Interactors!

By Rusty Klophaus
Last Updated: March 9, 2020

TL;DR

The Interactor pattern
makes it easier
to reason about apps
by providing a clean place
to put business logic.

What is the problem?

Fictional Example

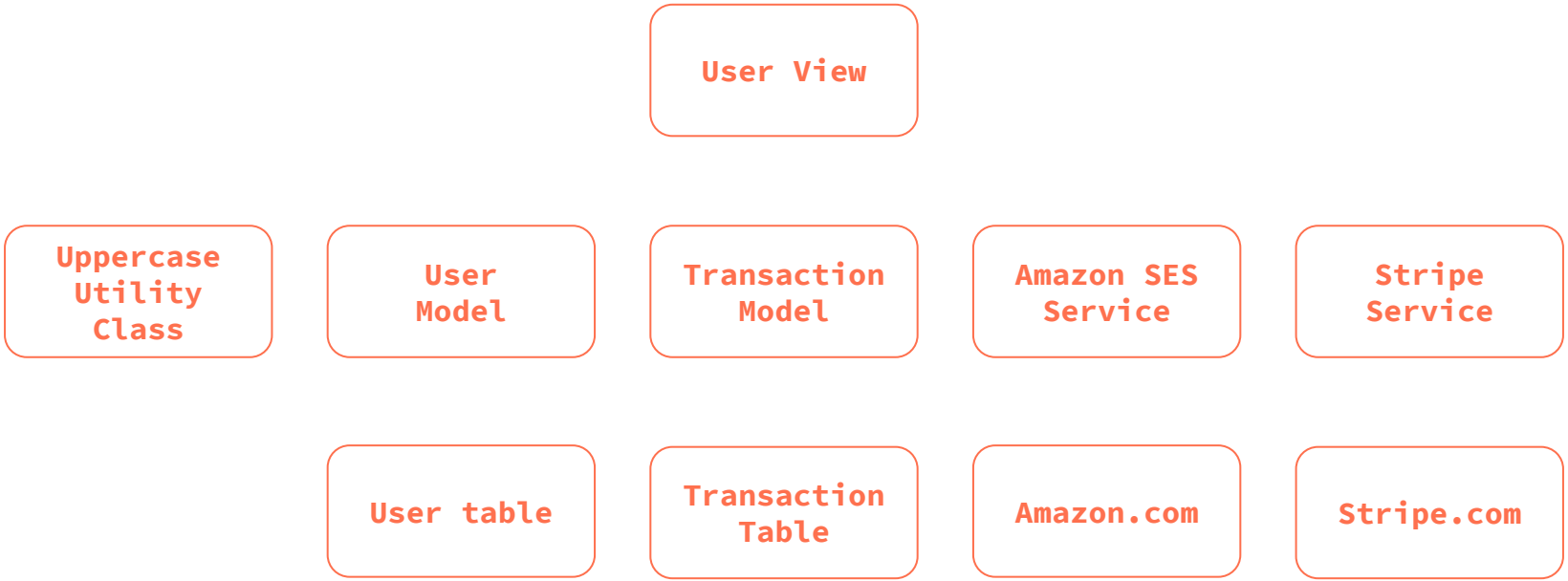
PBS Passport Gold - Let a viewer UPPERCASE their last name.

```
# Update the User record to make last name uppercase.
# Send confirmation email to user via Amazon SES
# Generate a new one-time donation Transaction record
# Charge the user via Stripe API
```

