flask-boiler

Release 0.0.1.dev5

Oct 20, 2021

Contents:

1 onto **1**
- 1.1 Connectors supported . . . 2
- 1.2 What I am currently trying to build . . . 2
- 1.3 What it already does . . . 2
- 1.4 Installation . . . 2
- 1.5 Object Lifecycle . . . 5
- 1.6 Typical ViewMediator Use Cases . . . 5
- 1.7 Comparisons . . . 7
- 1.8 Contributing . . . 8
- 1.9 License . . . 8

2 Migrate from google.cloud.firestore **9**
- 2.1 Add data . . . 9
- 2.2 Read data . . . 9
- 2.3 Save data . . . 10
- 2.4 Get data . . . 11
- 2.5 Simple queries . . . 11
- 2.6 Query operators . . . 12
- 2.7 Declare Models . . . 12
- 2.8 Field name conversion . . . 13

3 Context Management **15**
- 3.1 Auto Configuration with boiler.yaml . . . 15
- 3.2 Manual Configuration . . . 15

4 Quickstart **17**
- 4.1 Retrieve credentials . . . 17
- 4.2 Configure service account . . . 17
- 4.3 Add flask-boiler and the server client library to your app . . . 18
- 4.4 Create a Document As View . . . 18
- 4.5 Declare a Domain Model . . . 19
- 4.6 Add Entrypoint . . . 20
- 4.7 Create a form service . . . 21

5 Features **25**
- 5.1 Native to Distributed Systems . . . 25
- 5.2 Exchangable Components . . . 25

5.3 Object-Oriented Code . . . 26
5.4 N+1 Considerations . . . 26
5.5 Product Roadmap . . . 27

6 flask-boiler **29**
6.1 Domain Model . . . 29
6.2 View Model . . . 29
6.3 Flask REST API View Mediator . . . 29
6.4 Document View Mediator . . . 29
6.5 Query Delta View Mediator . . . 29
6.6 Context . . . 29
6.7 Auth . . . 29

7 Indices and tables **31**

CHAPTER 1

onto

Build Status Coverage Status Documentation Status

Demo:

When you change the attendance status of one of the participants in the meeting, all other participants receive an updated version of the list of people attending the meeting.

Untitled_2

Some reasons that you may want to use this framework or architectual practice:

- You want to build a reactive system and not just a reactive view.
- You want to build a scalable app that is native to distributed systems.
- You want a framework with a higher level of abstraction, so you can exchange components such as transportation protocols
- You want your code to be readable and clear and written mostly in python, while maintaining compatibility to different APIs.
- You have constantly-shifting requirements, and want to have the flexibility to migrate different layers, for example, switch from REST API to WebSocket to serve a resource.

This framework is at ***beta testing stage***. API is not guaranteed and ***may*** change.

Documentations: readthedocs

Quickstart: Quickstart

API Documentations: API Docs

Example of a Project using onto (now named onto): gravitate-backend

Related Technologies

1.1 Connectors supported

Implemented:

- REST API (Flask and Flasgger)
- GraphQL (Starlette)
- Firestore
- Firebase Functions
- JsonRPC (flask-jsonrpc)
- Leancloud Engine
- WebSocket (flask socketio)

To be supported:

- Flink Table API
- Kafka

1.2 What I am currently trying to build

Front end creates mutations in graphql. "Onto" receives the view model, and triggers action on domain model. A method in domain model is called (which lives in Flink Stateful Functions runtime). Different domain models communicate to persist a change, and save the output view into Kafka. Another set of system statically interprets "view model definition code" as SQL, and submit jobs with Flink SQL to assemble "view model". Eventually, the 1NF view of the data is sent to Kafka, and eventually delivered to front end in forms of GraphQL Subscription.

(Write side has Serializable-level consistency, and read side has eventual consistency)

1.3 What it already does

- Serialization and deserialization
- GraphQL/Flask server
- Multiple table join
- ...

1.4 Installation

In your project directory,

```
pip install onto
```

See more in Quickstart.

1.4.1 State Management

You can combine information gathered in domain models and serve them in Firestore, so that front end can read all data required from a single document or collection, without client-side queries and excessive server roundtrip time.

There is a medium article that explains a similar architecture called "reSolve" architecture.

See `examples/meeting_room/view_models` on how to use onto to expose a "view model" in firestore that can be queried directly by front end without aggregation.

1.4.2 Processor Modes

`onto` is essentially a framework for source-sink operations:

```
Source(s) -> Processor -> Sink(s)
```

Take query as an example,

- Boiler
- NoSQL
- Flink
  - staticmethods: converts to UDF
  - classmethods: converts to operators and aggregator's

1.4.3 Declare View Model

```
from onto.attrs import attrs

class CityView(ViewModel):

    name: str = attrs.nothing
    country: str = attrs.nothing

    @classmethod
    def new(cls, snapshot):
        store = CityStore()
        store.add_snapshot("city", dm_cls=City, snapshot=snapshot)
        store.refresh()
        return cls(store=store)

    @name.getter
    def name(self):
        return self.store.city.city_name

    @country.getter
    def country(self):
        return self.store.city.country

    @property
    def doc_ref(self):
        return CTX.db.document(f"cityView/{self.store.city.doc_id}")
```

