AST Guard Rails

Gobi Dasu

Ma[redacted]s Thesis
Departmen[redacted]omputer Science
Stanford University

[redacted]duction

Code review is a time-consuming activity for software engineering teams, managers, and instructors [2,9]. The practice is however necessary in a modern context not only to prevent code defects, but also to improve learning, team awareness, and codebase organization and style [1]. Otherwise, projects that involve multiple, potentially remote, engineers can easily devolve into spaghetti code. This is especially true of modern web applications [12]. Monitoring codebase organization and style guide compliance is a painstaking task. Hence, most professional software engineering teams use linters in their continuous integration processes [16]. However, linters alone cannot necessarily improve the quality of code style and organization. For instance, code produced by novice engineers working remotely are particularly prone to quality problems. This setting is interesting because of the rise of remote work as the future of work and an increasing number of novice engineers due to the present high demand for programmers [3]. We propose *AST Guard Rails*, an ultra-strict opinionated style guide enforcer that can actually help novice remote engineers produce professional code.

R[redacted]d [redacted]ks

There is a select amount of literature at the intersection of *abstract syntax trees* and *linters*. This literature can be divided into two groups: *pedagogical* and *code review*. The former deals with using abstract syntax trees to provide students code feedback and the latter helps teams perform automated code review.

In the *pedagogical* group, researchers have shown that abstract syntax trees can be used to parse student Intro-CS code submissions and group them, thereby allowing instructors to scale themselves, providing feedback to groups of students who made the same mistakes [7,9,10,13]. Other examples have focused on using AST to isolate specific structures such as variable names and leverage a mixed approach to provide feedback on poor names [6]. One system, *HelpMeOut*, assists a programmer by surfacing similar code predicaments other programmers faced and the corresponding fixes they made. Other methods have used neural networks to learn encodings of submissions and propagate feedback [14]. Overall, AST technology has been used in a variety of clever ways to support education. In this paper, instead of grouping AST parsings for feedback, we recurse over them to enforce rules of[redacted] style and organization.

In the *code review* group, research systems have analyzed millions of code samples to perform *statistical linting*, the practice of warning a programmer of unusual potentially buggy constructs in their code [5]. One system, *Diggit*, was able to use historical data mined from software repositories (which included code review threads) to produce automated code reviews, while another one *Meta* was able to outright replace poorly written functions with more robust equivalents produced by the crowd [4]. One of the most important lessons from the authors of Google static file checking systems has been to make sure that error messages are actionable, an important point that would inspire our system design [16]. While new linters are written frequently for new languages, literature on sta[illegible]y checking coding conventions is more sparse; researchers have produced a tool calle[illegible] *Coco* to specify and check CSS conventions [8,15]. Overall, there have been a number of successful applications of static file analysis using AST to improve the software development experience. That being said, static analysis tools are not popular with developers because of false positives [11]. Our team found this was especially true of open source linters, which we ran along with our AST checker and whose warnings we often had to suppress. This lesson inspired the design of our style guide and code conventions. Rules were only included if they would need to be followed in all foreseeable circumstances.

System

We introduce *AST Guard Rails*, a system that can help novice remote engineers produce profession[illegible] a generic backend REST API. The system comprises of a customizable boilerplate [illegible]I, an opinionated style guide, an AST-based (abstract syntax tree) code checker, and a git pre-receive hook which blocks engineers from pushing a codebase that is not style guide compliant.

The Boilerplate REST API

The boilerplate REST API was built in Django REST Framework (DRF). It consists of a User model with signup and authentication and a Sample Model with CRUD operations (create read update delete) and permissions for CRUD operations. It includes an admin interface, tests (which mainly test CRUD and CRUD permissions), and an API browser which lets API users test the URL endpoints of the API. The boilerplate is provided here: [illegible]ilerplate-draj-exam.