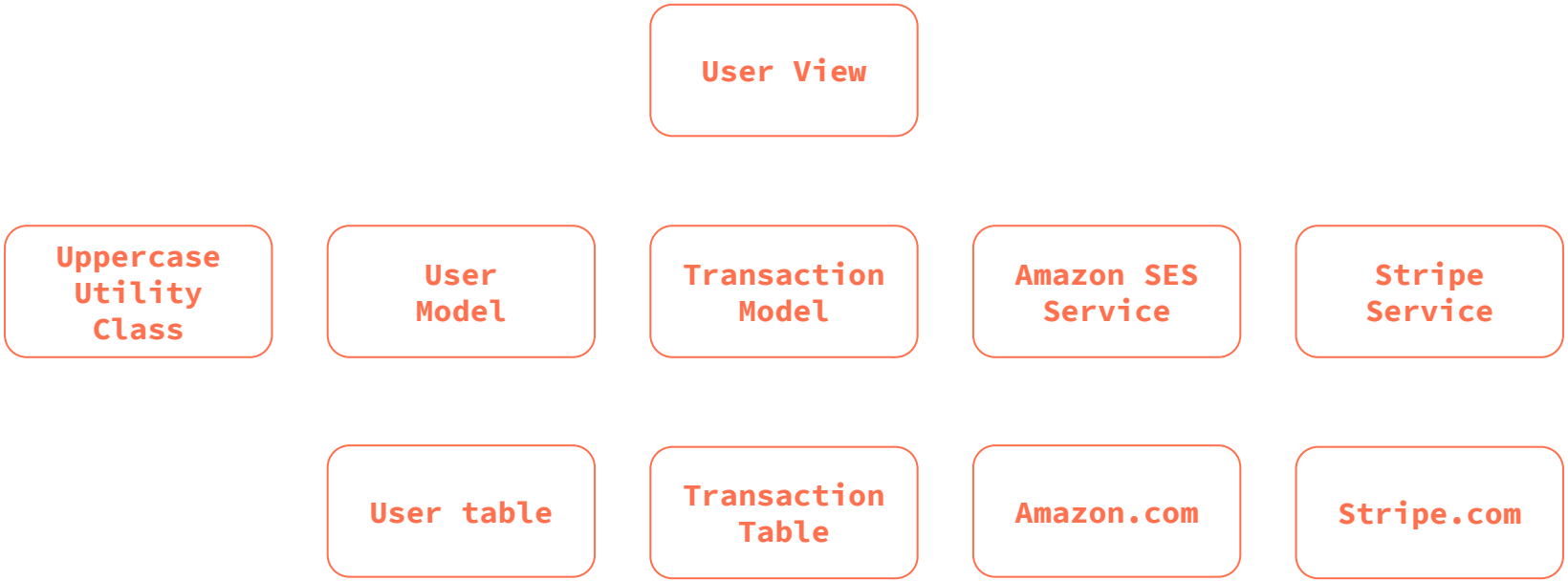
Touches 2 database tables, 2 third party services.

Where should this code live?

