Managing Complexity with Architecture Patterns in Python

Author: Chad Furman

Claps: 155

Date: Oct 6, 2021

Does your source code feel like a big ball of mud? Are dependencies interwoven throughout your codebase to the point where change feels dangerous or impossible?

As the business grows and the domain model (the business problem youâ€™re solving in the application) becomes more complex, how do we untangle the mess weâ€™ve created without re-writing everything from scratch? Better yet, how do we avoid having the mess in the first place?

Birdâ€™s-eye View

Below is a brief summary of the techniques presented in Architecture Patterns in Python:

Layered Architecture

- Single Responsibility
- Views vs Services vs Repositories vs ORM vs Domain
- Dependency Inversion
- High vs Low Level Modules
- Abstractions

Domain Driven Design

- â€œBusiness Speakâ€� first
- Domain Modeling (Event Storming, etcâ€¦)
- Entities vs ValueObjects vs Domain Services
- Data Classes

Test Driven Development

- What is TDD
- Testing in High Gear at the Service Layer
- Testing in Low Gear at the Domain

Design Patterns

- Repository Pattern
- Service Layer Pattern
- Unit of Work Pattern
- Aggregate Pattern

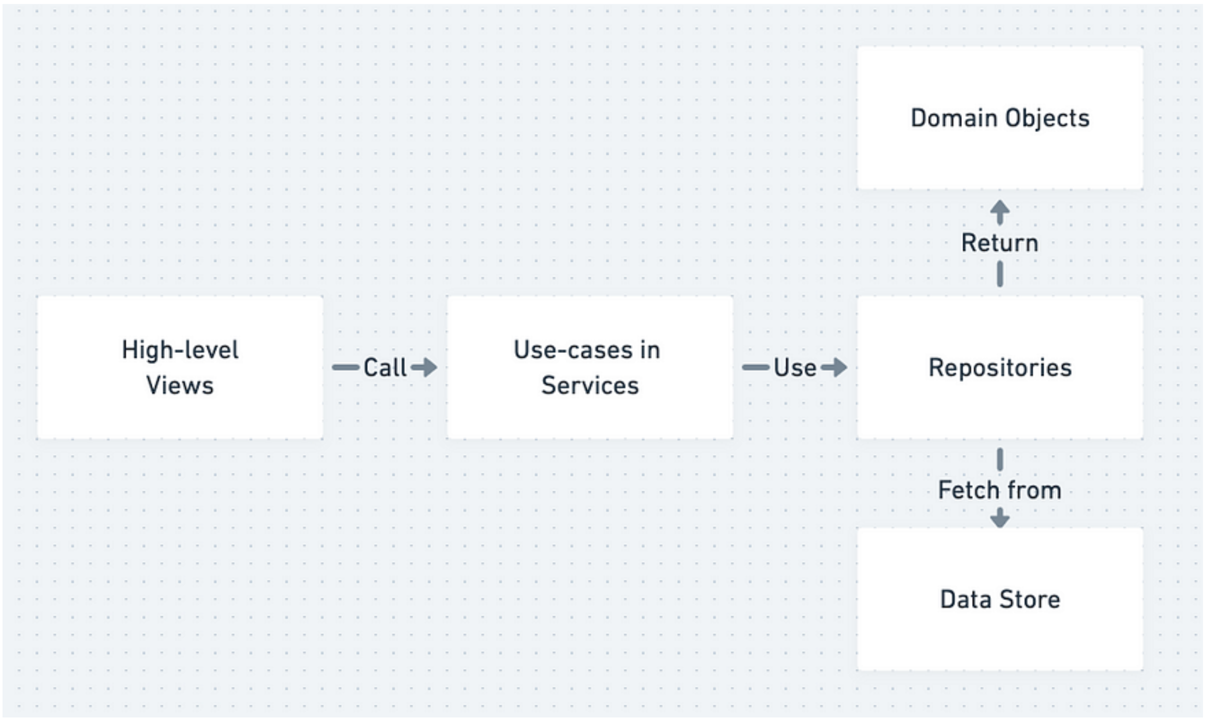
Event Driven Architecture

- Events

- Message Bus
- Event Handlers as the Service Layer
- Temporal Decoupling
- Queues and Brokers
- Idempotency, Failures, and Monitoring
- Commands
- CQRS
- Simple Reads vs Complex Commands
- Denormalization, Caching, and Eventual Consistency

