it being used to perform key functionality.

We therefore need to sanitise and validate inputs, server-side, in our Sinatra applications.



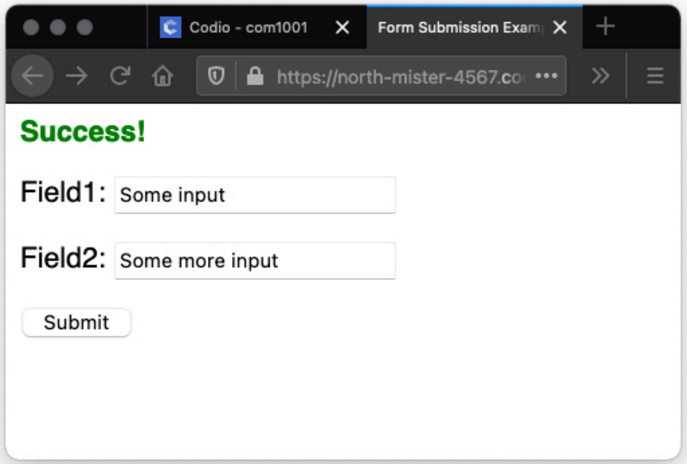


A View with Validation

```
<html>
  <head>
    <title>Form Submission Example</title>
    <link rel="stylesheet" href="style/style.css">
  </head>
  <body>
    <% if @form_was_submitted %>
      <% if @submission_error %>
        <p><strong class="error">Please correct the errors below</strong></p>
      <% else %>
        <p><strong class="success">Success!</strong></p>
      <% end %>
    <% end %>
    <form>
      <p>
        Field1: <input type="text" name="field1" value="<%= @field1 %>" />
        <% if @field1_error %>
          <br /><strong class="error"><%= @field1_error %></strong>
        <% end %>
      </p>
      <p>
        Field2: <input type="text" name="field2" value="<%= @field2 %>" />
        <% if @field2_error %>
          <br /><strong class="error"><%= @field2_error %></strong>
        <% end %>
      </p>
      <p><input type="submit" value="Submit"></p>
    </form>
  </body>
</html>
```

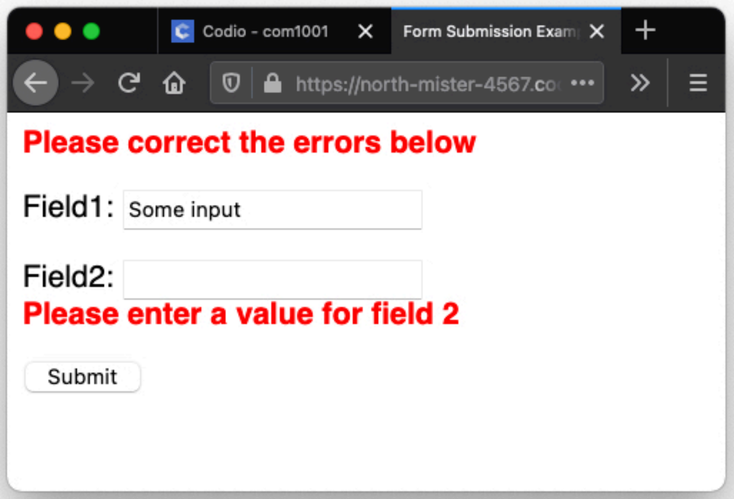
forms/validation_and_sanitisation/views/form.erb

If everything was entered correctly, the page displays a success message.



```html
<html>
  <head>
    <title>Form Submission Example</title>
    <link rel="stylesheet" href="style/style.css">
  </head>
  <body>
    <% if @form_was_submitted %>
      <% if @submission_error %>
        <p><strong class="error">Please correct the errors below</strong></p>
      <% else %>
        <p><strong class="success">Success!</strong></p>
      <% end %>
    <% end %>
    <form>
      <p>
        Field1: <input type="text" name="field1" value="<%= @field1 %>" />
        <% if @field1_error %>
          <br /><strong class="error"><%= @field1_error %></strong>
        <% end %>
      </p>
      <p>
        Field2: <input type="text" name="field2" value="<%= @field2 %>" />
        <% if @field2_error %>
          <br /><strong class="error"><%= @field2_error %></strong>
        <% end %>
      </p>
      <p><input type="submit" value="Submit"></p>
    </form>
  </body>
</html>
```

forms/validation_and_sanitisation/views/form.erb



The `form` tag has no attributes. That means that by default, it's using the `get` method, and the action is to send the data to the same URL as the form.

The view will conditionally show error messages depending on the values of certain variables, such as here.

The Controller

In this example, the **controller** is managing the validation and sanitisation. As we'll see later, it can also be performed by the **model**.

Read the params into local variables.

Since this route is sent its own form data, we don't know (without checking) whether the form is being viewed for the first time or not. This variable checks for the presence of submitted data.

If there was data sent, we can proceed to sanitisation and validation.

Sanitisation in this example is just removing leading and trailing whitespace with `strip!`

Here, we set the appropriate variables for the view if there are validation errors, asking the user to re-enter the data.

```ruby
get "/form" do
  @field1 = params["field1"]
  @field2 = params["field2"]

  @form_was_submitted = !@field1.nil? || !@field2.nil?

  @submission_error = nil
  @field1_error = nil
  @field2_error = nil

  if @form_was_submitted
    # sanitise the values by removing whitespace
    @field1.strip!
    @field2.strip!

    # now proceed to validation
    @field1_error = "Please enter a value for field 1" if @field1.empty?
    @field2_error = "Please enter a value for field 2" if @field2.empty?
    @submission_error = "Please correct the errors below" unless @field1_error.nil? && @field2_error.nil?
  end

  erb :form
end
```

forms/validation_and_sanitisation/controllers/form.rb

Unit Testing Validation Code

```
describe "Form Submission" do
  it "says Success when the data is ok" do
    get "/form", "field1" => "Some Text", "field2" => "Some Text"
    expect(last_response.body).to include("Success!")
  end

  it "rejects the form when field1 is not filled out" do
    get "/form", "field2" => "Some Text"
    expect(last_response.body).to include("Please correct the errors below")
    expect(last_response.body).to include("Please enter a value for field 1")
  end

  it "rejects the form when field2 is not filled out" do
    get "/form", "field1" => "Some Text"
    expect(last_response.body).to include("Please correct the errors below")
    expect(last_response.body).to include("Please enter a value for field 2")
  end
end
```

`forms/validation_and_sanitisation/spec/unit/form_spec.rb`

Thinking of a route in terms of a function (`params` = **input**, **HTML = output**) is a good idea when it comes to unit testing routes.

A page does not necessarily always receive its data via a user, via a form. It could receive the inputs directly via a script or robot.

So considering how the route should behave with different types of input or missing inputs is a good idea and can uncover logic bugs in your code.