

Google Summer of Code



DJANGO

[ADDING TYPE HINTS]

Name: Sachi Jain

GitHub: 🔗Skyiesac

LinkedIn: 🔗Sachi Jain

Discord: 🔗Skyiesac

Email: jainsachi1202@gmail.com

Time Zone: Indian Standard Time (IST), UTC +5:30

Project Size: large

Mentor: Thibaut Decombe

TABLE OF CONTENT

ABOUT 3
Executive Summary 4
Background (Why now?) 5
SOLUTION OVERVIEW 6
1. Ratchet based CI , instead of big bang : 6
Why mypy ? 6
Setup : 6
Optional: On pyright (to discuss with Django community): 7
SCOPE ANALYSIS 11
In Scope (12-week deliverables) 11
Phase 1 – Infrastructure and core (Week 1-2) : 13
Phase 2 – HTTP Layer (Week 3-4) : 13
Phase 3 – URLRouter (Week 5-6) : 14
Phase 4 — View Base Class (Week 7) 14
Phase 5 — ORM Protocol Foundations (Weeks 8–9) 15
Phase 6 — Ecosystem Compatibility and py.typed (Weeks 10–11) 16
Week 12 — Buffer and final review 17
PROJECT SCHEDULE 18
CONTRIBUTIONS SO FAR 19
Django: 19
Django ecosystem: 19
WHY ME ? 21
REFERENCES 22
FUTURE WORK 23

ABOUT

Greetings! I'm **Sachi Jain**, currently pursuing my pre-final year in B-Tech Computer Engineering at Dr. A.P.J Abdul Kalam Technical University, U.P., India. Over the past two years, I've immersed myself in the world of coding, cultivating a strong proficiency in Python , Django . My strong desire to explore and continuously learn motivates me to constantly discover new fields and technologies, leading to my recent selection for the **2026 global Djangonaut Space** (Team Mars).

My technical engagement with **Django core** includes implementing for migrations and database related PR and enhancing codebase integrity by introducing CI/CD workflows for migration checks. I have also contributed to the deprecation of legacy MariaDB versions and security-related suppressions in Python 3.14. These experiences have given me a deep understanding of the framework's internal architecture and the rigorous standards required for maintaining a world-class open-source project.

I am drawn to Django not only for its robust framework but for its culture. The mentorship I've received from maintainers and the Django community has been invaluable, and I am eager to give back to the community that has been so fundamental to my growth. I thrive in Django's well-documented and active environment and am committed to upholding the high standards of the organization through this GSoC project.

Contributing to Django has not only made me better at coding, but I also got to meet some amazing folks of the community and got to learn so much from them.

Executive Summary

The Add Types to Django project aims to transform Django from a framework that relies on third-party stub packages for type information into one that ships verified inline annotations as a first-class feature. By introducing type annotations directly into Django's source files, this project creates a live contract between Django's internals and the tools developers already use - mypy, pyright, and IDE hover documentation without requiring any external package.

The implementation centres on a phased ratchet approach: a scoped CI gate enforces typed modules can never regress as each layer is annotated in sequence, starting with the HTTP request and response layer, moving through URL routing and the view base class, and culminating in Protocol definitions that formally capture the ORM's expression contracts for the first time. The project integrates with Django's existing contribution workflow through a new typing guide, coordinates with the django-stubs maintainers to reduce duplicate maintenance, and lands py.typed only after an explicit compatibility sweep confirms the wider Django ecosystem is unaffected.

Background (Why now?)

The 2020 Technical Board explicitly rejected inline type annotations, citing three concerns:

1. Python typing was too young and in flux. Since then, the ecosystem has stabilized enormously. pyright ships in VS Code by default, mypy has reached 1.x, and ParamSpec, TypeVarTuple, typing.Self (PEP 673), and TypeAlias (PEP 613) are all stable stdlib features.

2. Competing tools made it impossible to pick one. The community has popularly been around mypy and pyright/pylance. The 2022 Django Developer Survey found 80% of Django developers use an IDE offering inline hover documentation , all of which depend on type information in the source.

