**Chapter 15: Search**

Search is a fundamental feature of most websites and certainly anything e-commerce related like our Bookstore. In this chapter we will learn how to implement basic search with forms and filters. Then we will improve it with additional logic and touch upon ways to go even more deeply with search options in Django. We only have three books in our database now but the code here will scale to as many books as we’d like.

Search functionality consists of two parts: a form to pass along a user search query and then a results page that performs a filter based on that query. Determining “the right” type of filter is where search becomes interesting and hard. But first we need to create both a form and the search results page.

We could start with either one at this point, but we’ll configure the filtering first and then the form.

**Search Results Page**

We’ll start with the search results page. As with all Django pages that means adding a dedicated URL, view, and template. The implementation order doesn’t particularly matter, but we will add them in that order.

Within books/urls.py, add a search/ path with a URL name of search\_results that uses a view called SearchResultsListView.

# books/urls.py

from django.urls import path

from .views import BookListView, BookDetailView, SearchResultsListView

urlpatterns = [

path("", BookListView.as\_view(), name="book\_list"),

path("<uuid:pk>", BookDetailView.as\_view(), name="book\_detail"),

path("search/", SearchResultsListView.as\_view(),

name="search\_result"), # new

]

Next up is the view SearchResultsListView which is, for now, a listing of all available books. That’s a prime candidate for using ListView. Its template will be called search\_results.html and live within the templates/books/ directory. The only new code is for SearchResultsListView as

we have previously imported both ListView and the Book model at the top of the file.

# books/views.py

...

class SearchResultsListView(ListView): # new

model = Book

context\_object\_name = "book\_list"

template\_name = "books/search\_results.html"

Last up is creating our template, templates/books/search\_results.html, in the text editor. For now it will list all available books by title, author, and price.

<!-- templates/books/search\_results.html -->

{% extends "\_base.html" %}

{% block title %}Search{% endblock title %}

{% block content %}

<h1>Search Results</h1>

{% for book in book\_list %}

<div>

<h3><a href="{{ book.get\_absolute\_url }}">{{ book.title }}</a></h3>

<p>Author: {{ book.author }}</p>

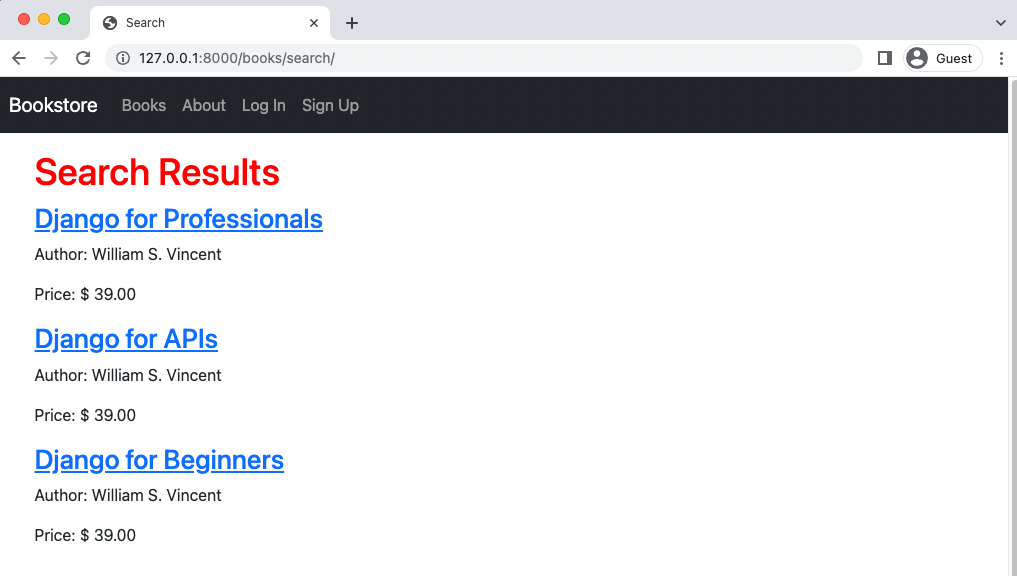
<p>Price: $ {{ book.price }}</p>

</div>

{% endfor %}

{% endblock content %}

If you are still logged into a user account, log out now. The search results page is now available for all users at <http://127.0.0.1:8000/books/search/>.



Search page

And there it is!

**Basic Filtering**

In Django a QuerySet is used to filter the results from a database model. Currently our search results page doesn’t feel like one because it is outputting all results from the Book model. Ultimately we want to run the filter based on the user’s search query, but first we’ll work through multiple filtering options.

It turns out there are multiple ways to customize a queryset including via a manager on the model itself but to keep things simple, we can add a filter with just one line. So let’s do that!

