Appendix

Changes in Django 5.1 and 5.2

This appendix will provide a summary of the changes introduced in Django 5.1 and 5.2. Here, new features will be covered so that you can understand how they can be utilized in your projects. Alongside new features, we will also discuss deprecations that may require you to consider changes, and removed features, as you will need to take action if your project uses them.

Changes in Django 5.1

This section will provide a summary of the Django 5.1 changes, including:

- Python compatibility
- `{% querystring %}` template tag
- PostgreSQL connection pools
- Authentication by default

The full release notes can be found at `https://docs.djangoproject.com/en/5.1/releases/5.1/`.

Python compatibility

The Python versions supported by Django 5.1 are 3.10, 3.11, 3.12, and 3.13 (introduced in 5.1.3). It is recommended that the latest release for each version is used, as this will ensure that a supported version is in use.

{% querystring %} template tag

New in Django 5.1 is the `{% querystring %}` template tag. This tag makes generating links that maintain existing query parameters much easier. The obvious use case for this is in pagination, where query parameters would typically need to be iterated through in order to add a new value for the next page.

The HTML for pagination might now look like this:

```
<div class="pagination">
  <span class="step-links">
    {% if page.has_previous %}
```

```
        <a href="{% querystring page=page.previous_page_number %}"> Previous</a>
      {% endif %}
      <span class="current">
        Page {{ page.number }} of {{ page.paginator.num_pages }}.
      </span>
      {% if page.has_next %}
        <a href="{% querystring page=page.next_page_number %}">Next</a>
      {% endif %}
    </span>
  </div>
```

The new template tag takes care of URL parameters while taking keyword arguments to update the URL parameters with data relevant to the pagination.

An example of where this would be useful could be a page of results from a race, where data displayed in a table includes the name of participants, their nationality, and their result. With a form alongside this table, there would be a search field, a filter for nationality, and an option for ordering by the fastest or slowest time. On form submission, the browser would make a GET request and fields would be added to the URL as parameters, which would look like this: `www.example.com/results?name=John&order_by=asc&page=1`.

These URL parameters would be maintained by the template tag instead of manually by the pagination markup, which has been necessary until now.

PostgreSQL connection pools

Creating database connections is time-consuming, requiring CPU time, memory, and network negotiation. This adds latency to the request/response cycle, which connection pools can reduce. Without connection pools, the application opens a new connection when it needs to access the database and then closes the connection when it completes the task.

Connection pool support for PostgreSQL was introduced in Django 5.1. Connection pools create connections to the database for the application to make use of. When it needs to access the database, a connection is requested from the pool. If one is available, then it is borrowed by the task in hand and returned once complete. By doing so, performance is improved, and resources are better managed because there is a limit on database connections imposed by the pool, thus preventing overloading of the database.

A new option has been added for `psycopg` called `pool`, which requires `psycopg[pool]` to be installed. This will ensure that the additional package `psycopg-pool` is installed alongside `psycopg`. The option is included inside the `OPTIONS` dictionary, as a dictionary containing `min_size`, `max_size`, and `timeout`, or by simply setting the pool option to `True` to use `ConnectionPool` defaults. Here is an example:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': config('DB_NAME'),
```