3. Writing correct type hints is hard and raises the contribution barrier. This remains true for complex ORM internals, which is exactly why this proposal targets well-bounded, non-dynamic APIs first, and treats the ORM only at the Protocol / contract level.

The 2025 forum thread (Revisiting Types / DEP-14) shows that core contributors including Simon Charette, Jacob Walls, Carlton Gibson, and Thibaud Colas now support moving forward. Steering Council member Emma confirmed the issue has been moved forward and the next step is to "set up a team around this idea and write a proposal on the path forward."

SOLUTION OVERVIEW

1. Ratchet based CI , instead of big bang :

The CI setup for this project follows the same ratchet principle that governs the annotations themselves: start with zero scope, never regress, expand incrementally. The type checking job passes on day one because the strict scope is empty. Modules are added to the strict scope only after their annotations are complete and reviewed. This means no existing contributor's workflow is disrupted.

Why mypy ?

- **Rigor.** mypy strict with `disallow_untyped_defs = True` enforces complete signatures, justified ignores, and internal consistency. This gives the ratchet its teeth.
- **Ecosystem fit.** mypy is what `django-stubs and django-types` is built around. Using it as the gate means coordination with `django-stubs` will be a bit easy.
- **Familiarity.** Django's community already knows mypy. When a PR fails the mypy job, the error is immediately actionable – contributors know the tool and implementation.

Setup :

mypy.ini

```
[mypy]
python_version = 3.10
warn_unused_ignores = True

# Entire django package permissive by default
[mypy-django.*]
ignore_errors = True

# === STRICT SCOPE – expands week by week ===
# (sections added here as each module is completed)

# Added Week 2:
[mypy-django.utils.datastructures]
strict = True

[mypy-django.core.exceptions]
```

```
strict = True
```

.github/workflows/typing.yml

```
jobs:
  mypy:
    name: "Type checking (mypy) - strict mode“
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-python@v5
        with: { python-version: "3.12" }
      - run: pip install "mypy>=1.0"
      - run: mypy django/  # mypy.ini controls what is actually checked strictly

  pyright:          #optional
    name: "Type checking (pyright) - advisory"
    runs-on: ubuntu-latest
    continue-on-error: true  # Never fails the build; output visible for review
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-python@v5
        with: { python-version: "3.12" }
      - run: pip install pyright
      - run: pyright django/
```

Optional: On pyright (to discuss with Django community):

Most Django *users* experience type checking through pyright, not mypy because most IDE's pylance is powered by pyright. mypy and pyright disagree on variance inference, recursive `TypeAlias` resolution, and `Protocol` structural matching. Annotations that pass mypy strictly can produce wrong hover text in VS Code. (Source : Discussion)

Adding pyright to CI would validate the user-facing experience. The reason it is included here is that it introduces a second checker with different rules and error messages. It can be discussed in the community to be added as a non-blocking advisory job.

Concern: Django's CI is already one of the heaviest - dozens of jobs running. Adding a second type checker, even an advisory one, means another job consuming runner time on every PR.

Discussion : Ratchet Idea by ryanhiebert

2. Inline .py annotations, not .pyi stubs

Type information lives directly in Django's source files. Stubs drift when Django's source changes, Inline annotations stay correct automatically. They also power IDE hover documentation without requiring any external package.

The annotation style used throughout – `from __future__ import annotations` at the top of every annotated file ,means annotations are stored as strings at runtime and never evaluated, so there is zero performance cost:

```
# django/http/response.py

from __future__ import annotations
from collections.abc import Mapping, Iterator
from typing import Any

class HttpResponseBase:
    status_code: int
    charset: str | None

    def __init__(
        self,
        content_type: str | None = None,
        status: int = 200,
        reason: str | None = None,
        charset: str | None = None,
        headers: Mapping[str, str] | None = None,
    ) -> None: ...

    def __setitem__(self, header: str, value: str) -> None: ...
    def __getitem__(self, header: str) -> str: ...
    def has_header(self, header: str) -> bool: ...
    def items(self) -> Iterator[tuple[str, str]]: …
```

Example :

https://github.com/Skyiesac/django/commit/cd199c2ef308b30a1f3ad4fe3b7eb2483eb0e374

I've added the broad type hints in http - response.py , you can check out my work for it.

