The Repository Pattern

Author: Charlie Steele

Claps: 376

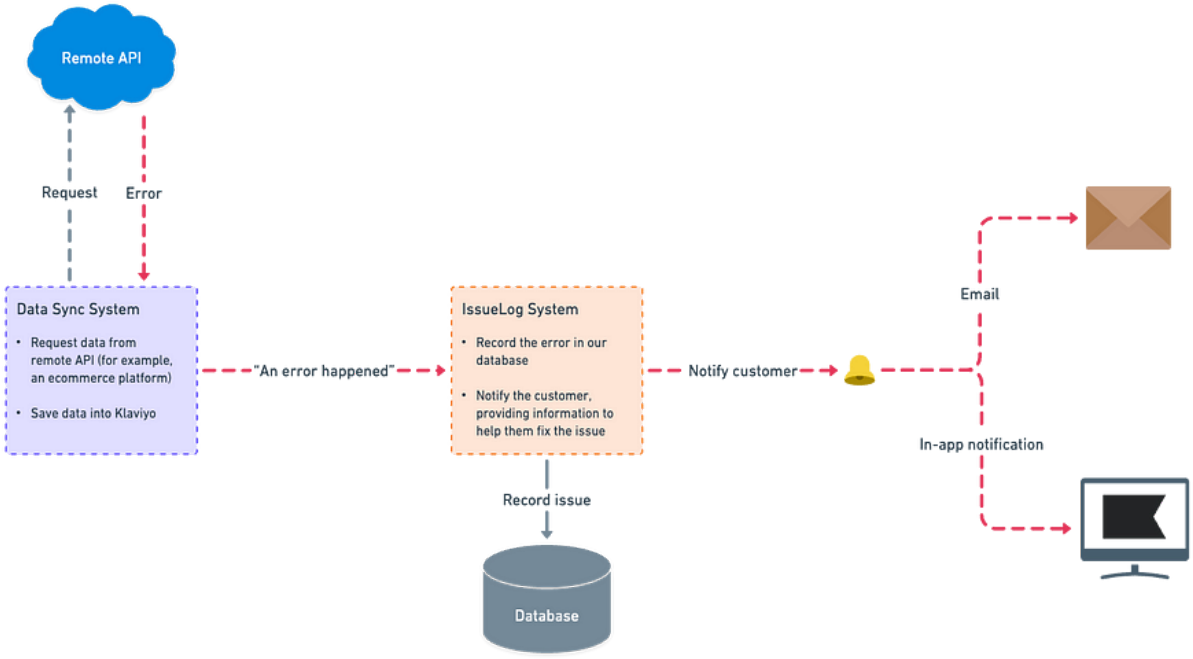
Date: Nov 14

Untangling a web of messy code is a challenge every seasoned engineer has faced. Itâ€™s a narrative of frustration, a story of lost time, and a battle against complexity. Enter the *repository pattern*, a guide for engineering teams grappling with disorganized code, especially code in which business logic and data storage are overly intertwined. In this post, I show how the pattern helped us refactor old code and set us up for scalability and maintainability.

To give some background

It all started back in the spring of 2021. We embarked on a project to deliver more actionable alerts to customers experiencing issues with their inbound data syncs.

The system we use to catch these issues and alert our customers is called `IssueLog`.



