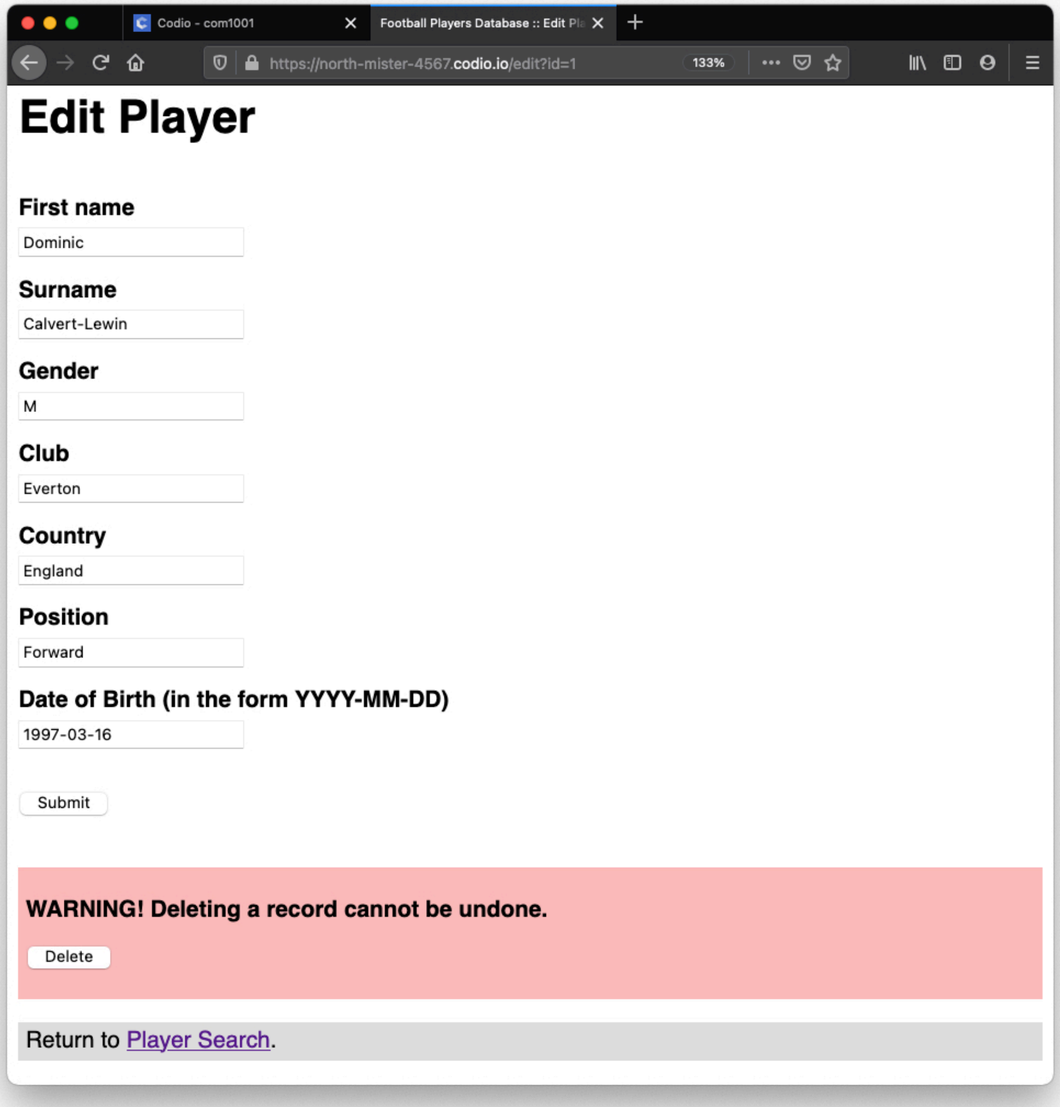
COM 1001

Introduction to Software Engineering

Professor Phil McMinn

Forms, `get` and `post`

Forms

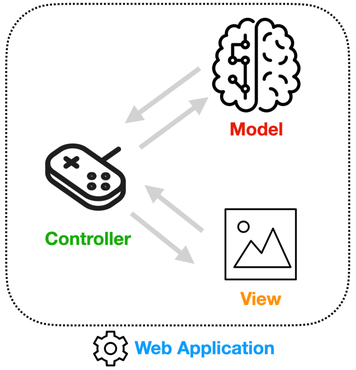

Codio - com1001
Football Players Database :: Edit Pl
https://north-mister-4567.codio.io/edit?id=1
133%
Edit Player
First name
Dominic
Surname
Calvert-Lewin
Gender
M
Club
Everton
Country
England
Position
Forward
Date of Birth (in the form YYYY-MM-DD)
1997-03-16
Submit
WARNING! Deleting a record cannot be undone.
Delete
Return to Player Search.

The Life of a Form

1

User goes to the URL of a web page with a form.

2

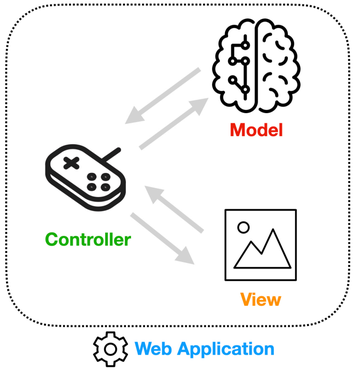


Sinatra maps requested URL to a route that generates web page, including form.

3

User enters data into form. Submitting the form causes the browser to generates another web page request. Browser sends form data as part of the request.

4



Sinatra maps URL to a route. The user's form data is put into a hash called `params`. In the Sinatra block for the route we can write code to process the form data and decide what to do with it…