The Opinionated Style Guide

T[illegible]inionated style guide mandates that engineers leave the given boilerplate largely u[illegible]hed, except that they may customize the following files:

1) `models.py` is a file in DRF that lets a programmer define database models. It may be customized but the customization must follow the following format and rules:

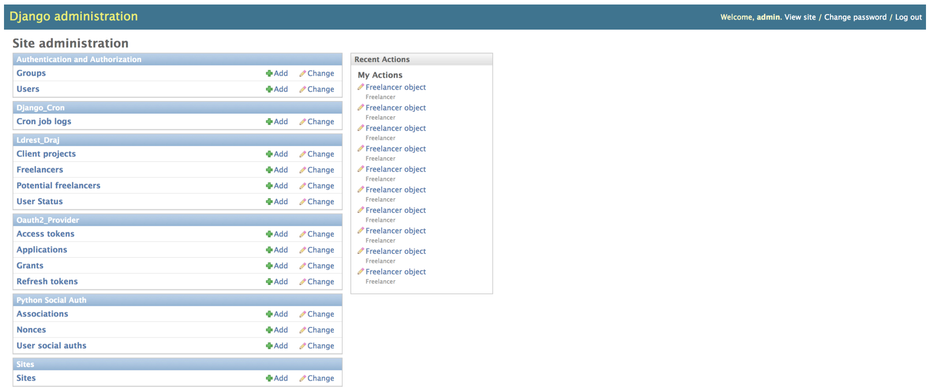
- The first character of the model name must be an uppercase letter, and the rest of the characters must be alphanumeric and lower case.
- Or, the model itself could serve as a many-to-many mapping table. In this case, the model name may combine multiple other existing model names, in which case the model should contain `Foreign Key` fields mapping the current model to each of the individual models within the model name, e.g. A model `SampleUser` should have foreign key fields to the models Sample and User.
- The file should look like the following:

```
Header content (don’t modify from boilerplate)
...
class Sample(models.Model):
        created = [illegible]s.DateTimeField(default=datetime.datetime.utcnow, blank=True, null=True)
        last_modifi[illegible] models.DateTimeField(default=datetime.datetime.utcnow, blank=True,
                nul[illegible]e)
        owner =  models.ForeignKey(User, blank=True, null=True, related_name='owner_sample')
        [add 1 or more fields here]
        # make sure no 2 fields have the same name
        def __unicode__(self):
                return self.[some text field]
        def __str__(self):
                return self.[some text field] # same as unicode
[Add 0 or more models below that respect the same format as Sample model described above, except
with “Sample” or “sample” replaced with the respective model name.]
```

- Finally, there was the restriction that only following fields are acceptable:

```
DateTimeField(default=datetime.datetime.utcnow, blank=True, null=True)

TextField(default=’*’, blank=True, null=True)

BooleanField(default=*)

IntegerField(default=*, blank=True, null=True)

DecimalField(default=*, blank=True, null=True)

ForeignKey(‘Model’, null=True, blank=True, related_name=’fieldname_currentmodel’)
```

2) `admin.py` is a file in DRF that lets a programmer define what data is shown in the admin dashboard of the web application. An example of the Django admin follows:


Django administration
Welcome, admin. View site / Change password / Log out

Site administration

Authentication and Authorization		
Groups	Add	Change
Users	Add	Change
Django_Cron		
Cron job logs	Add	Change
Ldrest_Draj		
Client projects	Add	Change
Freelancers	Add	Change
Potential freelancers	Add	Change
User Status	Add	Change
Oauth2_Provider		
Access tokens	Add	Change
Applications	Add	Change
Grants	Add	Change
Refresh tokens	Add	Change
Python Social Auth		
Associations	Add	Change
Nonces	Add	Change
User social auths	Add	Change
Sites		
Sites	Add	Change

Recent Actions

My Actions
Freelancer object
Freelancer
Freelancer object
Freelancer
Freelancer object
Freelancer
Freelancer object
Freelancer
Freelancer object
Freelancer
Freelancer object
Freelancer
Freelancer object
Freelancer
Freelancer object
Freelancer
Freelancer object
Freelancer
Freelancer object
Freelancer

It may be customized but the customization must follow the following format and rules:

- Each class in `admin.py` corresponds to the admin of a model defined in `models.py`. Each admin class should use `get_all_field_names` (for a horizontal list display of fields) and `search_fields` (for enabling search by fields). Each class should also register each model.

```
Header content (don't modify from boilerplate)
...
class SampleAdmin(admin.ModelAdmin):
    list_display = Sample._meta.get_all_field_names()
    search_fields = ('text', )
    # allow client to search by at least one of the text fields per model

admin.site.register(Sample, SampleAdmin)
[Add 0 or more admin classes below that respect the same format as SampleAdmin class described
above, except with "Sample" or "sample" replaced with the respective model name.]
```

