Automated Django Integration Testing Setup

Team 1

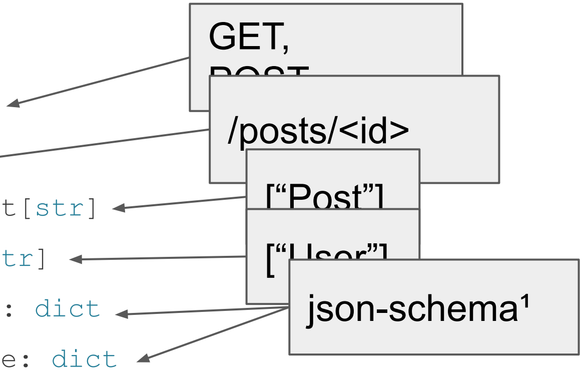
Murad Bashirov, Injoon Hwang, Sanga Choi, Letian Zhang

Project Recap

- Problem definition
  - When writing an integration test, it takes much more time to write a testing setup than to write a testing oracle.
    - Testing setup: register a user → user writes a post → user writes a chat → read the chat
    - Test oracle: `assert(chat.marked("AUTHOR"))`
- Goal
  - Automatically generate boilerplate testing setup codes for Django, so developers can focus on core logic.
- Method
  - Analyze codebase to make dependency graphs of models.
  - Given the dependency graph, generate API endpoint tests containing the required sequences of API calls to test endpoints.
- Evaluation
  - Run the generated tests, then count the number of valid tests and measure the coverage.

Extracting API information: What’s needed

```
@dataclass
class APINode:
    method: str
    path: str
    creates: list[str]
    uses: list[str]
    request_type: dict
    response_type: dict
```

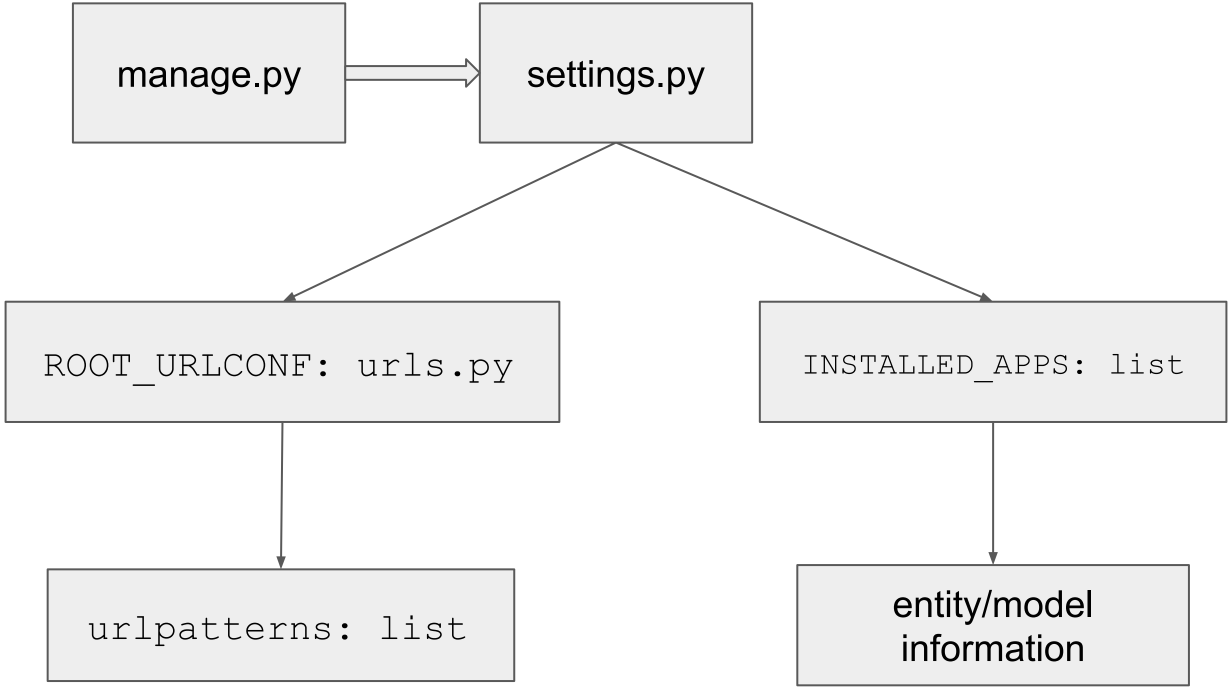


[1]json-schema: https://json-schema.org/

How to extract them?