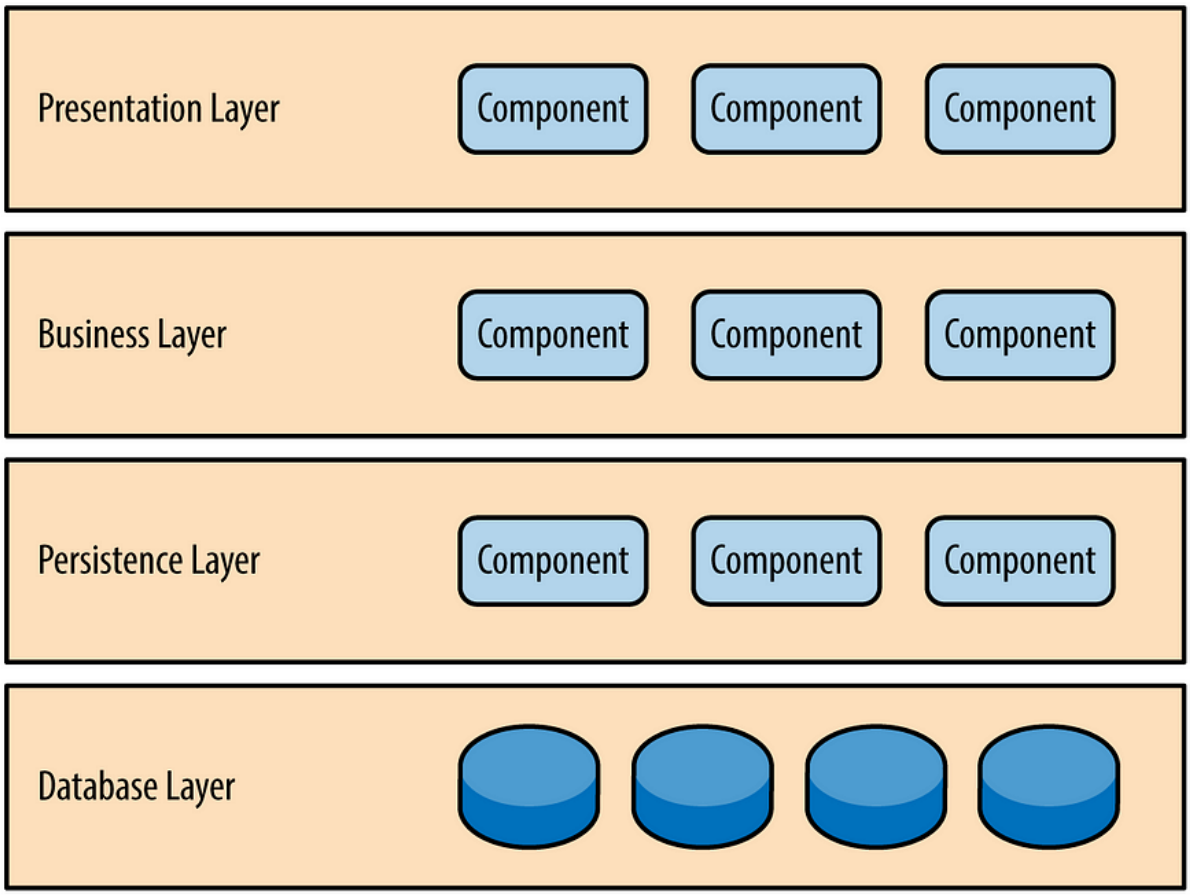
The `IssueLog` system sends users alerts when their data syncs are no longer functioning as expected. However, these alerts contained minimal data to help users get their syncs back up and running. For example, â€œYour periodic sync is no longer functioning as expectedâ€ isnâ€™t helpful and doesnâ€™t tell you anything about what you need to do. Our goal was to make these alerts more actionable. For example, â€œYour periodic sync is failing with an authentication error. Hereâ€™s how you can fix it.â€

We read through the product spec, reviewed the code, and quickly found ourselves in a classic Tidy First? scenario: making small, seemingly trivial changes would cost significant engineering effort unless we first invested time into refactoring the `IssueLog` code. For example, there was

strong coupling between our business logic and data access logic and many of our public service methods contained nearly identical functionality but with subtle differences, making it difficult to figure out which methods to use.

So we set out on a mission to improve the overall organization and structure of the IssueLog code. This would make it easier to build out new features and also write automated tests for those new features.

We knew that we wanted to move towards a layered architecture (similar to many other parts of the product), which organizes code into horizontal layers where each layer has a specific role within the application.



Layered Architecture (Oâ€™Reilly)

The `IssueLog` code had been neglected over the years â€” you know youâ€™re working with old code when you see commits from AB, our CEO. It had become some of the oldest code at Klaviyo, and had turned into a Big Ball of Mud as a result. Providing structure, modularization, and separation of responsibility was our primary goal.

The codebase also predated our use of data access layers, which are a common pattern at Klaviyo. One issue with our widespread usage of DALs was that it naturally led to inconsistencies in how the pattern was implemented across the codebase. We sought consistency.

Enter the Repository Pattern.

