

Learning Objectives

- Use Django Debug Toolbar to monitor performance
- Implement database indexes to speed up finding records
- Use `select_related` to fetch all of the data in one query
- Use bulk operations (create, update, delete) to reduce the number of queries

Clone Blango Repo

Clone Blango Repo

Before we continue, you need to clone the blango repo so you have all of your code. You will need the SSH information for your repo.

In the Terminal

- Clone the repo. Your command should look something like this:

```
git clone git@github.com:<your_github_username>/blango.git
```

- You should see a blango directory appear in the file tree.

You are now ready for the next assignment.

Introduction to Optimization

Introduction to Optimization

Before we start trying to speed up our Django apps (or indeed any other software) we need to know which piece of code is in need of the most help. There's an old saying (by [Sir Tony Hoare](#)) that “premature optimization is the root of all evil”. That means that we shouldn't waste time implementing optimizations unless we know that they will fix the bottlenecks of our application.

As a general rule, write performant code if it can be done in about the same amount of time as regular code.

▼ Writing Performant Code

Here's a trivial, non-Django example to illustrate this point. It lists all the items in the root directory.

```
i = 0
while True:
    directory_list = listdir("/")
    print(f"Item at {i} is {directory_list[i]}")
    i = i + 1
    if i == len(directory_list):
        break
```

Immediately we can see that this is inefficient, as we're re-listing the directory in each loop iteration. We can optimize the code by moving just one line, so the directory is only listed once:

```
directory_list = listdir("/")
i = 0
while True:
    print(f"Item at {i} is {directory_list[i]}")
    i = i + 1
    if i == len(directory_list):
        break
```

This type of optimization is probably fine to do without analyzing the code, as it's no extra work. Of course this example is a little contrived and a more pythonic way of doing this same thing would be like this:

```
for i, item in enumerate(listdir("/")):
    print(f"Item at {i} is {directory_list[i]}")
```

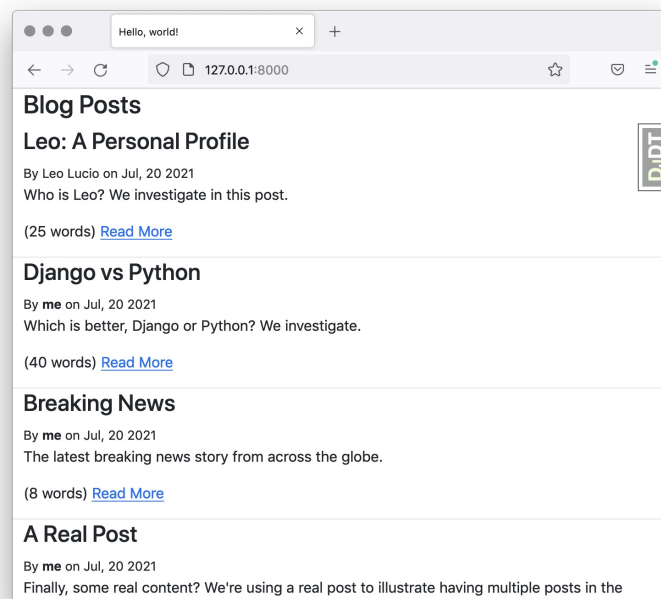
Now back to Django.

We'll first look at how to use the tool Django Debug Toolbar to see where our application is slow.

Django Debug Toolbar

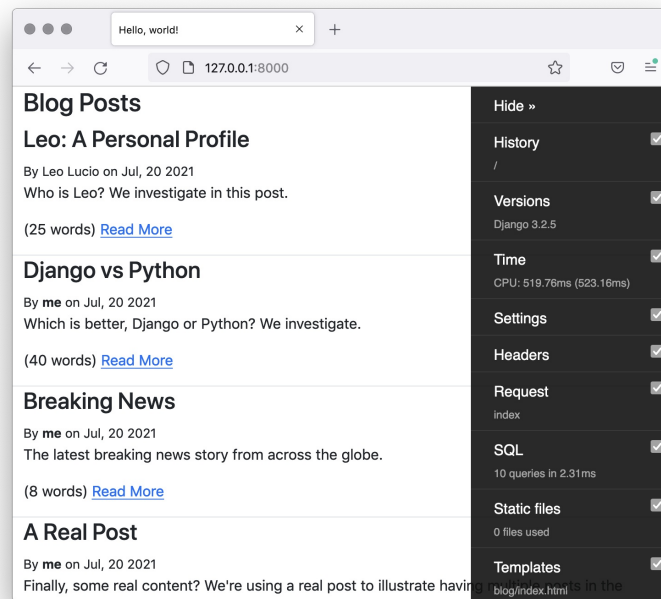
Django Debug Toolbar is a third party library that adds a toolbar to the side of your pages rendered by Django.