3) `serializers.py` is a file in DRF that lets the programmer specify how database model instances are converted to python objects and vice versa. It may be customized but the customization must follow the following format:

```
Header content (don't modify from boilerplate)
...
class SampleSerializer(serializers.ModelSerializer):
     [foreign key field] = [Foreign Key Field Name]Serializer(read_only=True,
        required=False)
     [foreign key field]_id = serializers.PrimaryKeyRelatedField(queryset=[Foreign Key
        Field Name].objects.all(), source='[foreign key field]', write_only=True)
     # the above is from http://stackoverflow.com/questions/29950956/drf-simple-foreign-key-
     assignment-with-nested-serializers
     class Meta:
        model = Sample
        fields = # list out ALL model fields and ALL related name fields of other
                models where this model is the Foreign Key (can use '__all__' syntax if
                not related name fields)
[Add 0 or more serializer classes below that respect the same format as SampleSerializer class
described above, except with "Sample" or "sample" replaced with the respective model name.]
```

4) `urls.py` is a file in DRF that lets the programmer specify the API endpoints. It may be customized but the customization must follow the following format and rules:

- We leverage the DRF's default URL registration method. Just register the name of the model (make it lowercase) with its model view set as the url. Do it below the Sample model registration line which looks like: `router.register(r'user-status', views.UserStatusViewSet).`

5) `views.py` is a file in DRF that lets the programmer specify the APIs reaction to `GET`, `POST`, `PATCH`, and `DELETE` requests made on the URL endpoints. It may be customized but the customization must follow the following format and rules:

- The file should look like the following:

```
class SampleViewSet(ModelViewSet):
    '''
    * Sample Model Description: This is a description of the sample model.
    * CRUD on Sample model
    * C - CREATE - POST /sample/ : [description of who has access to what for this method]
    * R - READ - GET /sample/ (list) : [description of who has access to what for this method]
    * R - READ - GET /sample/[id]/ (detail): [desc of who has access to what for this method]
    * U - UPDATE - PATCH /sample/[id]/ : [description of who has access to what for this method]
    * D - DELETE - DELETE /sample/[id]/ : [description of who has access to what for this method]
    * Note in the case of a nested model A where a field f points to an instance of another model
       B, you can set f's value to an instance b of B by PATCHing or POSTing with f_id = [the id
       of b].
    '''
    permission_classes = (SampleModelPermissions, )
    queryset = Sample.objects.all()
    filter_backends = (filters.DjangoFilterBackends, filters.SearchFilter,
       filters.OrderingFilter,)
    filter_fields = ('id', 'owner', 'stext', 'created', 'last_modified', )
    # This should list all model fields
    search_fields = ('stext',)
    # allow client to search by at least one of the text fields per model
    serializer_class = SampleSerializer

    def get_queryset(self):
        if self.request.user.is_superuser:
            [can have custom code here]
            return Sample.objects.all()
        [can have custom code here instead of the line below]
        return Sample.objects.filter(owner=self.request.user)
```

- `get_queryset` can have custom code written by the freelancer that returns a result list (from a Django model query)
- `get_queryset` custom code should follow the logging and exceptions rules

Logging will be required and enforced in the following circumstances (Only enforce on `permissions.py` and `views.py` -- the only custom code other than tests.py. `tests.py` doesn't have logging.):

- Every time a function or method call is made -- at the beginning and end of the call (like `has_permission` and `has_object_permission`)
- Every time a database (i.e. Django model) call is made, do a `try/except` around it and log what happens
    - o Valid database calls: `.all`, `.filter`, `.get`, `.create`, `.update`, `.delete`
    - o Disallowed database calls: `.save`
- Every time a third party API call is made (via an http request or via a python driver), then do a `try/except` around it.
- Every time an email is sent (with the Mailchimp API), then do a try/except around it.
- Never use print. Use logger for logging messages instead. Logging messages can be of the following types:
    - o `logger.debug('Use this solely for testing/debugging purpose only.')`
    - o `logger.info('Useful for showing progress of a particular task. Do not log credentials in info logging.')`
    - o `logger.error('Log error messages in case of Internal server error.')`
    - o `logger.exception('Useful for printing stacktrace. Shows code path and helps in debugging.')`