Iâ€™ll touch on each of these topics briefly, but Iâ€™m not going to re-print the book in this blog post. These will be my own words and my interpretation, so if you want the â€œreal-dealâ€� I suggest you go to the source and grab yourself a copy of this book :)

Layered Architecture



A simplified summary of Layered Architecture

The **SOLID** principles are heavily present in good design. Briefly, Iâ€™ll explain what these are if you donâ€™t know. S, for **Single Responsibility**, means code should have one reason to change and only one reason. O, for **Open-Closed**, means that your code should be open for extension but closed for modification. L, for **Liskov Substitution**, means that instances of child classes can replace usages of their parent classes without altering behavior. I, for **Interface Segregation**, means that your code should not be forced to implement behaviors it does not use. And finally, D, for **Dependency Inversion**, implies a sort of loose coupling.

Single-Responsibility is the motivation behind the Layered Architecture. That is, your Django views have the responsibility of handling the HTTP transactions â€” getting the inputs, sending the outputs and the status codes. These views should delegate to Services which orchestrate the business logic. Services implement the use-cases and should be dependent on abstractions around the low-level details, and these abstractions can include Repositories (for storage abstractions) and Units of Work (for transactional or atomic operation management).

These layers (Views, Services, Repositories/UoWs) start at the high level where your business is concerned with a specific use-case / endpoint / webpage. Then they use layers of abstraction down to the low-level operations where we write to the database (in Repositories) or talk to other systems etc. This is the principle of Dependency Inversion.

The Principle of **Dependency Inversion** has two parts. First, that high level modules should not depend on low level modules, and both should depend on abstractions. Second, that abstractions should not depend on details, but details should depend on abstractions. Because this is such a complex topic, I wonâ€™t dwell on it and if youâ€™re interested I suggest you find further reading here, here, or even better in this book!

Domain Driven Design

source: https://pixabay.com/photos/engineer-engineering-4941336/

Also known as DDD. Be a master of your domain! What is the domain? Well, actually, that depends on what business problem youâ€™re trying to solve! No, Iâ€™m not being facetious. It literally depends â€” the definition of the domain is the business problem youâ€™re trying to solve!