Here we see Django Debug Toolbar (DjDT), collapsed, in a browser window.



Django Debug toolbar closed

Clicking the **DjDT** button on the right will open up the menu:



Django Debug toolbar open

We won't go into detail on every section, but here's a brief overview:

- **History** shows a list of recent requests that have been made to the Django server recently. You can load the requests back into DjDT to view information about them.
- **Versions** shows the versions of Django, Python, DjDT and other libraries you have installed.
- **Time** shows how much time it took to generate the response.
- **Settings** will tell you all your current Django settings, including default ones which you won't be able to see in `settings.py`. It automatically hides sensitive setting values.
- **Headers** shows the HTTP headers in the request and response.
- **Request** shows lots of information about the request, such as which view handled it, the arguments passed to the view function, cookies, session data and GET and POST data.
- **SQL** we will look at in more detail, but in brief, it shows the SQL queries made by the Django ORM.
- **Static files** can help you debug problems with your static files. It shows you which static files are available, which path can be used to access them, and which were used in the request. This could be useful if you have two static files with the same name and Django is not loading the one you expect.

- **Templates** show how your template was rendered, how it extends other templates, and which templates it includes.
- **Cache** shows how you used the Django cache in generating the request, as well as telling you how many cache hit (keys were set in the cache) and misses (keys were not set) there were.
- **Signals** tells you which Django signals were triggered and which objects received them.
- **Logging** will show you the log messages that have been generated by your application. Note that since Blango has a custom LOGGING setting, we won't see anything in this panel. [This StackOverflow Answer](#) details how to re-add the DjDT logger handler to the LOGGING setting, if you want to be able to use this feature.
- **Intercept Redirects** is a setting that you can turn on and off. If on, then if Django sends a redirect to another page, you won't be redirected. Instead, DjDT intercepts the response and shows you where you *would have* been redirected to. You can then click a link and continue to that page.
- **Profile** shows you profile information about your Python code. This is displayed as a stack trace showing how long each function call took, how many times it was called, and the total time for each. The [Python Profile Documentation](#) gives more information on how to interpret this output. The output can help you know where to optimize your Python code: the best targets are functions that take a long time to run, or functions that might be pretty fast but are called 100s or 1000s of times. You probably won't gain much by trying to optimise a fast function that's only called once! Profiling data is turned off by default as running code through the profiler can slow it down, so to use it, you need to check the **Profiling** checkbox then refresh to page to view it.

Now that we've got an overview of the data that DjDT can show us, let's get it set up.

Installing & Configuring Django Debug Toolbar

Installing and Configuring Django Debug Toolbar

You'll now go through the process of getting DjDT installed and set up. For security, DjDT will only show up for clients whose IP address you've designated as being allowed. If you're running Django on your local machine, this is easy, just use `127.0.0.1`. Allowing access when running on the Codio Platform is a bit trickier as you first need to find out what Django "sees" your IP address is. Note that this will be different than your *actual* IP address, since requests to the Codio platform pass through a load balancer/reverse proxy we're actually getting that IP address.

We'll create a new view to return the IP address that's connected to Django. This is in `request.META["REMOTE_ADDR"]`. Create a view function like this:

```
def get_ip(request):
    from django.http import HttpResponse
    return HttpResponse(request.META['REMOTE_ADDR'])
```

[Open urls.py](#)

Then add a URL mapping to the view. The mapping will look something like this:

```
path("ip/", blog.views.get_ip)
```

Start the Django dev server and visit the URL you set up. You should see the IP address; it will probably start with `192.168`, for example, `192.168.11.179`. Remember this IP address as we'll need it later. **Important**, we need to keep `get_ip` and the URL mapping, as this might come in handy later.

[View Blog](#)

You need to stop the dev server before continuing. Press `Ctrl + C` on the keyboard. Now we can install DjDT, as usual it's installed with `pip` in the terminal:

```
pip3 install django-debug-toolbar
```

After it's installed, we need to make a few changes to our settings file. First, `debug_toolbar` needs to be added to the `INSTALLED_APPS` setting in your project.