- Models
- Url patterns
- View functions
- Serializers

Background


manage.py
settings.py
ROOT_URLCONF: urls.py
INSTALLED_APPS: list
urlpatterns: list
entity/model information

Models

- Pretty straightforward: {app_name}/models.py
- Collect models schema and what it depends on

```
class Post(models.Model):
    created_at = models.DateTimeField(auto_now_add=True)
    title = models.CharField(max_length=255)
    content = models.TextField()
    author = models.ForeignKey(User, related_name="posts", on_delete=models.CASCADE)


class Comment(models.Model):
    created_at = models.DateTimeField(auto_now_add=True)
    author = models.ForeignKey(User, related_name="comments", on_delete=models.CASCADE)
    content = models.TextField()
    post = models.ForeignKey(Post, related_name="comments", on_delete=models.CASCADE)
```

Url Patterns

- Pretty straightforward (2)

```
urlpatterns = [
   path('posts/', posts.views.PostList.as_view()),
   path('posts/<int:pk>/', post_views.PostDetail.as_view()),
   path('comments/', post_views.CommentList.as_view()),
   path('comments/<int:pk>/', post_views.CommentDetail.as_view()),
   path('users/', UserList.as_view()),
   path('users/<int:pk>/', UserDetail.as_view()),
]
```

```
path -> Class
```

Url Patterns

- Pretty straightforward (2)

```
urlpatterns = [
   path('posts/', posts.views.PostList.as_view()),
   path('posts/<int:pk>/', post_views.PostDetail.as_view()),
   path('comments/', post_views.CommentList.as_view()),
   path('comments/<int:pk>/', post_views.CommentDetail.as_view()),
   path('users/', UserList.as_view()),
   path('users/<int:pk>/', UserDetail.as_view()),
]
```

```
from posts import views as post_views
```

```
posts/views.py
```

View functions

- Currently only support subclasses of `APIView`

```
class PostList(APIView):
   def get(self, request, format=None):
       posts = Post.objects.all()
       serializer = PostSerializer(posts, many=True)
       return Response(serializer.data)

   def post(self, request, format=None):
       serializer = PostSerializer(data=request.data)
       if serializer.is_valid():
           serializer.save()
           return Response(serializer.data, status=status.HTTP_201_CREATED)

       return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

View functions

- Currently only support subclasses of `APIView`

```
class PostList(APIView):
    def get(self, request, format=None):
        posts = Post.objects.all()
        serializer = PostSerializer(posts, many=True)
        return Response(serializer.data)

    def post(self, request, format=None):
        serializer = PostSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data, status=status.HTTP_201_CREATED)

        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

View functions: Serializer Calls in view functions

GET list: `PostSerializer(posts, many=True)`
- First argument: Queryset
- Second arg: `many=True`

GET detail: `PostSerializer(post)`
- No second argument

POST: `PostSerializer(data=request.data)`
- No DB object passed
- `data` keyword argument is set

PATCH: `PostSerializer(post, data=request.data, partial=True)`
- DB object passed in
- `data` keyword argument is set
- assert `partial=True` because this is PATCH not PUT

DELETE: No serializer call, instead there's `<db_object>.delete()` call

Serializers

- Pretty straightforward (3)