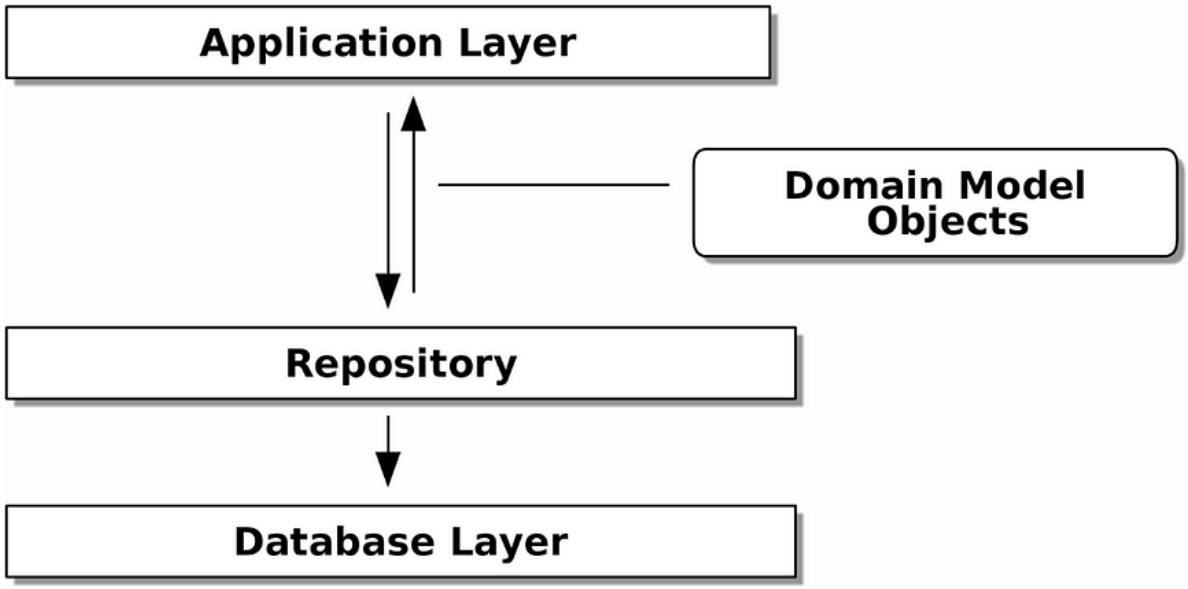
What is the repository pattern?

We first encountered the repository pattern during one of our weekly book club meetings, where we were reading a book called Architecture Patterns with Python. This is an excellent book and Iâ€™d highly recommend it to anyone who wants to level up their Python game. Thereâ€™s a free online version here. To quote the book,

> *â€œThe Repository pattern is an abstraction over persistent storage. It hides the boring details of data access by pretending that all of our data is in memory.â€ (Architecture Patterns with Python, chapter 2)*

The repository pattern is all about the concept of dependency inversion. ORMs, like the Django ORM or SQLAlchemy, add coupling between domain models and their representation in persistent storage (the database schema). The Repository Pattern inverts this dependency and says, *actually, your domain models should have no dependencies and the ORM should depend on your domain models instead of the other way around*.

When your application layer needs to read or write data, it passes domain model objects to the repository layer, and your repository layer handles the DB interactions. As a result, your application layer never handles raw ORM (e.g. Django) models. This has benefits such as separation of concerns and code isolation/encapsulation, both of which lend themselves nicely to service boundaries and domain decomposition.



Architecture Patterns with Python, chapter 2

Letâ€™s walk through an example

We can illustrate the pattern with a contrived example. Suppose that we have a data model called â€œProduct,â€ which has a stock keeping unit (SKU), name, and inventory quantity:

```
class Product(BaseModel):
    sku: models.CharField()
    name: models.CharField()
    inventory_quantity: models.IntegerField()
```

Letâ€™s say we have a service for dealing with our products. The service has a public method to update the product inventory when an order takes place.

```
class ProductService:
    def update_product_inventory(self, sku: str, amount: int):
        product = Product.objects.get(sku=sku)
        # make sure we have enough inventory.
        if product.inventory_quantity - amount >= 0:
            product.inventory_quantity = product.inventory_quantity - amou
            product.save(update_fields=["inventory_quantity"])
        else:
           # raise an exception
           ...
```

The logic here is simple:

1. Retrieve the product with the given SKU.
2. If the difference between the current inventory quantity and the given amount is greater than or equal to zero, update the inventory quantity and save the object back to the database.
3. Otherwise, raise an exception.

