**Chapter 16: Performance**

The first priority for any website is that it must work properly and contain proper tests. But if your project is fortunate enough to receive a large amount of traffic the focus quickly shifts to performance and making things as efficient as possible. This is a fun and challenging exercise for many engineers, but it can also be a trap.

The computer scientist Donald Knuth has a famous quote worth reading in its entirety:

“The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times; premature optimization is the root of all evil (or at least most of it) in programming.”

While it’s important to set up proper monitoring so you can optimize your project later on, don’t focus too much on it upfront. There’s no way to properly mimic production environments locally. And there is no way to predict exactly how a site’s traffic will look. But it is possible to spend far too much time seeking out tiny performance gains in the early stages instead of talking to users and making larger code improvements!

In this chapter we’ll focus on the broad strokes of Django-related performance and highlight areas worth further investigation at scale. Generally speaking performance comes down to four major areas: optimizing database queries, caching, indexes, and compressing front-end assets like images, JavaScript, and CSS.

**django-debug-toolbar**

Before we can optimize our database queries we need to see them. The go-to tool for this in the Django community is the third-party package django-debug-toolbar. It comes with a configurable set of panels to inspect the complete request/response cycle of any given page.

Per usual we can install it by adding the latest version to our requirements.txt file.

requirements.txt

asgiref==3.5.2

Django==4.0.4

psycopg2-binary==2.9.3

sqlparse==0.4.2

django-crispy-forms==1.14.0

crispy-bootstrap5==0.6

django-allauth==0.50.0

environs[django]==9.5.0

pillow==9.0.1

django-debug-toolbar==5.2.0

Then stop our container, rebuild the image, and run the container again.

docker-compose down

docker-compose up -d –build

docker-compose logs web --tail=50 (to check if there is/are any error(s))

There are three separate configurations to set in our django\_project/settings.py file:

1. INSTALLED\_APPS

2. Middleware

3. INTERNAL\_IPS

First add Debug Toolbar to the INSTALLED\_APPS configuration. Note that the proper name is debug\_toolbar not django\_debug\_toolbar as might be expected.

INSTALLED\_APPS = [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

'django.contrib.sites',

# Third-party

'crispy\_forms',

'crispy\_bootstrap5',

'allauth',

'allauth.account',

'debug\_toolbar', # new

# Local

'accounts.apps.AccountsConfig',

'pages.apps.PagesConfig',

'books.apps.BooksConfig',

]

Second, add Debug Toolbar to the Middleware where it is primarily implemented.

MIDDLEWARE = [

'django.middleware.security.SecurityMiddleware',

'django.contrib.sessions.middleware.SessionMiddleware',

'django.middleware.common.CommonMiddleware',

'django.middleware.csrf.CsrfViewMiddleware',

'django.contrib.auth.middleware.AuthenticationMiddleware',

'django.contrib.messages.middleware.MessageMiddleware',

'django.middleware.clickjacking.XFrameOptionsMiddleware',

'django\_toolbar.middleware.DebugToolbarMiddleware', # new

]

And third, set the INTERNAL\_IPS as well. If we were not in Docker this could be set to

'127.0.0.1', however, since we’re running our web server within Docker an additional step is required so that it matches the machine address of Docker. Add the following lines at the bottom of config/settings.py.

# django\_project/settings.py

...

# django-debug-toolbar

import socket

hostname, \_, ips = socket.gethostbyname\_ex(socket.gethostname())

INTERNAL\_IPS = [ip[:-1] + "1" for ip in ips]

Phew. That looks a bit scary, but basically it ensures that our INTERNAL\_IPS matches that of our Docker host.

There’s one last step and that is to update our URLconf. We only want Debug Toolbar to appear if DEBUG is true so we’ll add logic to display it only in this case at the bottom of the django\_project/urls.py file.

# django\_project/urls.py

...