A Simple Form – HTML

The `method` attribute for the `<form>` HTML tag specifies what HTTP method to use. Until now we've only used `get`.

We can leave this attribute out and the browser will assume we mean `get`.

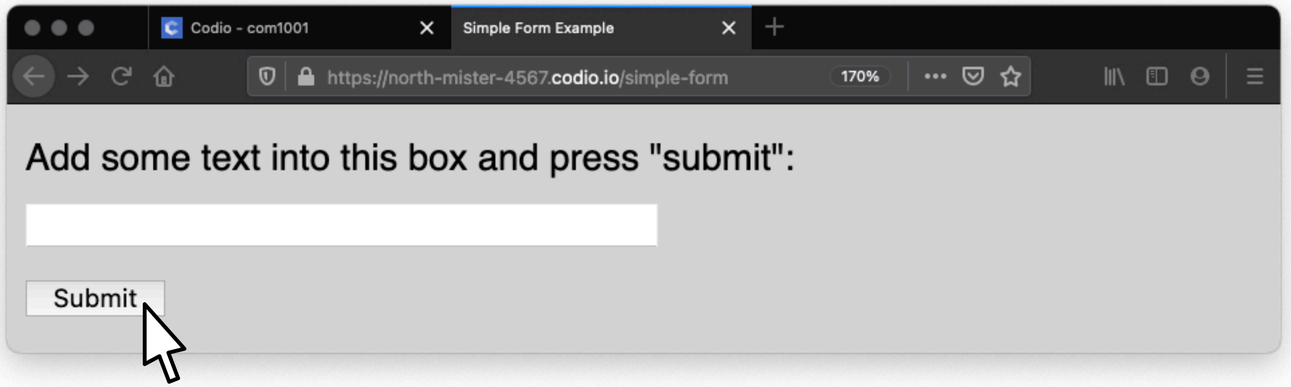
We need some form fields – HTML elements where the user can enter some data. Here, we have a text field. This is specified by the `type="text"` attribute. The name attribute is also important, as we'll see later.

```
<html>
  <head>
    <title>Simple Form Example</title>
    <link rel="stylesheet" href="style/style.css">
  </head>
  <body>
    <form method="get" action="/process-simple-form">
      <p>
        Add some text into this box and press "submit": <br />
        <input type="text" name="text_field" />
      </p>
      <p><input type="submit" value="Submit"></p>
    </form>
  </body>
</html>
```

forms/simple_forms/view/simple_form.erb

The `action` attribute for the `<form>` HTML tag specifies the URL that the browser should request when the form is submitted, and where the form data should be sent.