Exceptions `try: ... except: ...` will be required and enforced in the following circumstances (Only enforce on `permissions.py` and `views.py` -- the only custom code other than tests.py. `tests.py` doesn't have logging.):

- Every time a database call is made (`.all`, `.filter`, `.get`, `.create`, `.update`, `.delete`)
- Every time a third party API call is made (via an http request or via a python driver).
- Every time an email is sent.

6) `permissions.py` is a file in DRF that lets the programmer calculate the CRUD permissions the user making the request has. It may be customized but the customization must follow the following format and rules:

```
class SampleModelPermissions(permissions.DjangoObjectPermissions):
    def has_permission(self, request, view):
        if request.method in permissions.SAFE_METHODS:
            [see return rules]
        elif request.method == 'POST':
             [see return rules]
        elif request.method == 'PATCH':
            return True
        elif request.method == 'DELETE':
            return True
        else :
            return False
    def has_object_permission(self, request, view, obj):
        if request.method in permissions.SAFE_METHODS:
            [see return rules]
        elif request.method == 'PATCH':
             [see return rules]
        elif request.method == 'DELETE':
             [see return rules]
        else :
            return False
```

Return Rules:

- The following should be returned in each `if`/`elif` clause body:
    - either:
        - simply `return True` or `return False`
    - or have the following:
        - at least 1 `if` condition
        - the `obj` variable is used in the code for `has_object_permission`
        - `request.user` is used in the code (because `obj` fields and `request.user` should be compared at some point, i.e. for example, `request.user == obj.owner`)

7) `tests.py` is a file in DRF that lets the programmer test their API endpoints. In our case, since most custom code is permissions logic, our tests focus on testing CRUD permissions. It may be customized but the customization must follow the following rules:

- `permissions.py` has custom logic so it needs testing (views.py `get_queryset` ALSO has custom logic BUT the tests on the permissions classes cover the queryset creation logic)

- Should have a test class for each model's permissions class
- Should test all HTTP methods. Should handle at least one case where permission is given and one case where it is not.
- other points: http://stackoverflow.com/questions/3258733/new-to-unit-testing-how-to-write-great-tests
- `tests.py` should contain (refer to boilerplate tests.py):
  - `SampleTests` (should be same as boilerplate)
  - for `[OtherModel]Tests`, the test class should contain
    - super user creation (exact same as boilerplate)
    - app object creation (exact same as boilerplate)
    - site object creation (exact same as boilerplate)
    - client creation (exact same as boilerplate)
    - at least one call to `client.force_authenticate`
    - for every call to `client.force_authenticate`, there should be a call to `client.logout`
    - between two calls to `force_authenticate`, there should be a call to `client.logout`, i.e. if a call to `force_authenticate` is not the first instance of the call, then there should exist a call to `client.logout` before the `force_authenticate` call.
    - there should be at least one call to one of the following calls
    - `client.post` calls
    - `client.get` calls
    - `client.patch` calls
    - `client.delete` calls
    - following calls can exist as needed
    - `activate_user` calls
    - `self.assertEqual` calls
    - `assert` statements
    - creating lists and database call parsing (see `samples_owned_ids` and `samples_listed_ids`)
    - additional testing code is optional

Despite these strict restrictions, any API (that does only model data CRUD) can be built with this boilerplate and opinionated style guide, because it makes no domain or project-specific restrictions. That being said, the API does not support static file upload/download and other common use-cases of web APIs. The current system can be extended to support such use-cases.

The AST-based (abstract syntax tree) code checker

The AST code checker for enforcing the style guide rules leverages the python AST parser library. It parses each file and recursively walks through the AST tree of the file ensuring that each node meets style requirements. Whenever an error is encountered, the rule violated and the line number of the offending code are printed in red. The checker can be found here: pre-receive-draj.py.