Now, letâ€™s convert this to use the repository pattern. The first thing we need is a domain model object:

```
@dataclass
class ProductDomainModel:
    sku: str
    name: str
    inventory_quantity: int
    id: int | None = None
```

We use Python `dataclasses` for our domain model objects, but there are other libraries like `attrs` that work well too. Next, we create the repository:

```
class ProductRepository:
    def get(self, sku: str) -> ProductDomainModel:
        product = Product.objects.get(sku=sku)
        return self._to_domain(product)

def create_or_update(self, product: ProductDomainModel) -> ProductDomainMo
    product, _ = Product.objects.update_or_create(
        id=product.id,
        defaults={
            "sku": product.sku,
            "name": product.name,
            "inventory_quantity": product.inventory_quantity,
        }
    )
    return self._to_domain(product)

def _to_domain(self, django_model: Product) -> ProductDomainModel:
    return ProductDomainModel(
        id=django_model.id,
```

```
            sku=django_model.sku,
            name=django_model.name,
            inventory_quantity=django_model.inventory_quantity
        )
```

Note that each method returns a ProductDomainModel object, which means that this repository class is the only place where we actually interface with the ORM.

One last thing worth mentioning about this class is the `_to_domain` method. With the repository pattern, itâ€™s necessary to map your domain model objects to your ORM model objects. Many ORMs, such as SQLAlchemy, have built-in mapping utilities. The Django ORM, however, does not.

Now, upstream service classes no longer have to deal directly with the ORM:

```
class ProductService:
    def __init__(self, repository: AbstractProductRepository):
        self._repository = repository
    def update_product_inventory(self, sku: str, amount: int) -> None:
        product = self._repository.get(sku=sku)
        # make sure we have enough inventory.
        if product.inventory_quantity - amount >= 0:
            product.inventory_quantity = product.inventory_quantity - amou
            self._repository.create_or_update(product)
        else:
            # raise an exception
            ...
```

Itâ€™s also lean and only contains business logic. The service class doesnâ€™t know any details about how products are stored; that responsibility has been delegated and hidden behind the repository abstraction.

Benefits of the Repository Pattern

Enables domain separation

The first and most obvious reason to use the repository pattern is for domain separation purposes. Not giving application code access to the underlying ORM (Django in our case) models helps to establish service boundaries. Your higher level services, where business logic lives, can only perform the storage operations specified by your repository, and have no knowledge of how the data is being stored under the hood. The application layer simply knows about your domain model and the business logic that needs to be performed on that domain model, while the repository layer encapsulates everything related to storage.

Improves code testability

The repository pattern makes testing the application layer easier because it allows leveraging dependency injection to create fake database interactions within tests. Letâ€™s say you have an application-level service that reads from the database and performs some business logic. Rather than figuring out which database calls are being made and mocking those calls out to return the correct data for your test case, you can implement a fake repository that returns the data you want, and pass that repository to your service class.

As an example, letâ€™s say we want to write a unit test for our `ProductService.update_product_inventory` method. First, weâ€™d create a fake repository that implements the same protocol as our `ProductRepository`:

```
class FakeProductRepository:
    def get(self, sku: str) -> ProductDomainModel:
        return ProductDomainModel(
            sku=sku,
            name=â€œFake Productâ€,
            inventory_quantity=1,
        )
    def create_or_update(self, product: ProductDomainModel) -> ProductDoma
        return product
```

Notice that our fake repository doesnâ€™t actually talk to the database, it just returns fake data.

Our test would look something like this:

```
def test_update_product_inventory():
    product_service = ProductService(repository=FakeProductRepository())
    # test to make sure we correctly update product inventory according to
    ...
```

For a more in-depth, end-to-end example of how the repository pattern facilitates testing, check out Architecture Patterns with Python.

Simplifies swapping data storage technologies

The repository layer provides a contract for reading and writing data, and the application layer should only interact with the repository layer. If you decide to switch the persistence layer backing your domain model from MySQL to Postgres, for example, all you need to do is create a Postgres repository that implements the same interface as your MySQL repository. No changes are needed to application code besides swapping out which repository is being used.

The IssueLog Refactor

So far weâ€™ve covered the theory behind the repository pattern, and given a contrived example. Now letâ€™s take a look at how we applied this pattern to our codebase to revamp the `IssueLog` system, letting us then easily build the feature our customers see: alerts that tell them how to fix problems with inbound data syncs.

