

Django Tutorial Part 9: Working with forms

Created by 이윤준 (yoonyoon.lee@gmail.com)

June, 2019

We'll show

- how to work with HTML Forms in Django, and, in particular,
- the easiest way to write forms to create, update, and delete model instances.

We'll extend the *LocalLibrary* website so that librarians can renew books, and create, update, and delete authors using our own forms.

Overview

An HTML Form is a group of one or more fields /widgets on a web page, which can be used to collect information from users for submission to a server. Forms are a flexible mechanism for collecting user input because there are suitable widgets for entering many different types of data, including text boxes, checkboxes, radio buttons, date pickers, etc. Forms are also a relatively secure way of sharing data with the server, as they allow us to send data in `POST` requests with cross-site request forgery protection.

We've already encountered them in the Django Admin site.

Add book

Title:

Author:

Summary:




Enter a brief description of the book

ISBN:

13 Character ISBN number

Genre:

Science Fiction
Fantasy
Western
French Poetry

Select a genre for this book Hold down "Control", or "Command" on a Mac, to select more than one.

Save and add another

Save and continue editing

SAVE

Working with forms can be complicated. Developers need to

- write HTML for the form,
- validate and properly sanitise entered data on the server,
- repost the form with error messages to inform users of any invalid fields,
- handle the data when it has successfully been submitted, and
- finally respond to the user in some way to indicate success.

Django Forms take a lot of the work out of all these steps, by providing a framework that lets us define forms and their fields programmatically, and then use these objects to both generate the form HTML code and handle much of the validation and user interaction.

We're going to show

- a few of the ways we can create and work with forms, and
- in particular, how the generic editing form views can significantly reduce the amount of work you need to do to create forms to manipulate your models.

We'll extend our *LocalLibrary* application by adding a form to allow librarians to renew library books, and we'll create pages to create, edit and delete books and authors.

HTML forms

First a brief overview of HTML Forms. Consider a simple HTML form, with a single text field for entering the name of some "team", and its associated label:

Enter name:

The form is defined in HTML as a collection of elements inside `<form>...</form>` tags, containing at least one input element of `type="submit"`.

```
<form action="/team_name_url/" method="post">
  <label for="team_name">Enter name: </label>
  <input id="team_name" type="text"
    name="name_field" value="Default name for team.">
  <input type="submit" value="OK">
</form>
```

While here we just have one text field for entering the team name, a form may have any number of other input elements and their associated labels. The field's `type` attribute defines what sort of widget will be displayed. The `name` and `id` of the field are used to identify the field in JavaScript/CSS/HTML, while `value` defines the initial value for the field when it is first displayed. The matching team label is specified using the `label` tag (see "Enter name"), with a `for` field containing the `id` value of the associated input.

The `submit` input will be displayed as a button that can be pressed by the user to upload the data in all the other input elements in the form to the server (in this case, just the `team_name`). The form attributes define the HTTP `method` used to send the data and the destination of the data on the server (`action`):

- **action**: The resource/URL where data is to be sent for processing when the form is submitted. If this is not set (or set to an empty string), then the form will be submitted back to the current page URL.
- **method**: The HTTP method used to send the data: *post* or *get*.
 - The **POST** method should always be used if the data is going to result in a change to the server's database because this can be made more resistant to cross-site forgery request attacks.
 - The **GET** method should only be used for forms that don't change user data (e.g. a search form). It is recommended for when you want to be able to bookmark or share the URL.

The role of the server is first to render the initial form state — either containing blank fields or pre-populated with initial values.

After the user presses the submit button, the server will receive the form data with values from the web browser and must validate the information. If the form contains invalid data, the server should display the form again, this time with user-entered data in "valid" fields and messages to describe the problem for the invalid fields.

Once the server gets a request with all valid form data, it can perform an appropriate action (e.g. saving the data, returning the result of a search, uploading a file etc.) and then notify the user.

