Statement of Purpose

By JD Margulici :: linkedin.com/in/jdmargulici/ :: February 4, 2021

Overview

This document constitutes a statement of purpose for the Anaximander framework. As indicated by the tag line, Anaximander is intended as an open-source Python package that organizes and simplifies the code base of data-intensive application backends. In a nutshell, Anaximander provides primitives for declaring data models and interfacing Python programs with data storage engines. One way to think about it is as a generalization of object-relational mappers to both relational and non-relational data in order to support applications that employ a plurality of database technologies -which has effectively become the norm for most systems.

Context

Evidently the purpose of a programming framework is to solve problems faced by application developers, and I can think of no better way to describe the problems that Anaximander intends to address than to share the personal stories that have led me here.

Since 2013, I have been the founding Chief Technology Officer of three different startup companies. With each project I had to design and develop a new software application from scratch, and it also happens that these projects all involved complex data transformations. In practice I had to be both a data scientist and a software engineer. And because of that, I each time faced a conundrum: should I first focus on the science and then port it over to a production-grade application, or should I first establish a data infrastructure to host prototypes or early customer demos, and then delve into the science? In actuality, both approaches were necessary -or neither, depending on how you look at the question. By now we are all bought into the lean startup paradigm, but it does not readily answer my question unless we provide a lot more context about the business environment.

Many readers may already have an opinion about the proper articulation between data science and software engineering. This is a hot topic that bogs down many organizations and has prompted the emergence of numerous data platforms and products. Generally, the narrative depicts data scientists running models and experiments in a Python sandbox until they have coded a pipeline that can be incorporated into production systems with minimal modifications. To enable this, said data scientists need repeatable and consistent access to production data and metadata in Python code. Hence the same chicken-and-egg problem is found at every stage of the product lifecycle, but mature organizations have dedicated teams working on both sides.

In hindsight, my biggest frustration came from how much time my teams and myself had to spend writing boiler-plate code while also needing to think deeply about our models and algorithms. In all honesty, the pain was actually amplified by my desire to keep it under control: I realized early on that I needed a framework, but writing library code added its own share of complexity and failures. In the process, I ended up writing three formal versions of Anaximander, and many other related modules. The premise remained relatively consistent throughout these attempts: how to combine the expressive power and separation of concerns inherent to object-oriented programming with the data science

toolkit, all while minimizing the amount of code that needs to be written and maintained. Specifically, much of my efforts has gone into leveraging Pandas dataframes in production. Dataframe objects are self-describing but their flexibility makes them tricky to process in a repeatable manner, as there is no built-in mechanism to validate a dataframe against a schema. Hence a core concept of Anaximander is the DataObject, which is a way to expose data such as a dataframe and supplement it with metadata and validation methods so that it can safely be used in production. A robust discussion of this concept can be found in my 2020 paper *Tabular Data Archetypes*.

Each one of my previous attempts to write Anaximander aborted for essentially the same reasons: hitting the sweet spot of functionality and usability while keeping complexity manageable has proved a difficult endeavor. While some earlier versions of Anaximander have run in production, they were not usable enough to deserve publication. The takeaway for me is that success necessarily requires a clear and singular purpose with every layer of functionality. This document is my initial take at an overarching scope, a set of design objectives, and an overview of the use cases of the framework.

Metamodel

Before enunciating the objectives of the framework, a set of modeling assumptions is in order. This section provides a high-level overview of the metamodel underlying Anaximander. This is also a good segue to explain how its name was attributed. Anaximander (c. 610 – c. 546 BC) is one of the earliest known Greek philosopher, a disciple of Thales, whom he succeeded as the master of the so-called Milesian school. A keen observer of nature, Anaximander contributed to multiple scientific disciplines but is most famous for his theories on the origins of the universe and its component parts. As such he is sometimes referred to as the first metaphysicist.

The first system that I designed was a mapping application and involved large sets of spatiotemporal data. This led me to a deep exploration into the nature of information as a representation of reality -a metaphysical quest of sort, which I recognize to be a rabbit hole, but which nonetheless keeps yielding interesting insights. This is what now enables me to propose a set of information primitives that at least superficially appears to cover the bulk of use cases encountered in application modeling.

Traditionally, computer science has defined data types from the bottom up: data types exist because they correspond to different bit-level representations and interpretations. As a dynamically typed language, Python partially frees programmers from reasoning about types. But since the entire software stack makes data types a primary concern, it is hard to escape. If I create an integer index column in a database table, the column is defined as an integer column first, and secondarily as the table's index.