This form field, with `type="submit"`, is the submit button. When the user clicks this button, the browser is directed to the URL of the action parameter of the form, with the form data (i.e., the value in the text field in this example).



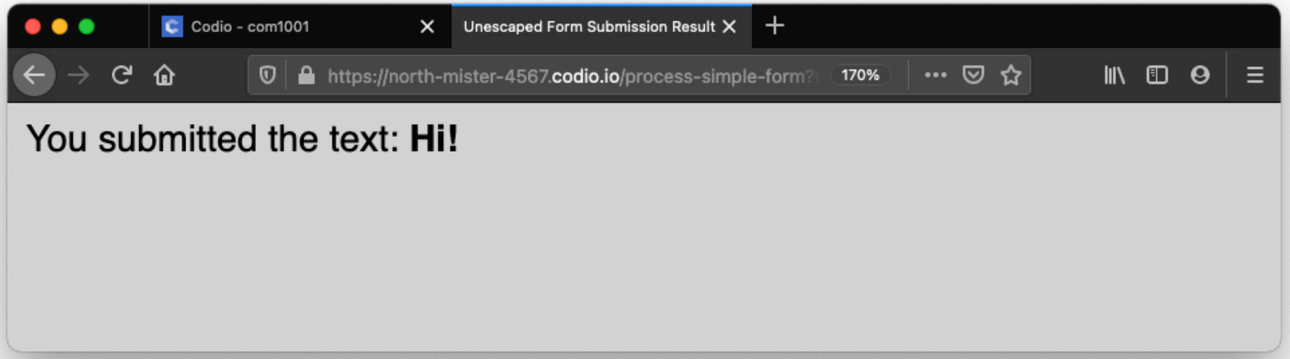

```ruby
get "/simple-form" do
  erb :simple_form
end

get "/process-simple-form" do
  @submitted_text_field_value = params["text_field"]
  erb :process_simple_form
end
```

forms/simple_forms/controllers/simple_form.rb

```html
<html>
  <head>
    <title>Simple Form Example</title>
    <link rel="stylesheet" href="style/style.css">
  </head>
  <body>
    <form method="get" action="/process-simple-form">
      <p>
        Add some text into this box and press "submit": <br />
        <input type="text" name="text_field" />
      </p>
      <p><input type="submit" value="Submit"></p>
    </form>
  </body>
</html>
```

The form is submitted to `/process-simple-form`. The Sinatra route picks up the form data in a hash called `params`. The keys of the hash of the names of the form fields from the original form, and the values consist of the data that the user entered into the form.