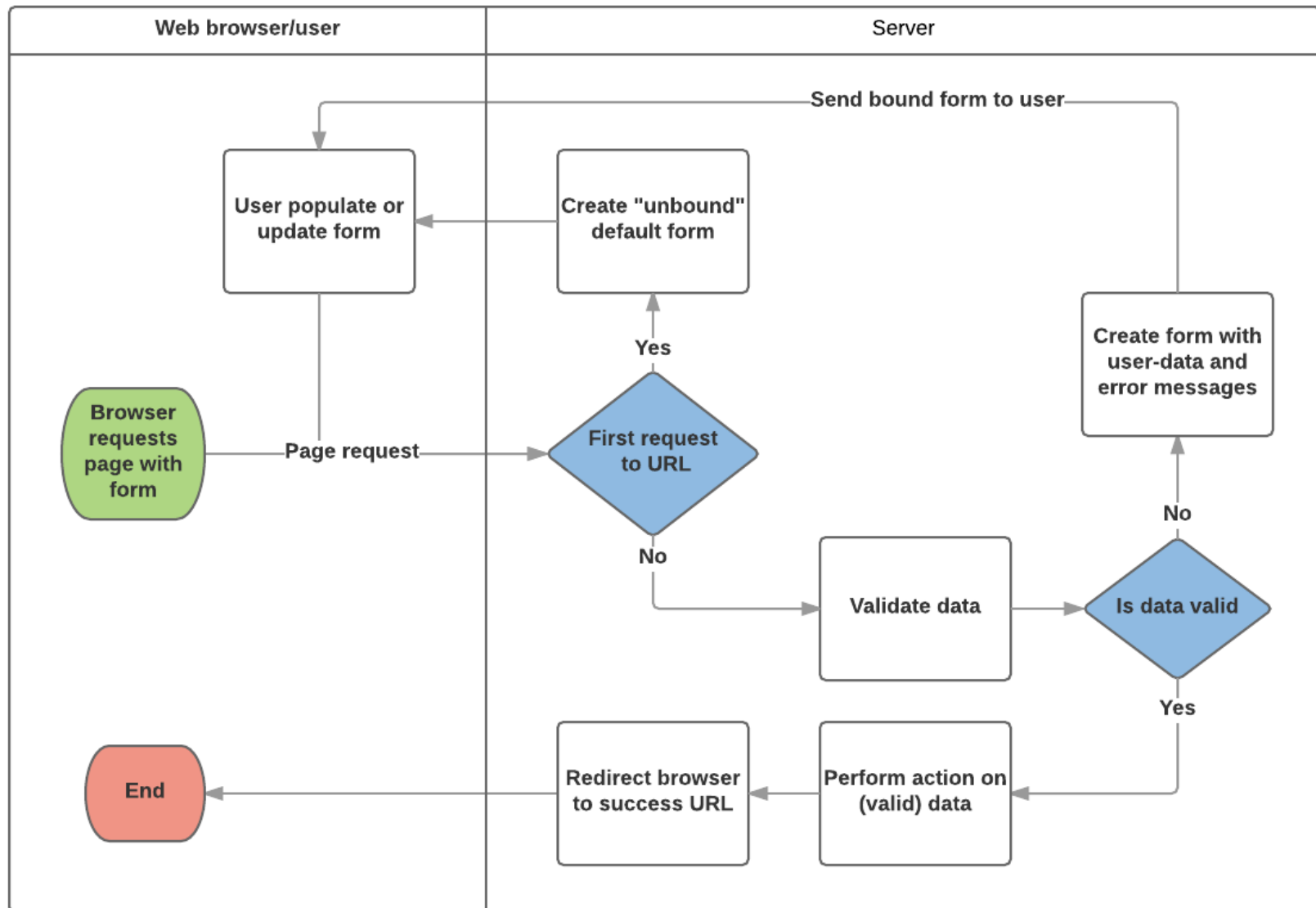
As we can imagine, creating the HTML, validating the returned data, re-displaying the entered data with error reports if needed, and performing the desired operation on valid data can all take quite a lot of effort to "get right". Django makes this a lot easier, by taking away some of the heavy lifting and repetitive code.

Django form handling process

Django's form handling uses all of the same techniques:

- the view gets a request, performs any actions required including reading data from the models,
- then generates and returns an HTML page (from a template, into which we pass a context containing the data to be displayed).

What makes things more complicated is that the server also needs to be able to process data provided by the user, and redisplay the page if there are any errors.



Based on the diagram above, the main things that Django's form handling does are:

1. Display the default form the first time it is requested by the user.
 - The form may contain blank fields (e.g. if you're creating a new record), or it may be pre-populated with initial values (e.g. if you are changing a record, or have useful default initial values).
 - The form is referred to as unbound at this point, because it isn't associated with any user-entered data (though it may have initial values).
2. Receive data from a submit request and bind it to the form.
 - Binding data to the form means that the user-entered data and any errors are available when we need to redisplay the form.

3. Clean and validate the data.

- Cleaning the data performs sanitization of the input (e.g. removing invalid characters that might be used to send malicious content to the server) and converts them into consistent Python types.
- Validation checks that the values are appropriate for the field (e.g. are in the right date range, aren't too short or too long, etc.)

4. If any data is invalid, re-display the form, this time with any user populated values and error messages for the problem fields.

5. If all data is valid, perform required actions (e.g. save the data, send an email, return the result of a search, upload a file etc.)

6. Once all actions are complete, redirect the user to another page.

Django provides a number of tools and approaches to help you with the tasks detailed above. The most fundamental is the `Form` class, which simplifies both generation of form HTML and data cleaning/validation.

We describe how forms work using the practical example of a page to allow librarians to renew books.

Renew-book form using a Form and function view

We're going to add a page to allow librarians to renew borrowed books.

- Create a form that allows users to enter a date value.
- Seed the field with an initial value 3 weeks from the current date, and add some validation to ensure that the librarian can't enter a date in the past or a date too far in the future.
- When a valid date has been entered, write it to the current record's `BookInstance.due_back` field.

The example will use a function-based view and a `Form` class. We explain how forms work, and the changes we need to make to our ongoing *LocalLibrary* project.

Form

The `Form` class is the heart of Django's form handling system.

It specifies the fields in the form, their layout, display widgets, labels, initial values, valid values, and (once validated) the error messages associated with invalid fields.

The class also provides methods for rendering itself in templates using predefined formats (tables, lists, etc.) or for getting the value of any element (enabling fine-grained manual rendering).

Declaring a Form

The declaration syntax for a `Form` is very similar to that for declaring a `Model`, and shares the same field types (and some similar parameters).

This makes sense because in both cases we need to ensure that each field handles the right types of data, is constrained to valid data, and has a description for display/documentation.

Form data is stored in an application's [forms.py](#) file, inside the application directory. Create and open the file **locallibrary/catalog/forms.py**.

To create a `Form`, we import the `forms` library, derive from the `Form` class, and declare the form's fields. A very basic form class for our library book renewal form is shown below — add this to your new file:

```
from django import forms

class RenewBookForm(forms.Form):
    renewal_date = forms.DateField(help_text =
        "Enter a date between now and 4 weeks (default 3).")
```


Form fields

We have a single `DateField` for entering the renewal date that will render in HTML with a blank value, the default label "*Renewal date:*", and some helpful usage text: "*Enter a date between now and 4 weeks (default 3 weeks).*"

As none of the other optional arguments are specified the field will accept dates using the `input_formats`: `YYYY-MM-DD` (2016-11-06), `MM/DD/YYYY` (02/26/2016), `MM/DD/YY` (10/25/16), and will be rendered using the default widget: `DateInput`.