```
        'USER': config('DB_USER'),
        'PASSWORD': config('DB_PASSWORD'),
        'HOST': config('DB_HOST'),
        'OPTIONS': {
            "pool": {
                "min_size": 2,
                "max_size": 4,
                "timeout": 5
            },        },    }
}
```

Authentication by default

The new middleware, `LoginRequiredMiddleware`, will cause any unauthenticated requests to be redirected to the login page. This would be useful for sites where most of the content is private because it would require a user to be logged in by default. Without this middleware, all views that should be kept private would need the `login_required()` decorator to be added to enforce authentication.

To require authentication, add `"django.contrib.auth.middleware.LoginRequiredMiddleware"` to your `MIDDLEWARE` setting.

To allow exceptions to this, there is a `new login_not_required()` decorator.

Backward-incompatible changes

GeoDjango

If you have `django.contrib.gis` in your `INSTALLED_APPS` setting, then you should consider that support has been removed for old dependencies, which should be updated to ensure support and security fixes. These dependencies include:

- PostGIS 2.5 (release 2018)
- PROJ < 6 (version 6 released in 2019)
- GDAL 2.4 (released 2018).

When provided with a directory path, GeoIP2 will no longer open the city and country databases, now preferring the city database if available. Although the country database can still be used by passing the path to the constructor, it is a subset of the city database, so typically, both are not needed.

Dropped support

Dropped support for MariaDB 10.4, which was released in 2018.

Dropped support for PostgreSQL 12, which was released in 2019.

Deprecated features

For the full list of deprecated features, see the official release notes, but of those, the ones likely to impact most projects are:

- Overriding existing URL converters is deprecated. From Django 6.0, you will need to register custom path converters with `django.urls.register_converter()` and not overrides of the default converters.
- When calling `Model.save()` or `Model.asave()`, positional arguments are deprecated. Keyword-only arguments should be used instead.

Removed features

Features deprecated in Django 4.2 that have now been removed include:

- The `BaseUserManager.make_random_password()` method
- The `index_together` model meta option
- The SHA1, unsalted SHA1, and unsalted MD5 password hashers
- The `DEFAULT_FILE_STORAGE` and `STATICFILES_STORAGE` settings
- The `django.core.files.storage.get_storage_class()` function

Changes in Django 5.2

This section will provide a summary of the Django 5.2 changes, including:

- Python compatibility
- Automatic model imports in the shell
- Composite primary keys
- Easier `BoundField` overrides

The full release notes can be found at `https://docs.djangoproject.com/en/5.2/releases/5.2/`.

Python compatibility

The Python versions supported by Django 5.2 are 3.10, 3.11, 3.12, and 3.13. It is recommended that the latest release for each version is used because this will ensure that a supported version is in use.

Automatic model imports in the shell

When running the `shell` management command, all models from the installed apps will be automatically imported. Running `shell` with the `–verbosity` flag set to `2` or higher will provide further details of what is imported.

It is possible to customize this behavior by subclassing the shell command and overriding get_auto_imports, which will import each Python module path it returns. For example, create a custom shell command in your project's management commands (`project/management/commands/shell.py`):

```
from django.core.management.commands import shell

class Command(shell.Command):
    def get_auto_imports(self):
        return super().get_auto_imports() + [
             "django.conf.settings"
        ]
```

With this override, running `django-admin shell` would launch the Django shell with all the default automatic imports as well as the Django settings so you could execute `settings.DEBUG` without first having to import settings yourself.

Composite primary keys

With Django 5.2, it is possible to have primary keys consisting of values from multiple fields. To do this, define a `pk` field on a model as a `CompositePrimaryKey` and pass a list of model field names.

This change brings better alignment between Django and the capabilities of SQL databases. A typical use case for this would be on the through model of a many-to-many relationship, because these represent the combination of two foreign keys. This is more naturally represented by the combination of these two keys rather than an auto-incrementing integer value. For example:

```
from django.db import models

class StudentCourse(models.Model):
    pk = models.CompositePrimaryKey("course", "user")
    user = models.ForeignKey(
        settings.AUTH_USER_MODEL,
        on_delete=models.CASCADE
    )
    course = models.ForeignKey(Course, on_delete=models.CASCADE)
    enrollment_date = models.DateField()
```

Easier BoundField overrides

It is now possible to specify the `BoundField` of a `Form` class. This allows for more control over field rendering, such as custom CSS classes.

An example of how this would be used in a login form is:

```
from django import forms


class LoginBoundField(forms.BoundField):

    custom_class = "login_field"
```

```
        def css_classes(self, extra_classes=None):
            result = super().css_classes(extra_classes)
            if self.custom_class not in result:
                # add the custom class
                result += f" {self.custom_class}"
            return result.strip()


    class LoginForm(forms.Form):
        bound_field_class = LoginBoundField

        username = forms.CharField(
            label="Username",
            widget=forms.TextInput(attrs={"class": "username-class"}),
    )
        password = forms.CharField(widget=forms.PasswordInput)
```

The HTML rendered for this form would then wrap the fields with the custom class:

```
<div class="login_field">
    <label for="id_username">Username:</label>
    <input type="text" name="username" class="username-class" id="id_username">
</div>

<div class="login_field">
    <label for="id_password">Password:</label>
    <input type="password" name="password" id="id_password" required>
</div>
```

The rendered fields are bound by the `div` elements, which, themselves, are identified by the class set on the `LoginBoundField`. This is a useful way to do more with Django to improve the frontend HTML rendering and provide the classes for the styling to adapt to.

Minor features

Django admin

There are two enhancements if you're using `django.contrib.admin`:

- A new block, `extrabody`, is now available in the `admin/base.html` template, allowing for custom code to be added before the closing `</body>` tag. This allows for easier customization of the admin for adding JavaScript before the closing body tag.
- `URLField` values now render as a link, which is typically what you would want from a URL value in the admin. This makes it easier to navigate to the links stored.

Django authentication

To enhance security, Django periodically increases the tPBKDF2 password hasher iteration count, this time, from 870,000 to 1,000,000. This helps protect against brute-force attacks.

Asynchronous methods have also been added to the user manager, user model, and model backend. These include:

- `UserManager.acreate_user()`
- `User.aget_all_permissions()`
- `ModelBackend.aauthenticate()`

Password validator classes can now provide custom error messages through a new method, `get_error_message()`, to customize from the default message: `"This password is too short. It must contain at least <min_length> characters."`

Database backends

MySQL now defaults to the `utf8mb4` character set, which allows for the storage of 4-byte characters. This also allows for full Unicode support, meaning you can store emojis in the database where you might be storing blog comments or content where emojis would be commonly used.

Decorators

Asynchronous methods can now be wrapped with `method_decorator()`, for example:

```
from django.utils.decorators import method_decorator
from django.contrib.auth.decorators import login_required

class MyAsyncView(View):

    @method_decorator(login_required)
    async def get(self, request, *args, **kwargs):
        # Run any async processing here
        result = await some_async_operation()
        return HttpResponse(result)
```

In this example, you can see that an asynchronous function is able to now use the decorator to restrict it to logged-in users.

Forms

There are new form widgets:

- `ColorInput` – Meant for entering color in hexadecimal (rrggbb) format using the input type "color"
- `SearchInput` – Meant for entering queries using the input type "search"
- `TelInput` – Meant for entering phone numbers using the input type "tel"

Accessibility has also been improved for screen reader users with `aria-describedby`, used to associate error messages with their corresponding fields.

Migrations

`AlterConstraint` is a new no-op operation allowing migrations to modify database constraints without having to drop and recreate them.

Requests and responses

Areas that may impact the functionality of APIs or the use of HTTP requests in applications are as follows:

- A string representation of `HttpResponse.content` is now available with `HttpResponse.text`. This provides a string representation of the content, decoded using the defined charset, which defaults to UTF-8.
- A client's preferred media type can now be queried using the new method, `HttpRequest.get_preferred_type()`. This returns the preferred MIME type based on the given `Accept` header. Most browsers don't have a preference, so the first item defined by `media_types` would be chosen.
- There is a new `preserve_request` argument for `HttpResponseRedirect`, `HttpResponsePermanentRedirect`, which impacts the HTTP status codes used, and `redirect()`, which allows the user agent to reuse the HTTP method and body for redirection using specific status codes.

Backward-incompatible changes

Database backend API

A new model function, `_is_pk_set()`, can be used to establish whether a model instance's primary key is set.

`BaseDatabaseOperations.adapt_decimalfield_value()` is now a no-op and just returns the given value.GeoDjango

Support has been removed for:

- PostGIS 3.0 – released in 2019; 3.5 is the release at the time of writing
- GDAL 3.0 – released in 2019; 3.10 is the release at the time of writing

Dropped support for PostgreSQL 13

PostgreSQL ends support in November 2025.

However, Django 5.2 has support for PostgreSQL 14 and above.