3. ORM Protocols and structural typing

The ORM's expression pipeline has always depended on two implicit contracts that have never been formally defined. Any object passed into the ORM as an expression must be able to compile itself to a SQL string and params tuple (`Compilable`), and must be able to resolve its

field references against a query (`Resolvable`). These contracts exist everywhere in `F`, `Value`, `Func`, `Aggregate`, every custom expression ever written.

This project formalises them as `Protocol` classes in a new `django/db/models/expressions/protocols.py`, directly implementing Simon Charette's structural typing suggestion. The key insight is that `Protocol` does not require any existing class to change :

```
from __future__ import annotations
from typing import Any, Protocol, TYPE_CHECKING

if TYPE_CHECKING:
    from django.db.models import fields
    from django.db.models.sql import compiler, query


class Compilable(Protocol):
    """
    Any object that can produce a PEP 249 (sql_string, params_tuple) pair.
    May define as_<vendor> methods to specialise per backend.
    """
    def get_source_expressions(self) -> list[Compilable | None]: ...
    def as_sql(
        self,
        compiler: compiler.SQLCompiler,
        connection: Any,
    ) -> tuple[str, tuple[Any, ...]]: ...


class Resolvable(Protocol):
    """
    Any object that can resolve its field references against a query,
    returning a Compilable ready for execution.
    """
    output_field: fields.Field

    def get_source_expressions(self) -> list[Resolvable | None]: ...
    def resolve_expression(
        self,
        query: query.Query,
        allow_joins: bool = True,
        reuse: set[str] | None = None,
        summarize: bool = False,
        for_save: bool = False,
    ) -> Compilable: ...
```

Link : [Comment by Simon Charettes]

One non-obvious design decision: as_sql returns tuple[str, tuple[Any, ...]] in the Protocol definition, but some ORM expressions return tuple[str, list[Any]] at runtime. After auditing the core expression classes (Func, Aggregate, Value, F), the correct return type is `tuple[str, list[Any] | tuple[Any, ...]]`, with an alias SQLParams = list[Any] | tuple[Any, ...] defined in the protocols module for readability.

The protocols are validated by annotating core expression classes to implement them and using `assert_type` in a test fixture a wrong protocol definition fails visibly rather than silently:

```
from typing import assert_type
from django.db.models import F, Value
from django.db.models.expressions.protocols import Compilable, Resolvable

def check_f_is_resolvable(f: F) -> None:
    assert_type(f, Resolvable)

def check_value_is_compilable(v: Value) -> None:
    assert_type(v, Compilable)
```

Protocol Related PR : Added a Protocol in django-stubs

4. Guarding against import cycles with TYPE_CHECKING

Adding annotations to a large interconnected framework creates real circular import risk. `HttpRequest` references `ResolverMatch`, `ResolverMatch` references `HttpRequest`, and the import graph has a cycle. Two complementary tools make this structurally impossible:

```
from __future__ import annotations  # PEP 563  never evaluated at import time
from typing import TYPE_CHECKING

if TYPE_CHECKING:
    # These imports are evaluated ONLY by the type checker, never at runtime.
    # Completely invisible to Python's import system
    from django.http import HttpRequest
    from django.urls import ResolverMatch
    from django.db.models.sql import compiler
```

Link: Work by Daniel Moisset

5. IDE Hover Documentation :

Attribute-level docstrings. The 2022 Django Developer Survey found 80% of Django developers use an IDE offering inline hover documentation. Right now hovering over HttpRequest shows nothing. During this project , we can add attribute docstrings to support Hover Documentation where needed .