There are many other types of form fields, which you will largely recognise from their similarity to the equivalent model field classes:

BooleanField, CharField, ChoiceField,
TypedChoiceField, DateField, DateTimeField,
DecimalField, DurationField, EmailField, FileField,
FilePathField, FloatField, ImageField, IntegerField,
GenericIPAddressField, MultipleChoiceField,
TypedMultipleChoiceField, NullBooleanField,
RegexField, SlugField, TimeField, URLField,
UUIDField, ComboField, MultiValueField,
SplitDateTimeField, ModelMultipleChoiceField,
ModelChoiceField.

The arguments that are common to most fields are listed:

- `required`: If `True`, the field may not be left blank or given a `None` value. Fields are required by default, so you would set `required=False` to allow blank values in the form.
- `label`: The label to use when rendering the field in HTML. If a label is not specified, Django will create one from the field name by capitalizing the first letter and replacing underscores with spaces (e.g. *Renewal date*).
- `label_suffix`: By default a colon is displayed after the label (e.g. Renewal date:). This argument allows you to specify a different suffix containing other character(s).
- `initial`: The initial value for the field when the form is displayed.
- `widget`: The display widget to use.
- `help_text`: Additional text that can be displayed in forms to explain how to use the field.

- `error_messages`: A list of error messages for the field. You can override these with your own messages if needed.
- `validators`: A list of functions that will be called on the field when it is validated.
- `localize`: Enables the localization of form data input.
- `disabled`: The field is displayed but its value cannot be edited if this is `True`. The default is `False`.

Validation

Django provides numerous places where we can validate our data. The easiest way to validate a single field is to override the method `clean_<fieldname>()` for the field you want to check. So for example, we can validate that entered `renewal_date` values are between now and 4 weeks by implementing `clean_renewal_date()`.

Update your [forms.py](#) file:

```
import datetime

from django import forms
from django.core.exceptions import ValidationError
from django.utils.translation import ugettext_lazy as _

class RenewBookForm(forms.Form):
    renewal_date = forms.DateField(help_text=
        "Enter a date between now and 4 weeks (default 3).")

    def clean_renewal_date(self):
        data = self.cleaned_data['renewal_date']

        # Check if a date is not in the past.
        if data < datetime.date.today():
            raise ValidationError(_('Invalid date - renewal in past'))

        # Check if a date is in the allowed range (+4 weeks from today).
        if data > datetime.date.today() +
            datetime.timedelta(weeks=4):
            raise ValidationError(
                _('Invalid date - renewal more than 4 weeks ahead'))

        # Remember to always return the cleaned data.
        return data
```

There are two important things to note.

1. We get our data using `self.cleaned_data['renewal_date']` and that we return this data whether or not we change it at the end of the function. This step gets us the data "cleaned" and sanitized of potentially unsafe input using the default validators, and converted into the correct standard type for the data (in this case a Python `datetime.datetime` object).
2. If a value falls outside our range we raise a `ValidationError`, specifying the error text that we want to display in the form if an invalid value is entered. The example above also wraps this text in one of Django's translation functions `ugettext_lazy()` (imported as `_()`), which is good practice if we want to translate our site later.

URL Configuration

Before we create our view, let's add a URL configuration for the *renew-books* page. Copy the configuration to the bottom of **locallibrary/catalog/urls.py**.

```
urlpatterns += [  
    path(  
        'book/<uuid:pk>/renew/',  
        views.renew_book_librarian,  
        name='renew-book-librarian'),  
]
```


The URL configuration will redirect URLs with the format **/catalog/book/<bookinstance id>/renew/** to the function named `renew_book_librarian()` in **views.py**, and send the `BookInstance` id as the parameter named `pk`. The pattern only matches if `pk` is a correctly formatted `uuid`.

View

The view has to render the default form when it is first called and then either re-render it with error messages if the data is invalid, or process the data and redirect to a new page if the data is valid.

In order to perform these different actions, the view has to be able to know whether it is being called for the first time to render the default form, or a subsequent time to validate data.

For forms that use a `POST` request to submit information to the server, the most common pattern is for the view to test against the `POST` request type (if `request.method == 'POST' :`) to identify form validation requests and `GET` (using an `else` condition) to identify the initial form creation request.

If we want to submit our data using a `GET` request then a typical approach for identifying whether this is the first or subsequent view invocation is to read the form data (e.g. to read a hidden value in the form).

The book renewal process will be writing to our database, so, by convention, we use the **POST** request approach. The code fragment shows the (very standard) pattern for this sort of function view.

```
import datetime

from django.shortcuts import render, get_object_or_404
from django.http import HttpResponseRedirect
from django.urls import reverse

from catalog.forms import RenewBookForm

def renew_book_librarian(request, pk):
    book_instance = get_object_or_404(BookInstance, pk=pk)
```

```
# If this is a POST request then process the Form data
if request.method == 'POST':

    # Create a form instance and populate it with data from the request
    # (binding):
    form = RenewBookForm(request.POST)

    # Check if the form is valid:
    if form.is_valid():
        # process the data in form.cleaned_data as required
        # (here we just write it to the model due_back field)
        book_instance.due_back = form.cleaned_data['renewal_date']
        book_instance.save()

        # redirect to a new URL:
        return HttpResponseRedirect(reverse('all-borrowed') )

# If this is a GET (or any other method) create the default form.
else:
    proposed_renewal_date = datetime.date.today()
        + datetime.timedelta(weeks=3)
    form = RenewBookForm(initial={'renewal_date': proposed_renewal_date})

context = {
    'form': form,
    'book_instance': book_instance,
}

return render(request, 'catalog/book_renew_librarian.html', context)
```

First, we import our form (`RenewBookForm`) and a number of other useful objects/methods used in the body of the view function:

- `get_object_or_404()`: Returns a specified object from a model based on its primary key value, and raises an `Http404` exception (not found) if the record does not exist.
- `HttpResponseRedirect`: This creates a redirect to a specified URL (HTTP status code 302).
- `reverse()`: This generates a URL from a URL configuration name and a set of arguments. It is the Python equivalent of the `url` tag that we've been using in our templates.
- `datetime`: A Python library for manipulating dates and times.