We can override the default queryset attribute on ListView which by default shows all results. The queryset documentation is quite robust and detailed, but often using contains (which is case sensitive) or icontains (which is not case sensitive) are good starting points. We will implement the filter based on the title that “contains” the name “beginners”.

# books/views.py

class SearchResultsListView(ListView):

model = Book

context\_object\_name = "book\_list"

template\_name = "books/search\_results.html"

queryset = Book.objects.filter(title\_\_icontains="beginners") # new

Refresh the search results page and now only a book with the title containing “beginners” is displayed. Success!



Search page for “beginners”

For basic filtering most of the time the built-in queryset methods of filter(), all(), get(), or

exclude() will be enough. However there is also a very robust and detailed QuerySet API available as well that is worthy of further study.

**Q Objects**

Using filter() is powerful and it’s even possible to chain filters together such as search for all titles that contain “beginners” and “django”. However often you’ll want more complex lookups that can use “OR” not just “AND”; that’s when it is time to turn to Q objects.

Here’s an example where we set the filter to look for a result that matches a title of either “beginners” or “api”. It’s as simple as importing Q at the top of the file and then subtly tweaking our existing query. The | symbol represents the “or” operator. We can filter on any available field: not just title but also author or price as desired.

As the number of filters grows it can be helpful to separate out the queryset override via get\_queryset(). That’s what we’ll do here but note that this choice is optional.

# books/views.py

from django.db.models import Q # new

...

class SearchResultsListView(ListView):

model = Book

context\_object\_name = "book\_list"

template\_name = "books/book\_list.html"

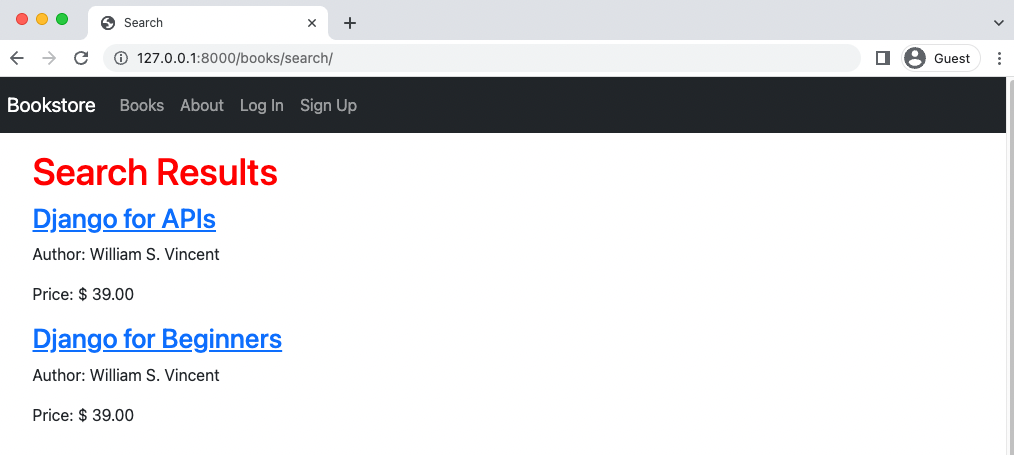
def get\_queryset(self): # new

return Book.objects.filter(

Q(title\_\_icontains="beginners") | Q(title\_\_icontains="api")

)

Refresh the search results page to see the new result.



Search with Q objects

Now let’s turn our attention to the corresponding search form so that rather than hardcode our filters in we can populate them based on the user’s search query.

**Forms**

Fundamentally a web form is simple: it takes user input and sends it to a URL via either a GET or POST method. However in practice this fundamental behavior of the web can be monstrously complex.

The first issue is sending the form data: where does the data actually go and how do we handle it once there? Not to mention there are numerous security concerns whenever we allow users to submit data to a website.

There are only two options for “how” a form is sent: either via GET or POST HTTP methods.

A POST bundles up form data, encodes it for transmission, sends it to the server, and then receives a response. Any request that changes the state of the database–creates, edits, or deletes data should use a POST.

A GET bundles form data into a string that is added to the destination URL. GET should only be used for requests that do not affect the state of the application, such as a search where nothing within the database is changing, basically we’re just doing a filtered list view.

If you look at the URL after visiting Google.com you’ll see your search query in the actual searchresults page URL itself.

For more information, Mozilla has detailed guides on both sending form data and form data validation that are worth reviewing if you’re not already familiar with form basics.

**Search Form**

Let’s add a basic search form to the navbar right now so it is available on every page. That means updating the \_base.html template and in particular, the bottom of the <ul></ul> tags after the if/else/elif logic.

Here is what the updated code looks like which we’ll explain below.

