Asynchronous support

Django has support for writing asynchronous ("async") views, along with an entirely async-enabled request stack if you are running under ASGI. Async views will still work under WSGI, but with performance penalties, and without the ability to have efficient long-running requests.

We're still working on async support for the ORM and other parts of Django. You can expect to see this in future releases. For now, you can use the `sync_to_async()` adapter to interact with the sync parts of Django. There is also a whole range of async-native Python libraries that you can integrate with.

Async views

Any view can be declared async by making the callable part of it return a coroutine - commonly, this is done using `async def`. For a function-based view, this means declaring the whole view using `async def`. For a class-based view, this means declaring the HTTP method handlers, such as `get()` and `post()` as `async def` (not its `__init__()`, or `as_view()`).

Note: Django uses `asgiref.sync.iscoroutinefunction` to test if your view is asynchronous or not. If you implement your own method of returning a coroutine, ensure you use `asgiref.sync.markcoroutinefunction` so this function returns `True`.

Under a WSGI server, async views will run in their own, one-off event loop. This means you can use async features, like concurrent async HTTP requests, without any issues, but you will not get the benefits of an async stack.

The main benefits are the ability to service hundreds of connections without using Python threads. This allows you to use slow streaming, long-polling, and other exciting response types.

If you want to use these, you will need to deploy Django using ASGI instead.

Warning: You will only get the benefits of a fully-asynchronous request stack if you have *no synchronous middleware* loaded into your site. If there is a piece of synchronous middleware, then Django must use a thread per request to safely emulate a synchronous environment for it.

Middleware can be built to support both sync and async contexts. Some of Django's middleware is built like this, but not all. To see what middleware Django has to adapt for, you can turn on debug logging for the `django.request` logger and look for log messages about *"Asynchronous handler adapted for middleware …"*.

In both ASGI and WSGI mode, you can still safely use asynchronous support to run code concurrently rather than serially. This is especially handy when dealing with external APIs or data stores.

If you want to call a part of Django that is still synchronous, you will need to wrap it in a `sync_to_async()` call. For example:

```
from asgiref.sync import sync_to_async

results = await sync_to_async(sync_function, thread_sensitive=True)(pk=123)
```

If you accidentally try to call a part of Django that is synchronous-only from an async view, you will trigger Django's asynchronous safety protection to protect your data from corruption.

Decorators

New in Django 5.0.

The following decorators can be used with both synchronous and asynchronous view functions:

- `cache_control()`
- `never_cache()`
- `no_append_slash()`
- `csrf_exempt()`
- `csrf_protect()`
- `ensure_csrf_cookie()`
- `requires_csrf_token()`
- `sensitive_variables()`
- `sensitive_post_parameters()`
- `gzip_page()`
- `condition()`

- conditional_page()
- etag()
- last_modified()
- require_http_methods()
- require_GET()
- require_POST()
- require_safe()
- vary_on_cookie()
- vary_on_headers()
- xframe_options_deny()
- xframe_options_sameorigin()
- xframe_options_exempt()

For example:

```
from django.views.decorators.cache import never_cache


@never_cache
def my_sync_view(request): ...


@never_cache
async def my_async_view(request): ...
```

Queries & the ORM

With some exceptions, Django can run ORM queries asynchronously as well:

```
async for author in Author.objects.filter(name__startswith="A"):
    book = await author.books.afirst()
```

Detailed notes can be found in Asynchronous queries, but in short:

- All `QuerySet` methods that cause an SQL query to occur have an `a` -prefixed asynchronous variant.
- `async for` is supported on all QuerySets (including the output of `values()` and `values_list()` .)

Django also supports some asynchronous model methods that use the database:

```
async def make_book(*args, **kwargs):
    book = Book(...)
    await book.asave(using="secondary")


async def make_book_with_tags(tags, *args, **kwargs):
    book = await Book.objects.acreate(...)
    await book.tags.aset(tags)
```

Transactions do not yet work in async mode. If you have a piece of code that needs transactions behavior, we recommend you write that piece as a single synchronous function and call it using `sync_to_async()`.