if settings.DEBUG: # new

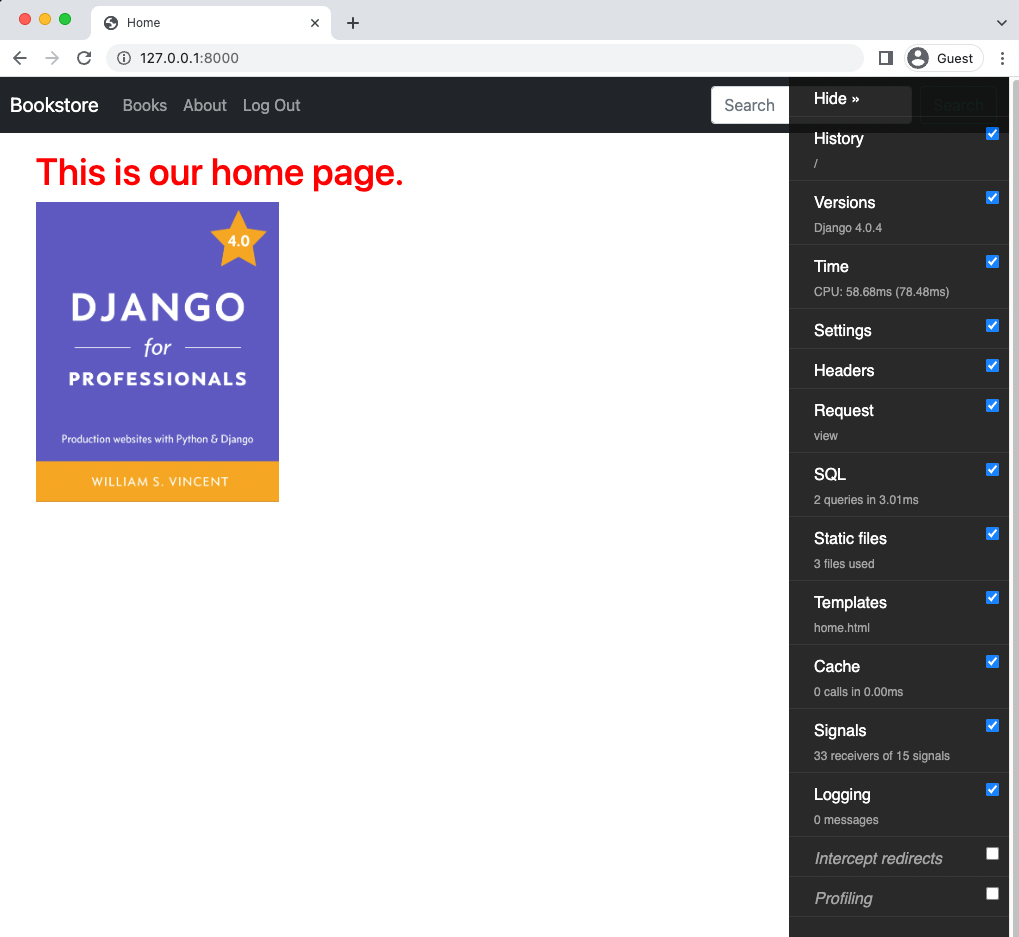
import debug\_toolbar

urlpatterns = [

path("\_\_debug\_\_/", include(debug\_toolbar.urls)),

] + urlpatterns

Now if you refresh the homepage you’ll see the django-debug-toolbar on the righthand side.



