Automated Django Integration Testing Setup

Murad Bashirov
muradb@kaist.ac.kr
KAIST
Daejeon, South Korea

Sanga Choi
sangachoi@kaist.ac.kr
KAIST
Daejeon, South Korea

Injoon Hwang
yuwol@kaist.ac.kr
KAIST
Daejeon, South Korea

Letian Zhang
hiris@kaist.ac.kr
KAIST
Daejeon, South Korea

ABSTRACT

This paper presents an automated approach to generate boilerplate testing code for Django applications, specifically targeting the integration testing phase. By analyzing the dependency relations between models and endpoints, our tool generates the necessary code to test each endpoint. Evaluation of projects, including a simplified Reddit version and complex projects, demonstrates the effectiveness of our approach in reducing the effort and time required for testing setup and improving test coverage. Our research aims to enhance the developer experience in testing Django-based web applications.

Source code and evaluation projects available at: https://github.com/m-spitfire/cs453-dj-inittest

[This paper is for the CS453 course project]

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

software testing, test setup generation, django

1 INTRODUCTION/MOTIVATION

In the fast-paced world of software engineering, ensuring the quality and reliability of applications is of paramount importance. According to Stack Overflow Annual Developer Survey (2022)[5], it was found that despite the critical role testing plays in software development, many companies still struggle to implement effective testing strategies. This can be attributed to the complexity and time-consuming nature of setting up tests, particularly for applications with numerous dependencies and intricate interactions between components.

This project report focuses on the challenges faced by software engineers, particularly when starting a new job and discovering that the existing code base lacks any form of testing. A common scenario involves the need to test a simple API, such as posting a like on a chat. In order to do this, the engineer must first register, post an article, and post a chat, which can be a cumbersome process when done manually on the development server. This often leads to engineers skipping the test-writing process altogether, resulting in a code base that is more prone to errors and difficult to maintain.

To address this issue, our project aims to automate the generation of boilerplate code for setting up tests, thereby reducing the time and effort required by software engineers. By analyzing the code base and creating a dependency graph, we can determine the relationships between different API calls and their underlying entities. This information can then be used to generate valid sequences of API calls that do not invoke any database errors, streamlining the test setup process.

We will also present real-world examples of test code that spends a significant amount of time on setup, rather than focusing on the test oracle. By automating the generation of boilerplate code for test setup, our project aims to encourage more companies to adopt testing best practices, ultimately leading to higher-quality software and improved user experiences.

It is important to note that our project focuses solely on the setup stage of testing and does not attempt to address the business logic aspect, which should be handled by developers. Our goal is to provide a solid foundation for software engineers to build upon, enabling them to concentrate on crafting effective test oracles and ensuring the overall quality of their applications.

2 BACKGROUND

In this section, we provide an overview of the key technologies and concepts that form the foundation of our automated testing system for Django projects. We will discuss the back end, Django as a high-level Python web framework, and the Django REST framework (DRF) as a toolkit for building web APIs, Django Models, and Django API endpoints.

2.1 Back end

The back end refers to the server side of a web application, responsible for managing data storage, processing, and business logic. It is part of the application that runs on the server and interacts with databases, APIs, and other services to provide the necessary functionality and data to the front end (client-side). Back-end development typically involves working with server-side programming languages, databases, and web frameworks to build the core components of a web application.

2.2 Django: High-Level Python Web Framework

Django is a popular, high-level web framework for building web applications using the Python programming language. It follows the Model-View-Controller (MVC) architectural pattern and provides a comprehensive set of tools and features to streamline the development process. Django promotes the use of reusable code, rapid development, and a clean, pragmatic design, making it an ideal choice for developers looking to build scalable and maintainable web applications.

2.3 Django REST Framework (DRF)

A Toolkit for Building Web APIs The Django REST framework (DRF) is a powerful and flexible toolkit for building web APIs using Django. It provides a set of tools and features that simplify the process of creating, testing, and deploying RESTful APIs, allowing developers to quickly build APIs that adhere to best practices and industry standards. DRF supports various authentication and permission schemes, serialization, query parameters, pagination, and more, making it a versatile choice for building web APIs in Django projects.

2.4 Django Models

In Django, models are the representation of database tables and define the structure of the data that will be stored. Models are Python classes that inherit from Django's Model class and define the fields and their data types, as well as any relationships between tables, such as foreign keys, one-to-one, and many-to-many relationships. Models serve as the foundation for creating, retrieving, updating, and deleting records in the database, and they play a crucial role in enforcing data integrity and consistency.