Remember, the goal of this refactor was to improve code organization and make it easier for us to implement new features. There are a million different design patterns that can be leveraged to refactor a messy codebase. When we first read about the repository pattern we thought, *oh this just sounds like a fancier and stricter version of the data access layer pattern that we follow elsewhere in the codebase.* We were already familiar with DALs, wanted to move towards a layered architecture, and wanted to see if the rules laid out by the repository pattern would help us achieve better consistency in how DALs are implemented across the codebase.

Data model

The `IssueLog` system has a simple data model: an `Issue` table to represent an error that is occurring within a customerâ€™s data sync, and a `Log` table to represent specific instances of that issue. An `Issue` can have many logs, and each `Log` belongs to a specific issue. Hereâ€™s a simplified version of how these models are defined via the Django ORM:

```
class Issue(BaseModel):
    code = models.IntegerField()
    service_id = models.IntegerField()
    latest_log_id = models.IntegerField()
    status = models.IntegerField(default=STATUS_OPEN)
    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(db_index=True, auto_now=True)

class Log(BaseModel):
    service_id = models.IntegerField()
    origin = models.CharField(max_length=255, null=True)
    issue_id = models.IntegerField(db_index=True, null=True)
    data = JSONField()
    created = models.DateTimeField(auto_now_add=True)
```

Refactoring the code using the repository pattern

Prior to the refactor we had a handful of different service classes, each of which called directly into the Django ORM to read and write issues and logs. As I mentioned above, this area of the code hadnâ€™t received any TLC in a long time, so naturally we had accumulated some tech debt. Overall, there were 31 public methods spread across several classes, all making similar queries:

```
1. IssueLogService.log_custom_integration_error
2. IssueLogService.generate_custom_issue_log_payload
3. IssueLogService.save_custom_integration_error_and_maybe_alert
4. IssueLogService.should_log
5. IssueLogService.log_integration_error
6. IssueLogService.redact_sensitive_keys
7. IssueLogService.create_issues_for_statistic_gaps
8. LogService.get
9. LogService.logs
10. LogService.logs_for_issue
11. LogService.logs_since
12. LogService.logs_latest
13. LogService.count
14. LogService.all_by_id_list
15. LogService.create
16. LogService.update_log
17. IssueService.issue_or_none
18. IssueService.issues
19. IssueService.most_recent_issue
20. IssueService.unresolved
21. IssueService.create_error
22. IssueService.create_warning
```

```
23. IssueService.create
24. IssueService.update
25. IssueService.has_logged_sampling_issue_recently
26. IssueService.flag_logged_sampling_issue
27. IssueService.trim_issue_logs
28. IssueService.resolve
29. IssueService.resolve_for_integration
30. IssueService.has_auto_resolved_recently
31. IssueService.company_integrations_with_issues
```

The challenge was to take this code and get it to a place where making changes and building out new features was safe and easy. However, it was difficult and overwhelming to read through all 31 public methods and try to suss out what each method was doing, where code could be consolidated, etc. Instead, we took a bottom-up approach:

1. Find all places where we were interacting with our database (via the ORM).
2. Create repository methods for each of these interactions, making sure to consolidate logic wherever possible so that we ended up with a relatively small number of methods that generally mapped to CRUD operations.
3. Replace all direct ORM calls with calls to the new repository methods.

Letâ€™s go through a couple examples: `record_new_issue` (previously `log_integration_error`) and `most_recent_issue`.

Weâ€™ll start with `record_new_issue`. This method is responsible for creating a new `Issue` record, and also creating a new `Log` record to associate with the issue.

Prior to the refactor, it looked something like this:

```
def record_new_issue(
 self, service_id, ex,
):
    data = ex.data
    code = ex.code

    # create a log entry
    log_entry = Log(
        service_id=service_id,
        data=data,
    )
    log_entry.save()
    # see if the issue exists
    issue = Issue.objects.filter(service_id=service_id, code=code, status=
    # create an issue if one doesn't exist. otherwise update the existing
    if not issue:
        issue = Issue(
            code=code, service_id=service_id, latest_log_id=log_entry.id,
        )
        issue.save()
    else:
        issue.latest_log_id = log_entry.id
        issue.save()
    # associate the new log entry with the issue
    log_entry.issue_id = issue.id
```

```
    log_entry.save()
    return issue
```