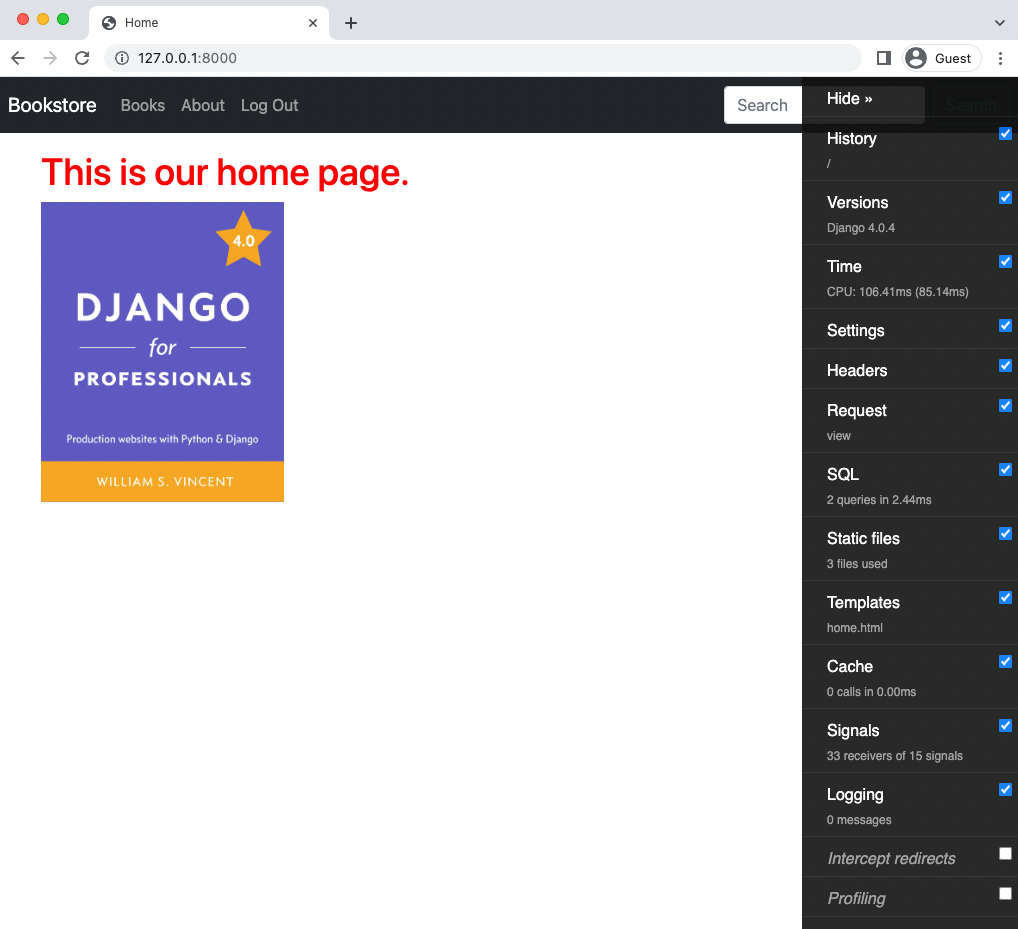
Debug Toolbar

If you click the “Hide” link on top it becomes a much smaller sidebar on the righthand side of the page.

**Analyzing Pages**

Debug Toolbar has many possible customizations but the default settings visible tell us a lot about our homepage. For instance, we can see the current version of Django being used as well as the Time it took to load the page.

Probably the most useful item, however, is SQL which shows the queries on a specific page. If you are logged out right now, there are no SQL queries on the homepage. So go ahead and log in with your superuser account and then return the homepage. Debug Toolbar shows that two queries are being run for logged in users.



Debug Toolbar

Over on the Django for Professionals book page there are six queries being run:

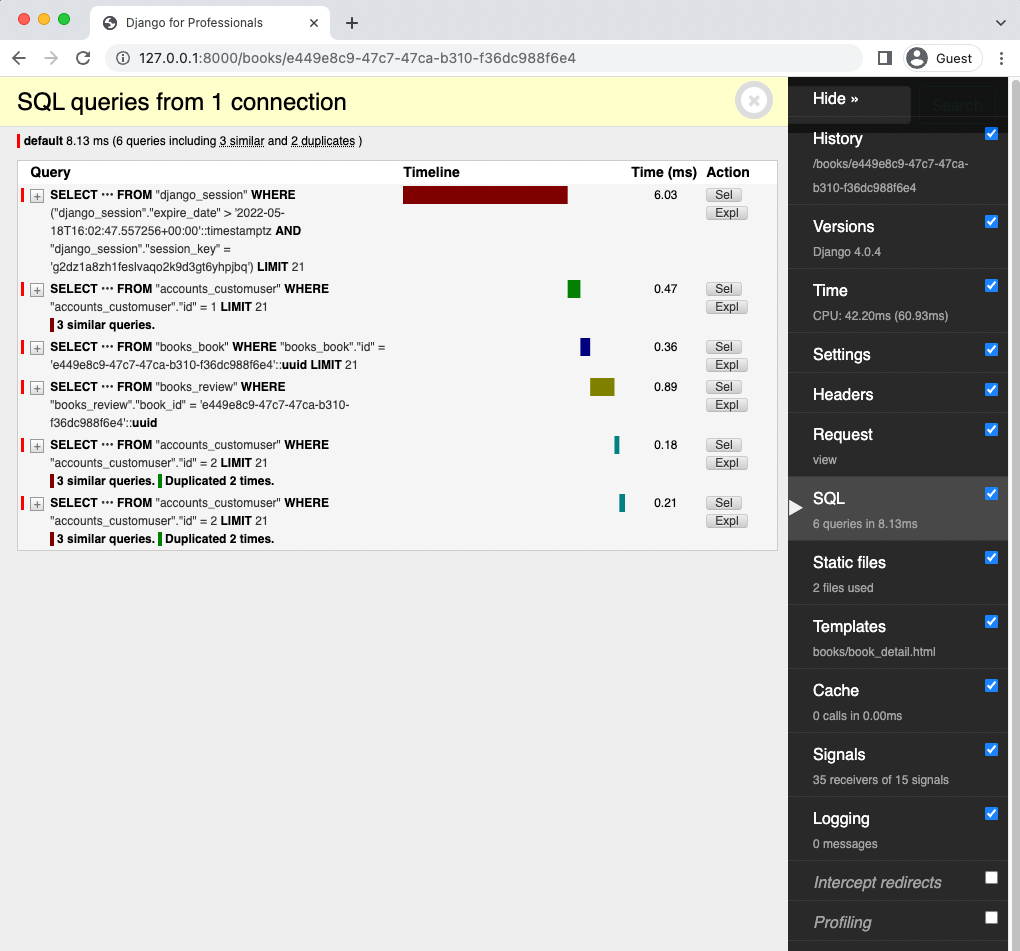
• the sessions framework to manage users