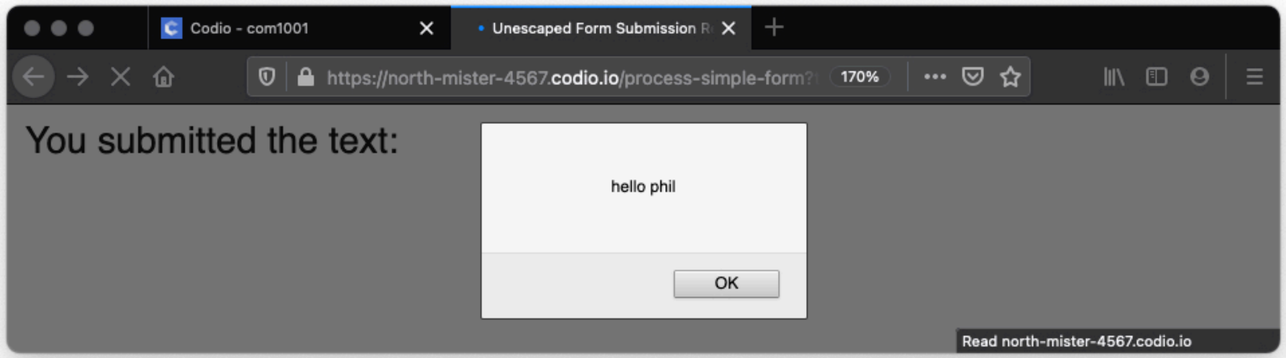

```ruby
get "/simple-form" do
  erb :simple_form
end

get "/process-simple-form" do
  @submitted_text_field_value = params["text_field"]
  erb :process_simple_form
end
```

```erb
<html>
  <head>
    <title>Unescaped Form Submission Result</title>
    <link rel="stylesheet" href="style/style.css">
  </head>
  <body>
    <% unless @submitted_text_field_value.nil? %>
      <p>You submitted the text:<strong>
      <%= @submitted_text_field_value %>
      </strong></p>
    <% end %>
  </body>
</html>
```

forms/simple_forms/view/unescaped_submission.erb

Never Trust User Inputs!



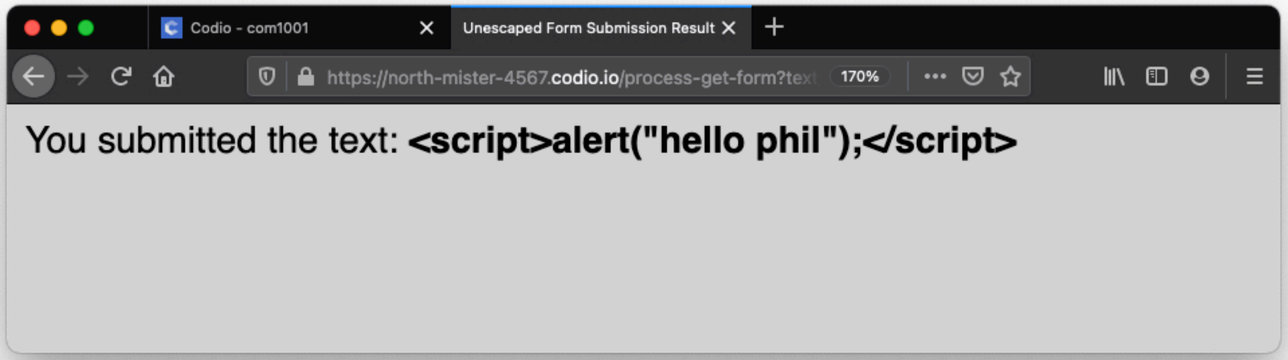
```erb
<html>
  <head>
    <title>Escaped Form Submission Result</title>
    <link rel="stylesheet" href="style/style.css">
  </head>
  <body>
    <% unless @submitted_text_field_value.nil? %>
      <p>You submitted the text:<strong>
      <%= h @submitted_text_field_value %>
      </strong></p>
    <% end %>
  </body>
</html>
```