Performance

When running in a mode that does not match the view (e.g. an async view under WSGI, or a traditional sync view under ASGI), Django must emulate the other call style to allow your code to run. This context-switch causes a small performance penalty of around a millisecond.

This is also true of middleware. Django will attempt to minimize the number of context-switches between sync and async. If you have an ASGI server, but all your middleware and views are synchronous, it will switch just once, before it enters the middleware stack.

However, if you put synchronous middleware between an ASGI server and an asynchronous view, it will have to switch into sync mode for the middleware and then back to async mode for the view. Django will also hold the sync thread open for middleware exception propagation. This may not be noticeable at first, but adding this penalty of one thread per request can remove any async performance advantage.

You should do your own performance testing to see what effect ASGI versus WSGI has on your code. In some cases, there may be a performance increase even for a purely synchronous codebase under ASGI because the request-handling code is still all running asynchronously. In general you will only want to enable ASGI mode if you have asynchronous code in your project.

Handling disconnects

New in Django 5.0.

For long-lived requests, a client may disconnect before the view returns a response. In this case, an `asyncio.CancelledError` will be raised in the view. You can catch this error and handle it if you need to perform any cleanup:

```
async def my_view(request):
    try:
        # Do some work
        ...
    except asyncio.CancelledError:
        # Handle disconnect
        raise
```

You can also handle client disconnects in streaming responses.

Async safety

`DJANGO_ALLOW_ASYNC_UNSAFE`

Certain key parts of Django are not able to operate safely in an async environment, as they have global state that is not coroutine-aware. These parts of Django are classified as "async-unsafe", and are protected from execution in an async environment. The ORM is the main example, but there are other parts that are also protected in this way.

If you try to run any of these parts from a thread where there is a *running event loop*, you will get a `SynchronousOnlyOperation` error. Note that you don't have to be inside an async function directly to have this error occur. If you have called a sync function directly from an async function, without using `sync_to_async()` or similar, then it can also occur. This is because your code is still running in a thread with an active event loop, even though it may not be declared as async code.

If you encounter this error, you should fix your code to not call the offending code from an async context. Instead, write your code that talks to async-unsafe functions in its own, sync function, and call that using `asgiref.sync.sync_to_async()` (or any other way of running sync code in its own thread).

The async context can be imposed upon you by the environment in which you are running your Django code. For example, Jupyter notebooks and IPython interactive shells both transparently provide an active event loop so that it is easier to interact with asynchronous APIs.

If you're using an IPython shell, you can disable this event loop by running:

```
%autoawait off
```

as a command at the IPython prompt. This will allow you to run synchronous code without generating `SynchronousOnlyOperation` errors; however, you also won't be able to `await` asynchronous APIs. To turn the event loop back on, run:

```
%autoawait on
```

If you're in an environment other than IPython (or you can't turn off `autoawait` in IPython for some reason), you are *certain* there is no chance of your code being run concurrently, and you *absolutely* need to run your sync code from an async context, then you can disable the warning by setting the `DJANGO_ALLOW_ASYNC_UNSAFE` environment variable to any value.

Warning: If you enable this option and there is concurrent access to the async-unsafe parts of Django, you may suffer data loss or corruption. Be very careful and do not use this in production environments.

If you need to do this from within Python, do that with `os.environ`:

```
import os

os.environ["DJANGO_ALLOW_ASYNC_UNSAFE"] = "true"
```

Async adapter functions

It is necessary to adapt the calling style when calling sync code from an async context, or vice-versa. For this there are two adapter functions, from the `asgiref.sync` module: `async_to_sync()` and `sync_to_async()`. They are used to transition between the calling styles while preserving compatibility.