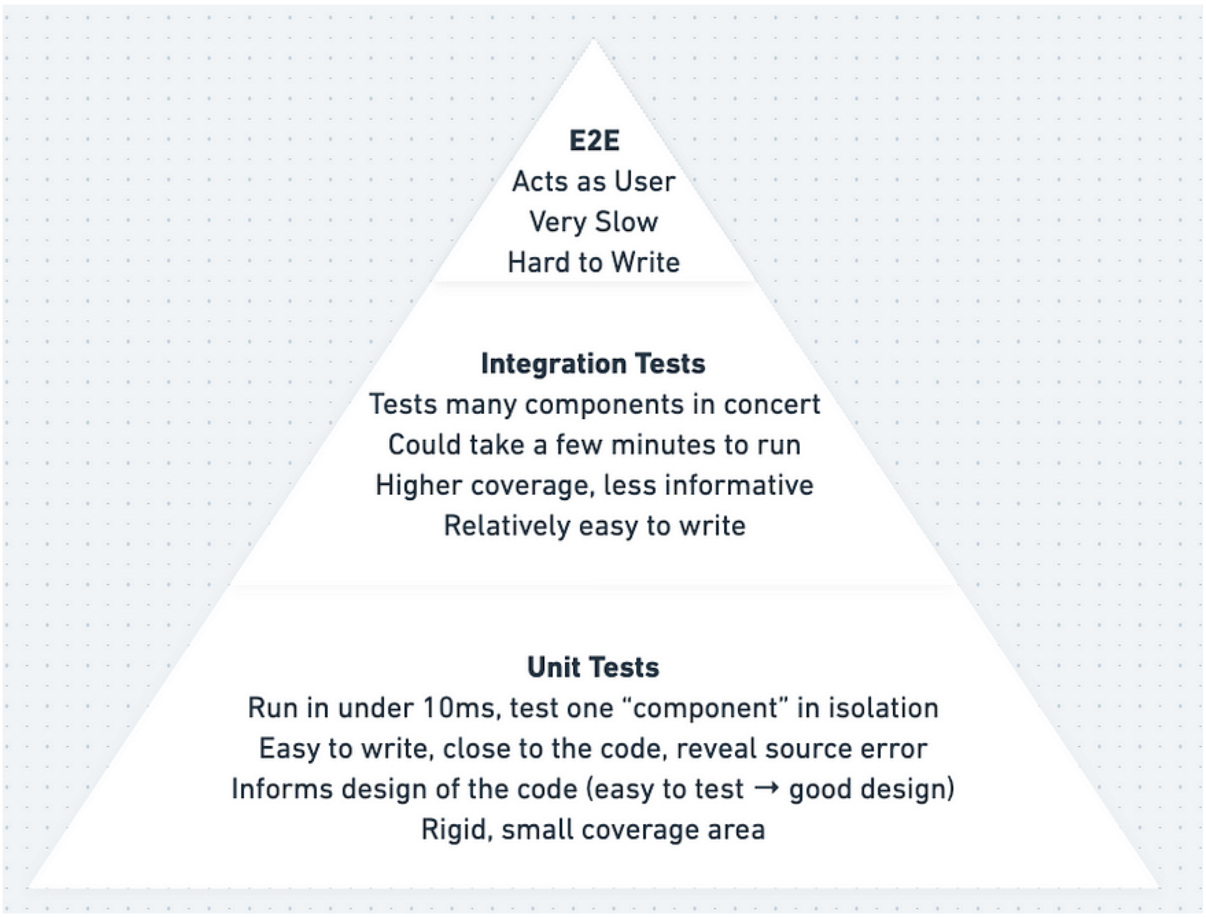
That is, if youâ€™re working for a shipping company, then when you go to model your domain youâ€™ll find you likely have â€œShippingContainersâ€� and â€œShipsâ€� or â€œTrucksâ€� etc. You likely have â€œSalesReportsâ€� and â€œPackingManifestsâ€�. But if you were to, say, be working for a Software company, then these Domain Objects wouldnâ€™t make much sense and youâ€™d have a totally different domain model.

The process of figuring out your domain model is referred to asâ€¦ â€œDomain Modelingâ€�. You can use several different techniques for this, and one of my favorite is â€œEvent Stormingâ€� (https://eventstorming.com/). Basically, though, the TLDR is that you need to sit down with the stakeholders (the people who need the problem solved) and figure out the language theyâ€™re using. Write down nouns and verbs, connect them together, and figure out how your domain works. Get this right, and it makes the rest of the process easier.

Youâ€™ll need to then turn this Domain Model into actual code. For our purposes, we focus on â€œEntitiesâ€� and â€œValueObjectsâ€� â€” the difference being that an Entity has a permanent identity (like an ID field, for example) and a ValueObject changes identity based on itsâ€¦ well.. valuesâ€¦ To give an example, a â€œUserâ€� would have an ID field and you could change the Userâ€™s email without changing the actual user. A ValueObject, however, would be something like an Address. If you change the value of the address, you have a new address! See how that works?

You can represent your Domain Model in python pretty simply using â€œ@dataclassâ€� which sets up your constructors for you and some other neat things. This can give you a pretty simple object designed just to store specific properties (like city, state, zip, for example, or like firstname, lastname, etc). Then you can return these objects from your Repositories and youâ€™ll have a consistent structure for passing around your application. Make your Domain Models reference each other by IDs and hydrate as necessary, optionally storing in a cache, and youâ€™re off to the races.