Open settings.py

```
INSTALLED_APPS = [  
    # other existing settings truncated for brevity  
    "debug_toolbar",  
]
```

Then, `"debug_toolbar.middleware.DebugToolbarMiddleware"` must be added to the `MIDDLEWARE` setting, you can just add it as the first element in the list.

```
MIDDLEWARE = [  
    "debug_toolbar.middleware.DebugToolbarMiddleware",  
    # other existing settings truncated for brevity  
]
```

▼ DjDT Configuration Warning

The official Django Debug Toolbar configuration guide gives the following warning:

The order of `MIDDLEWARE` is important. You should include the Debug Toolbar middleware as early as possible in the list. However, it must come after any other middleware that encodes the response's content, such as `GZipMiddleware`.

Finally we need to add an `INTERNAL_IPS` setting, which is the list of IP address that are allowed to use DjDT. This will be the IP address you got from your `get_ip` view.

```
INTERNAL_IPS = ["192.168.11.179"]
```

Those are all the setting changes we need. Next, we need to add a URL mapping to DjDT's URLs. We'll configure this to only be set up if our `DEBUG` setting is `True`, for some extra security. Open your `urls.py`. Start by adding some imports, we need to import `debug_toolbar`, settings from `django.conf` and include from `django.urls`. All your imports should look like this after you've added them:

Open urls.py


```
import debug_toolbar
from django.conf import settings
from django.contrib import admin
from django.urls import path, include

import blog.views
```

Then we'll map the path `__debug__` to the DjDT's URLs, but only if we're in debug mode. After the existing `urlpatterns` definition, add this code:

```
if settings.DEBUG:
    urlpatterns += [
        path("__debug__/", include(debug_toolbar.urls)),
    ]
```

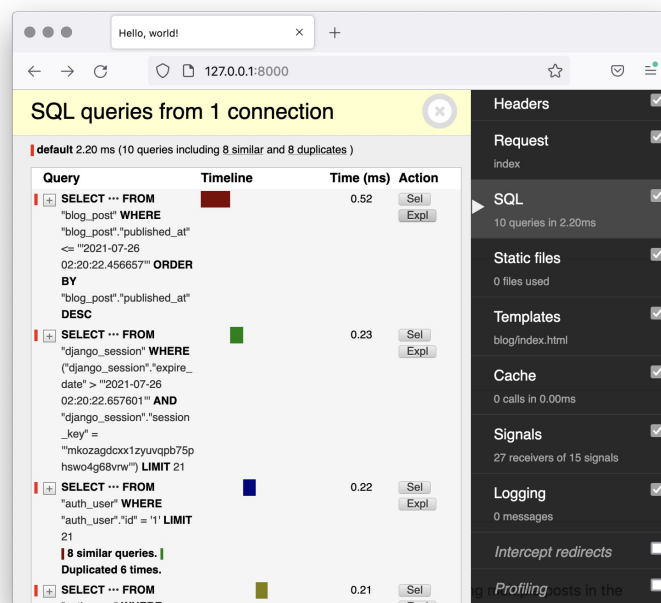
Now you're ready to try our DjDT. Load up the Blango main page in a browser, and check that you can see the DjDT on the side of the screen (refer back to the first image on the previous page).

[View Blog](#)

Explaining SQL Queries

Explaining SQL Queries

Now let's see how we can use DjDT to help us optimize our database. Click on the **SQL** panel, and you'll see a list of the queries that were made to the database.



django debug toolbar sql panel

▼ DjDT is no longer loading

If you started a new Codio session between when you installed DjDT and now, Codio has assigned your blog a new IP address. Add `/ip` in the blog URL to get the new IP address. Update `INTERNAL_IPS` in the `settings.py` file with this new value.