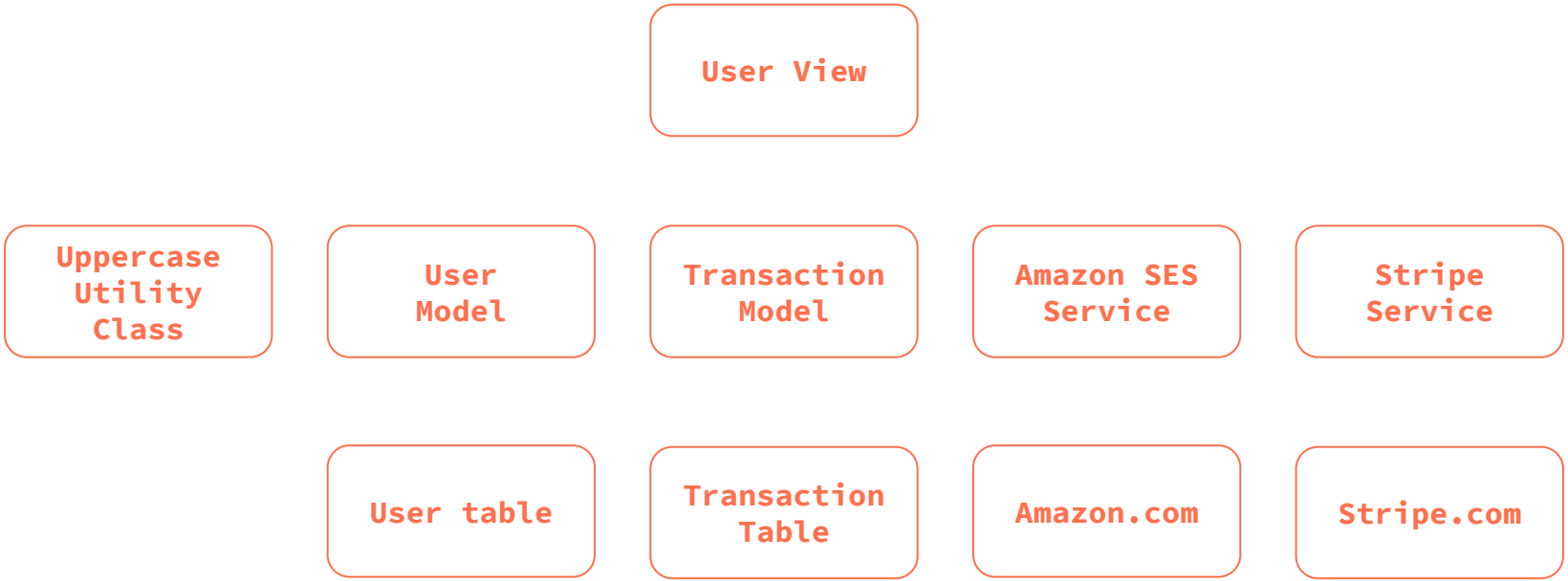
Where should this code live?



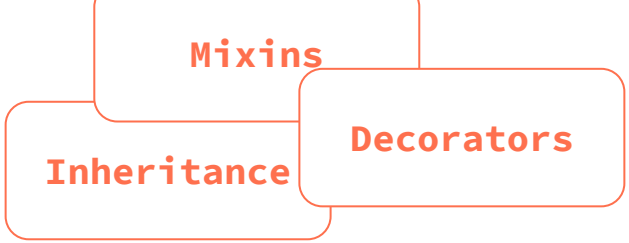
Who should know about who?

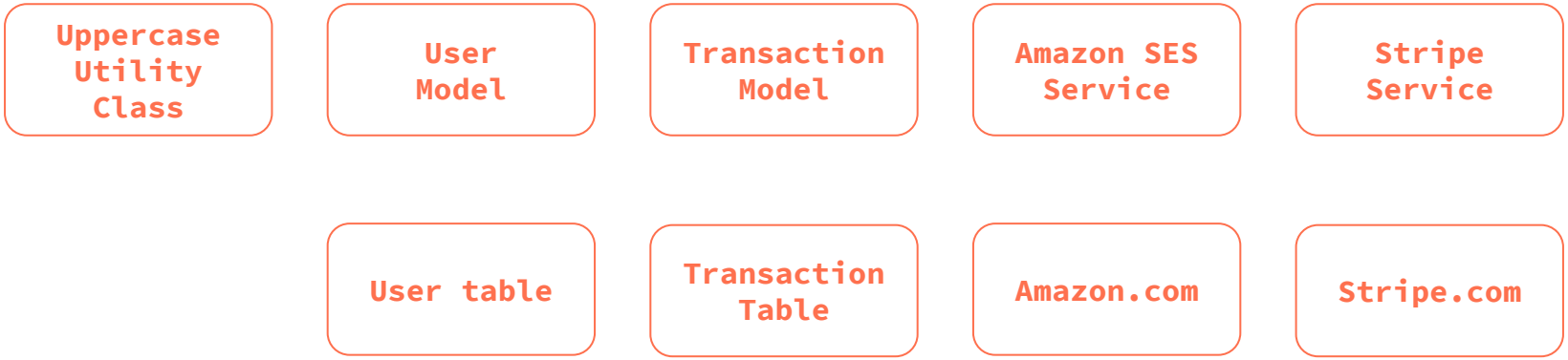


Put code in the view?