In the view we first use the `pk` argument in `get_object_or_404()` to get the current `BookInstance` (if this does not exist, the view will immediately exit and the page will display a "not found" error).

If this is not a `POST` request (handled by the `else` clause) then we create the default form passing in an `initial` value for the `renewal_date` field (as shown, this is 3 weeks from the current date).

```
book_instance = get_object_or_404(BookInstance, pk=pk)

# If this is a GET (or any other method) create the default form
else:
    proposed_renewal_date = datetime.date.today()
        + datetime.timedelta(weeks=3)
    form = RenewBookForm(initial={'renewal_date': proposed_renewal_date})

context = {
    'form': form,
    'book_instance': book_instance,
}

return render(request, 'catalog/book_renew_librarian.html', context)
```

After creating the form, we call `render()` to create the HTML page, specifying the template and a context that contains our form. In this case, the context also contains our `BookInstance`, which we'll use in the template to provide information about the book we're renewing.

However, if this is a `POST` request, then we create our `form` object and populate it with data from the request. This process is called "binding" and allows us to validate the form.

We then check if the form is valid, which runs all the validation code on all of the fields — including both the generic code to check that our date field is actually a valid date and our specific form's `clean_renewal_date()` function to check the date is in the right range.

```
book_instance = get_object_or_404(BookInstance, pk=pk)

# If this is a POST request then process the Form data
if request.method == 'POST':

    # Create a form instance and populate it with data
    # from the request (binding):
    form = RenewBookForm(request.POST)

    # Check if the form is valid:
    if form.is_valid():
        # process the data in form.cleaned_data as required
        # (here we just write it to the model due_back field)
        book_instance.due_back = form.cleaned_data['renewal_date']
        book_instance.save()

        # redirect to a new URL:
        return HttpResponseRedirect(reverse('all-borrowed') )

context = {
    'form': form,
    'book_instance': book_instance,
}

return render(request, 'catalog/book_renew_librarian.html', context)
```

If the form is not valid we call `render()` again, but this time the form value passed in the context will include error messages.

If the form is valid, then we can start to use the data, accessing it through the `form.cleaned_data` attribute (e.g. `data = form.cleaned_data['renewal_date']`). Here we just save the data into the `due_back` value of the associated `BookInstance` object.

The final step in the form-handling part of the view is to redirect to another page, usually a "success" page. In this case we use `HttpResponseRedirect` and `reverse()` to redirect to the view named `'all-borrowed'`. If you didn't create that page consider redirecting to the home page at URL `'/'`).

We still need to restrict access to the view to librarians. We should probably create a new permission in `BookInstance` ("`can_renew`"), but, to keep things simple here, we just use the `@permission_required` function decorator with our existing `can_mark_returned` permission.

The final view is as shown. Please copy this into the bottom of **`locallibrary/catalog/views.py`**.

```
import datetime

from django.contrib.auth.decorators import permission_required
from django.shortcuts import get_object_or_404
from django.http import HttpResponseRedirect
from django.urls import reverse

from catalog.forms import RenewBookForm

@permission_required('catalog.can_mark_returned')
def renew_book_librarian(request, pk):
    """View function for renewing a specific BookInstance by librarian."""
    book_instance = get_object_or_404(BookInstance, pk=pk)

    # If this is a POST request then process the Form data
    if request.method == 'POST':

        # Create a form instance and populate it
        # with data from the request (binding):
        form = RenewBookForm(request.POST)

        # Check if the form is valid:
        if form.is_valid():
            # process the data in form.cleaned_data
            # as required (here we just write it to the model due_back field)
            book_instance.due_back = form.cleaned_data['renewal_date']
            book_instance.save()

            # redirect to a new URL:
            return HttpResponseRedirect(reverse('all-borrowed')) )
```

```
# If this is a GET (or any other method) create the default form.
else:
    proposed_renewal_date = datetime.date.today() + datetime.timedelta(weeks=3)
    form = RenewBookForm(initial={'renewal_date': proposed_renewal_date})

context = {
    'form': form,
    'book_instance': book_instance,
}

return render(request, 'catalog/book_renew_librarian.html', context)
```

The template

Create the template referenced in the view (**/catalog/templates/catalog/book_renew_librarian.html**) and copy the code into it:

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

{% block content %}
    <h1>Renew: {{ book_instance.book.title }}</h1>
    <p>Borrower: {{ book_instance.borrower }}</p>
    <p>{% if book_instance.is_overdue %} class="text-danger"{% endif %}>
        Due date: {{ book_instance.due_back }}</p>