[Open settings.py](#)

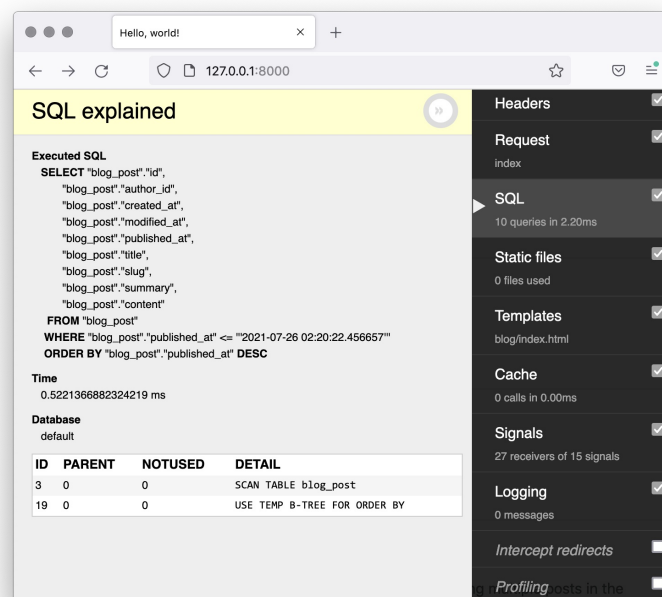
The *Timeline* column gives you a graphical look at how long each query took, and the **Time** column shows this numerically. Under the **Actions** column you can click **Sel** to show the full query that was executed and the

data that was returned, but we're more interested in the **Expl** button. Clicking this will show the *explain* statement for the SQL that was executed.

▼ What is a SQL explain?

When a SQL query is constructed (either written by hand or by the Django ORM) we can have the database explain *how* it will execute the query. For example, when running in a database shell, instead of executing `SELECT * FROM table...` we would instead do `EXPLAIN SELECT * FROM TABLE...`. This output will tell us if the query will use indexes or if the entire table needs to be scanned to get the results. The full EXPLAIN output needs database expertise to be completely understood, but there are a few things you can look out for. If you see that your query performs a table scan, then it's a sign that you'll need an index.

Click **Expl** for the top SQL query (the one that begins with `SELECT ... FROM "blog_post" WHERE`). You'll see the explanation of the query in the table at the bottom of the window.



The screenshot shows the Django debug toolbar with the 'SQL explained' panel open. The panel displays the executed SQL query, the time taken to execute it, and a table with database operation details.

Executed SQL

```
SELECT "blog_post"."id",
       "blog_post"."author_id",
       "blog_post"."created_at",
       "blog_post"."modified_at",
       "blog_post"."published_at",
       "blog_post"."title",
       "blog_post"."slug",
       "blog_post"."summary",
       "blog_post"."content"
FROM "blog_post"
WHERE "blog_post"."published_at" <= "2021-07-26 02:20:22.456657"
ORDER BY "blog_post"."published_at" DESC
```

Time
0.5221366882324219 ms

Database
default

ID	PARENT	NOTUSED	DETAIL
3	0	0	SCAN TABLE blog_post
19	0	0	USE TEMP B-TREE FOR ORDER BY

The right sidebar of the toolbar shows various debugging options: Headers, Request, SQL (selected), Static files, Templates, Cache, Signals, Logging, Intercept redirects, and Profiling.

django debug toolbar sql explained

We can see the two rows under the *DETAIL* column, each of these basically shows what operation the database performed to fetch and order the data. In our case, the database performed a table scan (`SCAN TABLE blog_post`) and then ordered the data (`USE TEMP B-TREE FOR ORDER BY`).

By looking at the query and thinking about how the database might work, we can surmise what's happening. Since we're filtering by `published_at < now`, the database must be scanning the whole table to check each record's `published_at` matches the criteria. The records are then ordered using a

binary tree. Since there are only a handful of records in the database currently, it's still fast even though we're scanning the whole table. But as our data and table grow, we need to sure we don't start getting bottlenecked. A good way to prevent table scans is to make sure an index is used.

Database Indexes

A database index is like a lookup table that the database stores to tell it where to locate records. Without it, the database has to scan the whole table to find matching rows. Think of it like using a dictionary in Python. If you've used the right keys, you can fetch the record for that key instantly. Compare that to trying to find an element in a list, you have to iterate over records until you find the one you want.

We're going to add a database index to the `published_at` column. This will speed up both filtering and ordering against that column.

Open `blog/models.py` and find the `published_at` field definition in the `Post` model. Add a `db_index=True` argument to it:

[Open models.py](#)

```
published_at = models.DateTimeField(blank=True, null=True,  
                                   db_index=True)
```

Now, we need to run the makemigrations and migrate management commands. In the terminal:

- Stop the server - *Control + C*
- Make migrations - `python3 manage.py makemigrations`
- Migrate - `python3 manage.py migrate`
- Start the server again - `python3 manage.py runserver 0.0.0.0:8000`

Now refresh the post list page in your browser, and go into the SQL explain for the first query again. You should see that the explain *DETAIL* has only one row now.