• to accounts\_customuser to load our user

• to books\_book to load the “Django for Professionals” book

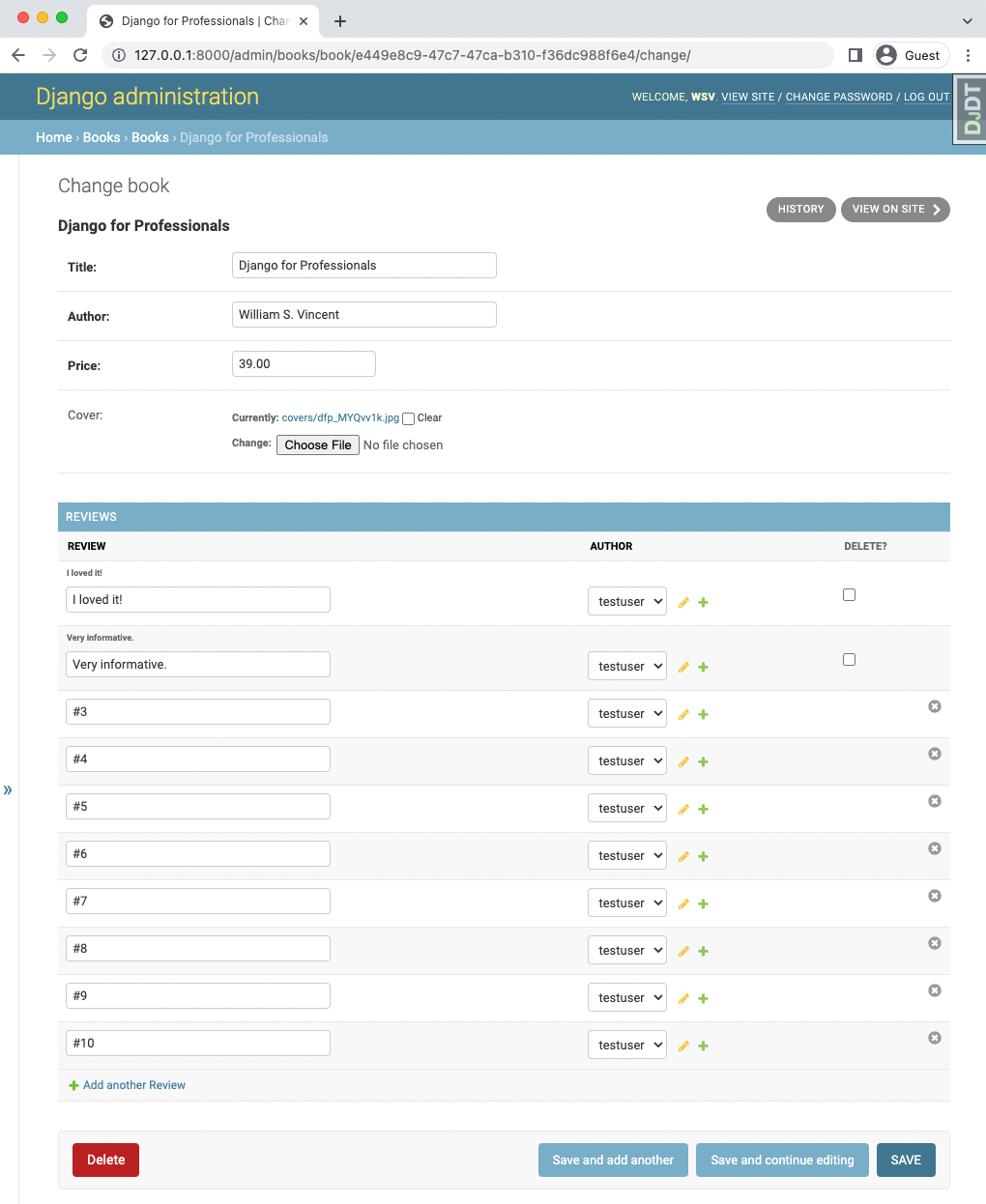
• to books\_review to load the reviews

• and then 2 more queries to accounts\_customuser

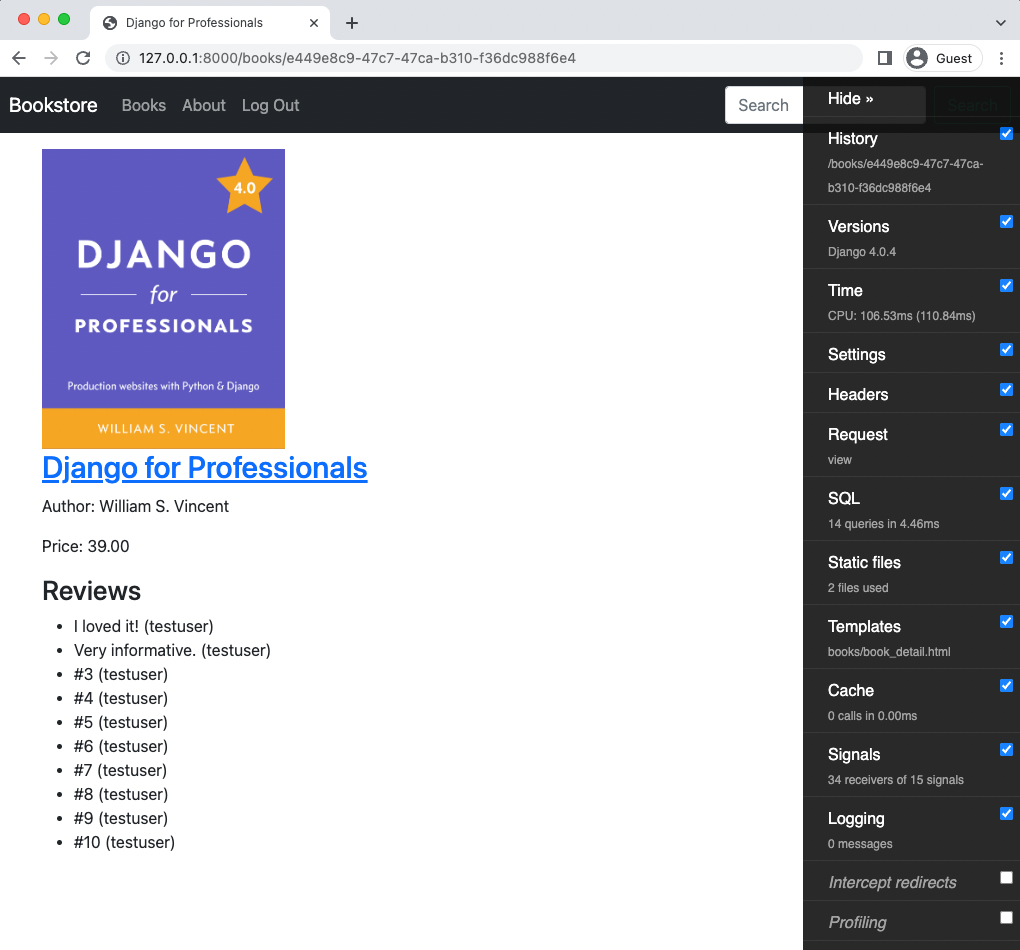


Debug Toolbar SQL Queries

Six queries doesn’t seem too bad. But what if we had a lot of reviews for a book? Go into the admin and to the entry for the “Django for Professionals” book. Add eight more reviews–all by testuser–so there are now ten total and click “Save.”