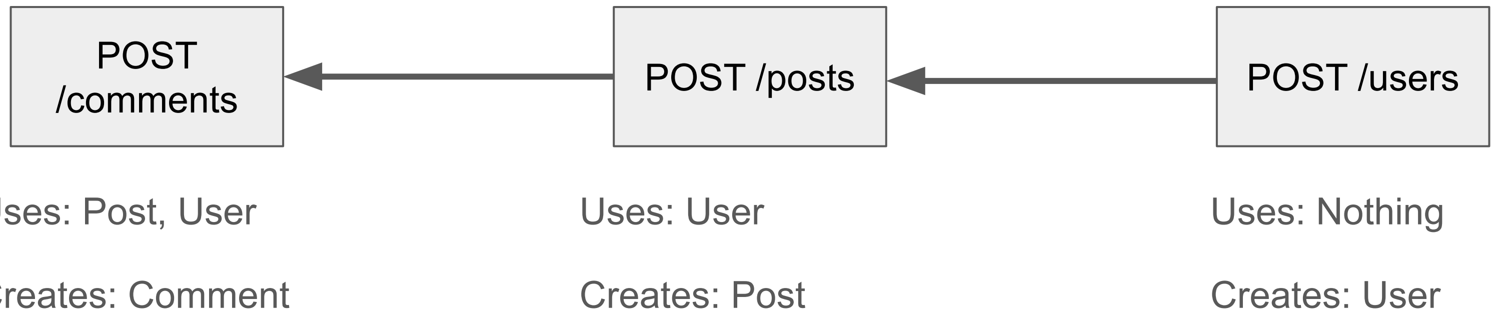
```
class ArticleSerializer(serializers.ModelSerializer):

    class Meta:
        model = Article
        fields = ("id", "headline", "publications")
        extra_kwargs = {"publications": {"required": False}}
```

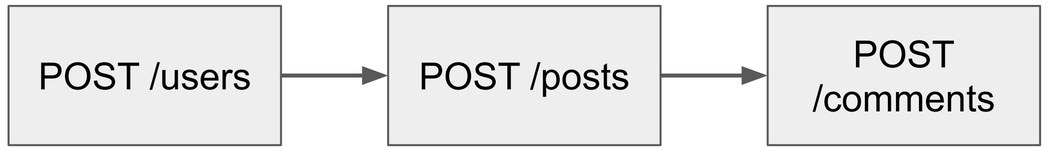
Generating API Sequence

Each API can be called if and only if the models it ***uses*** are all created

Calling each API would make models as it specifies in the ***creates*** list



Valid sequence:

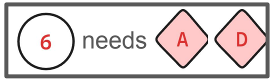


The "Conditional Graph" Algorithm

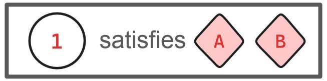
Nodes Vertex vs. Condition

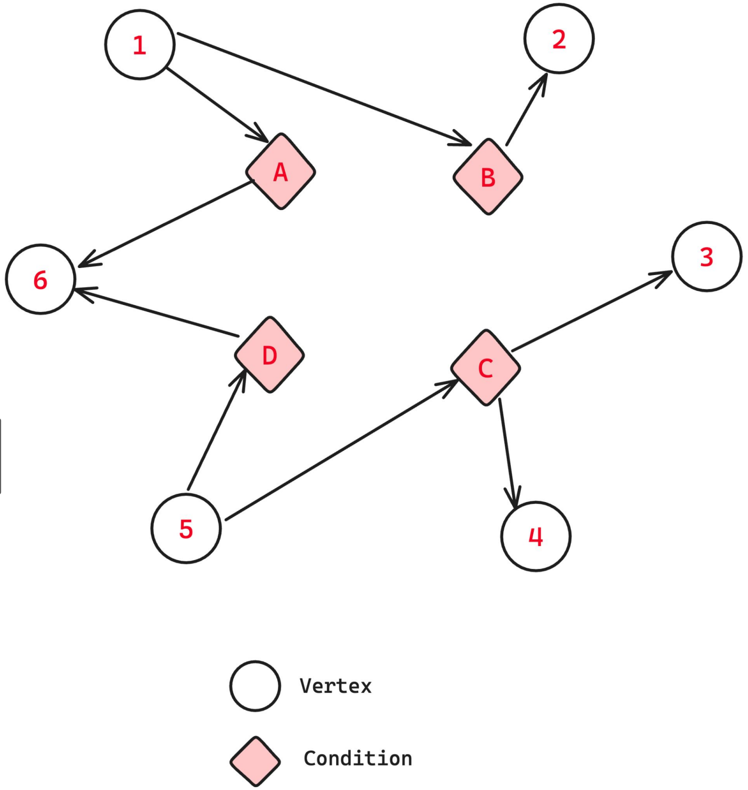
Edges One end is vertex, the other end is condition