The Pre-receive Hook

The hook itself is a pre-receive hook which means that it runs when an engineer tries to push a commit. In our case, the hook returns an error code when code is not style guide compliant, and the push is blocked.

Evaluation

We created an "exam" out of the boilerplate in order to evaluate our *AST Guard Rails*. The following instructions were sent to 22 engineers, out of which 16 actually completed the exam:

"The goal of the 00-boilerplate-draj-exam is to implement simple "to do"-list functionality using the provided boilerplate for backend development.

Your task is to implement a model called "Item" and its associated classes. Users should be able sign up and create a list of "to do" items, which only they can view/edit/delete. The Django superuser owning the database should be able to view/edit/delete any item. Permissions are important!

You need to visualize what the backend of the example project todomvc.com/examples/angularjs/ should look like, and implement it within the provided boilerplate framework.

When you implement this simple functionality, you need to submit the 00-boilerplate-draj-exam, with your additions by committing it to your personal gitlab area."

[The rest of the instructions were technical setup related and are linked here for reference: 00-boilerplate-draj-exam]

The exam would report to the engineer whether they passed or not based on several tests that ran on their code. However, the quality of their code itself was not analyzed.

Then we set up a compliance checker which used an AST parser to enforce style guide rules: 00-boilerplate-draj-exam/wikis/home. We wanted to test whether the rules and checker were effective, resulted in better quality code, and prevented bad practices. In other words, the engineers were guided by the system to write better quality code that respects standards. We sent the following message to 9 engineers we were still in contact with, out of which 6 engineers completed the exam again with AST Guard Rails:

"Hi @channel, in order to assess the effectiveness of the compliancy system we have setup, we would like to request each of you to complete the same 00-boilerplate-draj exam as before, except this time your pushes will only be accepted if you follow the rules laid out in 00-boilerplate-draj/wikis/home. Please proceed."

For the 6 people who completed the exam both with and without AST Guard Rails, we compared pair wise their two submissions. We acknowledge the following biases that such a system evaluation would introduce:

- The fact that we are only focusing on the sample size that stayed with us means that we may have tested on a particularly persistent population or a population that did not receive other job offers during the course of the evaluation.
- We cannot make statistical claims that AST Guard Rails can improve performance in general. We can only explain in what ways AST Guard Rails improved engineer codebase quality in our particular case study.
- It is possible that there was a learning effect over the course of the months between taking the first and second exam, and so we also looked at code performance in categories that were not checked by the AST Guard Rails

Based on the without-AST submissions we designed the following set of error codes:

- [F] File system organization:
  - [f1] Migrations performed
- [A] Appropriate Choice of Constructs:
  - [a1] Appropriate Field Choice (i.e. TextField instead of CharField which is too restricting)
  - [a2] DRF Comments Written so they serve as Self-Documenting APIs, or API docs written separately
  - [a3] No Unnecessary Use of Superclasses
  - [a4] No mixing of constructs -- permissions checking in admin or models or views
- [C] Code Cleanliness:
  - [c1] No Unnecessary or Confusing Comments*
  - [c2] No Unnecessary or Redundant Code Lines*
  - [c3] Clean Code "Punctuation" (i.e. no stray commas)*
  - [c4] No Erroneous items in field lists
  - [c5] No Unnecessary Additional Imports
- [N] Naming:
  - [n1] Appropriate URL Endpoint Names
- [CO] Comprehensiveness:
  - [co1] Filter fields were Sufficiently Comprehensive*
- [T] Testing:
  - [t1] Performs Logging in Permissions
  - [t2] Organizes Tests So It Tests The API Calls
- [TI] Time:
  - [ti1] Inverse of Number of commits needed till success (passing tests).

The items marked with a star (*) are attributes that our AST Guard Rails did not check for, and can therefore be used as controls. All of the variables above were labeled by 2 independent reviewers as 0 (no) or 1 (yes) for both of the exams of each of the 6 subjects. Each of the categories and subcategories are positive (good) code attributes, so a 1 would be "good" and a 0 "bad".

Results

Results of the evaluation follow. The columns represent average ratings for each error code category and rows represent codebases without and with AST guard rails.