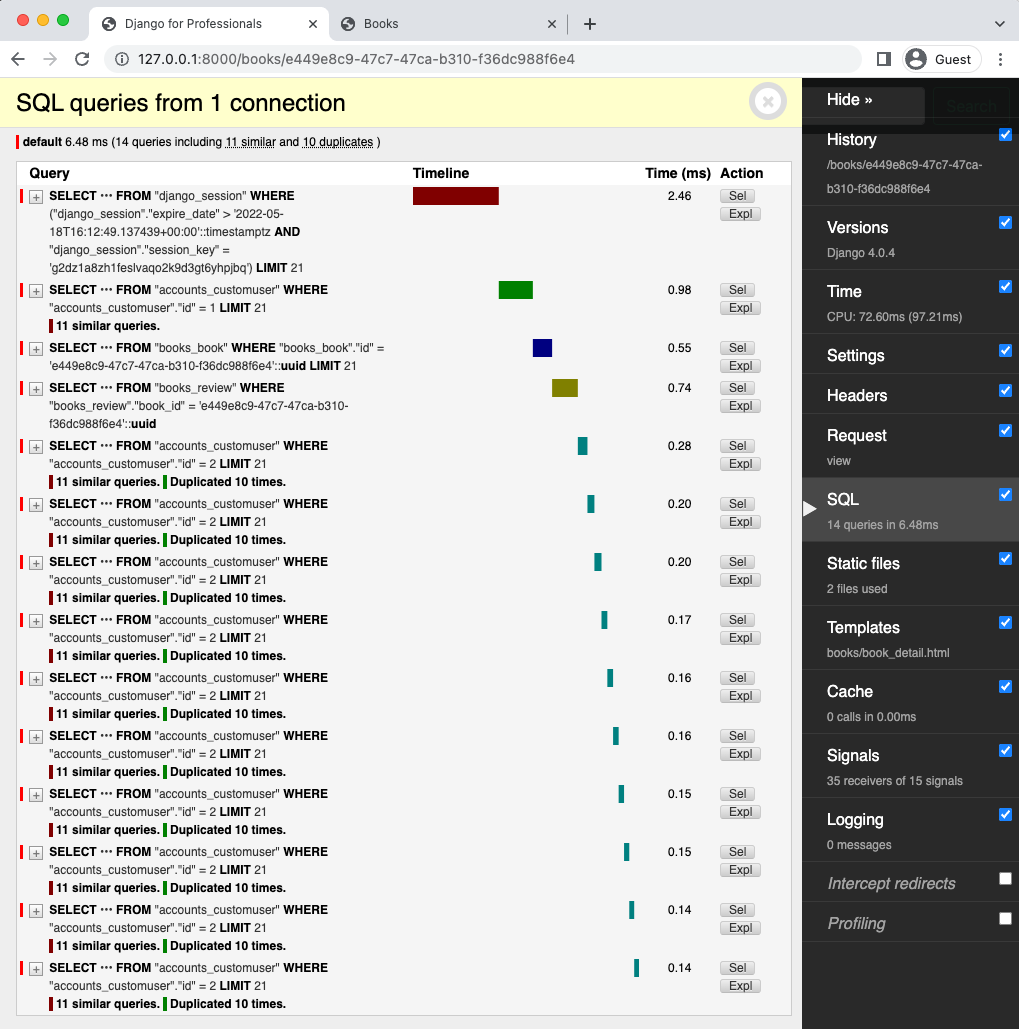
Admin Ten Reviews

How many SQL queries do you think will now be on our detail page for this book? Let’s head over and check! 

Books Page Ten Reviews

As perhaps expected there are now eight more queries to coincide with the eight new reviews meaning 14 total queries for this one page.

If you look at the SQL detail view in Django Debug Toolbar it is actually quite helpful in flagging duplicate queries. Is it really necessary to call accounts\_customuser eleven times? There’s like a way to reduce the number of queries since, after all, the books query manages to import three books with just one query.



11 Similar Queries

What we are experiencing now is a very common issue known generally as the “N+1 Queries problem.” There are several third-party packages that can help identify the issue. Fortunately there are tools to help.

**select\_related and prefetch\_related**

Reducing SQL queries is one of the first steps when optimizing a Django project. There are two powerful methods included in the Django ORM to help us that can boost performance by creating a single more complex QuerySet rather than multiple, smaller queries:

• select\_related for Foreign Key relationships

• prefetch\_related() for Many to Many relationships

select\_related returns a QuerySet that “follows” foreign-key relationships–either One-to-One or One-to-Many–selecting additional related-object data as needed. Under the hood the Django ORM creates a SQL join and includes the fields of a related object in the SELECT statement, which results in all related objects being included in a single more complex database query.

prefetch\_related performs a separate lookup for each relationship and “joins” them together with Python, not SQL. This allows it to prefetch many-to-many and many-to-one objects, which cannot be done using select\_related.

At first glance it might seem the issue here is a Foreign Key one and calls for select\_related. But in fact it is a Many-to-Many issue of authors to reviews since for one article there can be many reviews by many authors.

Studying your QuerySets deeply is vital for intermediate-to-advanced Django developers. There are many built-in methods available and understanding how QuerySets are built, the fact they are unique and also lazy, can yield insight into speeding up performance.

We can access the Python shell directly from the command line to play around with our current QuerySets. After starting the shell the first step is to import the Book model from our books app. Then we can select all of it to see our three books.

docker-compose exec web python manage.py shell

>>> from books.models import Book

>>> Book.objects.all()

<QuerySet [<Book: Django for APIs>, <Book: Django for Beginners>, <Book: Django for Profe\

ssionals>]>

We want to look at just the “Professionals” book now. If we were relying on the default ids for each Book, we could simply use the get() method but since we are using UUIDs we will instead use the contains() method and store the result as a variable called pros. Finally we can look at the all the reviews stored on it.

>>> pros = Book.objects.get(title\_\_contains="Professionals")

>>> pros

<Book: Django for Professionals>

>>> pros.reviews.all()

<QuerySet [<Review: 1>, <Review: 2>, <Review: 3>, <Review: 4>, <Review: 5>, <Review: 6>, \

<Review: 7>, <Review: 8>, <Review: 9>, <Review: 10>]>

A very useful third-party package is django-extensions which comes with a number of Django command extensions. Among them is shell\_plus, which automatically loads all app models in the shell. This can save a lot of time over the life of a Django project and makes working with the Django ORM much easier. But for now, we can exit the Django shell by typing Ctrl + d.

By default a DetailView will return a queryset that calls the all() method. In other words,

Book.objects.all(). We want to update this to prefetch\_related all the reviews for each author in one go. Here is how the update looks in our books/views.py file.