[View Blog](#)

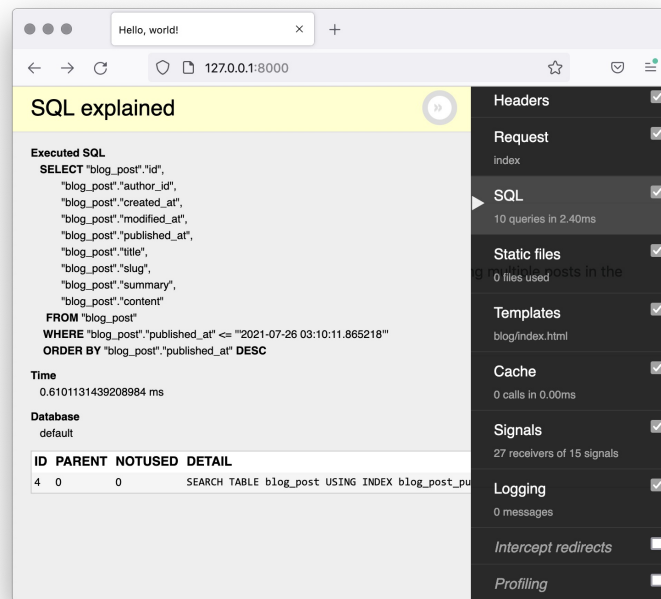


Figure 5

The content of the row is `SEARCH TABLE blog_post USING INDEX blog_post_published_at_9524a659` (`published_at<?`) so we know that our index is being used.

▼ Why don't we just index everything?

You might think that we can save a lot of time by creating indexes on every column to begin with. But this doesn't work too well. Indexes take up disk space and memory, which can waste resources. It also takes time to update an index when a record is inserted or updated, which can slow down these processes. So it's better to try to be aware of which of your columns you might be filtering and ordering by, and then adding the actual indexes you need accordingly.

We're going to quickly look at one other way of explaining queries, by using the `explain()` method on a `QuerySet`.

Explaining QuerySets

Explaining QuerySets

Sometimes you might want to explain queries without having to go through a view and Django Debug Toolbar. For example, in the Django Python shell you can quickly update a QuerySet and then explain it, without having to update view code, refresh the browser, and dig into DjDT.

The `explain()` method on a QuerySet will return a string in the same format as what you've seen in the Explain panel of DjDT. Let's use it to optimize comment fetching.

The Django Python shell should be in the bottom-left panel. Import the `Post` model.

Since the method of fetching comments is the same for any `Post` object, we can just do a one-liner to fetch the first one, then run `explain()` on its `comments` attribute:

```
In [1]: from blog.models import Post

In [2]: Post.objects.first().comments.explain()
Out[2]: '0 0 0 SEARCH TABLE blog_comment USING INDEX
        blog_comment_content_type_id_e26f0063
        (content_type_id=?)'
```

We can see it's using an index for the `content_type` field, but not when ordering (by `created_by`). Also, we know that when we're fetching `Comment` objects that their `object_id` must match the `Post` objects primary key, so the query should be filtering by `object_id`. We don't see an index being used when filtering by `object_id`, and indeed there isn't one. We can therefore optimize this query by adding an index on the `object_id` and `created_at` fields.

Jump back over to the `blog/models.py` and add the `db_index=True` argument to the `created_at` and `object_id` fields (these fields are in different classes). Run the `makemigrations` and `migrate` management commands, then restart your Django Python shell.

- Exit the shell - `exit()`
- Make migrations - `python3 manage.py makemigrations`
- Migrate - `python3 manage.py migrate`
- Start the shell again - `python3 manage.py shell`

Re-execute the `explain` and you should get output like this:

```
In [1]: from blog.models import Post

In [2]: Post.objects.first().comments.explain()
Out[2]: '0 0 0 SEARCH TABLE blog_comment USING INDEX
        blog_comment_object_id_134c93ed (object_id=?)'
```

Notice that now the query will use the `object_id` index, which should make queries faster, as we assume there are less `Comment` objects for a particular `object_id` than there are for a given `content_type`. But the query still uses a temporary binary tree for ordering, instead of the `created_at` index. Why is this? Without knowing the internals of the database it's hard to say. But one might guess that because there are only a few rows returned it's not necessary to use both indexes. Depending on the exact data, the database may change how it fetches data, what indexes it uses, and how it iteratively filters down results.