After creating domain models and moving all database operations to `Repository` classes:

```
def record_new_issue(self, service_id, ex):
    # create a log
    log = LogDomainModel(
        service_id=service_id,
        data=ex.data,
    )
    LogRepository().create_or_update(log)

    # get or create an issue record
    issue = IssueRepository().get_or_create(service_id, ex, ex.code)

    # set the latest_log_id
    issue.latest_log_id = log.id
    IssueRepository().create_or_update(issue)

    # associate the new log entry with the issue
    log.issue_id = issue.id
    LogRepository().create_or_update(log)
```

Next, letâ€™s refactor `most_recent_issue`. This method is responsible for retrieving the most recent issue for a given `service_id`. Originally, our code looked like this:

```
def most_recent_issue(
    self, service_id, historical=False,
):
    issue_query = Issue.objects.filter(
        service_id=service_id,
        level=IntegrationIssue.LEVEL_ERROR,
        status=IntegrationIssue.STATUS_OPEN,
    ).order_by("-updated")
    # Limit to 10 issues so we don't run tons and tons of SQL queries.
    recent_non_periodic_sync_issues = issue_query.filter(
        monitor_item_id__isnull=True
    )[:10]

    for record in recent_non_periodic_sync_issues:
        log_query = Log.objects.filter(
            service_id=service_id,
            issue_id=issue_id,
        )
        is_origin_historical_sync = log_query[0].is_origin_historical_sync
        # If it's historical, the origin needs to be likewise. If it's not
        if (
            historical and is_origin_historical_sync
        ) or not is_origin_historical_sync:
            return record
    return issue_query.first()
```

After the refactor:

```
def get_most_recent_issue(
    self, service_id: int, is_historical: bool
) -> IntegrationIssue:
    issues = IssueRepository().get_many_with_order_and_limit(
        service_id=service_id,
        order_by=IssueRepository.ORDER_BY_UPDATED,
        status=IssueStatus.OPEN,
        limit=20,
    )
   for issue in issues:
       logs = LogRepository().get_many_with_order_and_limit(
           issue_id=issue.id, limit=1
       )
       log = next(logs, None)

       if historical == log.is_origin_historical_sync:
           return issue
```

We continued on like this until all of our service methods were calling into our Repository classes. Once we were finally done, we had a clear separation of business logic and data storage logic.

So, you moved a little bit of code around. Whatâ€™s the big deal?

Well, after we finished converting all ORM calls to use our new repository methods, **things really started falling into place**. Now that all data access logic lived in our `Repository` classes, and business logic was being encoded via parameters to public methods defined on those `Repository` classes, we started to notice that the only difference between many of our public service methods were the parameters we were passing down to the data access methods.

For example, in our `IssueService` we had three different ways to log a new issue:

1. `log_custom_integration_issue`
2. `save_custom_integration_error_and_maybe_alert`
3. `log_integration_error`

Once we had moved the data storage logic out of these methods, and we could more easily identify the subtle differences in business logic between them, we were able to consolidate them into a single method called `record_new_issue`.

Apply this to all 31 methods, and it scales very well. At the end of the refactor we had reduced the number of public service methods from 31 to 10. This reduction in complexity allowed us to consolidate all our service methods under a single class, `IssueLogService`, which became the new public interface for the `IssueLog` system.

Hereâ€™s what our public interfaces looked like after the refactor:

1. `IssueLogService.get_most_recent_logs_for_issue_id`
2. `IssueLogService.get_issue_by_id`
3. `IssueLogService.get_most_recent_periodic_sync_issue`
4. `IssueLogService.get_most_recent_issue`
5. `IssueLogService.get_most_recent_issues_and_logs`
6. `IssueLogService.resolve_issues`
7. `IssueLogService.record_new_issue`

```
8. IssueLogService.delete_issue_logs_before_date
9. IssueLogService.has_auto_resolved_recently
10. IssueLogService.get_data_from_exception
```

If you ask me, this interface is much cleaner and more user-friendly than what we had before.