# books/views.py

...

class BookDetailView(

LoginRequiredMixin,

PermissionRequiredMixin,

DetailView):

model = Book

context\_object\_name = "book"

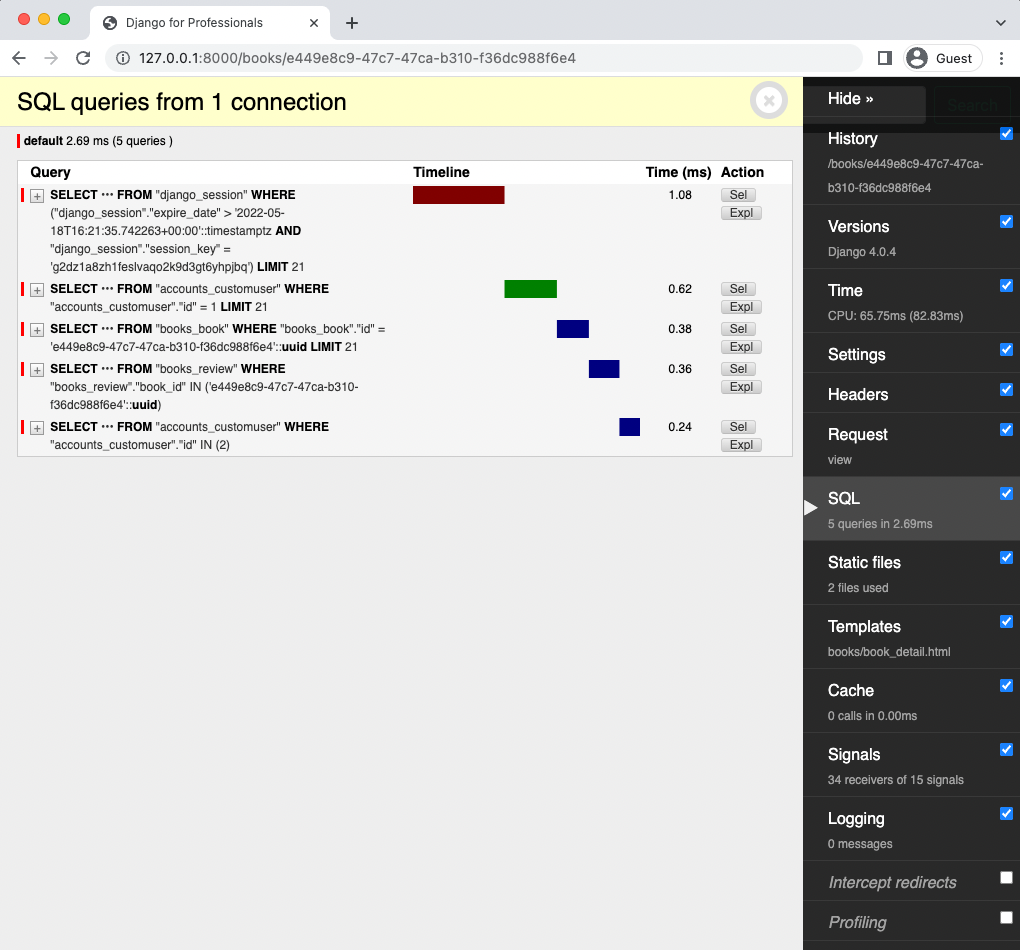
template\_name = "books/book\_detail.html"

login\_url = "account\_login"

permission\_required = "books.special\_status"

queryset = Book.objects.all().prefetch\_related('reviews\_\_author',)

The double underscore here is a lookup which is commonly used to filter QuerySets.

If we refresh the web page for the “Professionals” book now django-debug-toolbar shows that we’ve reduced our SQL queries from 14 to 5! Which makes sense since instead of 10 different queries for each review and author we now do it once (assuming all the reviews are by the same author). 

Five Queries

**Indexes**

Indexing is a common technique for speeding up database performance. It is a separate data structure that allows faster searches and is typically only applied to the primary key in a model. The downside is that indexes require additional space on a disk so they must be used with care.

Tempting as it is to simply add indexes to primary keys from the beginning, it is better to start without them and only add them later based on production needs. A general rule of thumb is that if a given field is being used frequently, such as 10-25% of all queries, it is a prime candidate to be indexed.

Historically an index field could be created by adding db\_index=True to any model field. For example, if we wanted to add one to the id field in our Book model it would look as follows (don’t actually implement this though).

# books/models.py

...

class Book(models.Model):

id = models.UUIDField(

primary\_key=True,

db\_index=True, # new

default=uuid.uuid4,

editable=False)

...

This change would need to be added via a migration file and migrated.

Starting in Django 1.11, class-based model indexes were added so can include in the Meta section instead. So you could write the previous index as follows instead:

# books/models.py

...

class Book(models.Model):

id = models.UUIDField(

primary\_key=True,

default=uuid.uuid4,

editable=False)

…

class Meta:

indexes = [

models.Index(fields=["id"], name="id\_index"),

]

permissions = [

("special\_status", "Can read all books"),

]

Since we’ve changed the model we must create a migrations file and apply it.

docker-compose exec web python manage.py makemigrations books

Migrations for 'books':

books/migrations/0005\_book\_id\_index.py

- Create index id\_index on field(s) id of model book

docker-compose exec web python manage.py migrate

Operations to perform:

Apply all migrations: account, accounts, admin, auth, books, contenttypes, sessions, si\

tes

Running migrations:

Applying books.0005\_book\_id\_index... OK

**Caching**

A cache is an in-memory storing of an expensive calculation. Once executed it doesn’t need to be run again!