forms/simple_forms/view/escaped_submission.erb

Always Escape Values of Untrusted Variables in the View

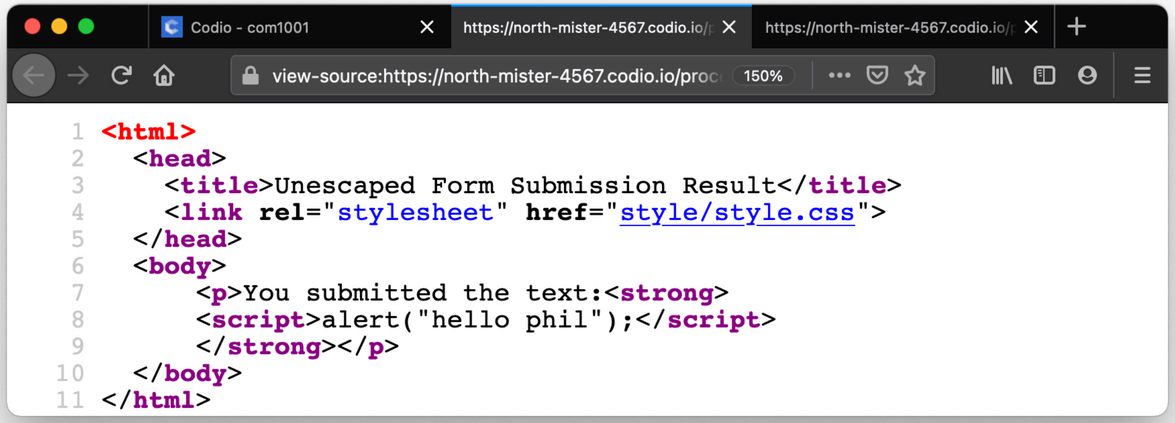
Recall how we escaped HTML special characters from database values using the h **method.**

We need to do this for form values too. This will prevent the browser interpreting any user inputs as part of the HTML of the page, such as maliciously injecting scripts and potentially compromising the security of our web application.

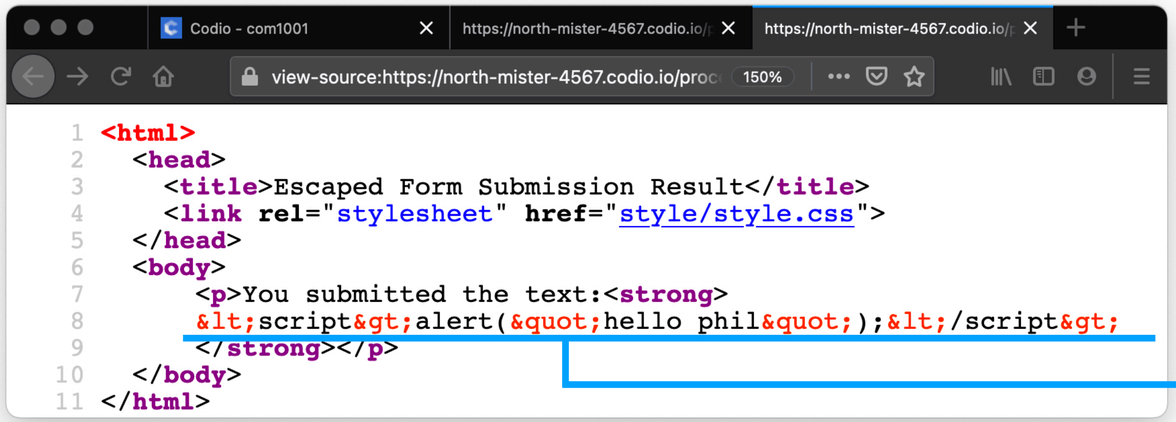


The Great Escape

Unescaped (view not using h):



Escaped (view uses h):



We can see the impact of escaping HTML characters if we view the source of the generated page.