Test Driven Development


E2E
Acts as User
Very Slow
Hard to Write
Integration Tests
Tests many components in concert
Could take a few minutes to run
Higher coverage, less informative
Relatively easy to write
Unit Tests
Run in under 10ms, test one “component” in isolation
Easy to write, close to the code, reveal source error
Informs design of the code (easy to test → good design)
Rigid, small coverage area

The â€œTesting Pyramidâ€� with explanations

TDD for some is a controversial topic. If youâ€™re not familiar, the basic premise of TDD is that there are just three rules:

1. Youâ€™re not allowed to write any code unless you have a failing test.

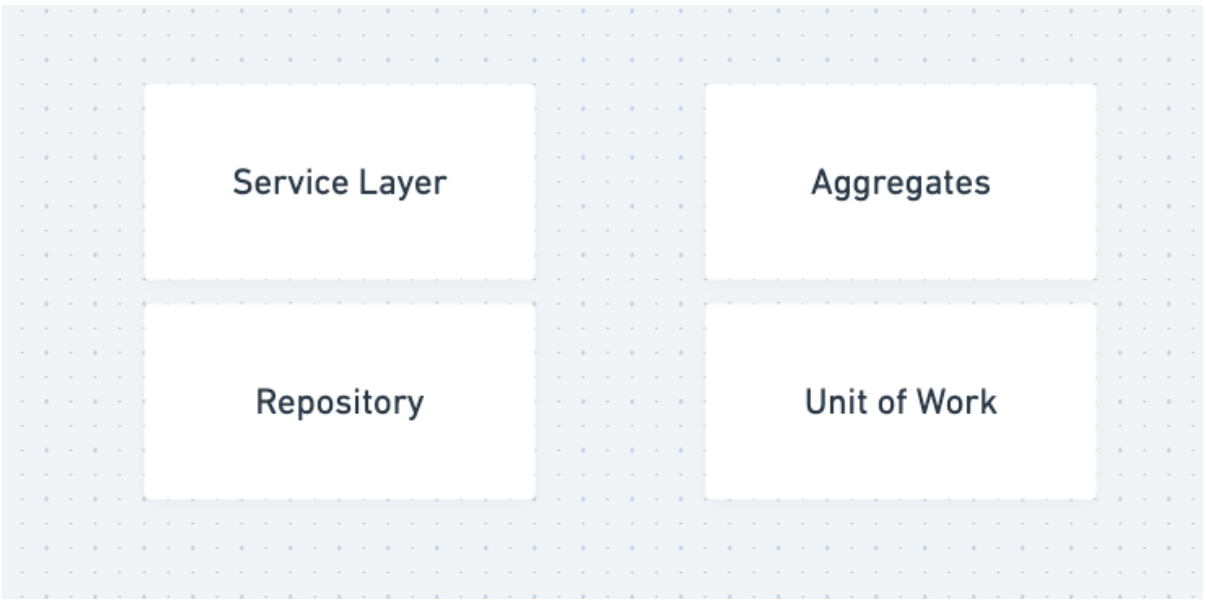
2. Youâ€™re only allowed to write one test case at a time, and it should start out failing.
3. Once you have a failing test, you should only write just enough code to make that test pass.

Thatâ€™s it. Then you repeat. People say this full loop is a 30-second process â€” I suspect theyâ€™ve been practicing it a bit more than I have. This is also known as â€œoffensiveâ€� testing rather than the â€œdefensiveâ€� testing weâ€™re all used to â€” that is, defensive testing is when you write your tests after the fact to â€œprotectâ€� yourself. Defensive testing gives you some protection, but itâ€™s much harder to get high coverage. Offensive testing gives you 100% coverage out of the gate and *forces* you to write testable code using abstractions and the like.

That said, TDD is not a magic bullet. It is not a religion. There are (rarely) cases where TDD does not work. TDD also does not prevent you from writing bugs or writing bad code (you can still also write bad tests, too). With this in mind, itâ€™s important to maximize the value of your tests by testing in â€œhigh gearâ€� when possible and in â€œlow gearâ€� when necessary.

Testing in High Gear vs Low Gear is a concept that is discussed in this book. To summarize, â€œhigh gearâ€� is when youâ€™re writing tests at the service layer or with other high-level modules (see â€œLayered Architectureâ€� above). They tend to cover more code and are best when adding new features or fixing simple bugs. Testing in â€œlow gearâ€� is at the domain level and with other low-level modules. Low gear is best when facing especially difficult bugs or when doing a very large refactor.