SCOPE ANALYSIS

To ensure a successful delivery, the implementation is divided into three parts based on complexity:

In Scope (12-week deliverables)

Module / Area	Why	Risk
`django.utils.datastructures` - `MultiValueDict` Generic	**Must be typed before `QueryDict`** it is the base class; skipping it makes HTTP typing impossible	Medium
`django.core.exceptions` - exception hierarchy	Purely structural, no dynamics; unblocks HTTP annotations	Low
`Django.http.response` - `HttpResponseBase`, `HttpResponse`, `StreamingHttpResponse`, `FileResponse`, `JsonResponse`	Well-bounded, minimal dynamic behaviour	Low
`django.http.request` - `HttpRequest`, `QueryDict`, `HttpHeaders`	Most-used Django object; huge IDE impact	Medium depends on `MultiValueDict` above
`django.http.multipartparser` - `MultiPartParser`, `LazyStream`	Foundation for `HttpRequest.FILES` types	Low-Medium
`django.urls` - `path()`, `re_path()`, `include()`, `reverse()`, `resolve()`, `ResolverMatch`	Clear, stable signatures	Low-Medium
`Django.views.base`- `View.as_view()`, `View.dispatch()`, `View.setup()` only (time bound)	High user visibility; base class only, not mixins	Medium

ORM foundations - `Compilable` and `Resolvable` `Protocol` definitions	charettes' explicit proposal; minimal surface, high contributor leverage	Medium
ORM expressions – `Expression`, `F`, `Value`, `Func` typed to implement the above protocols	Validates the protocols are correct	Medium

Explicitly Out of Scope

- Full QuerySet generic typing (explicitly out of 12 weeks timeline , requires mypy plugin; too dynamic for inline)
- Manager.from_queryset() (mypy issue #2813)
- Field.__init__ kwargs expansion (django-stubs)
- **All CBV mixins**: TemplateResponseMixin, ContextMixin, get_context_data(**kwargs) – the **kwargs-threading patterns here are a known scope trap that would consume the entire project
- Settings module (plugin territory - PEP 649's is not stable enough right now , but it can help in better implementation.)
- Template engine internals
- py.typed before Week 12 … explicitly deferred until ecosystem compatibility is verified

Implementation Plan

Phase 1 – Infrastructure and core (Week 1-2) :

1.0.1 Set up scoped CI (mypy Strict mode)

Add `mypy.ini` to the repo root. The key design is per-module opt-in: only modules listed in the strict sections are checked. The rest of Django is explicitly excluded, so the CI job passes immediately on an empty strict scope and modules are ratcheted in as they are annotated.

1.0.2 Contributing typing guide docs

Add `docs` covering: how to run mypy locally, how to justify adding type hints, and the relationship with `django-stubs`. Written in Week 1, updated throughout.

1.0.3 `django/utils/datastructures.py`

This is a hard prerequisite for the entire HTTP layer and must come first. `MultiValueDict` is Django's internal multi-valued mapping used by `QueryDict`, `HttpRequest.FILES`, and form data. Without typing its base class first, `QueryDict` annotations are impossible.

MultiValueDict **internally stores lists** but **pretends to return single values** when you use `[]`. The trick is to declare the base class honestly as `dict[_KT, list[_VT]]`, then override `__getitem__` to return `_VT` so the type checker understands both the internal truth and the public interface.

1.0.4 `django/core/exceptions.py`

The cleanest target: a flat exception hierarchy with no dynamics, no generics, no circular imports. Annotating this unblocks HTTP annotations that reference `SuspiciousOperation`, `DisallowedHost`, `ValidationError`, etc.

Phase 2 – HTTP Layer (Week 3-4) :

2.0.1 `django/http/response.py`

2.0.2: `django/http/request.py`

The inline annotation will be:
from django.contrib.auth.base_user import AbstractBaseUser
from django.contrib.auth.models import AnonymousUser
class HttpRequest: user: `AbstractBaseUser | AnonymousUser  #or protocol`

2.0.3: `django/http/multipartparser.py` and `cookie.py`

`MultiPartParser` and `LazyStream` have clear, stable method signatures. Annotating this module completes the HTTP layer.

`[mypy-django.http.*]` → strict scope.

Phase 3 – URLRouter (Week 5-6) :

URL routing has clean, stable signatures. `TypeAlias` makes the view callable definition readable and reusable across the codebase.

