HTML forms

In HTML, a form is a collection of elements inside <form>...</form> that allow a visitor to do things like

enter text, select options, manipulate objects or controls, and so on, and then send that information back to

the server.

Some of these form interface elements - text input or checkboxes - are built into HTML itself. Others are much

more complex; an interface that pops up a date picker or allows you to move a slider or manipulate controls

will typically use JavaScript and CSS as well as HTML form <input> elements to achieve these effects.

As well as its <input> elements, a form must specify two things:

• where: the **URL** to which the data corresponding to the user’s input should be returned

• how: the **HTTP** method the data should be returned by

As an example, the login form for the Django admin contains several <input> elements: one of **type="text"**

for the username, one of **type="password"** for the password, and one of **type="submit"** for the **“Log in”**

button. It also contains some hidden text fields that the user doesn’t see, which Django uses to determine

what to do next.

It also tells the browser that the form data should be sent to the URL specified in the <form>’s action attribute

**- /admin/** - and that it should be sent using the HTTP mechanism specified by the method attribute - post.

When the **<input type="submit" value="Log in**"> element is triggered, the data is returned to /admin/

# GET and POST - - -

**GET** and **POST** are the only **HTTP** methods to use when dealing with forms.

Django’s login form is returned using the **POST** method, in which the browser bundles up the form data,

encodes it for transmission, sends it to the server, and then receives back its response.

**GET**, by contrast, bundles the submitted data into a string, and uses this to compose a URL. The URL contains

the address where the data must be sent, as well as the data keys and values. You can see this in action

if you do a search in the Django documentation, which will produce a URL of the form **https://docs.**

**djangoproject.com/search/?q=forms&release=1**.

**GET** and **POST** are typically used for different purposes.

Any request that could be used to change the state of the system - for example, a request that makes changes

in the database - should use **POST**. **GET** should be used only for requests that do not affect the state of the

system.

**GET** would also be unsuitable for a password form, because the password would appear in the **URL**, and thus,

also in browser history and server logs, all in plain text. Neither would it be suitable for large quantities of

data, or for binary data, such as an image. A web application that uses GET requests for admin forms is a

security risk: it can be easy for an attacker to mimic a form’s request to gain access to sensitive parts of the

system. **POST**, coupled with other protections like Django’s **CSRF** protection offers more control over access.

On the other hand, **GET** is suitable for things like a web search form, because the URLs that represent a GET

request can easily be bookmarked, shared, or resubmitted.

Forms in Django - - -

# The Django Form class - - -

At the heart of this system of components is Django’s Form class. In much the same way that a Django model

describes the logical structure of an object, its behavior, and the way its parts are represented to us, a Form

class describes a form and determines how it works and appears.

In a similar way that a model class’s fields map to database fields, a form class’s fields map to **HTML** form

**<input>** elements. (A **ModelForm** maps a model class’s fields to **HTML** form **<input>** elements via a Form;

this is what the Django admin is based upon.)

A form’s fields are themselves classes; they manage form data and perform validation when a form is submitted. A **DateField** and a **FileField** handle very different kinds of data and have to do different things

with it.

A form field is represented to a user in the browser as an **HTML** “**widget**” - a piece of user interface machinery.

Each field type has an appropriate default Widget class, but these can be overridden as required

# Instantiating, processing, and rendering forms - - -

When rendering an object in Django, we generally:

1. get hold of it in the view (fetch it from the database, for example)

2. pass it to the template context

3. expand it to HTML markup using template variables

Rendering a form in a template involves nearly the same work as rendering any other kind of object, but

There are some key differences. In the case of a model instance that contained no data, it would rarely if ever be useful to do anything with it in a template. On the other hand, it makes perfect sense to render an unpopulated form - that’s what we do when we want the user to populate it.

So, when we handle a model instance in a view, we typically retrieve it from the database. When we’re dealing

with a form we typically instantiate it in the view.

When we instantiate a form, we can opt to leave it empty or prepopulate it, for example with:

• data from a saved model instance (as in the case of admin forms for editing)

• data that we have collated from other sources

• data received from a previous HTML form submission

The last of these cases is the most interesting, because it’s what makes it possible for users not just to read a

website, but to send information back to it too.

-------------------------- END ---------------