2.5 Django API Endpoints

API endpoints are the points of interaction between the client-side and server-side components of a web application. In the context of Django, API endpoints are defined using views and URL configurations, which map specific URLs to their corresponding view functions. These view functions handle incoming HTTP requests, process the data, and return an appropriate HTTP response, such as JSON data or an HTML template. By defining and implementing API endpoints, developers can expose the functionality of their Django application to external clients, such as web browsers, mobile apps, or other services.

In summary, the background section provides an overview of the key technologies and concepts that underpin our automated testing system for Django projects. By understanding the role of the backend, Django as a web framework, the Django REST framework, Django Models, and API endpoints, we can better appreciate the challenges and opportunities associated with building and testing web applications in this context.

3 METHOD

3.1 Analyzing

The first step of our implementation is to analyze the existing code base to extract the information needed. We extract the information in the format of Python classes as in the listing 1. Here is each property explained

- `method`: The method of API Endpoint, e.g. `GET`, `POST`
- `path`: The path of API endpoint, e.g. `/comments/`
- `creates`: List of models that API would create
- `uses`: List of models that API would use. Note that a model has an optional field.
- `request_type`: The request type defined in terms of JSON Schema[1]
- `response_type`: The response type is defined in terms of JSON Schema.

```python
@dataclass
class Model:
    name: str
    optional: bool = False

@dataclass
class API:
    method: str
    path: str
    creates: List[Model]  # outgoing
    uses: List[Model]  # incoming (prerequisite)
    request_type: dict
    response_type: dict
```

Listing 1: API Info format

$$\begin{tikzcd}
\texttt{manage.py} \arrow[r] & \texttt{settings.py} \arrow[dl] \arrow[dr] & \\
\texttt{ROOT\_URLCONF: urls.py} \arrow[d] & & \texttt{INSTALLED\_APPS: list} \arrow[d] \\
\texttt{urlpatterns: list} & & \text{model information}
\end{tikzcd}$$

Figure 1: Django Project Hierarchy

To extract this information analyzer gathers information from the code base in 4 steps: extract models, URL patterns, view functions, and serializers. Let's start with models. The model definition in the Django project is located in `{app_dir}/models.py`. To know what apps are in the project, we find the settings file first by analyzing the entry point, `manage.py`, then we extract all the models in the project as shown in figure 1. We extract what is a model's schema, i.e. the column names, and what it depends on by analyzing the foreign key attributes. The next step is to find URL patterns, which are also located in their own file, which is defined in the settings file too. URL patterns are just a list that maps a pattern to a view function, like shown in listing 2. By analyzing the imports on the file, we find the locations of the view functions and go to their definition to actually extract all the information that's left. We currently only support the class-based views that inherit from `APIView`, as shown in listing 3

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

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

```
    path('comments/<int:pk>/', post_views.CommentDetail.
      as_view()),
    path('users/', UserList.as_view()),
    path('users/<int:pk>/', UserDetail.as_view()),
]
```

Listing 2: URL Patterns

```
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)
```

Listing 3: View Class

If we look at individual view functions, we can spot one thing that for each view function the serializer call is different. We have analyzed all the possible view functions that our analyzer will cover and mapped the methods (methods imply the payloads) to serializer calls in the following way

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

GET detail `PostSerializer(post)`
- First argument: DB object
- 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

Moving on to serializers, they are pretty easy to handle, as there's nothing fancy in them. We only support `ModelSerializer`, as in listing 4

```
class ArticleSerializer(serializers.ModelSerializer):
    class Meta:
        model = Article
        fields = ("id", "headline", "publications")
```

Listing 4: Serializer

Here we map the serializer to the model to handle `creates`, and to find the schema of the model. The `fields` has to be a subset of columns defined in the model's schema.

3.2 Generating sequence with conditional graph algorithm

Each API can be called if the models it requires are all created. Calling each API satisfies the models it creates. We used a conditional graph to represent these dependencies between APIs.

3.2.1 Conditional Graph. We propose a unique data structure coined Conditional Graph, as shown in Figure 2, which is a derivative of a dependency graph. Conditional Graph has two types of nodes, one is vertex and the other is condition. Edges are only between different kinds of nodes, i.e. vertex to condition or condition to vertex. A `vertex` requires all conditions such that are the other ends