Whatever the case, when you know about explaining queries, you have a starting point for optimizations when you have a much bigger database on a production site.

Now we'll look at changing the Python code to optimize how Django fetches data from the database.

Selecting Related Objects

In the Django Debug Toolbar **SQL** panel, on our main page (the post list), you might have noticed that it does a lot of queries. One to get the list of `Post` objects, and then another for each `Post` to get the author's `User` object. This means for n posts, you'll make $n + 1$ database queries.

- Exit the shell - `exit()`
- Start the dev server - `python3 manage.py runserver 0.0.0.0:8000`
- [View Blog](#)

We can speed up the process by having Django fetch all the data in one query. In the backend, the database is performing a *join*, which means joining two tables together and fetching data from both simultaneously.

The name Django gives this is *select related*, and it's done with the `select_related()` method on a `QuerySet`. It takes one or more string arguments, which are the name of the fields for which to fetch related objects.

First open the index page and examine the DjDT **SQL** panel. Take a note of how many queries are executed and how long they take. In the authors case, with 7 `Post` objects, there are 10 queries which take 3.01ms.

▼ DjDT is no longer loading

If you started a new Codio session between when you installed DjDT and

now, Codio has assigned your blog a new IP address. Add `/ip` in the blog URL to get the new IP address. Update `INTERNAL_IPS` with this new value.

Now open your `blog/views.py` file. In the index view, add a `selected_related("author")` call to the posts fetching `QuerySet`:

[Open views.py](#)

```
posts =
    Post.objects.filter(published_at__lte=timezone.now()).select_related(
        "author")
```

That's all there is to it, go back to your browser and refresh the page, then examine the number of queries and time again. For the author, it's gone down to 3 queries and 1.12ms.

The reason this is faster is because there's a lot of overhead in opening the connection to the database and running the query, so it can be faster to open the connection once and send a single "bigger" query.

In summary, `select_related()` is quick to add and can make for some good speedups if you know you're going to use the related objects for each record you're querying.

Next we'll look at reducing the amount of data that is returned when fetching from the database.

Fetching Only Some Columns

Fetching data with the Django ORM is a two-stage process. First the data must be retrieved from the database, and then converted from raw bytes into a Python object (a model instance).

If we don't need all the data for a particular model, we can use the `QuerySet` methods `defer()` and `only()` to control which columns are retrieved and in-turn converted to Python. These methods are basically the opposite of each other: `defer()` takes one or more string arguments which are columns to *not* load data for. On the contrary, `only()` takes the columns to **only** load data for. That's not to say that the unloaded fields aren't accessible though. If you try to access one of them in Python, Django makes a database query to fetch that field.

As an example, on our post list page we only use a `Post`'s title, summary, content, slug, author and `published_at`. We don't use `created_at` or `modified_at`.

As an aside, we also don't use tags or comments but since these are related objects they're not fetched until they're used anyway.

These two methods of filtering the fetched fields are equivalent:

```
posts = (
    Post.objects.filter(published_at__lte=timezone.now())
    .select_related("author")
    .only("title", "summary", "content", "author",
         "published_at", "slug")
)
```

And

```
posts = (
    Post.objects.filter(published_at__lte=timezone.now())
    .select_related("author")
    .defer("created_at", "modified_at")
)
```

Try it out:

Try updating the index view to use `only()` and `defer()`, as per the two examples above. Use the Django Debug Toolbar (DjDT) to make sure that no extra SQL queries are run. Then, try removing a field from the `only()` call, or adding one to `defer()`. In DjDT, you should see that another query is now being made for each `Post` object to fetch that missing value.

[Open views.py](#)

When you're finished, revert your changes so you're fetching `Post` objects like this again:

```
posts = Post.objects.filter(published_at__lte=timezone.now()).select_related(
    "author")
```

Why are we reverting these changes? There are a couple of reasons. First, using `only()` and `defer()` doesn't always give us a speed boost. As we saw with the use of `select_related()`, most of the overhead was in creating connections and filtering not sending back the data. So you need to be first that having too many fields is a bottleneck before trying to resolve it.

The second reason is that these method's arguments need to be kept up to date. If you add a new field to a model, there's a reasonable assumption that the field is available for use in, say, a template, without worrying about if it's been loaded or not. If you don't keep `only()` or `defer()` up to date then you might end up making extra queries to get the data you need and

thus slow down your site without knowing about it. For this reason, if you do think you need to use one, prefer `defer()`, so that if you add fields they'll be pre-fetched and ready to use without having to update anything else.