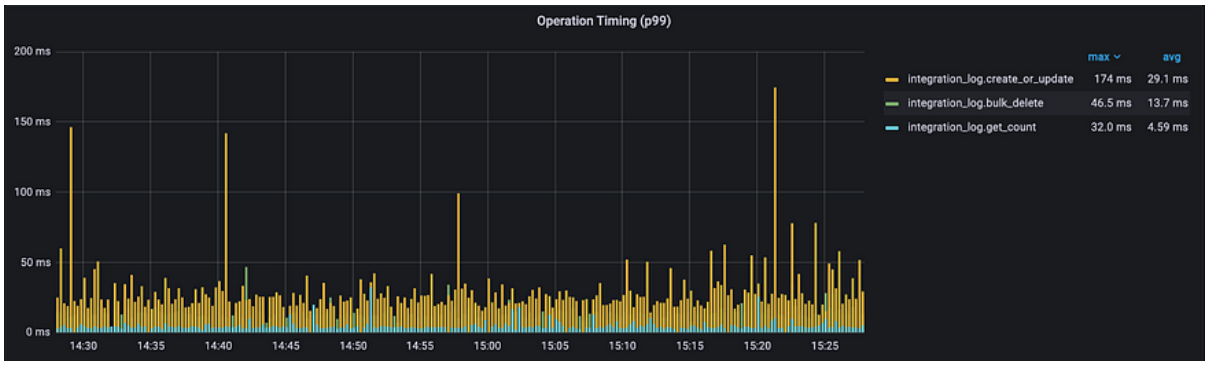
Additional Benefits of the Repository Pattern

Facilitates optimizations to data access

Consolidating all data access logic to a `Repository` class makes it easier to make performance optimizations such as adding caching or making tweaks to how you interact with your data stores.

Prior to the refactor, making an optimization to how we were querying our MySQL database would have required us to make changes in several different locations. It was also more difficult to identify and track down non-performant queries due to the lack of centralization. Now that all data access lives in one place, we have a much better grasp on our query patterns and itâ€™s easier to do things like optimize SQL queries and add indexes. We also were able to easily add instrumentation to all data access operations, a task that wouldâ€™ve been more difficult with data access methods scattered across many different methods.

Check out this Grafana dashboard, for example:



Simplifies error handling

Another side effect of moving all data storage operations to a repository layer is that it becomes easier to add error handling. Transient errors are a common occurrence when interacting with data stores over the network. One common approach to dealing with these errors is to catch and retry them, which requires some additional code.

Prior to the refactor, adding error handling would have required us to make changes across all locations where we access our data storage. Now, we can isolate this logic to a single location in the `Repository` classes.

Learnings

We have embraced the repository pattern, and have since introduced several new uses of it in our codebase. However, every design pattern has tradeoffs so Iâ€™d be remiss if I didnâ€™t talk about some of the issues weâ€™ve encountered using this pattern in production.

Model object mappings

As shown in the example code above, our implementation of the repository pattern requires us to maintain an explicit mapping between Django model fields and domain model fields. This means that anytime a field is added, there are three different places that need to be updated. This is error-prone, and has been the cause of some head scratching for engineers just getting familiar with the pattern.

Other ORMs, such as SQLAlchemy, provide utilities for automating this mapping. In the future, we may decide to build our own mapping utilities for the Django ORM, which would remove the need to maintain them explicitly.

Complexity

The pattern introduces an additional level of indirection. Indirection can make code more difficult to follow, especially for engineers ramping up. The added complexity might not make sense for a simple service, but pays off in terms of maintenance costs as a service grows.

Conclusion

Implementing the repository pattern to refactor messy code is a strategic move that can transform a convoluted and error-prone codebase into one thatâ€™s clean, maintainable, and adaptable. By decoupling the data access layer from the applicationâ€™s business logic, the repository pattern not only promotes separation of concerns but also enhances testability, code reusability, and flexibility with data sources. This approach offers a well-structured solution for tackling the complexity of data management and query operations, resulting in a more efficient and organized codebase that is easier to work with and maintain in the long run. Embracing the repository pattern is not just about making your code cleaner; itâ€™s a commitment to a more scalable and sustainable software architecture

My teamâ€™s commitment to this pattern is illustrated by the fact that we use it for all new projects, and have gone back and refactored other systems (in addition to `IssueLog`) to use it.

PS: We discovered this pattern while reading the book Architecture Patterns with Python. *This book is mentioned in another blog post, Top 10 Books for Mastering the Art of Building Software at Scale, which also lists other books that have been helpful to us in designing and building scalable software at Klaviyo. Feel free to check it out!*