Subject	F	A	C	N	CO	T	TI
1-w/o	1	1	**0.6**	1	1	**0.5**	**0.5**
1-w/AST	1	1	**1**	1	1	**1**	**1**
2-w/o	1	**0.5**	**0.4**	0	1	1	0.33
2-w/AST	1	**1**	**0.8**	1	1	1	0.25
3-w/o	1	1	1	1	1	1	**0.33**
3-w/AST	1	1	1	1	1	1	**1**
4-w/o	1	**0.5**	**0.6**	1	1	**0.5**	0.5
4-w/AST	1	**1**	**0.8**	1	1	**1**	0.5
5-w/o	1	1	1	1	1	1	**0.25**
5-w/AST	1	1	1	1	1	1	**0.5**
6-w/o							
6-w/AST							

Figure 1: Two independent reviewers rated both codebases (with and without AST Guard Rails) according to error codes.

As we can see above, AST Guard Rails were able to help engineers produce better quality code. We see examples of improvement in the following categories: appropriate choice of constructs, code cleanliness, testing, and time. There was not much change in file system organization or comprehensiveness. Because several months passed between engineers taking the exam without and with AST Guard Rails, we investigated whether improvements occurred in the starred (*) error codes:

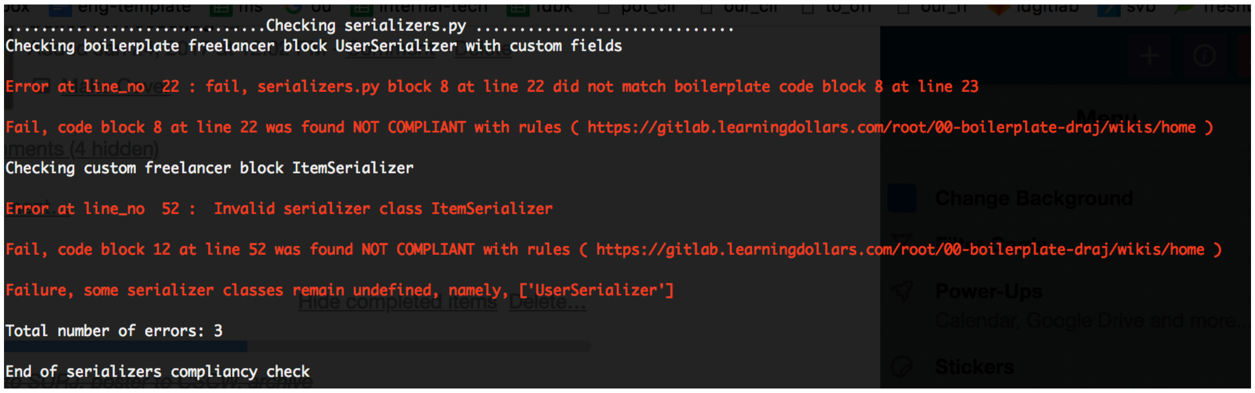
Subject	c1	c2	c3	co1
1-w/o	0	0	1	1
1-w/AST	1	1	1	1
2-w/o	1	1	0	1
2-w/AST	1	1	0	1
3-w/o	1	1	1	1
3-w/AST	1	1	1	1
4-w/o	0	1	1	1
4-w/AST	0	1	1	1
5-w/o	1	1	1	1
5-w/AST	1	1	1	1
6-w/o				
6-w/AST				

Figure 2: Ratings for starred (*) error codes.

We see no obvious trend of improvement due to AST Guard Rails, as expected for these starred (*) subcategories.