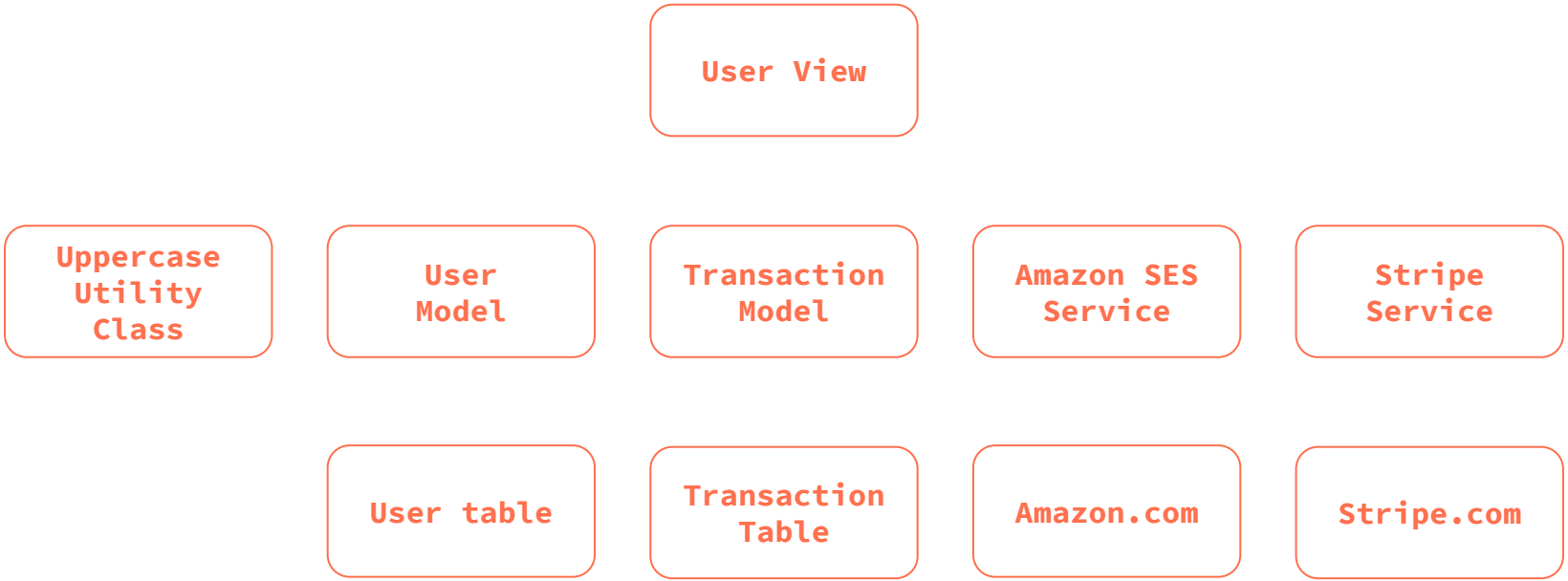


Solving Fat Views with Dark Magic