3.0.1: `django/urls/resolvers.py`

3.0.2: `django/urls/conf.py`

3.0.3: `django/urls/base.py`

3.0.4: `django/urls/converters.py`, `django/urls/utils.py`

```
@type_check_only
class _Converter(Protocol):
    regex: str
    def __init__(self) -> None:
    def to_python(self, value: str) -> Any:
    def to_url(self, value: Any) -> str:
```

These are easier to tackle files . As these are light weighted and can be type syntax easily.

A new Protocol will be introduced here to make codebase easy.

`[mypy-django.urls.*]` → strict scope

Phase 4 — View Base Class (Week 7)

4.0.1 `django/views/base.py` View class only, no mixins

@classonlymethod is a custom Django descriptor. mypy cannot introspect custom descriptors natively and misreports as_view as a regular instance method. The # type: ignore[misc] suppression is the correct fix , it is explicitly justified in-line so reviewers understand it is intentional. The -> ViewCallable return type is still enforced by mypy at all call sites even with the ignore present.

View.as_view() is a classmethod that accepts **initkwargs, passes them to __init__, and returns a callable that accepts an HttpRequest plus URL kwargs and returns an HttpResponse. Typing this correctly requires ParamSpec:

```
_P = ParamSpec("_P")

class View:
    http_method_names: ClassVar[list[str]]
    @classonlymethod
    def as_view(cls, **initkwargs: Any) -> ViewCallable: ...  # type: ignore[misc]
    # The type: ignore[misc] is intentional and justified:
    # mypy cannot introspect @classonlymethod (a custom Django descriptor)
    # and misreports the return as a bound method. The -> ViewCallable
    # return type IS enforced at all call sites despite the suppression.

    def setup(
        self,
        request: HttpRequest,
        *args: Any,
        **kwargs: Any,
    ) -> None: ...
```

CBV mixins are explicitly deferred. `TemplateResponseMixin`, `ContextMixin`, and `get_context_data(**kwargs)` involve `**kwargs`-threading across multiple inheritance chains. Annotating them incorrectly is actively worse than leaving them unannotated. Only `View` base class methods – `as_view()`, `dispatch()`, `setup()`, and `http_method_not_allowed()` are annotated here. Mixins are left for future work.

`[mypy-django.views.base]` → scope.

Phase 5 — ORM Protocol Foundations (Weeks 8–9)

5.0.1 `django/db/models/expressions/protocols.py` (new module)

This implements Simon Charette's January 2025 proposal. The Django ORM has always relied on two informal contracts objects that compile to SQL and objects that resolve field references in the context of a query. These contracts exist in every expression, every annotation, every aggregate, but have never been formally defined. A new module introduces two `Protocol` classes that formalise them for the first time.

`from __future__ import annotations` at the top of the module ensures the self-referential `list[Compilable | None]` in the protocol body does not require a forward reference string .

The two protocols are `Compilable` (any object that can produce a SQL string and params tuple via `as_sql()`) and `Resolvable` (any object that can resolve its field references against a query and return a `Compilable`). These are the contracts the ORM has always depended on informally , this makes them explicit and checkable.

5.0.2 Validate protocols against core ORM expressions

`Expression`, `F`, `Value`, `Func`, and `Aggregate` are annotated to formally implement the protocols above. This step serves two purposes simultaneously: it validates the protocol definitions are correct (if a core class cannot satisfy the protocol, the definition needs adjustment), and it delivers immediately useful types for ORM contributors writing custom expressions.

Validation uses `assert_type` in a dedicated typing test fixture so that a wrong protocol definition causes a visible type-check failure rather than silently passing.

`[mypy-django.db.models.expressions.protocols]` → strict scope.

Phase 6 — Ecosystem Compatibility and py.typed (Weeks 10–11)

6.0.1 Ecosystem compatibility sweep

Before adding `py.typed`, the newly annotated Django is installed into three key third-party packages and mypy is run on their source. The test question is simple: does the new Django produce any new mypy errors that did not exist before? Packages tested:

- **django-filter** — extensive `QueryDict` and URL parameter usage; expected risk around the `QueryDict` generic binding.
- **django-allauth** — overrides `View` subclasses and uses `HttpRequest` heavily; expected risk around `View.dispatch` signature compatibility.