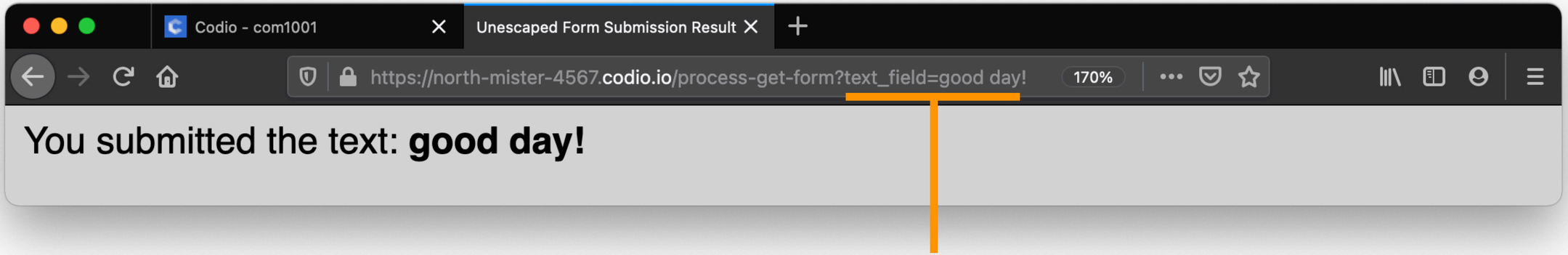
Firefox: Tools → Web Developer → Page Source

Chrome: View → Developer → View Source

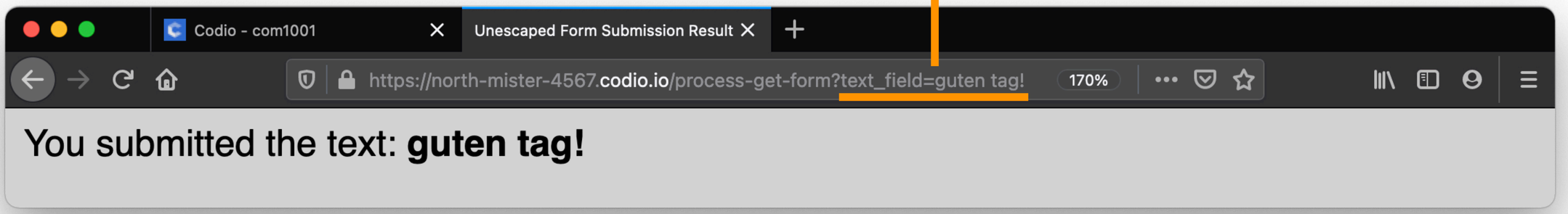
Others: Google it!

Special HTML characters like “<”and “>” are replaced with equivalent HTML “entities” (`<` and `>` respectively) that will render those characters as text, so as not to confuse the browser into thinking they are part of the HTML of the page.

The `params` hash for `get`



Check out the URL in your browser when a form is submitted via get. The URL contains the data you submitted. If you edit the part of the URL after the "`text_field=`" and hit enter, you'll realise you don't have to use the form to send data to the receiving route and for it to end up in the `params` hash. This is another good reason **not to trust any raw inputs** that you retrieve from the the `params` hash!



The post Form Submission Method

```
<html>
  <head>
    <title>POST Form Example</title>
    <link rel="stylesheet" href="style/style.css">
  </head>
  <body>
    <form method="post" action="/process-post-form">
      <p>
        Add some text into this box and press "submit": <br />
        <input type="text" name="text_field" />
      </p>
      <p><input type="submit" value="Submit"></p>
    </form>
  </body>
</html>
```

forms/simple_forms/view/simple_form.erb

The post form submission method is an alternative to get that does not expose form data as part of the submission URL.

To use the post method, we set it as the method attribute in the form

The `post` HTTP method

In order to handle the form submission sent using the `post` method, we need to use the `post` verb to prefix the route in our Sinatra app.