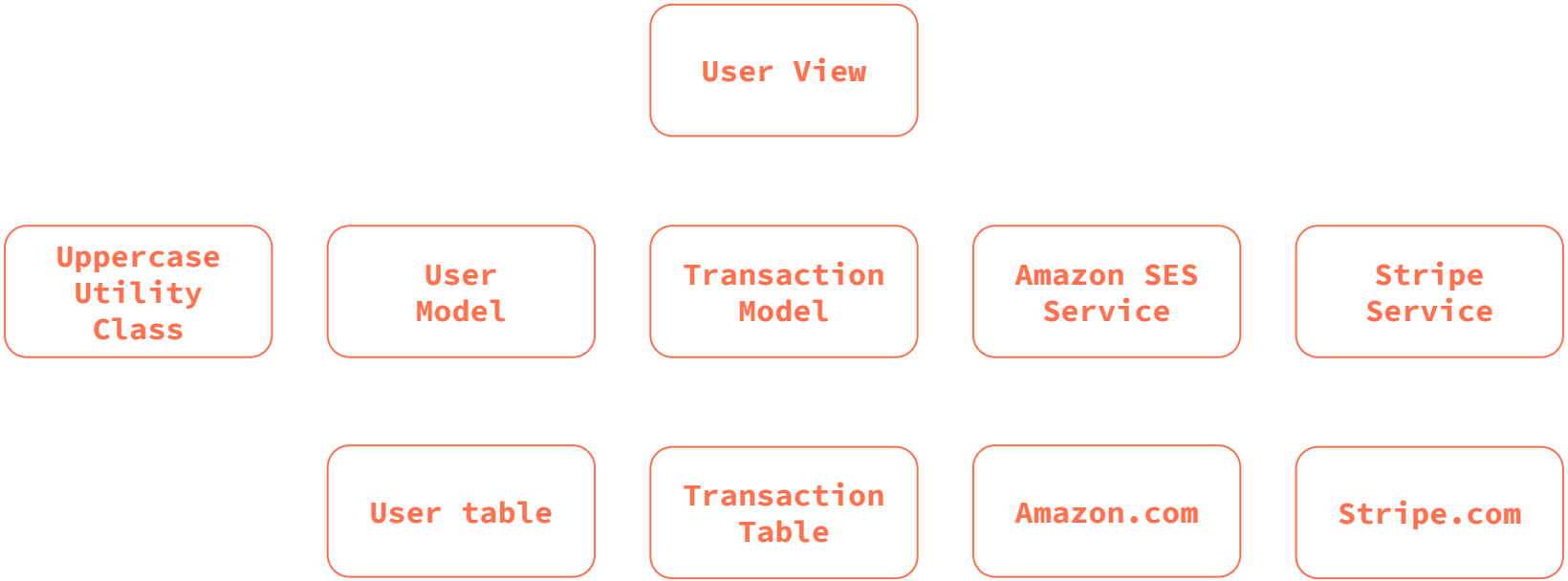




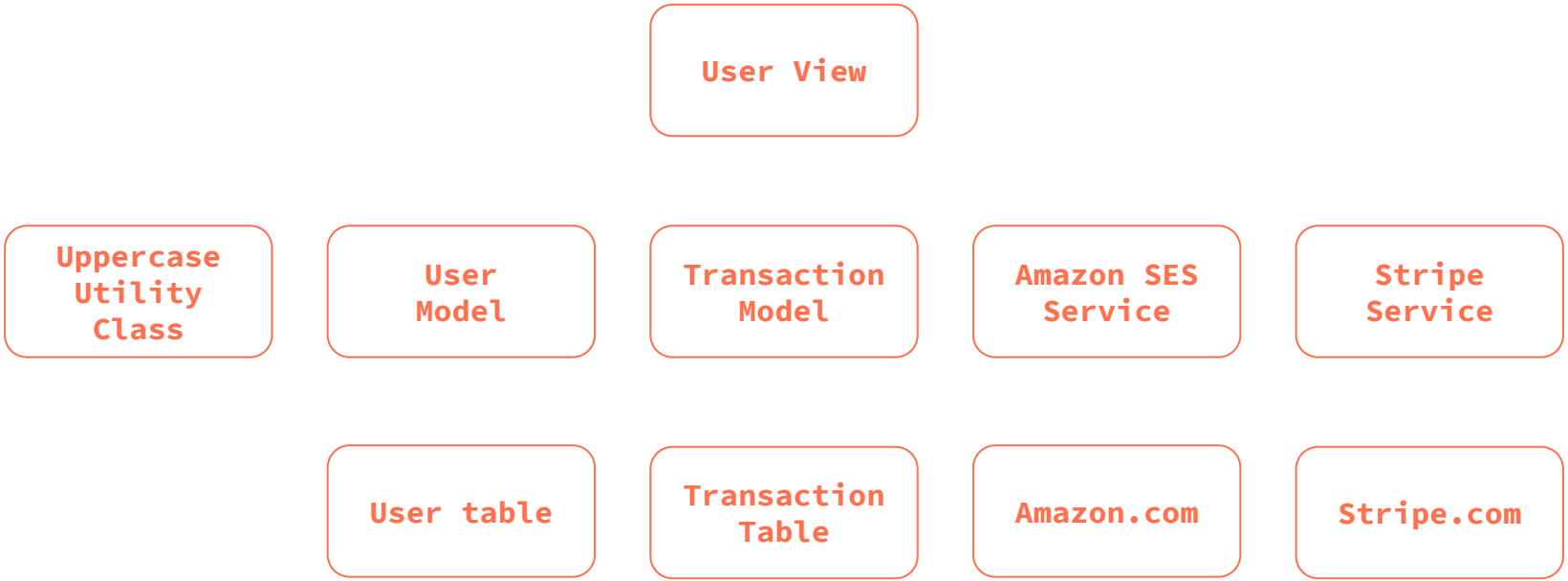
Put code in the model?