- **Django REST Framework** — wraps `HttpRequest` in its own `Request` class; highest risk package; expected issues around `HttpRequest` attribute compatibility, `FILES`, `META`, and `user`.

Where issues are found the handoff is active , minimal reproduction scripts and suggested `mypy.ini` config snippets are provided directly to the affected maintainers so they have concrete, actionable material rather than just a problem report.

6.0.2 Adding py.typed and further ORM

Only after the compatibility sweep passes does the `py.typed` marker get added. It is an empty file in the `django/` package root, as required by PEP 561. This tells every type checker that Django now ships inline type information for its annotated modules. Adding it before the sweep would flood every third-party project's CI with false-positive errors. It is the capstone of the project, not the starting point.

6.0.3 django-stubs coordination and final documentation

A detailed coordination issue is filed on the `django-stubs` repo documenting which modules Django now ships inline types for, every significant annotation decision and its rationale (the `MultiValueDict` generic design, the `request.user` union, the `JSONValue` recursive TypeAlias, the `ConverterProtocol` design), and a proposal that `django-stubs` defer to Django's inline types for these modules to eliminate duplicate maintenance.

The contributing typing guide started in Week 1 is finalised with complete examples and merged into Django's official documentation.

Week 12 — Buffer and final review

Reserved as an explicit buffer. The ORM validation in Phase 5 is the most likely source of unexpected complexity covariance and contravariance issues in protocol method signatures only surface at type-check time, not at writing time. Any outstanding compatibility sweep issues, documentation gaps, or PR review cycles that spilled over are resolved here before final submission.

PROJECT SCHEDULE

Week	Dates	Deliverable
0	May 1 – May 24	Community bonding: django-stubs/django-types audit, local mypy environment
1–2	May 25– June 9	Phase 1: mypy CI setup, typing guide skeleton, MultiValueDict generic, core/exceptions.py
3–4	June 10 – June 25	Phase 2: http/response.py, http/request.py, http/multipartparser.py, http/cookie.py HTTP layer strict scope locked
5–6	June 26– July 11	Phase 3: urls/exceptions.py, urls/resolvers.py, urls/base.py, urls/conf.py, urls/converters.py (Protocol), urls/utils.py URL strict scope locked
7	July 12 – July 19	Phase 4: views/base.py View.as_view(), dispatch(), setup(), http_method_not_allowed() views strict scope locked
8–9	July 20 – August 4	Phase 5: Compilable + Resolvable Protocol definitions; Expression, F, Value, Func, Aggregate validation ORM protocols scope
10–11	August 5 – August 20	Phase 6: Ecosystem sweep (django-filter, allauth, DRF); repro scripts for any issues; py.typed added; documentation
12	August 20 – August 27	The project will be reviewed and final reports will be submitted to the mentors.

CONTRIBUTIONS SO FAR

Django:

Ticket	Title	Status
#36639	Added CI step to run makemigrations --check against test models.	Merged
#36812	Dropped support for MariaDB < 10.11.	Merged
#36376	Fixed color suppression in argparse help on Python 3.14+	Merged
#36280	Exception checks with assertRaisesMessage()	Merged
#36528	Removed the unnecessary title attributes	Merged
#27574	Added support for ST_Distance_Sphere function on MySQL	In Process
#31834	Fixed sqlmigrate failure when altering unique_together.	Unreviewed
#36145	Fixed the FIRST_DAY_OF_WEEK setting for admin calendar.	Unreviewed
#36459	Added Aria labels to the buttons inside the AdminDate Widget.	In Process
#36436	Made CookieStorage.signer attribute private.	Unreviewed

Django ecosystem:

Django-stubs	Replaced IO[bytes] with _PostDataProtocol to fix HttpRequest.	Merged

Django project .com	Added missing labels for donation form input. (Djangonaut'26)	Merged
Django project .com	Fixed the missing labels for community images. (Djangonaut'26)	Merged

OTHER MINOR CONTRIBUTIONS :