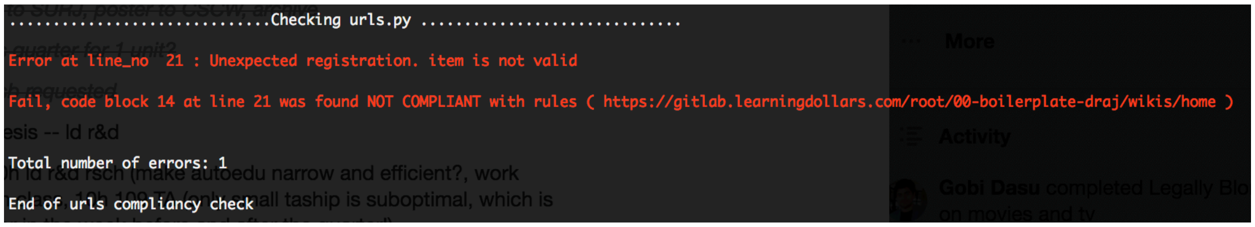
Discussion

AST Guard Rails were able to help engineers produce better quality code. We tried to understand how the system was able to do this without being excessively restrictive. Speaking with the evaluation subjects, we were able to find out that the system provided actionable feedback. It was able to effectively inform engineers of exactly what they had to fix to be able to push their code.



```
.............................Checking serializers.py .............................
Checking boilerplate freelancer block UserSerializer with custom fields

Error at line_no  22 : fail, serializers.py block 8 at line 22 did not match boilerplate code block 8 at line 23

Fail, code block 8 at line 22 was found NOT COMPLIANT with rules ( https://gitlab.learningdollars.com/root/00-boilerplate-draj/wikis/home )

Checking custom freelancer block ItemSerializer

Error at line_no  52 :  Invalid serializer class ItemSerializer

Fail, code block 12 at line 52 was found NOT COMPLIANT with rules ( https://gitlab.learningdollars.com/root/00-boilerplate-draj/wikis/home )

Failure, some serializer classes remain undefined, namely, ['UserSerializer']

Total number of errors: 3

End of serializers compliancy check
```

Figure 3: In this snippet of AST Guard Rails output, we see the system telling the engineer that they have removed essential serializers that came with the boilerplate starter code.



```
.............................Checking urls.py .............................

Error at line_no  21 : Unexpected registration. item is not valid

Fail, code block 14 at line 21 was found NOT COMPLIANT with rules ( https://gitlab.learningdollars.com/root/00-boilerplate-draj/wikis/home )

Total number of errors: 1

End of urls compliancy check
```

Figure 4: In this snippet of AST Guard Rails output, we see the system telling the engineer that they have registered a URL incorrectly. When the engineer checked, they found they did not register the corresponding view.

It turned out that correcting the error messages was easy and fun because a few small tweaks led to the suppression of a number of errors, creating a positive feeling of “progress”. This level of detail was possible because the AST Guard Rails recursively walked through code complaining whenever something was wrong and specifying which rule was violated.

That being said there are a number of limitations of this system. Firstly this system only works on the specific boilerplate which we setup. It is not even generic to *any* application that is built in Django and React. Moreover, in practice different teams have different needs, in which case such *AST Guard Rails* would have to be customized defeating the purpose of an out-of-the-box solution.

Another limitation of the evaluation is that it was done over the course of a maximum of 10 commits per developer. We have not assessed whether engineers would become *frustrated* with such a restrictive system over the course of months or years.

We did not get negative feedback from participating engineers regarding their experience with the system. However we did get feedback from an engineering manager who used the system over the course of 2 months. He decided to deactivate the system because it was a legacy codebase which was being retrofitted into the AST Guard Rails boilerplate and too many errors were being thrown.

Most of the categories that AST Guard Rails didn't check did not witness code improvement. However, there were a few rare improvements in codebases in categories that were not checked by the AST Guard Rails, bringing into question whether the system encouraged engineers to write cleaner code even in dimensions that were not explicitly checked. Such a phenomenon would require further research and more subjects to investigate.

A final limitation could be that such strict restrictions would prevent engineers from leveraging the most appropriate constructs. For example, the current system does not support regular `ViewSet`s, only `ModelViewSet`s. However, a particular application may find it more efficient to have the backend perform a number of complex operations using one `ViewSet` as opposed to having the frontend update models through CRUD operations alone. We did not observe any such worsening of code quality because of AST Guard Rails, but the todo application is not a general and complex enough use case.

Conclusion

In conclusion, AST Guard Rails are capable of helping engineers, particularly novice remote ones, produce professional quality code. We witnessed code quality improvements in a number of categories. That being said there are a number of limitations of the AST Guard Rails system. They include the fact that the system is not easily generalizable to different codebases, was not tested over long periods of time, is not easy for legacy codebases to retrofit into, and does not guarantee the best choice of constructs in all circumstances. Further research would include investigating each of these limitations and designing more robust and general systems.