    <form action="" method="post">
        {% csrf_token %}
        <table>
            {{ form.as_table }}
        </table>
        <input type="submit" value="Submit">
    </form>
{% endblock %}
```

We extend the base template and then redefine the content block. We are able to reference `{{ book_instance }}` (and its variables) because it was passed into the context object in the `render()` function, and we use these to list the book title, borrower, and the original due date.

First we declare the `form` tags, specifying where the form is to be submitted (`action`) and the `method` for submitting the data (in this case an "HTTP POST").

An empty action means that the form data will be posted back to the current URL of the page (which is what we want!).

Inside the tags, we define the `submit` input, which a user can press to submit the data. The `{% csrf_token %}` added just inside the form tags is part of Django's cross-site forgery protection.

All that's left is the `{{ form }}` template variable, which we passed to the template in the context dictionary.

This provides the default rendering of all the form fields, including their labels, widgets, and help text — the rendering is below:

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

{% block content %}
    <h1>Renew: {{ book_instance.book.title }}</h1>
    <p>Borrower: {{ book_instance.borrower }}</p>
    <p>{% if book_instance.is_overdue %} class="text-danger"
        {% endif %} Due date: {{ book_instance.due_back }}</p>

    <form action="" method="post">
        {% csrf_token %}
        <table>
            {{ form.as_table }}
        </table>
        <input type="submit" value="Submit">
    </form>
{% endblock %}
```

If you were to enter an invalid date, you'd additionally get a list of the errors rendered in the page.

```
<tr>
  <th><label for="id_renewal_date">Renewal date:</label></th>
  <td>
    <ul class="errorlist">
      <li>Invalid date - renewal in past</li>
    </ul>
    <input id="id_renewal_date" name="renewal_date"
      type="text" value="2015-11-08" required>
    <br>
    <span class="helptext">
      Enter date between now and 4 weeks (default 3 weeks).
    </span>
  </td>
</tr>
```

Other ways of using form template variable

Using `{{ form.as_table }}`, each field is rendered as a table row. We can also render each field as a list item (using `{{ form.as_ul }}`) or as a paragraph (using `{{ form.as_p }}`).

It is also possible to have complete control over the rendering of each part of the form, by indexing its properties using dot notation. So, for example, we can access a number of separate items for our `renewal_date` field:

- `{{ form.renewal_date }}`: The whole field.
- `{{ form.renewal_date.errors }}`: The list of errors.
- `{{ form.renewal_date.id_for_label }}`: The id of the label.
- `{{ form.renewal_date.help_text }}`: The field help text.

Testing the page

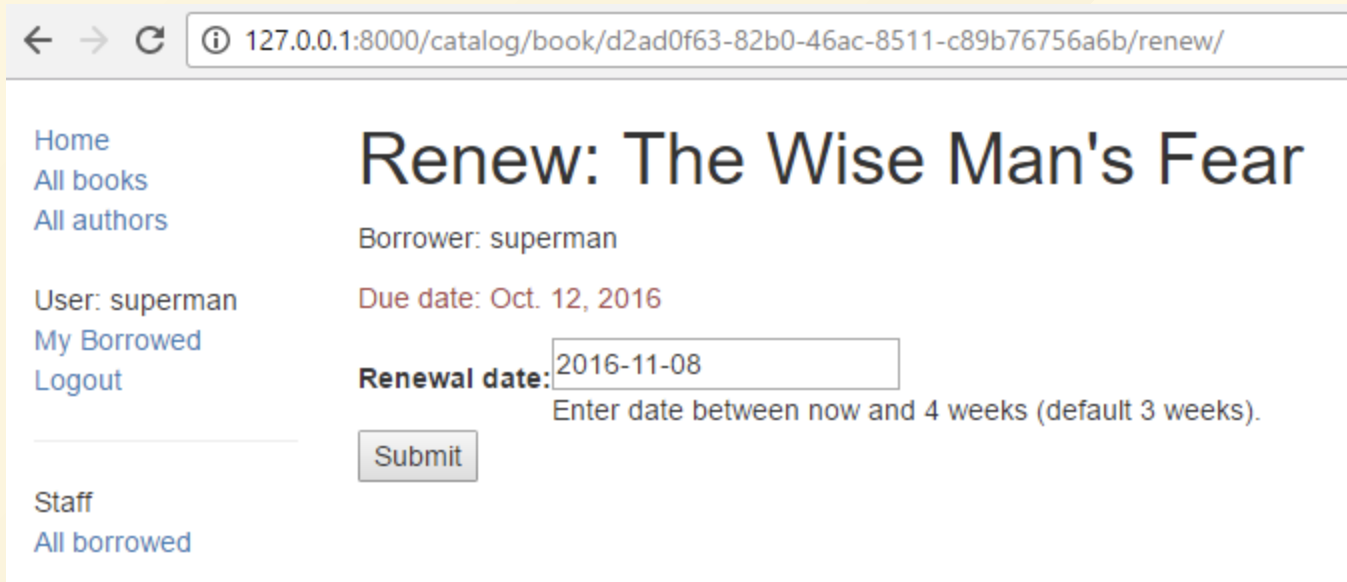
If you accepted the "challenge" in Django Tutorial Part 8: User authentication and permissions you'll have a list of all books on loan in the library, which is only visible to library staff. We can add a link to our renew page next to each item using the template code below.

```
{% if perms.catalog.can_mark_returned %}-  
  <a href="{% url 'renew-book-librarian' bookinst.id %}">  
    Renew</a>  
{% endif %}
```

You can alternatively manually construct a test URL http://127.0.0.1:8000/catalog/book/<bookinstance_id>/renew/.

What does it look like?

If you are successful, the default form will look like this:



The screenshot shows a web browser window with the address bar displaying the URL: 127.0.0.1:8000/catalog/book/d2ad0f63-82b0-46ac-8511-c89b76756a6b/renew/. The page has a light blue header with navigation links: Home, All books, All authors, User: superman, My Borrowed, Logout, Staff, and All borrowed. The main content area is titled 'Renew: The Wise Man's Fear'. It displays the borrower's name as 'superman' and the due date as 'Oct. 12, 2016'. The 'Renewal date' field is a text input box containing '2016-11-08'. Below the input box is a 'Submit' button. A note below the button states: 'Enter date between now and 4 weeks (default 3 weeks)'.