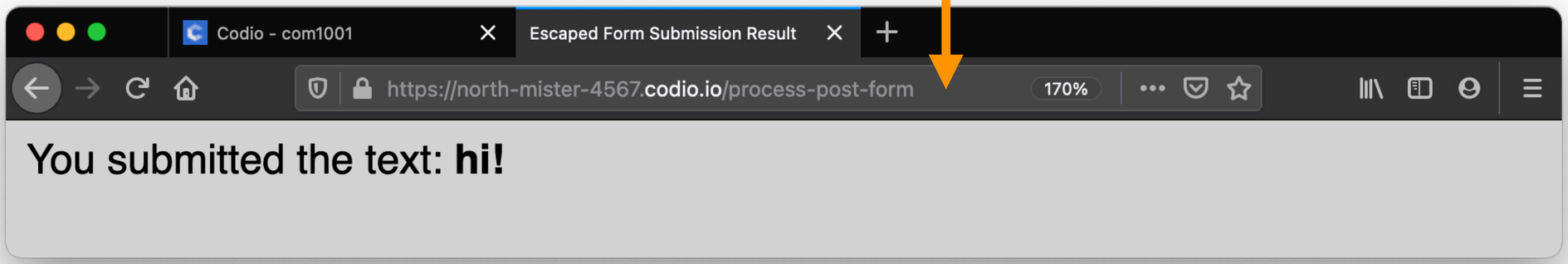
```ruby
get "/post-form" do
  erb :post_form
end

post "/process-post-form" do
  @submitted_text_field_value = params["text_field"]
  erb :escaped_form_submission
end
```

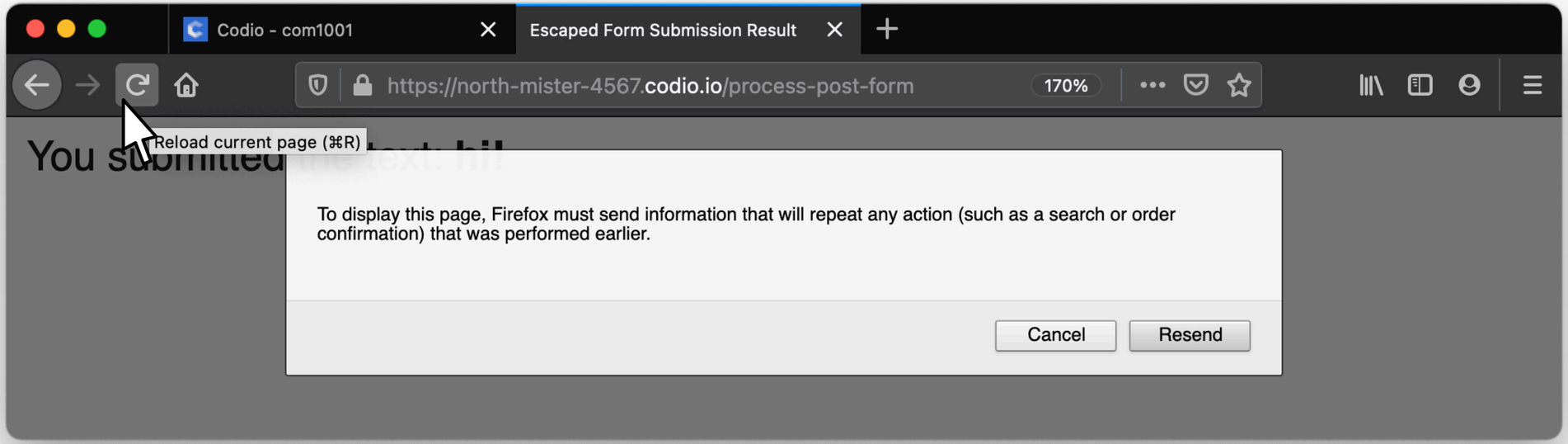
forms/simple_forms/controllers/post_form.rb

This route is inaccessible by typing the URL into your browser, since the browser will only generate a `get` request, which will not match this route, since it uses `post` instead.

The `post` form submission method does not expose form data as part of the submission URL



Users cannot resubmit the form without the browser specially asking them whether they wish to repeat the action:



Characteristics of a `get` Form Submission

Since the data is displayed in the URL, the form submission…

… can be bookmarked (useful for repeating search queries)

… remains in the browser's history

… can be cached by the browser (no need to request the page repeatedly)

But:

- Data is restricted to **text only** (ASCII)
- **Security** is very weak – data can be stored in server logs etc.