- Litmuschaos (CNCF)
- alltheplaces
- pipeshub-ai

COLLABORATION WITH MENTORS

- **Guidance on Architecture & Design**: Provide insights on structuring the System.
- **Code Review & Best Practices**: Ensure the implementation aligns with Django's existing architecture and coding standards.
- **Technical Support**: Assist in debugging complex issues and refining key components.
- **Periodic Evaluation**: Assess the progress at each milestone and suggest improvements.
- **Integration Assistance**: Help with Static type integration and mapping.
- **PR shape and size.** Each phase will be one focused PR covering one logical layer , not a 2000-line diff that covers three phases at once.
- **Community decisions.** The typing guide content and the `py.typed` timeline both affect the broader Django community, not just this project, I will post them to the Django Forum for input.

WHY ME ?

I am an active contributor to the framework and have developed a deep understanding of its codebase through consistent, hands-on involvement. My work on a range of bug fixes and feature enhancements has given me a strong familiarity with Django's architecture, allowing me to quickly navigate the codebase and effectively extend its functionality.

I am a DjangoNaut 🚀 (Session- 6) in Team accessibility , which gives me an advantage of bonding with Django Team , also I am active on news related to Django. I have 2 years of experience in Django and internships , giving me a deep understanding of this cool framework.

Beyond the technical aspect, I genuinely resonate with the collaborative and open-source culture of the organization. I love interacting with maintainers and fellow contributors, participating in thoughtful discussions, and sharing insights. My clear communication skills enable me to express ideas well, write helpful documentation, and work smoothly with mentors and the broader community.

With my in-depth understanding and alignment with the organization's values, I'm confident in my ability to drive this project forward and make meaningful, sustainable improvements. Despite being an AI era , I will take full ownership of every single line of my code . Also I don't intend to send any proposal to any other Organization for Google Summer of Code 2026. Django will be my sole submission.I plan on continuing to contribute to Django even after GSoC along with solving any other issues or maintenance needs that come up in the future.

And aside from all that – I'm just a chill guy🎀 who loves clean code and Django.

REFERENCES

- Django Forum: "Revisiting types in Django / DEP-14" (Jan–Nov 2025)
- Django Developers Group: "Technical Board statement on type hints for Django" (Apr 2020)
- Django Developers Group: "PEP 484 type hinting in Django" (Aug 2016 – Oct 2019)
- Django Developers Group: "Transition Docs to Inline" (Jan 2024)
- DEP 0484 draft: Static type checking for Django (Oct 2019)
- GitHub: django/new-features issue #23
- Simon Charette's ORM Protocol sketch (Jan 2025, Django Forum post #11)
- Charettes ORM slides
- django-stubs: https://github.com/typeddjango/django-stubs
- django-types: https://github.com/sbdchd/django-types
- PEP 561: Distributing and Packaging Type Information
- PEP 563: Postponed Evaluation of Annotations
- PEP 613: TypeAlias
- PEP 673: typing.Self
- PEP 649: Deferred Evaluation of Annotations (in development)
- Adding type hints to ORM by Will mcgugan

FUTURE WORK

Being part of Django has honestly been such a cool experience. I got to dive deep into the formation of such a cool framework , picked up a bunch about CI, testing, and code coverage, and really saw how open-source works behind the scenes. It's been super rewarding and definitely leveled up my dev game.

What made it even better was the community, everyone's been so welcoming and helpful. It made me realize how much a solid developer community matters. Even when the GSoC will be wrapping up, I'm not planning to stop. I'll keep sending in solid PRs, help out newcomers, and throw around some new ideas to keep the project moving forward.

The CBV mixin chain (TemplateResponseMixin, ContextMixin, get_context_data) was explicitly deferred because **kwargs-threading across multiple inheritance chains cannot be annotated correctly without a community discussion about the right approach. That discussion is worth starting during Week 12.

The `django-stubs` coordination is a long-term relationship. As Django ships more inline annotations, django-stubs needs to keep reducing its duplicate coverage for those modules. I plan to remain involved in that coordination after GSoC ends and type hints for other parts .

Working on Django really made me appreciate clean, maintainable code and it got me thinking about how the founders even came up with such a great framework . Can't wait to keep building and joining the crew.

THANK YOU