← → ↻ ⓘ 127.0.0.1:8000/catalog/book/d2ad0f63-82b0-46ac-8511-c89b76756a6b/renew/

Home
All books
All authors

User: superman
My Borrowed
Logout

Staff
All borrowed

Renew: The Wise Man's Fear

Borrower: superman

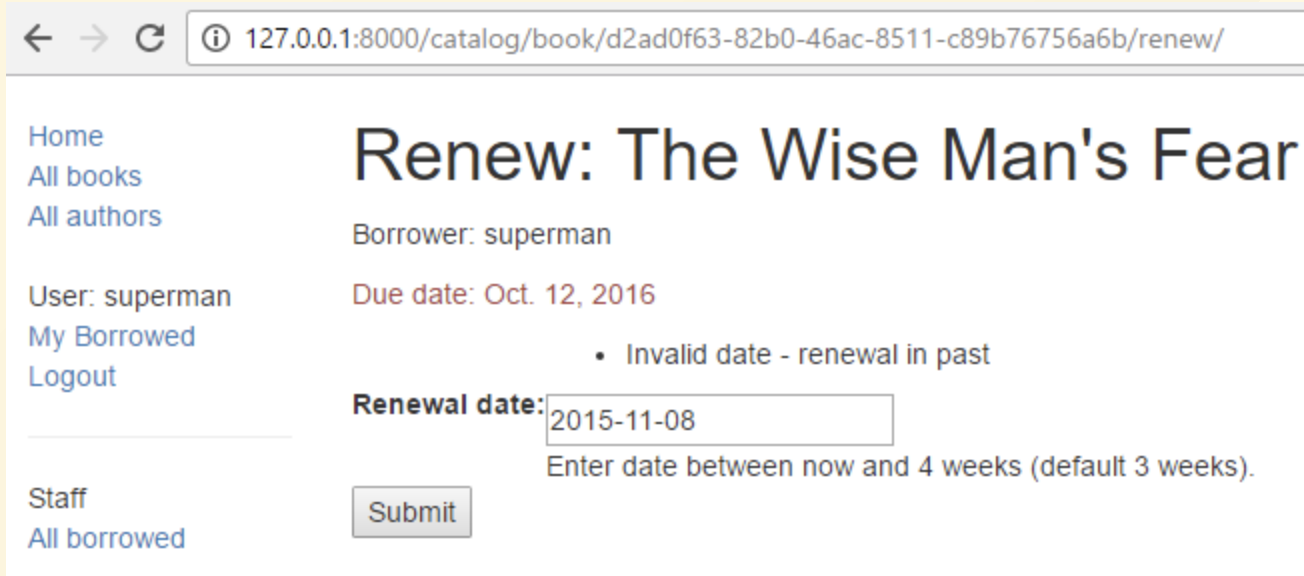
Due date: Oct. 12, 2016

Renewal date:

Submit

Enter date between now and 4 weeks (default 3 weeks).

The form with an invalid value entered will look like this:



A screenshot of a web browser displaying a library renewal page. The browser's address bar shows the URL: 127.0.0.1:8000/catalog/book/d2ad0f63-82b0-46ac-8511-c89b76756a6b/renew/. The page has a left sidebar with navigation links: Home, All books, All authors, User: superman, My Borrowed, Logout, Staff, and All borrowed. The main content area is titled "Renew: The Wise Man's Fear". It shows the borrower as "superman" and the due date as "Oct. 12, 2016". Below this, there is an error message: "Invalid date - renewal in past". The "Renewal date:" field contains the text "2015-11-08". A note below the field states: "Enter date between now and 4 weeks (default 3 weeks).". A "Submit" button is located at the bottom of the form.

← → ↻ ⓘ 127.0.0.1:8000/catalog/book/d2ad0f63-82b0-46ac-8511-c89b76756a6b/renew/

Home
All books
All authors

User: superman
My Borrowed
Logout

Staff
All borrowed

Renew: The Wise Man's Fear

Borrower: superman

Due date: Oct. 12, 2016

- Invalid date - renewal in past

Renewal date: 2015-11-08

Enter date between now and 4 weeks (default 3 weeks).

Submit

The list of all books with renew links will look like this:

[Home](#)
[All books](#)
[All authors](#)

User: [superman](#)
[My Borrowed](#)
[Logout](#)

[Staff](#)
[All borrowed](#)

All Borrowed Books

- [The Wise Man's Fear](#) (Oct. 12, 2016) - [superman](#) - [Renew](#)
- [The Name of the Wind](#) (Nov. 8, 2016) - [superman](#) - [Renew](#)
- [The Dueling Machine](#) (Nov. 8, 2016) - [johnc](#) - [Renew](#)

ModelForms

Creating a `Form` class using the approach described above is very flexible, allowing you to create whatever sort of form page you like and associate it with any model or models.

If we just need a form to map the fields of a single model then our model will already define most of the information that we need in your form: fields, labels, help text, etc.

Rather than recreating the model definitions in your form, it is easier to use the `ModelForm` helper class to create the form from your model. This `ModelForm` can then be used within our views in exactly the same way as an ordinary `Form`.

A basic `ModelForm` containing the same field as our original `RenewBookForm` is shown below.