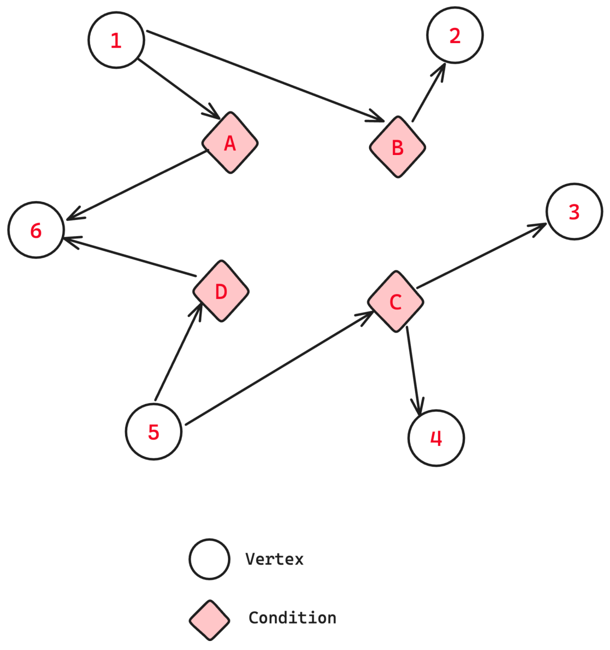


Figure 2: Conditional Graph

of incoming edges. A vertex satisfies all conditions such that the other ends of outgoing edges.

A `path` in a conditional graph is a sequence of vertices, where each vertex is visited only after satisfying all conditions it requires.

From the example graph Figure 2, Vertices are marked with numbers $(1, 2, \ldots, 6)$, while conditions are marked with alphabets (A, B, C, D). $[1, 5, 6]$ is one example of a valid path. Vertex 1 satisfies condition A, vertex 5 satisfies condition D, thus finally vertex 6 is reachable since all its requirements are satisfied.

3.2.2 Converting API list into conditional graph. There might be optional requirements for an API endpoint. Let's donate these optional requirements with dashed lines in the graph, as in Figure 3. Optional requirements and satisfactions between vertex and condition can be expanded into multiple vertices. If a vertex v has a total of N optional edges, $\{e_1, e_2, \ldots, e_N\}$, between various conditions, the vertex is duplicated into 2^N vertices and each duplicated vertex has the same edges with the original vertex except for the optional

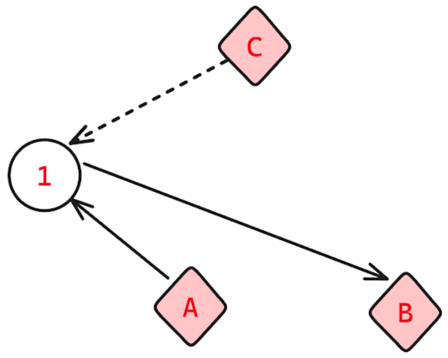


Figure 3: Graph with optional vertex

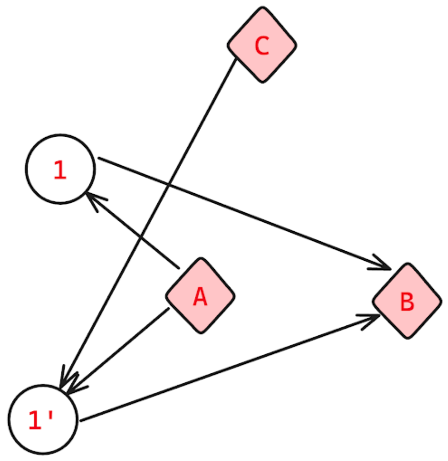


Figure 4: Graph with optional vertexes resolved

edges, and each duplicated vertex will have a different subset of optional edges than any other duplicated vertex as in Figure 4

3.2.3 Traversing conditional graph to generate valid API sequences. If we traverse all valid paths from a conditional graph, the generator will result in many paths visiting vertices that do not require any condition to meet. To generate more realistic sequences which only have APIs that are relevant to the data flow, we modified the conventional depth-first search algorithm to start from a target vertex and find all sequences that satisfy its requirements recursively.

3.3 Generating request payload

Other than a sequence of API calls, the payloads for each API are necessary to generate a valid test setup. If several APIs are called before the final API, then the payload for the final API should be relevant to the response of previous responses. This inference is based on our observations on the nature of the response and temporal locality of data. Each of them is as follows:

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

Therefore, to infer the payload of the next API call, we store payloads and expected responses of all previous API calls. After that, Algorithm 1 explains the procedure that we apply heuristics to fill up the request payload.