The next speed-up we'll look at is performing bulk operations.

Bulk Operations

Bulk Operations

This speedup is also about reducing the number of queries to the database. By using bulk operations, Django can perform creates, updates and deletes with a single query.

Bulk Create

Bulk create allows you to create many objects of the same type at once, using the `bulk_create()` method on the model classes manager (the `objects` attribute). For example, for a `Post` this method would be at `Post.objects.bulk_create()`. It takes three arguments:

- `objs`: a list of objects to insert; these must be instances of the same model class.
- `batch_size` (optional): the number of objects to insert per query.
- `ignore_conflicts` (optional, defaults to `False`): if set to `True`, Django will ignore any records that failed to insert due to conflicts like duplicate constraints being violated.

For example, to create a few new `Tag` objects at once:

```
Tag.objects.bulk_create(
    Tag(value="programming"),
    Tag(value="learning"),
    Tag(value="news")
)
```

(we'll do a try-it-out session after introducing all the bulk operations)

There are a few caveats though, which are listed in the [bulk_create documentation](#)

Next, updating objects in bulk.

Bulk Update

There are actually two ways to bulk update objects in Django. In the first method, we update all the attributes we want to change on the model instances, and then call `bulk_update()` on the object manager. `bulk_update()` takes three arguments.

- `objs`: a list of model instances to update.
- `fields`: a list of field names to update.
- `batch_size` (optional): the number of objects to update in each query.

For example, let's say we wanted to update the publication date on each post, and increment it by one day.

```
posts = list(Post.objects.all())
for p in posts:
    p.published_at = p.published_at + timedelta(days=1)

Post.objects.bulk_update(posts, ["published_at"])
```

As with `bulk_create()`, there are some caveats, listed at the [bulk update documentation](#).

The other method of bulk updating uses the `update()` method on a `QuerySet`. This can be more efficient as you don't need to first fetch all the objects from the database and update them in Python code, the operation is performed entirely in the database. The downside is that all objects must have the column set to the same value. The `update()` method takes keyword arguments of the values to set.

For example, to unpublish all our posts, by setting `published_at` to `None`, we could do this:

```
Post.objects.all().update(published_at=None)
```

Finally, let's look at bulk deletion.

Bulk Delete

Bulk deletion is also performed on a `QuerySet`, with the `delete()` method. For example, here's how to delete all `Post` objects that were created more than a year ago but have never been published.

```
Post.objects.filter(published_at__isnull=True,
                    created_at__lt=timezone.now() -
                    timedelta(days=365)).delete()
```

Try It Out

Let's have each of our authors leave a comment on the posts they've written, thanking the readers. Since authors are much too busy to do this themselves, we can automate the process.

You can run these commands in a Django Python shell.

First, import the Post and Comment models.

```
In [1]: from blog.models import Post, Comment
```

Then, iterate over each Post object and create a Comment for it; keep track of the comments in a list.

```
In [2]: comments = []

In [3]: for post in Post.objects.all():
...:     comments.append(
...:         Comment(creator=post.author, content="Thank you
...:             for reading my post!", content_object=post)
...:     )
...:
```

Finally, let's insert them all at once:

```
In [4]: Comment.objects.bulk_create(comments)
Out[4]:
[<Comment: Comment object (None)>,
 <Comment: Comment object (None)>,
 <Comment: Comment object (None)>,
 <Comment: Comment object (None)>,
 <Comment: Comment object (None)>,
 <Comment: Comment object (None)>,
 <Comment: Comment object (None)>]
```

Now load up any post detail page in your browser, and you should see the authors have been busy leaving comments.