Meanwhile we have coined the term "data science", but it is effectively a catch-all neologism for statistical modeling and related software techniques, whereas not much attention has been given to categorizing data -which you would think is what a science of data would focus on first and foremost. Yet anyone who has manipulated data would recognize that types exist at the semantic level. For instance, a time series of temperature measurements differ in some fundamental way from the list of student's heights in my eldest son's fifth-grade classroom. It is not just that they represent distinct physical quantities: the observations relate to one another differently (time series vs. categorical index). If one is to model a real-world problem with data, this kind of semantic categorization appears infinitely more relevant than the fact that both objects store floats. In fact, this difference in perspective reminds me a bit of the contrast between mathematics and physics: mathematics treat numbers in the abstract,

whereas physics use numbers to describe natural phenomena. This is not to imply any kind of value judgment on computer scientists, data scientists, mathematicians or physicists: every discipline has its own purpose and methods. In particular, software gets written on computers, so no matter how fancy we get in our application modeling, integers and floats have to be reckoned with at some level of the stack. My point is that as an application designer I would rather reason about data in terms of its usage and meaning than in terms of its encoding.

With these somewhat philosophical statements established, let us now examine the proposed metamodel. At its root, the framework recognizes five types -call them “archetypes”, of data objects, as follows:

1. **Entities**: entities form the backbone of the application domain. They encompass all the typical concepts you would find in a software application: users, products, places, things, etc. If I design a parking application, I may create entities to represent parking spaces, vehicles, and parking attendants, to name a few examples. Usually, the entities have defining attributes and are linked by relations -e.g. the second floor in a parking garage has a relation to all the individual parking spaces on that floor, and as such they tend to be stored in relational databases, though other models are possible as well.
2. **Event Logs**: phenomenological data is, almost by definition, time series of events. Whether we consider parking transactions, user clicks or moon phases, every natural manifestation takes place on a time continuum. Software engineers will tend to think of event logs as a dump of machine-generated messages where one looks for bugs. But in the context of Anaximander we are not directly concerned with modeling such logs (unless, that is, we are writing a software application whose domain happens to be computer systems). Here, event logs refer to all information that is indexed with timestamps and forms natural sequences. Event logs may be stored in any kind of backend, but column-oriented databases such as HBase, Bigtable or Cassandra are particularly well suited because their storage strategy preserves sequencing.
3. **Specifications**: the specification archetype refers to nested data, commonly stored as JSON or YAML files. This kind of data is pervasive throughout software applications: configuration files, document-oriented data models, or any kind of complex specification that a flat, tabular model fails to adequately represent. Nested data has its own storage engine in the form of document-oriented databases such as MongoDB, but another popular option is to store files in a Data Lake. Another pattern I have used is to attach complex specifications to an entity type and store it in a JSON column of the entity’s relational database table.
4. **Summaries**: this archetype refers to arbitrary tabular data that may be indexed in a variety of ways. One can generate a summary with a simple selection query against a database table, or it could be the result of complex transformation and summarization operations -hence the name. Tabular data is ultimately the bread and butter of most data-intensive software systems, and it may be generated from any storage backend.
5. **Collections**: finally, the collections archetype designate groups of data objects, classically organized as either sequential lists, key-value mappings, or unordered sets. A collection data object contains arbitrary data objects -though the contained objects must be homogeneous. Hence it could refer to a list of users, a mapping of summaries or a set of event logs. And of course, collections enable recursive structures, such that a collection object may itself contain collections.

An important observation that must be provided at this juncture is that the dividing lines between these categories are both strict in terms of their semantic interpretation, yet porous at the implementation level. For instance, it is obvious that a query against a database table of customers can be modeled either as a simple summary or as a list of entities. Likewise a time series fits the event logs archetype most naturally, but in some contexts it may be treated as a summary. In other words, archetypes are programming interfaces, and they do not silo application data. Anaximander will provide easy conversion methods to allow the same data to be represented as different archetypes, depending on the use case. If I query customers to run analytics on their characteristics, I am probably looking at my query result as a tabular summary. On the other end, if my workload is more transactional in nature, I might rather instantiate customer objects with attributes and methods that streamline my business logic. The intended value of the Anaximander framework is to let programmers declare a customer class as an entity type and use that class to specify backend storage and perform I/O operations with different kinds and shapes of data objects.