<!-- templates/\_base.html -->

{% endif %}

</ul>

<form class="d-flex" action="{% url 'search\_result' %}" method="get">

<input class="form-control me-2" type="search" name="q" placeholder="Search" aria-label="Search">

<button class="btn btn-outline-success" type="submit">Search</button>

</form>

</div>

</div>

</nav>

We are adding HTML <form> tags and using Bootstrap’s styling to make them look nice. The action specifies where to redirect the user after the form is submitted, which will be the URL named search\_results page. As with all URL links this is the URL name for the page. Then we indicate the desired method of get rather than post.

The second part of the form is the input which contains the user search query. We provide it with a variable name, q, which will be later visible in the URL and also available in the views file. We add Bootstrap styling with the class, specify the type of input is text, add a Placeholder which is default text that prompts the user. The last part, aria-label, is the name provided to screen reader users. Accessibility is a big part of web development and should always be considered from the beginning: include

aria-labels with all your forms!

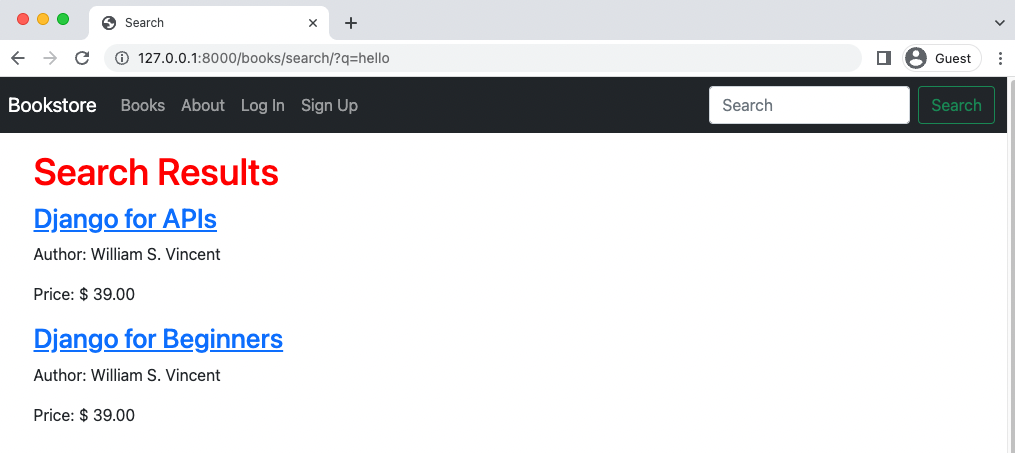
Finally we’re including a “Search” button to be clicked on to run the search query. A user can also just press Return to achieve the same result.

Navigate to the homepage–or any page really–and the new search box is present in the upper right.



**Homepage with search box**

Try inputting a search, for example for “hello.” Upon hitting Return or clicking the “Search” button you are redirected to the search results page. Note, in particular, the URL contains the search query /search/?q=hello.



**URL with query string**

However the results haven’t changed! And that’s because our SearchResultsListView still has the hardcoded values from before. The last step is to take the user’s search query, represented q in the URL, and pass it in to the actual search filters.

# books/views.py

class SearchResultsListView(ListView):

model = Book

context\_object\_name = "book\_list"

template\_name = "books/book\_list.html"

def get\_queryset(self):

query = self.request.GET.get("q") # new

return Book.objects.filter(

Q(title\_\_icontains="beginner") | Q(title\_\_icontains="api")

)

What changed? We added a query variable that takes the value of q from the form submission. Then updated our filter to use query on either a title or an author field. That’s it! Refresh the search results page–it still has the same URL with our query–and the result is expected: no results on either title or author for “hello”.

Go back to the homepage and try a new search such as for “api” or “beginners” to see the complete search functionality in action.

**Git**

Make sure to create a new Git commit for the changes in this chapter.

Bring docker down on chapter - docker-compose down

Remove-Item -Recurse -Force .git

git init

git status

git add .

git commit -m “Chapter 15. Search”

As always you can compare your code again the official source code on Github.

**Conclusion**

Our basic search is now complete, but we’ve only scratched the surface of potential search optimizations. For example, maybe we want a button added to the search form that could be clicked in addition to hitting the Return key? Or better yet include form validation. Beyond filtering with ANDs and ORs there are other factors if we want a Google-quality search, things like relevancy and much more.

There are several third-party packages with enhanced features such as django-watson or djangohaystack however, given that we’re using PostgreSQL as the database, we can take advantage of its full text search and other features which are built into Django itself.

A final option is either use an enterprise-level solution like ElasticSearch that must be running on a separate server (not the hardest thing with Docker), or rely on a hosted solution like Swiftype or Algolia.

In the next chapter we’ll explore the many performance optimizations available in Django as we prepare our Bookstore project for eventual deployment.

The end.