Design Patterns



A simple layout of the design patterns we talk about

There are quite a few design patterns that are worth knowing. Some other books, like â€œDesign Patterns: Elements of Reusable Object Oriented Softwareâ€� cover several more of them. Architecture Patterns in Python focuses on four specifically: the Repository Pattern, Service Layer Pattern, Unit of Work Pattern, and Aggregate Pattern.

Repositories are an abstraction around your storage mechanism. You can have a repository for Redis, for a CSV file, for a Database, etc. They can all meet a common interface and you could swap one for the other, if you really wanted to. The goal is to abstract away the low-level details so that your high level modules do not depend on low level details. This is important for layered architecture, and thatâ€™s why this book uses the Repository pattern extensively.

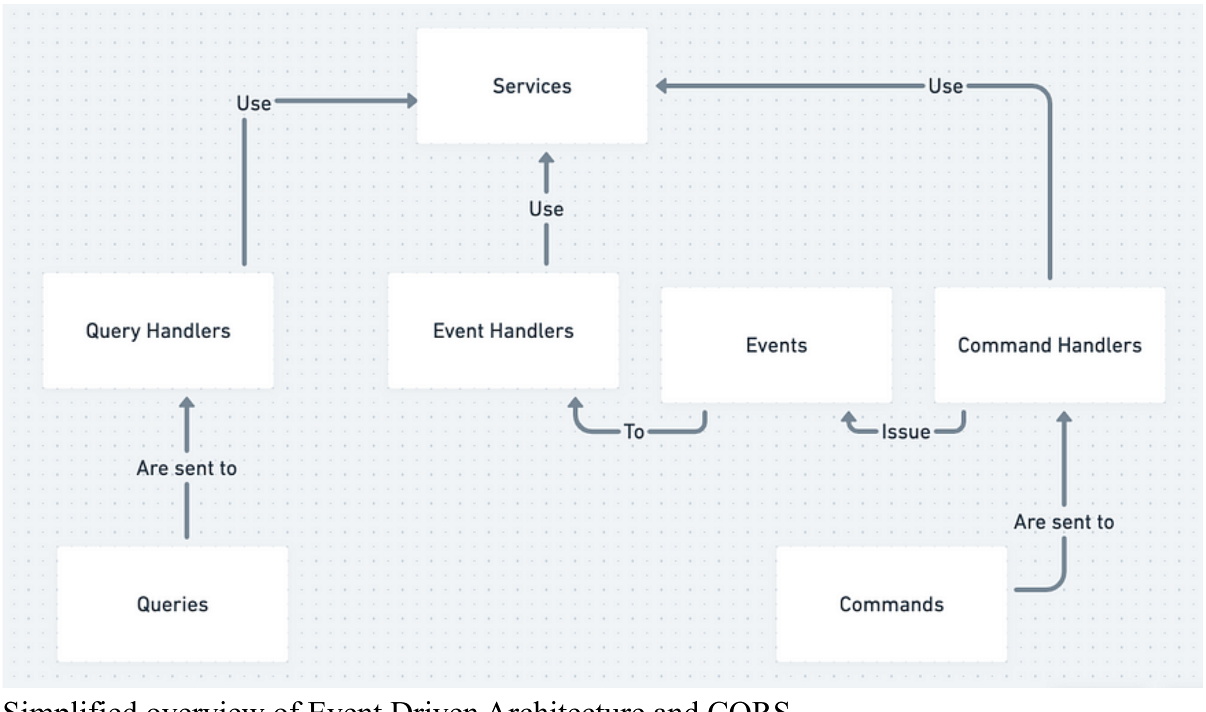
Service Layers are simply the orchestration for your business logic. When you first start writing API endpoints, the tendency is to put all your business logic in a single function that handles the API request. This violates the Single Responsibility Principle in that the API Endpoint Handler is now responsible for managing the HTTP input, response, and all the various aspects of the business logic like creating users, validating input, logging them in, etc. These lower level (albeit not the lowest level) tasks can be delegated to services which have a method for each use-case. That is, the service would have a method to sign up a user, log a user in, etc. These methods would call into the repositories and would receive back Domain Objects.