Scope and design objectives

One of my main sources of inspiration is the object-process methodology formulated by Technion professor Dov Dori in the 1990s. Its premise is straightforward: even after 50 years, the practice of information technology does not provide a robust systems engineering framework for application development. This is not to say that there are not plenty of best practices, thought leadership and excellent development platforms. Yet when it comes to laying the foundations of a new software application, there is not really a conceptual framework to go by. I would say the closest thing we have to that would be object-oriented programming and design patterns. But in the new world of big data these concepts do not really fit. Object-process distills the essence of a software application to its core: data representations, and atomic transformations. As a result, this is how I think about Anaximander: a framework for specifying and handling the application's data objects, upon which the program runs various operators. Having dabbed with both sides of this equation, I have arrived at the conclusion that there is plenty of usefulness that can be derived from data objects alone. The processing side, which one could envision as libraries of data operators, is therefore out of scope for the time being.

Another important premise of the Anaximander framework is that it must be understood as a programming interface. There is no intention here to create software functionality that is not already available through other means. In fact, the implementation of Anaximander will lean heavily on existing Python packages, and in some cases merely add a thin layer to glue them together. Rather, the functionality lies in an easy to use, object-oriented interface to the application's data, which serves both the interactive data science use case and a range of production code use cases.

The implication is that programmer-friendliness must be the overarching design goal in order for the framework to accomplish its purpose. In this respect, a great model to follow is the Python *attrs* framework, which considerably reduces the amount of boiler plate code one needs to write to obtain fully functional object classes. Anaximander will similarly aim for concise, declarative statements that pack a lot of convenient and reliable functionality. The difference is that *attrs* can work for any kind of Python class, irrespective of the use case, whereas Anaximander focuses on data classes with predefined semantics.

Having said all this, we now come to what is probably the most crucial question in this document: who would use Anaximander, and why? While I am confident in my assessment that there is a gap in the Python ecosystem that Anaximander can fill, why would one bother to learn yet another framework that does not even purport to provide additional functionality? Further, the data ecosystem is sprawling with data storage and processing systems that all interact as online services. Where does Anaximander fit?

To answer this question, my best bet is to once again go back to my personal experience. Let us imagine a single developer or small team setting out to write a new data application from scratch, and consider what they would need. Again, the stretch -and the opportunity, is that both the science and the product must be developed concurrently. This means that we require the following capabilities:

- Cloud storage for application data
- Local storage for low-latency development and testing
- Fault-tolerant and scalable cloud data pipelines
- To manage the cloud data pipelines, you need solid application logs and tracing
- Local data pipelines for low-latency development and testing
- To successfully program data pipelines, you ideally want interactive debugging
- Shared banks of data sets for development, training, validation, unit tests, integration tests, customer demos, and eventually production data organized by customer or project
- Because you are building, your schemas are going to be constantly changing, so you need easy ways to make and propagate those changes
- Likewise, the team is going to spend a lot of time making interactive data queries as they figure out how to cobble together different data sets, so clear and consistent interfaces will considerably improve productivity. This is not limited to writing queries: people need accessible metadata and directories to find data
- Data visualization not only for interactive development, but also for process testing and validation, as well as product prototyping

Now the good news is that all the enabling technologies to set up this stack are available at virtually no setup cost from cloud vendors. And further, these same vendors publish reference architectures that demonstrate how to build complete systems. The problem is simply that it is a lot of technologies and interfaces to wrap your head around, and it still doesn't tell you how to organize your code base.

Enter Anaximander! Start with your data ontology: what are the application's entities, the time series, the configurations? Encode those in Python and you literally already have an application stack: file-based storage for all manners of data objects in your domain, either local or in cloud buckets; orm-like base code for storing in a range of database engines: key-value stores, relational databases, columnar databases or document-oriented databases; conversely, base code for exposing data objects through REST or Socket-based APIs; a concise, object-oriented syntax for interactive and semantics-aware querying, plotting and summarization of data; a meta-architecture that drives code organization in a way that is consistent and predictable; all the power of the Python data ecosystem and clean, readable production code with batteries included (local and cloud logging, testing, visual debugging) to handle business logic.

In the end, I believe that the tagline aptly captures the statement of purpose: a rapid application development framework for data-intensive Python. Like any framework, it will have to trade off

usability, flexibility and performance, and it will make common use cases easier, but it will not satisfy every need. But I am willing to bet that there a lot of people out there who can program a good Pandas pipeline but would struggle to turn it into a production system, and this is who Anaximander wants to help.