Put code in a utility class?



Put code in a service class?



Why is it like this?

Frameworks optimize for CRUD apps.

CRUD stands for **C**reate / **R**ead / **U**pdate / **D**elete.
You can solve many business problems with simple CRUD apps.

The Good

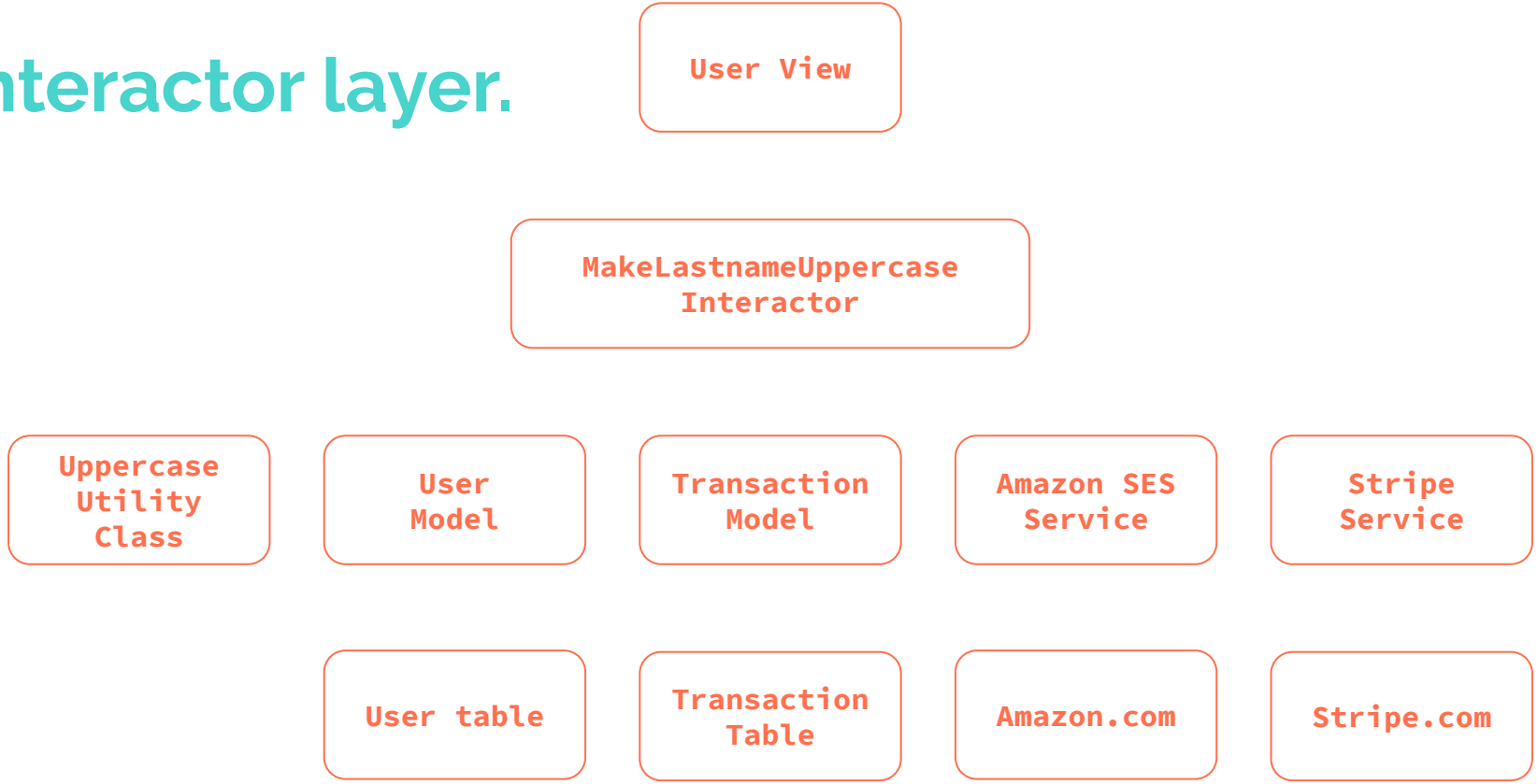
- Aligns nicely with the HTTP Verbs: POST / GET / PUT (or PATCH) / DELETE.
- Rapid development.