Consider that our Bookstore project is a dynamic website. Each time a user requests a page our server has to make various calculations including database queries, template rendering, and so on before servicing it. This takes time and is much slower than simply reading a file from a static site where the content does not change.

On large sites, though, this type of overhead can be quite slow and caching is one of the first solutions in a web developer’s tool bag. Implementing caching on our current project is definitely overkill, but we will nonetheless review the options and implement a basic version.

Django has its own cache framework which includes four different caching options in descending order of granularity:

1) The per-site cache is the simplest to set up and caches your entire site.

2) The per-view cache lets you cache individual views.

3) Template fragment caching lets you specify a specific section of a template to cache.

4) The low-level cache API lets you manually set, retrieve, and maintain specific objects in the cache.

Why not just cache everything all the time? Two big reasons. The first is that cache memory is expensive since it is stored as RAM. Think about the cost of going from 8GB to 16GB of RAM on your laptop vs. 256GB to 512GB of hard drive space! Another is the cache must be “warm,” that is filled with updated content, so depending upon the needs of a site, optimizing the cache so it is accurate, but not wasteful, takes quite a bit of tuning.

If you wanted to implement per-site caching, which is the simplest approach, you’d add

UpdateCacheMiddleware at the very top of the MIDDLEWARE configuration in django\_project/settings.py and FetchFromCacheMiddleware at the very bottom. Also set three additional fields CACHE\_MIDDLEWARE\_ ALIAS, CACHE\_MIDDLEWARE\_SECONDS, and CACHE\_MIDDLEWARE\_KEY\_PREFIX.

# django\_project/settings.py

MIDDLEWARE = [

'django.middleware.cache.UpdateCacheMiddleware', # new

'django.middleware.security.SecurityMiddleware',

'django.contrib.sessions.middleware.SessionMiddleware',

'django.middleware.common.CommonMiddleware',

'django.middleware.csrf.CsrfViewMiddleware',

'django.contrib.auth.middleware.AuthenticationMiddleware',

'django.contrib.messages.middleware.MessageMiddleware',

'django.middleware.clickjacking.XFrameOptionsMiddleware',

'debug\_toolbar.middleware.DebugToolbarMiddleware',

'django.middleware.cache.FetchFromCacheMiddleware' # new

]

CACHE\_MIDDLEWARE\_ALIAS = "default" # new

CACHE\_MIDDLEWARE\_SECONDS = 604800 # new

CACHE\_MIDDLEWARE\_KEY\_PREFIX = "" # new

The only default you might want to adjust is CACHE\_MIDDLEWARE\_SECONDS which is the default number of seconds (600) to cache a page. After the period is up, the cache expires and becomes empty. A good default when starting out is 604800 seconds or 1 week (60secs x 60minutes x 168hours) for a site with content that doesn’t change very often. But if you find your cache filling up rapidly or you are running a site where the content changes on a frequent basis, shortening this setting is a good first step.

Implementing caching is strictly optional at this point though. Once a website is up and running the need for caching–per site, per page, and so on–will quickly become apparent. As a site grows in size, a dedicated and separate caching server often makes sense. The two most popular options for this are Redis and Memcached which, as of Django 4.0, both come with built-in Django support.

**Front-end Assets**

Another major source of bottlenecks in a website is loading front-end assets. CSS and JavaScript can become quite large and therefore tools like django-compressor can help to minimize their size.

Images are often the first place to look in terms of asset size. The static/media file set up we have in place will scale to a quite large size, but for truly large sites it is worth investigating the use of a Content Delivery Network (CDN) for images instead of storing them on the server filesystem.

You can also serve different size images to users. For example, rather than shrink down a large book cover for a list or search page you could store a smaller thumbnail version instead and serve that where needed. The third-party easy-thumbnails package is a good place to start for this.

A fantastic free e-book on the topic is Essential Image Optimization by Addy Osmani that goes into depth on image optimization and automations.

As a final check there are automated tests for front-end speed such as Google’s PageSpeed Insights that will assign a score based on how quickly a page loads.

**Next Steps**

**Git**

There’s been a lot of code changes in this chapter so make sure to commit everything with Git.

Remove-Item -Recurse -Force .git

git init

git status

git add .

git commit -m “Chapter 15. Search”

If you have any errors make sure to look at your logs with docker-compose logs and compare your code with the official source code on Github.

**Conclusion**

There is an almost endless list of performance optimizations that can be applied to a project. But take care to recall Donald Knuth’s sage advice and not go too crazy on this. Bottlenecks will reveal themselves in production and should largely be addressed then; not in advance. There are also several third-party packages that can be helpful in identifying N+1 issues, most notably nplusone, django-zen-queries, and django-auto-prefetch.

You should remember that performance problems are a good problem to have! They are fixable and mean that your project is being heavily used which is far better than having a completely optimized but largely ignored website.