Vertex Visiting Rule

- Vertex can be visited only after satisfying all conditions it "*needs*": all the conditions(other ends) of its incoming edges visited 

- Visiting a vertex satisfies all conditions, which are the other ends of its outgoing edges





The "Conditional Graph" Algorithm

Condition == Model

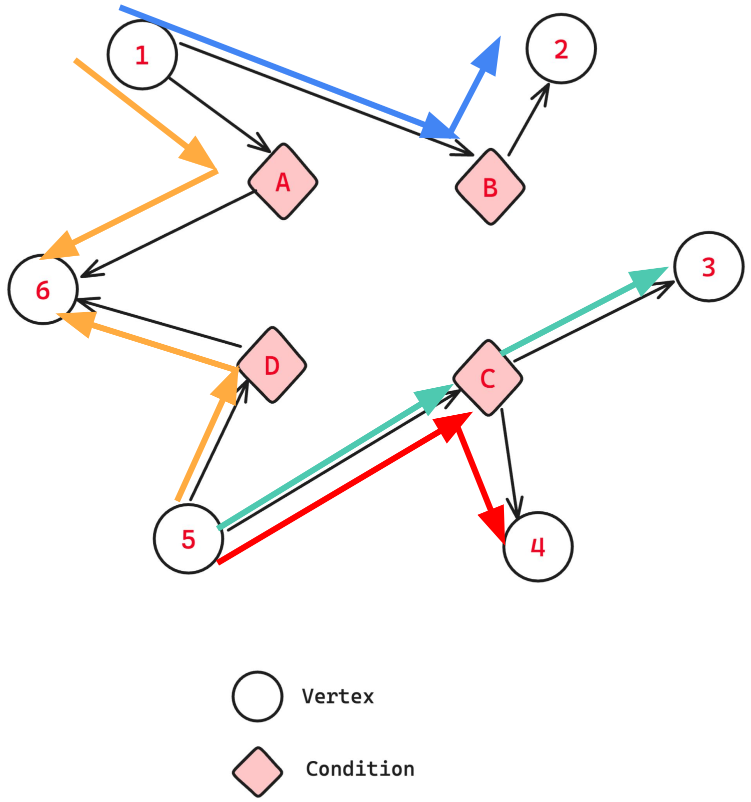
Vertex == API Call

Finding all valid sequences of API calls

==

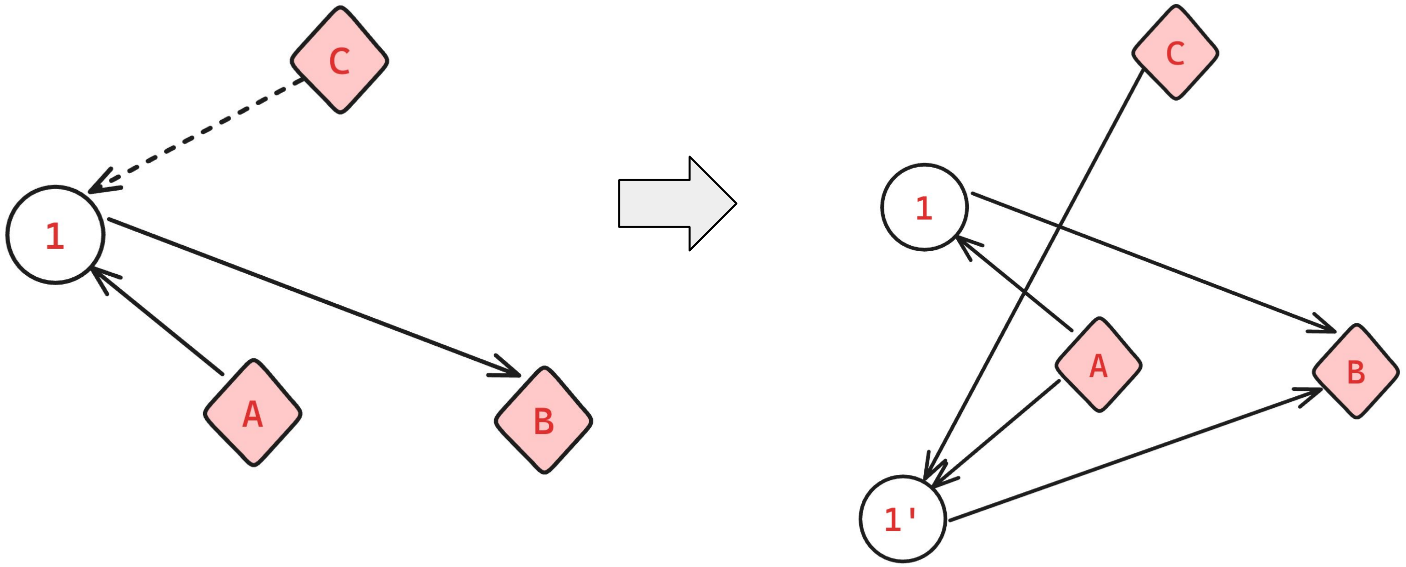
Finding all vertex-visiting-order list

which abide by the vertex visiting rule