All we need to do to create the form is add `class Meta` with the associated `model` (`BookInstance`) and a list of the model `fields` to include in the form (we can include all fields using `fields = '__all__'`, or we can use `exclude` (instead of `fields`) to specify the fields *not* to include from the model).

```
from django.forms import ModelForm

from catalog.models import BookInstance

class RenewBookModelForm(ModelForm):
    class Meta:
        model = BookInstance
        fields = ['due_back']
```

The rest of the information comes from the model field definitions. If these aren't quite right, then we can override them in our `class Meta`, specifying a dictionary containing the field to change and its new value. For example, in this form we might want a label for our field of "*Renewal date*", and we also want our help text to be specific to this use case.

The `Meta` shows us how to override these fields, and we can similarly set `widgets` and `error_messages` if the defaults aren't sufficient.

```
class Meta:
    model = BookInstance
    fields = ['due_back']
    labels = {'due_back': _('New renewal date')}
    help_texts = {'due_back':
        _('Enter a date between now and 4 weeks (default 3).')}
```

To add validation we can use the same approach as for a normal Form — we define a function named `clean_field_name()` and raise `ValidationError` exceptions for invalid values.

The only difference with respect to our original form is that the model field is named `due_back` and not "`renewal_date`". This change is necessary since the corresponding field in `BookInstance` is called `due_back`.

```
from django.forms import ModelForm

from catalog.models import BookInstance

class RenewBookModelForm(ModelForm):
    def clean_due_back(self):
        data = self.cleaned_data['due_back']

        # Check if a date is not in the past.
        if data < datetime.date.today():
            raise ValidationError(
                _('Invalid date - renewal in past'))
```

```
# Check if a date is in the allowed range  
# (+4 weeks from today).  
if data > datetime.date.today()  
    + datetime.timedelta(weeks=4):  
    raise ValidationError(_(  
        'Invalid date - renewal more than 4 weeks ahead'))  
  
# Remember to always return the cleaned data.  
return data
```

```
class Meta:  
    model = BookInstance  
    fields = ['due_back']  
    labels = {'due_back': _('Renewal date')}  
    help_texts = {'due_back':  
        _('Enter a date between now and 4 weeks (default 3).')}
```

The class `RenewBookModelForm` is now functionally equivalent to our original `RenewBookForm`.

We could import and use it wherever we currently use `RenewBookForm` as long as we also update the corresponding form variable name from `renewal_date` to `due_back` as in the second form declaration:

```
RenewBookModelForm(initial={'due_back':  
proposed_renewal_date}
```

.

Generic editing views

The form handling algorithm we used in our function view example represents an extremely common pattern in form editing views.

Django abstracts much of this "boilerplate" for us, by creating generic editing views for creating, editing, and deleting views based on models.

Not only do these handle the "view" behavior, but they automatically create the form class (a `ModelForm`) for us from the model.

We're going to use generic editing views to create pages to add functionality to create, edit, and delete **Author** records from our library — effectively providing a basic reimplementation of parts of the Admin site (this could be useful if we need to offer admin functionality in a more flexible way that can be provided by the admin site).

Views

Open the views file (**locallibrary/catalog/views.py**) and append the code block to the bottom of it:

```
from django.views.generic.edit
    import CreateView, UpdateView, DeleteView
from django.urls import reverse_lazy

from catalog.models import Author

class AuthorCreate(CreateView):
    model = Author
    fields = '__all__'
    initial = {'date_of_death': '05/01/2018'}

class AuthorUpdate(UpdateView):
    model = Author
    fields = ['first_name', 'last_name',
              'date_of_birth', 'date_of_death']

class AuthorDelete(DeleteView):
    model = Author
    success_url = reverse_lazy('authors')
```

To create, update, or delete the views we need to derive from `CreateView`, `UpdateView`, and `DeleteView` (respectively) and then define the associated model.

For the "create" and "update" cases we also need to specify the fields to display in the form (using the same syntax as for `ModelForm`). In this case, we show both the syntax to display "all" fields and how we can list them individually.

We can also specify initial values for each of the fields using a dictionary of *field_name/value* pairs (here we arbitrarily set the date of death for demonstration purposes — you might want to remove that!).

By default, these views will redirect on success to a page displaying the newly created/edited model item, which in our case will be the author detail view we created in a previous tutorial.

We can specify an alternative redirect location by explicitly declaring parameter `success_url` (as done for the `AuthorDelete` class).

The `AuthorDelete` class doesn't need to display any of the fields, so these don't need to be specified.

We do however need to specify the `success_url`, because there is no obvious default value for Django to use.

In this case, we use the `reverse_lazy()` function to redirect to our author list after an author has been deleted — `reverse_lazy()` is a lazily executed version of `reverse()`, used here because we're providing a URL to a class-based view attribute.

Templates

The "create" and "update" views use the same template by default, which will be named after our model: *model_name_form.html* (you can change the suffix to something other than **_form** using the

`template_name_suffix` field in your view, e.g.

```
template_name_suffix = '_other_suffix')
```

Create the template file

locallibrary/catalog/templates/catalog/author_form.html and copy in the text.


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

{% block content %}
    <form action="" method="post">
        {% csrf_token %}
        <table>
            {{ form.as_table }}
        </table>
        <input type="submit" value="Submit">
    </form>
{% endblock %}
```

This is similar to our previous forms and renders the fields using a table. Note also how again we declare the `{% csrf_token %}` to ensure that our forms are resistant to CSRF attacks.

The "delete" view expects to find a template named with the format *model_name_confirm_delete.html* (again, we can change the suffix using `template_name_suffix` in our view).

Create the template file **locallibrary/catalog/templates/catalog/author_confirm_delete.html** and copy in the text.

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

{% block content %}

<h1>Delete Author</h1>

<p>Are you sure you want to delete the author: {{ author }}?</p>

<form action="" method="POST">
    {% csrf_token %}
    <input type="submit" value="Yes, delete.">
</form>

{% endblock %}
```