Units of Work are for atomic operations. Think â€œdatabase transactionsâ€� and â€œlocksâ€� and generally encapsulating related operations. If you need to â€œbook a hotel roomâ€� then you could have a â€œUnit of Workâ€� that wraps this logic. If, during looking up available rooms and assigning the room to a person and processing the personâ€™s payment information there happens to be some sort of error, the Unit of Work will nicely roll back all of this logic for you. You could rely on low-level database transactions (and your Unit of Work may be doing this under the hood), but to in-line that logic in your service function starts to muddy up your code. Using a Unit of Work for handling these atomic operations provides a clean interface that can take advantage of Pythonâ€™s powerful â€œwithâ€� statement and automatically clean up after you as needed.

Aggregates are collections of Domain Objects with a common consistency boundary. Things like a Shopping Cart could be an aggregate â€” thereâ€™s several domain objects inside the shopping cart, and there may even be other aggregates inside the shopping cart. Itâ€™s useful, though, during checkout, to treat the Shopping Cart as a single unit. You can think of an Aggregate like a tree of objects, and you can refer to the aggregate by the root.

One additional note about Aggregates is that you should have one aggregate per repository. Put another way, you should not have Repositories for Domain Objects which are not Aggregates. In this way, Aggregates form the â€œpublicâ€� API of your Domain Model.

Event Driven Architecture



Simplified overview of Event Driven Architecture and CQRS

Put simply, EDA is when you use â€œeventsâ€� as an input into your system. An event (or Domain Event) is a ValueObject, and you can have Internal and External events. Internal events never leave your system and are usually handled by something like a Message Bus (a simple router that maps events to event handlers). External events get sent to other systems and are great for â€œtemporal decouplingâ€� â€" you can issue an event to a message broker which manages a series of queue workers asynchronously.

All events can fail, and how we handle the failure is important. We need monitoring to know when events are failing and which events failed. We also need our event handlers to be idempotent so when we retry the events, nothing unexpected happens. Regular events can fail somewhat safely without affecting the overall operation, and this is an important distinction between events and commands.

Commands are a special type of event. Whereas a regular event can have multiple handlers, a command only has one handler. A command, when it fails, should re-throw the exception up the stack whereas when an event fails, there should be some graceful exception handling. Commands usually modify data and trigger side effects, and separating this from the â€œreturning dataâ€� operation is the goal of CQRS (Command/Query Responsibility Segregation).

The primary motivation behind CQRS is that Commands are expensive and complex and usually necessitate atomicity to a certain extent as well as immediate consistency. Queries, on the other hand, are simple read operations. Queries do not usually rely on the domain (the business logic), where Commands typically do rely on the domain. Queries can be executed against a read replica, where commands are typically best done against the primary data store. Queries can also take advantage of denormalized data and eventual consistency. This is great because thereâ€™s typically several orders of magnitude more queries than there are commands and this helps the system scale better.

Applying all of this

In summary, itâ€™s important to go piece-by-piece. You donâ€™t need to do all of this at once. If youâ€™re hesitant to try out Units of Work or you donâ€™t have any immediate use for Aggregates or you donâ€™t even have a Domain Model, thatâ€™s okay! One of the easiest and most effective things you can start with is by approaching Layered Architecture â€" see if you can decouple your lower-level modules from your higher-level modules using services. See if you can isolate your storage logic into a repository that your services use. Even better if you can also map out some simple data classes to represent your domain objects and have your ORM depend on these.

If you group logic that depends on itself together and you separate modules with abstractions, youâ€™ll be part of the way there. See where your seams are and begin to split your code out into testable chunks. For some excellent examples of this, check out â€œWorking Effectively with Legacy Codeâ€� which is both a great book and is cited by â€œArchitecture Patterns in Pythonâ€�.

Oh, and if you havenâ€™t yet, read â€œArchitecture Patterns in Pythonâ€� and especially note the epilogue! This will get you much more context on everything I glossed over above. I summarized a 300+ page book in about 5 pages, so thereâ€™s definitely some stuff I left out :)