Algorithm 1 Fill Up API Payload and Expected Response

```
1:  procedure FillUpAPI(sequence of API calls)
2:      Initialize empty payload for the next API call
3:      Initialize empty expected response for the next API call
4:      for each API call in the sequence from the most recent to the first do
5:          for each field in the next API call do
6:              if field found in the expected response of the current API call then
7:                  Fill the field in the payload with the matching data from the expected response
8:                  Fill the corresponding field in the expected response with the same data
9:              else if field found in the payload of the current API call then
10:                 Fill the field in the payload with the data from the same field in the current API call
11:                 Fill the corresponding field in the expected response with the same data
12:             else
13:                 Randomly generate data based on the field type
14:                 Fill the field in the payload with the generated data
15:                 Fill the corresponding field in the expected response with the same data
16:             end if
17:         end for
18:     end for
19: end procedure
```

After filling up the payload, fill up the *expected response data* in the same way.

However, if the request payload should be decided dynamically, we cannot fill it up in static value. The most representative and critical issue concerns the id field. For example, when posting an article, the id field of an article is filled from DB, therefore posting a comment to the article requires tracking the id of the created article. To resolve the issue, we store the index of the referenced API and the trace to find the dynamic value from the response of the referenced API if the field to infer the value has a type of foreign key.

3.4 Writing Django test automatically

For creating a test suite, we used the `django.test` module. It provides handy testing tools such as `TestCase` and `Client` classes:

- `TestCase`: A Python class consisting of methods that test all HTTP requests.
- `Client`: A dummy web browser for calling HTTP requests and getting responses from them.

API sequence is an order of API calls that must be called before calling the target API. Each call includes an HTTP request method, endpoint URL, request payload, et cetera. Using this information, we generate test cases and check if response status codes are less than 400; no errors.

The actual test file is generated by using the Python standard library `ast`. The resulting test suite looks as below:

```python
class MyTestCase(TestCase):
    def test_post_products(self):
        res0 = self.client.post(
            "/manufacturers/",
            {"name": "elit. esse odit"},
            "application/json",
        )
        assert res0.status_code < 400
        res1 = self.client.post(
            "/categories/",
            {"name": "elit. esse odit"},
            "application/json",
        )
        assert res1.status_code < 400
        res2 = self.client.post(
            "/products/",
            {
                "name": "elit. esse odit",
                "category": res1.data["id"],
                "manufacturer": res0.data["id"],
            },
            "application/json",
        )
        assert res2.status_code < 400
    ...
```

Listing 5: Example Test Output

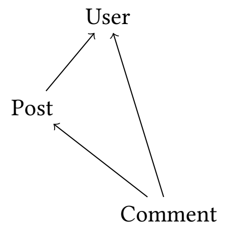
4 EVALUATION/RESULTS

Our evaluation method consists of two parts: ratio (1) of valid sequences to generated sequences and the coverage.

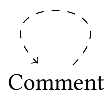
$$R = \frac{n_v}{N} \tag{1}$$

We used 4 projects we wrote to evaluate the project. Note that the following graphs are not the aforementioned Conditional Graphs in 3.2.1, but just an entity relation graph, with dashed line showing optional relation.

(1) `simple_app`: This app has 3 models: Post, Comment, User. The entity relation between them is as follows:



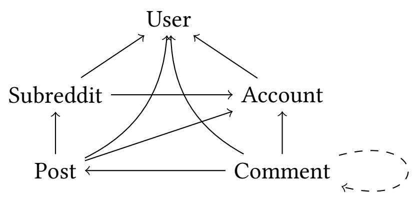
(2) `cycle_simple`: This app only has 1 model, Comment, with cyclic relation



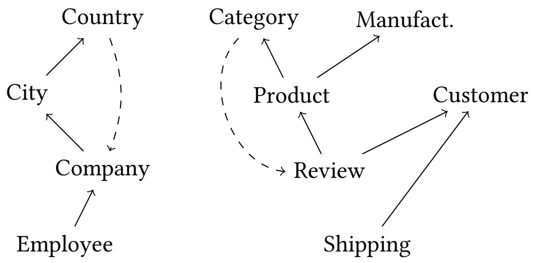
(3) `reddit`: This app has 5 models: User, Account, Subreddit, Post, Comment

Table 1: Evaluation results

Project	Generated	Valid	R	Coverage
`simple_app`	15	15	1	91%
`cycle_simple`	3	3	1	96%
`reddit`	56	56	1	90%
`many_models`	93	93	1	89%



(4) `many_models`: This app has 10 models:



The results are shown in the table 1. The reason why the coverage is getting smaller as the number of sequences increases is that there are more API endpoints and for each endpoint, there is a 404 or 400 handler in the code, as shown in the `post` function in the listing 3 which is not in the scope of our project.

5 RELATED WORK

In the field of automated test case generation for RESTful APIs, several research papers have tried to solve the dependency problem between endpoints in some way. In this section, we discuss relevant works that have addressed similar objectives.