URL configurations

Open your URL configuration file (**locallibrary/catalog/urls.py**) and add the configuration to the bottom of the file:

```
urlpatterns += [  
    path('author/create/',  
         views.AuthorCreate.as_view(),  
         name='author_create'),  
    path('author/<int:pk>/update/',  
         views.AuthorUpdate.as_view(),  
         name='author_update'),  
    path('author/<int:pk>/delete/',  
         views.AuthorDelete.as_view(),  
         name='author_delete'),  
]
```

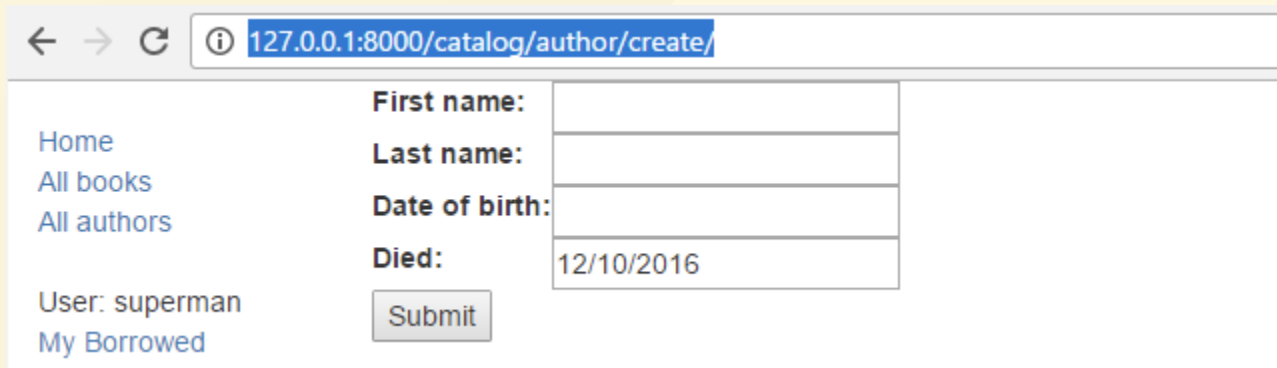
We can see that the views are classes, and must hence be called via `.as_view()`, and we should be able to recognize the URL patterns in each case. We must use `pk` as the name for our captured primary key value, as this is the parameter name expected by the view classes.

The author create, update, and delete pages are now ready to test.

Testing the page

First, log in to the site with an account that has whatever permissions you decided are needed to access the author editing pages.

Then navigate to the author create page:
<http://127.0.0.1:8000/catalog/author/create/>, which should look like the screenshot.



The screenshot shows a web browser window with the address bar displaying `127.0.0.1:8000/catalog/author/create/`. The page layout includes a left sidebar with navigation links: [Home](#), [All books](#), [All authors](#), [User: superman](#), and [My Borrowed](#). The main content area contains a form with the following fields: **First name:** (empty text input), **Last name:** (empty text input), **Date of birth:** (empty text input), and **Died:** (text input containing `12/10/2016`). A **Submit** button is located below the **Died:** field.

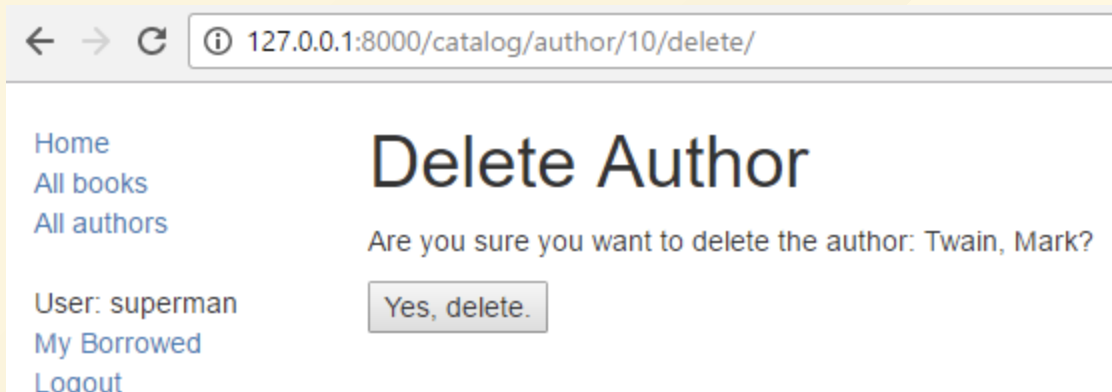
Enter values for the fields and then press **Submit** to save the author record. You should now be taken to a detail view for your new author, with a URL of something like

<http://127.0.0.1:8000/catalog/author/10>.

You can test editing records by appending */update/* to the end of the detail view URL (e.g.

<http://127.0.0.1:8000/catalog/author/10/update/>).

Finally, we can delete the page by appending delete to the end of the author detail-view URL (e.g. <http://127.0.0.1:8000/catalog/author/10/delete/>). Django should display the delete page. Press **Yes, delete.** to remove the record and be taken to the list of all authors.



Challenge yourself

Create some forms to create, edit, and delete **Book** records. You can use exactly the same structure as for **Authors**. If your **book_form.html** template is just a copy-renamed version of the **author_form.html** template, then the new "create book" page will look like the screenshot:



127.0.0.1:8000/catalog/book/create/

[Home](#)
[All books](#)
[All authors](#)

User: superman
[My Borrowed](#)
[Logout](#)

[Staff](#)
[All borrowed](#)

Title:

Author:

Summary:

Enter a brief description of the book

ISBN:

13 Character [ISBN number](#)

Genre:

Science Fiction
Fantasy
Western
French Poetry

Select a genre for this book

Language:

Submit