The Bad

- Frameworks often OVER-optimize for CRUD. See TodoMVC, Todo-Backend
- No clear conventions for other business logic.

Solution? New layer.

Interactor layer.



This is not a new idea...

"Robert C. Martin talks about a layer called UseCase on his book Clean Architecture. Eric Evans calls it as Service layer on his book Domain-Driven design. Martin Fowler calls it Service layer as well, but sometimes he refers to a similar layer, the Controller layer."

Business Logic in Django Projects

I prefer the term "Interactors."
The other words already mean something.

What do interactors look like?

Filesystem Location

Some hypothetical interactors:

Located at top level of application source.

```
.../interactors/register_user.py
.../interactors/become_standard_member.py
.../interactors/become_gold_member.py
.../interactors/handle_gdpr_request.py
.../interactors/cancel_membership.py
.../interactors/generate_monthly_usage_report.py
.../interactors/handle_expired_credit_card.py
```

Best Practices for Location and Naming

Put all interactors in one directory.
Keep other classes out of the directory.
Name files “verb_adjective_noun.py”
Name classes “VerbAdjectiveNoun”

Makes capabilities of the system clear.
Helps new team members ramp up quickly.
Makes code navigation easier.
Makes code coverage reports more meaningful.

Placeholder: Sample Interactor

Best Practices for Writing Interactors

Curate your import statements.
Reveals dependencies and side-effects.

Avoid handling ‘view’ logic.
Define errors locally, give them descriptive names.
Give the constructor explicit parameters.
Put fiddly logic into class methods with “_private” names.
Write ‘_private’ methods in a functional style.
Makes unit tests obvious.
Makes unit testing easy.

Do validation and authorization stuff at the top.
Makes the logic really clear.

Placeholder: Sample Unit Test

Best Practices for Testing Interactors

First, test “_private” methods.
Should be written in a functional style, easy to test, quick win.
Also, highest value to test -- it contains the tricky code.

Next, test that all exceptions are thrown.
Ensures that failure cases are working as expected.
Especially important for stuff related to security.

Finally, verify the main success flow using mock objects.
If done properly, main flow should be straightforward and fairly short.

Placeholder: Sample Caller

Best Practices for Migration

Only migrate if necessary.
Some apps don’t need it.

Only migrate the necessary parts.
Let the framework do what it does best.

Migrate slowly and gradually.
Grand rewrites never work.

Recap

“Interactors” are nothing more
than a few simple conventions around
grouping, naming, and structuring files.

But if you do it well,
you gain a HUGE amount of code clarity.