RESTful API Automated Test Case Generation [2] This paper by Arcuri (2017) presents an approach for automated test case generation for RESTful APIs. The author uses an evolutionary algorithm to generate test cases from OpenAPI[2] specification of the API. This algorithm is a full black box, and it tends to generate failing sequences until it finds the MIO[1] algorithm of EvoMaster. It targets RESTFul services running on JVM.

Automatic Generation of Test Cases for REST APIs: A Specification Based Approach [3] In this work, Ed-douibi et al. (2018) propose a specification-based approach for automatically generating test cases for REST APIs. The authors utilize the OpenAPI specification to generate API calls and construct test cases. But they do not use any algorithm, instead, they do random testing, resulting in generating invalid tests most of the time.

Mining Unit Test Cases to Synthesize API Usage Examples[4] Ghafari et al. (2017) present a technique for synthesizing API usage examples by mining existing unit test cases. Our project's

[2]https://www.openapis.org/

target is the code bases that have zero testing; however, this work highlights the importance of leveraging existing test cases to generate meaningful API usage scenarios.

These related works provide insights into automated test case generation for REST APIs. They address various challenges such as generating valid API sequences, leveraging API specifications, and mining existing test cases. The project aims to contribute to this research domain by proposing a novel approach tailored for Django projects and extracting a comprehensive dependency graph to facilitate effective test case generation.

6 FUTURE WORK

As our project continues to evolve, there are several areas of improvement and expansion that we plan to focus on in order to enhance the capabilities of our automated testing system for Django projects. In this section, we outline the key areas of future work that will be undertaken to further refine and extend the system.

Evaluating and Debugging for More Complex Projects: As the system matures, it is crucial to evaluate and debug its performance on increasingly complex Django projects. This will involve employing various testing methodologies, tools, and techniques to assess the reliability, performance, and scalability of the automated testing system.

Fine-graining the output: The current implementation outputs a big test case class that has all the test functions. A better approach would be splitting it into several classes based on the "app"(here "app" refers to a Django app[3]) and generating utility functions that call one sequence to use in more complex tests.

Continuous Integration and Deployment: To further streamline the development process, we plan to explore the integration of our automated testing system with continuous integration and deployment pipelines. This will enable developers to automatically run tests as part of their development workflow, ensuring that any issues are detected and resolved early in the development process. This integration will help to maintain high-quality code and reduce the likelihood of errors reaching the end user.

In conclusion, the future work outlined in this section will help to refine and expand our automated testing system for Django projects, ultimately leading to higher-quality software and improved user experiences. By addressing these areas of improvement, we aim to make a significant impact on the way developers approach testing and contribute to the ongoing advancement of software engineering best practices.

7 CONCLUSION

To summarize, we made an automated boilerplate testing code generator for Django. It parses core files for Django, such as `manage.py` and `settings.py`, to get information on models, serializers, views, and URLs. Then, using a conditional graph algorithm, it generates sequences of API calls for each endpoint. These are used to generate an actual test file that performs integration testing for the given project. We then used the `coverage.py` module to measure the coverage of the generated test file.

As far as we know, this is the first attempt to use white-box testing to make Django boilerplate testing code supporting complex relations such as foreign keys. Our tool reduces the burdensome process of writing repetitive, meaningless codes and helps developers focus on writing tests about the core logic.

[3]https://docs.djangoproject.com/en/4.2/ref/applications/

REFERENCES

[1] Andrea Arcuri. 2017. Many independent objective (MIO) algorithm for test suite generation. In vol. 10452, 3–17. arXiv: 1901.01541[cs]. DOI: 10.1007/978-3-319-66299-2_1.

[2] Andrea Arcuri. 2017. Restful api automated test case generation. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, 9–20. DOI: 10.1109/QRS.2017.11.

[3] Hamza Ed-douibi, Javier Luis Cánovas Izquierdo, and Jordi Cabot. 2018. Automatic generation of test cases for rest apis: a specification-based approach. In *2018 IEEE 22nd International Enterprise Distributed Object Computing Conference (EDOC)*, 181–190. DOI: 10.1109/EDOC.2018.00031.

[4] Mohammad Ghafari, Konstantin Rubinov, and Mohammad Mehdi Pourhashem Kallehbasti. 2017. Mining unit test cases to synthesize api usage examples. *Journal of Software: Evolution and Process*, 29, (Dec. 2017), e1841. DOI: 10.1002/smr.1841.

[5] Stack Overflow. 2022. Stack overflow annual developer survey. https://insights.stackoverflow.com/survey/2022. (2022).