References

1. Alberto Bacchelli and Christian Bird. 2013. Expectations, outcomes, and challenges of modern code review. In *Proceedings - International Conference on Software Engineering*. https://doi.org/10.1109/ICSE.2013.6606617
2. Robert Chatley and Lawrence Jones. 2018. Diggit: Automated code review via software repository mining. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 567–571. https://doi.org/10.1109/SANER.2018.8330261
3. Alpana Dubey, Kumar Abhinav, Sakshi Taneja, Gurdeep Virdi, Anurag Dwarakanath, Alex Kass, and Mani Suma Kuriakose. 2016. Dynamics of Software Development

Crowdsourcing. In *2016 IEEE 11th International Conference on Global Software Engineering (ICGSE)*, 49–58. https://doi.org/10.1109/ICGSE.2016.13

4. Ethan Fast and Michael S Bernstein. 2016. Meta : Enabling Programming Languages to Learn from the Crowd. *Proc. UIST '16*. https://doi.org/10.1145/2984511.2984532
5. Ethan Fast, Daniel Steffee, Lucy Wang, Joel R Brandt, and Michael S Bernstein. 2014. Emergent, Crowd-scale Programming Practice in the IDE. *Proceedings of the 32Nd Annual ACM Conference on Human Factors in Computing Systems*: 2491–2500. https://doi.org/10.1145/2556288.2556998
6. Elena L Glassman, Lyla Fischer, Jeremy Scott, and Robert C Miller. 2015. Foobaz : Variable Name Feedback for Student Code at Scale. *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology - UIST '15*. https://doi.org/10.1145/2807442.2807495
7. Elena L Glassman, Jeremy Scott, Rishabh Singh, Philip J. Guo, and Robert C. Miller. 2015. OverCode: Visualizing Variation in Student Solutions to Programming Problems at Scale. *ACM Transactions on Computer-Human Interaction*. https://doi.org/10.1145/2699751
8. Boryana Goncharenko and Vadim Zaytsev. 2016. Language design and implementation for the domain of coding conventions. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering - SLE 2016*, 90–104. https://doi.org/10.1145/2997364.2997386
9. Andrew Head, Elena Glassman, Gustavo Soares, Ryo Suzuki, Lucas Figueredo, Loris D'Antoni, and Björn Hartmann. 2017. Writing Reusable Code Feedback at Scale with Mixed-Initiative Program Synthesis. In *Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale - L@S '17*, 89–98. https://doi.org/10.1145/3051457.3051467
10. Jonathan Huang, Chris Piech, Andy Nguyen, and Leonidas Guibas. 2013. Syntactic and functional variability of a million code submissions in a machine learning MOOC. In *CEUR Workshop Proceedings*. https://doi.org/10.1080/09523987.2015.1053287
11. Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs? In *Proceedings - International Conference on Software Engineering*. https://doi.org/10.1109/ICSE.2013.6606613
12. Tommi Mikkonen and Antero Taivalsaari. 2008. Web applications - Spaghetti code for the 21st century. In *Proceedings - 6th ACIS International Conference on Software Engineering Research, Management and Applications, SERA 2008*. https://doi.org/10.1109/SERA.2008.16
13. Andy Nguyen, Christopher Piech, Leonidas Guibas, and Jonathan Huang. 2014. Codewebs : Scalable Homework Search for Massive Open Online Programming Courses. *Proceedings of the 23rd International Conference on World Wide Web*. https://doi.org/10.1145/2566486.2568023
14. Chris Piech, Jonathan Huang, Andy Nguyen, Mike Phulsuksombati, Mehran Sahami, and Leonidas Guibas. 2015. Learning Program Embeddings to Propagate Feedback on Student Code. Retrieved June 8, 2018 from http://arxiv.org/abs/1505.05969
15. Matic Potočnik, Uroš Čibej, and Boštjan Slivnik. 2014. Linter: A Tool for Finding Bugs and Potential Problems in Scala Code. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, 1615–1616. https://doi.org/10.1145/2554850.2555129

16. Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. 2018. Lessons from building static analysis tools at Google. *Communications of the ACM* 61, 4: 58–66. https://doi.org/10.1145/3188720