Characteristics of a `post` Form Submission

Since the data is *not* displayed in the URL, the form submission…

… cannot be bookmarked (useful for repeating search queries)

… does not remain in the browser's history

… cannot be cached by the browser

However:

- Data is can be **binary** (allows for document uploads…)
- **Security** is stronger – especially when connections are encrypted with SSL – useful for logging in with confidential credentials.

get or post?

	get	post
Cachable by Browser?	Yes	No
Remain in Browser History?	Yes	No
Bookmarkable?	Yes	No
Restriction on Length?	Yes	No
Restriction on Data?	ASCII only	Binary Allowed
Data displayed in URL?	Yes	No
Security	Weak – data part of URL, can be cached, bookmarked, stored in web server logs etc.	Stronger – especially when connections are encrypted with SSL

How to Decide Whether to Use `get` or `post`

`get` works well for search queries on insensitive data:

- queries can be bookmarked
- direct URLs can be constructed for linking to specific search results (and can be embedded in `<a href="...">` links)

`post` works best when a user needs to:

- submit sensitive data (e.g. logging into a system)
- or is providing one-time information (e.g., job application data) or performing a one-time action (e.g., deleting some data)

Query Strings

As we've already seen, we don't need a form to submit parameters via `get`. We can just craft a URL.

The form of the URL is always `some_url?key1=value1&key2=value2`

That is, the parameters to a get request are provided by suffixing the URL with a "`?`", and then providing the key-value pairs in the form `key=value`, separated by "`&`"s.

The part after the "`?`" is called the **query string**.

Crafting a Query String

```ruby
require "uri"

get "/construct-querystring" do
  person = { "name" => "Phil McMinn",
             "job" => "Professor of Software Engineering",
             "address" => "Regent Court",
             "age" => "That would be telling..." }

  @querystring = URI.encode_www_form(person)

  erb :construct_querystring
end
```

forms/simple_forms/controllers/querystring.rb
(part 1/2)

Sometimes it's useful to pass values from one page to another via the URLs in `<a href="...">` tags. We can turn any arbitrary hash into a query string using the `URI.encode_www_form` method

```erb
<a href="/process-querystring?<%= h @querystring %>">Click me!</a>
```

forms/simple_forms/views/construct_querystring.erb

We always need to escape the query string in the view, since "&" (the key-value separator in a query string) is a HTML special character. This will only work properly if we've not already escaped any key-value pairs.

The value of `@querystring` is:

```
/process-querystring?name=Phil+McMinn&job=Professor+of+Software+Engineering&address=Regent+Court&age=That+would+be+telling...
```

```ruby
get "/process-querystring" do
  @name = params["name"]
  @job = params["job"]
  @address = params["address"]
  @age = params["age"]

  erb :process_querystring
end
```

`forms/simple_forms/controllers/querystring.rb`
(part 2/2)

The route corresponding to the URL we sent the query string with can then unpack the `params` hash and get the values we originally sent.

Note this code is no different to that had the values had been submitted by a form – the receiving route has no idea of the context, or how the data got in the `params` hash – just that it is there!

How Do We Unit Test Forms?

```ruby
describe "Get Form Example" do
  it "writes the submitted string to the page" do
    get "/process-get-form", "text_field" => "Some Text"
    expect(last_response.body).to include("Some Text")
  end
end

describe "Post Form Example" do
  it "writes the submitted string to the page" do
    post "/process-post-form", "text_field" => "Some Text"
    expect(last_response.body).to include("Some Text")
  end
end
```

forms/simple_forms/spec/unit/simple_forms_spec.rb

Testing the act of submitting a form is out of the scope of a unit test (we'll be learning later how to do this, however as part of acceptance testing).

We **can** test what our web application does when it receives data via `get` or `post`, as in these examples.

We provide the contents of the intended `params` hash as extra parameters to the route, and check that it responds with those parameters as we would expect.