info

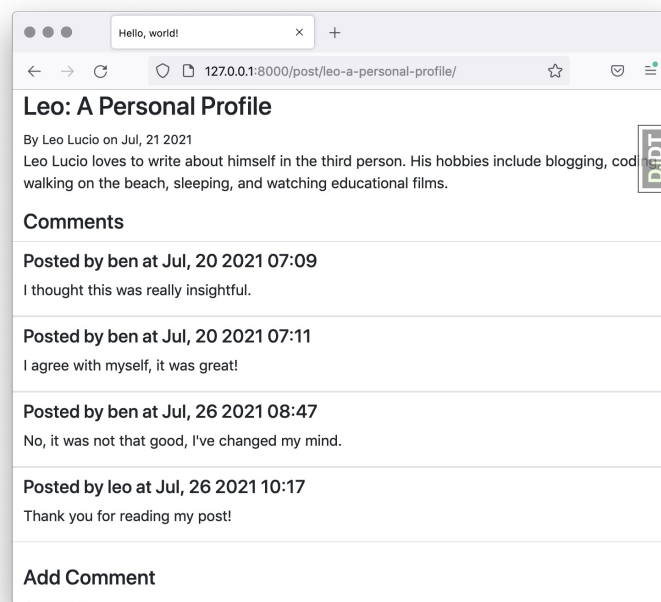
Starting a second terminal

Important, you will need to start a second terminal instance to run the dev server.

- Go to “Tools” in the Codio menu bar
- Select “Terminal”, a second terminal tab should appear in the workspace
- In the new terminal (not the shell), enter the following command:

```
python3 blango/manage.py runserver 0.0.0.0:8000
```

- Finally, [view the blog](#)



author comment added

Now we'll try a bulk update. We'll append the author's username to the end of the comment, to make it more personal. We'll need to fetch a reference to the comments to update. Switch back to the terminal tab with the shell.

```
In [5]: comments = Comment.objects.filter(content="Thank you for  
        reading my post!")
```

Then update the content of each one.

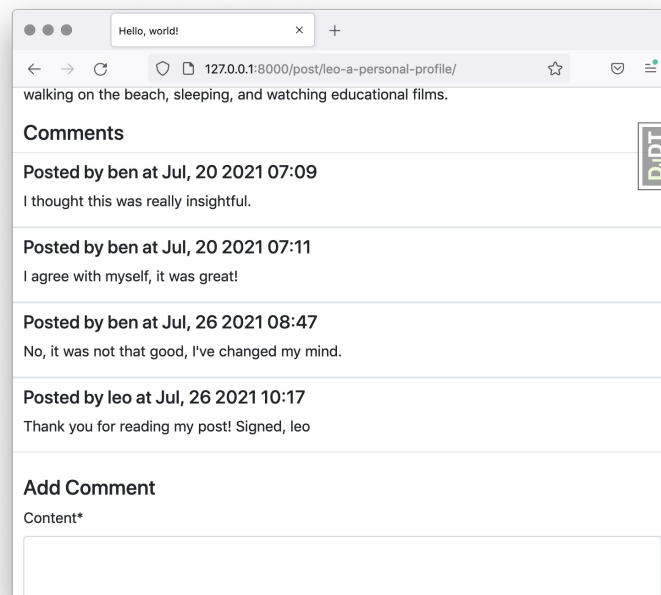
```
In [6]: for comment in comments:
...:     comment.content = comment.content + " Signed, " +
...:         comment.creator.username
...:
```

Then, we'll `bulk_update()` them all, making sure to indicate that it's the content field that's changed:

```
In [7]: Comment.objects.bulk_update(comments, ["content"])
```

Now refresh a post detail page in your browser, and you should see the newer, more personalized message.

[View Blog](#)



author comment updated

Unfortunately all our fraudulent commenting has caught up with us, and users have noticed the similarity of the comments. Let's remove all the comments before there's a huge backlash.

Something like this should do it:

```
In [8]: Comment.objects.filter(content__contains="Thank you for
...:         reading my post!").delete()
Out[8]: (14, {'blog.Comment': 14})
```

Refresh your browser for the last time, and notice that those bad comments are gone.

[View Blog](#)

Miscellaneous Tips

To summarize and give a few tips that don't fit anywhere else:

- Measure first, then optimize. You might be surprised where your bottlenecks actually are.
- Data filtering should take place in the database. It's much faster to filter in the database than iterate over results and filter in Python.
- Use indexes to make sure you're filtering efficiently.
- Retrieve the things you're going to need.
- Use bulk operations when you can.
- We've covered most of advice this contains, but for a more in depth look, read the [Django Database Access Optimization Documentation](#).

This wraps up Module 3. In Module 4, we'll look at some different ways of managing, registering and authenticating users.

Pushing to GitHub

Pushing to GitHub

Before continuing, you must push your work to GitHub. In the terminal:

- Commit your changes:

```
git add .  
git commit -m "Finish database optimization"
```

- Push to GitHub:

```
git push
```