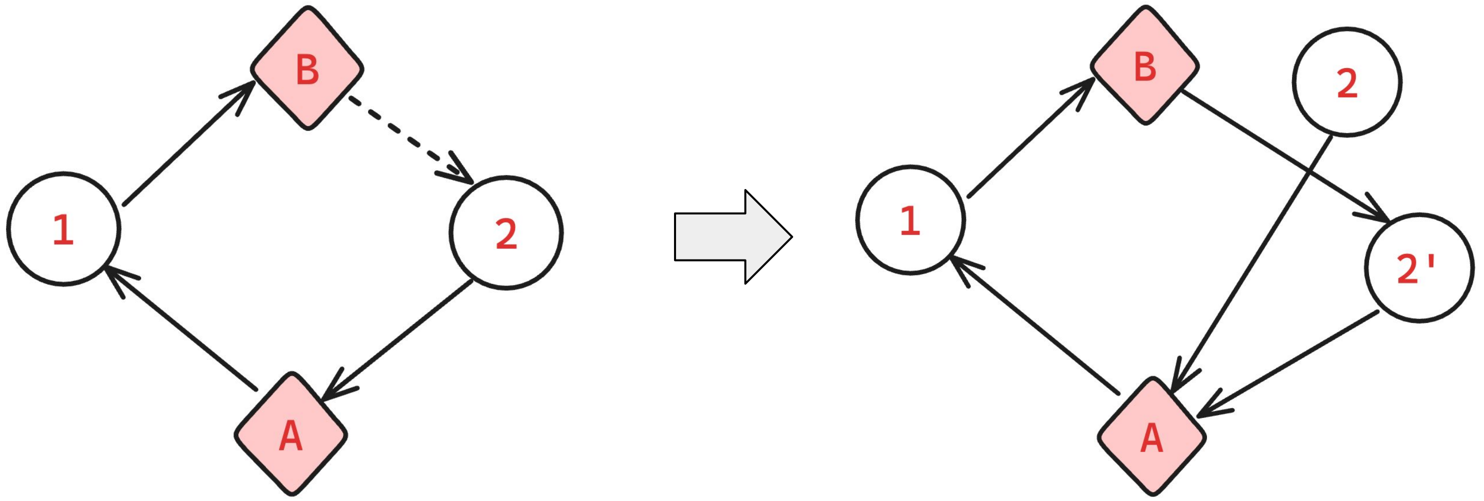
Expanding optional edges

- **Expand** a vertex with N optional edges into 2^N vertices
- Expanded vertex has same edges with original vertex except for the optional edges



Resolving cycles

Expanding & Visiting with the vertex visiting rule ensures correct visiting



Valid sequence:

$$\begin{tikzcd} \boxed{2} \arrow[r] & \boxed{1} \arrow[r] & \boxed{2'} \end{tikzcd}$$

Filling up the request payload

Given a sequence of API calls with payload and *expected* response + the next API call info:

What would be the payload of the next API call?

```
POST /users
payload
{
        "email": "sangachoi@kaist.ac.kr",
        ...
}


response (expected)
{
        "id": 10,
        "email": "sangachoi@kaist.ac.kr",
        ...
}
```

```
POST /posts
payload
{
        "author": 10,
        "title" "my post",
        ...
}

response (expected)
{
        "id": 21,
        "author": 10,
        "title": "my post"
        ...
}
```

```
POST /comments
payload
{
        "author": ?,
        "post": ?,
        "content" ?,
        ...
}
response (expected)
{
        "id": ?,
        "author": ?,
        "post": ?,
        "content": ?,
        ...
}
```

Observation

1. Nature of response: "Response data is for being used"
2. Temporal locality: "Data used recently are more likely to be used"

Heuristics to fill up requests

Given a sequence of API calls with payload and expected response + the next API call info

Fill up the payload **field by field** for the next API call :

1. From the most recent call to the first call, lookup the "expected" response data

    First run: Find exact match of field name and type

    Second run: Find match of type

2. If we cannot find the field from previous calls, randomly generate it based on type

After filling up payload, fill up "expected response data" same way

Referencing the field of previous response dynamically

Cannot hard code "id" field

Save it as "variable" and its value as trace

```
POST /comments
payload
{
        "author": $1,
        "post": $2,
        "content": "random text",
        ...
}
response (expected)
{
        "id": <random number> ,
        "author": $3,
        "post": $4,
        "content": "random text",
        ...
}
```

```
parameter map
{
        "$1": (1, ["author"]),
        "$2": (1, ["id"]),
        "$3": (1, ["author"]),
        "$4": (1, ["id"]
}
```

Trace of reference

```
calls[0]:   POST /users
calls[1]:   POST /posts
```

Index of response referenced

`post` should be come from response of `POST /posts` which is calls[1]

Generated Test

```python
from django.test import TestCase

class MyTestCase(TestCase):
    def test_delete_comments_detail(self):
        res0 = self.client.post('/users/', {'username': 'nfQ2CS9jMf', 'email': 'patrick66@example.net'}, 'application/json')
        assert res0.status_code < 400
        res1 = self.client.post('/posts/', {'title': 'reiciendis dolor illum reprehenderit Hic ipsum', 'content': 'exercitationem amet odit ipsum, adipisicing amet', 'author': res0.data['id']}, 'application/json')
        assert res1.status_code < 400
        res2 = self.client.post('/comments/', {'author': res1.data['author'], 'content': 'exercitationem amet odit ipsum, adipisicing amet', 'post': res1.data['id']}, 'application/json')
        assert res2.status_code < 400
        res3 = self.client.delete(f"/comments/{res2.data['id']}/", {}, 'application/json')
        assert res3.status_code < 400
    ...
```

Evaluation Result

```
----------------------------------------------------------------------
Ran 15 tests in 0.122s

OK
Name            Valid   Failed  Percentage
------------------------------------------
test_app        15      0       100.00%
------------------------------------------
Name                       Stmts   Miss  Cover
------------------------------------------
config/urls.py                 6      0   100%
posts/serializers.py          10      0   100%
posts/views.py                70      8    89%
users/serializers.py           7      0   100%
users/views.py                39      4    90%
------------------------------------------
TOTAL                        132     12    91%
```

Demo

https://github.dev/m-spitfire/cs453-dj-inittest

Limitations

- No authentication
  - Only testing API endpoints with an anonymous user
- Redundant codes
  - Creates a user in every test → Use fixtures that can be used over and over

Remaining Works

- Graph Expansion to handle basic cycle, optional parameters, and the case of multiple APIs creating the same model (WIP)
- Test on open source/complex examples (WIP)

Summary

- Goal
    - Automatically generate boilerplate testing setup codes for Django so developers can focus on core logic.
- What we have done
    - Parse Django APIs.
    - Find out the required sequence of API calls to test each endpoint using *the conditional graph algorithm*.
    - Fill up request payloads using a heuristic.
    - Generate Django tests.
    - Count the number of valid tests and measure the coverage.
- Remaining works
    - Expand the graph to handle cycle and optional dependencies.
    - Evaluate and debug for more complex projects