These adapter functions are widely used in Django. The asgiref package itself is part of the Django project, and it is automatically installed as a dependency when you install Django with `pip`.

```
async_to_sync()
```

```
async_to_sync(async_function, force_new_loop=False)
```

Takes an async function and returns a sync function that wraps it. Can be used as either a direct wrapper or a decorator:

```
from asgiref.sync import async_to_sync


async def get_data(): ...


sync_get_data = async_to_sync(get_data)


@async_to_sync
async def get_other_data(): ...
```

The async function is run in the event loop for the current thread, if one is present. If there is no current event loop, a new event loop is spun up specifically for the single async invocation and shut down again once it completes. In either situation, the async function will execute on a different thread to the calling code.

Threadlocals and contextvars values are preserved across the boundary in both directions.

`async_to_sync()` is essentially a more powerful version of the `asyncio.run()` function in Python's standard library. As well as ensuring threadlocals work, it also enables the `thread_sensitive` mode of `sync_to_async()` when that wrapper is used below it.

```
sync_to_async()
```

```
sync_to_async(sync_function, thread_sensitive=True)
```

Takes a sync function and returns an async function that wraps it. Can be used as either a direct wrapper or a decorator:

```
from asgiref.sync import sync_to_async

async_function = sync_to_async(sync_function, thread_sensitive=False)
async_function = sync_to_async(sensitive_sync_function, thread_sensitive=True)


@sync_to_async
def sync_function(): ...
```

Threadlocals and contextvars values are preserved across the boundary in both directions.

Sync functions tend to be written assuming they all run in the main thread, so `sync_to_async()` has two threading modes:

- `thread_sensitive=True` (the default): the sync function will run in the same thread as all other `thread_sensitive` functions. This will be the main thread, if the main thread is synchronous and you are using the `async_to_sync()` wrapper.
- `thread_sensitive=False` : the sync function will run in a brand new thread which is then closed once the invocation completes.

Warning: `asgiref` version 3.3.0 changed the default value of the `thread_sensitive` parameter to `True`. This is a safer default, and in many cases interacting with Django the correct value, but be sure to evaluate uses of `sync_to_async()` if updating `asgiref` from a prior version.

Thread-sensitive mode is quite special, and does a lot of work to run all functions in the same thread. Note, though, that it *relies on usage of* `async_to_sync()` *above it in the stack* to correctly run things on the main thread. If you use `asyncio.run()` or similar, it will fall back to running thread-sensitive functions in a single, shared thread, but this will not be the main thread.

The reason this is needed in Django is that many libraries, specifically database adapters, require that they are accessed in the same thread that they were created in. Also a lot of existing Django code assumes it all runs in the same thread, e.g. middleware adding things to a request for later use in views.

Rather than introduce potential compatibility issues with this code, we instead opted to add this mode so that all existing Django sync code runs in the same thread and thus is fully compatible with async mode. Note that sync code will always be in a *different* thread to any async code that is calling it, so you should avoid passing raw database handles or other thread-sensitive references around.

In practice this restriction means that you should not pass features of the database `connection` object when calling `sync_to_async()` . Doing so will trigger the thread safety checks:

```
# DJANGO_SETTINGS_MODULE=settings.py python -m asyncio
>>> import asyncio
>>> from asgiref.sync import sync_to_async
>>> from django.db import connection
>>> # In an async context so you cannot use the database directly:
>>> connection.cursor()
django.core.exceptions.SynchronousOnlyOperation: You cannot call this from
an async context - use a thread or sync_to_async.
>>> # Nor can you pass resolved connection attributes across threads:
>>> await sync_to_async(connection.cursor)()
django.db.utils.DatabaseError: DatabaseWrapper objects created in a thread
can only be used in that same thread. The object with alias 'default' was
created in thread id 4371465600 and this is thread id 6131478528.
```

Rather, you should encapsulate all database access within a helper function that can be called with `sync_to_async()` without relying on the connection object in the calling code.