1.4.4 Document View

```
class MeetingSessionGet(Mediator):

    from onto import source, sink

    source = source.domain_model(Meeting)
    sink = sink.firestore()  # TODO: check variable resolution order

    @source.triggers.on_update
    @source.triggers.on_create
    def materialize_meeting_session(self, obj):
        meeting = obj
        assert isinstance(meeting, Meeting)

        def notify(obj):
            for ref in obj._view_refs:
                self.sink.emit(reference=ref, snapshot=obj.to_snapshot())

        _ = MeetingSession.get(
            doc_id=meeting.doc_id,
            once=False,
            f_notify=notify
        )
        # mediator.notify(obj=obj)

    @classmethod
    def start(cls):
        cls.source.start()
```

1.4.5 Create Flask View

You can use a RestMediator to create a REST API. OpenAPI3 docs will be automatically generated in `<site_url>/apidocs` when you run `_ = Swagger(app)`.

```
app = Flask(__name__)

class MeetingSessionRest(Mediator):

    # from onto import source, sink

    view_model_cls = MeetingSessionC

    rest = RestViewModelSource()

    @rest.route('/<doc_id>', methods=('GET',))
    def materialize_meeting_session(self, doc_id):

        meeting = Meeting.get(doc_id=doc_id)

        def notify(obj):
            d = obj.to_snapshot().to_dict()
            content = jsonify(d)
            self.rest.emit(content)
```

(continues on next page)

(continued from previous page)

```
        _ = MeetingSessionC.get(
            doc_id=meeting.doc_id,
            once=False,
            f_notify=notify
        )

    # @rest.route('/', methods=('GET',))
    # def list_meeting_ids(self):
    #     return [meeting.to_snapshot().to_dict() for meeting in Meeting.all()]

    @classmethod
    def start(cls, app):
        cls.rest.start(app)

swagger = Swagger(app)

app.run(debug=True)
```

(currently under implementation)

1.5 Object Lifecycle

1.5.1 Once

Object created with `cls.new` -> Object exported with `obj.to_view_dict`.

1.5.2 Multi

Object created when a new domain model is created in database -> Object changed when underlying datasource changes -> Object calls `self.notify`

1.6 Typical ViewMediator Use Cases

Data flow direction is described as Source -> Sink. "Read" describes the flow of data where front end would find data in Sink useful. "Write" describes the flow of data where the Sink is the single source of truth.

1.6.1 Rest

Read: Request -> Response Write: Request -> Document

1. Front end sends HTTP request to Server
2. Server queries datastore
3. Server returns response

1.6.2 Query

Read: Document -> Document Write: Document -> Document

1. Datastore triggers update function
2. Server rebuilds ViewModel that may be changed as a result
3. Server saves newly built ViewModel to datastore

1.6.3 Query+Task

Read: Document -> Document Write: Document -> Document

1. Datastore triggers update function for document `d` at time `t`
2. Server starts a transaction
3. Server sets write_option to only allow commit if documents are last updated at time `t` (still under design)
4. Server builds ViewModel with transaction
5. Server saves ViewModel with transaction
6. Server marks document `d` as processed (remove document or update a field)
7. Server retries up to MAX_RETRIES from step 2 if precondition failed

1.6.4 WebSocket

Read: Document -> WebSocket Event Write: WebSocket Event -> Document

1. Front end subscribes to a ViewModel by sending a WebSocket event to server
2. Server attaches listener to the result of the query
3. Every time the result of the query is changed and consistent:
    1. Server rebuilds ViewModel that may be changed as a result
    2. Server publishes newly built ViewModel
4. Front end ends the session
5. Document listeners are released

1.6.5 Document

Read: Document -> Document Write: Document -> Document

1.6.6 Comparisons

| | Rest | Query | Query+Task | WebSocket | Document | |———————— |—— |——- |—————— |—————— |————- | | Guarantees | 1 (At-Most-Once) | 1 (At-Least-Once) | =1[^1] (Exactly-Once) | 1 (At-Most-Once) | 1 (At-Least-Once) | | Idempotence | If Implemented | No | Yes, with transaction[^1] | If Implemented | No | | Designed For | Stateless Lambda | Stateful Container | Stateless Lambda | Stateless Lambda | Stateful Container | | Latency | Higher | Higher | Higher | Lower | Higher | | Throughput | Higher when Scaled| Lower[^2] | Lower | Higher when Scaled | Lower[^2] | | Stateful | No | If Implemented | If Implemented | Yes | Yes | | Reactive | No | Yes | Yes | Yes | Yes |

[^1]: A message may be received and processed by multiple consumer, but only one consumer can successfully commit change and mark the event as processed. [^2]: Scalability is limited by the number of listeners you can attach to the datastore.

1.7 Comparisons

1.7.1 GraphQL

In GraphQL, the fields are evaluated with each query, but onto evaluates the fields if and only if the underlying data source changes. This leads to faster read for data that has not changed for a while. Also, the data source is expected to be consistent, as the field evaluation are triggered after all changes made in one transaction to firestore is read.

GraphQL, however, lets front-end customize the return. You must define the exact structure you want to return in onto. This nevertheless has its advantage as most documentations of the request and response can be done the same way as REST API.

1.7.2 REST API / Flask

REST API does not cache or store the response. When a view model is evaluated by onto, the response is stored in firestore forever until update or manual removal.

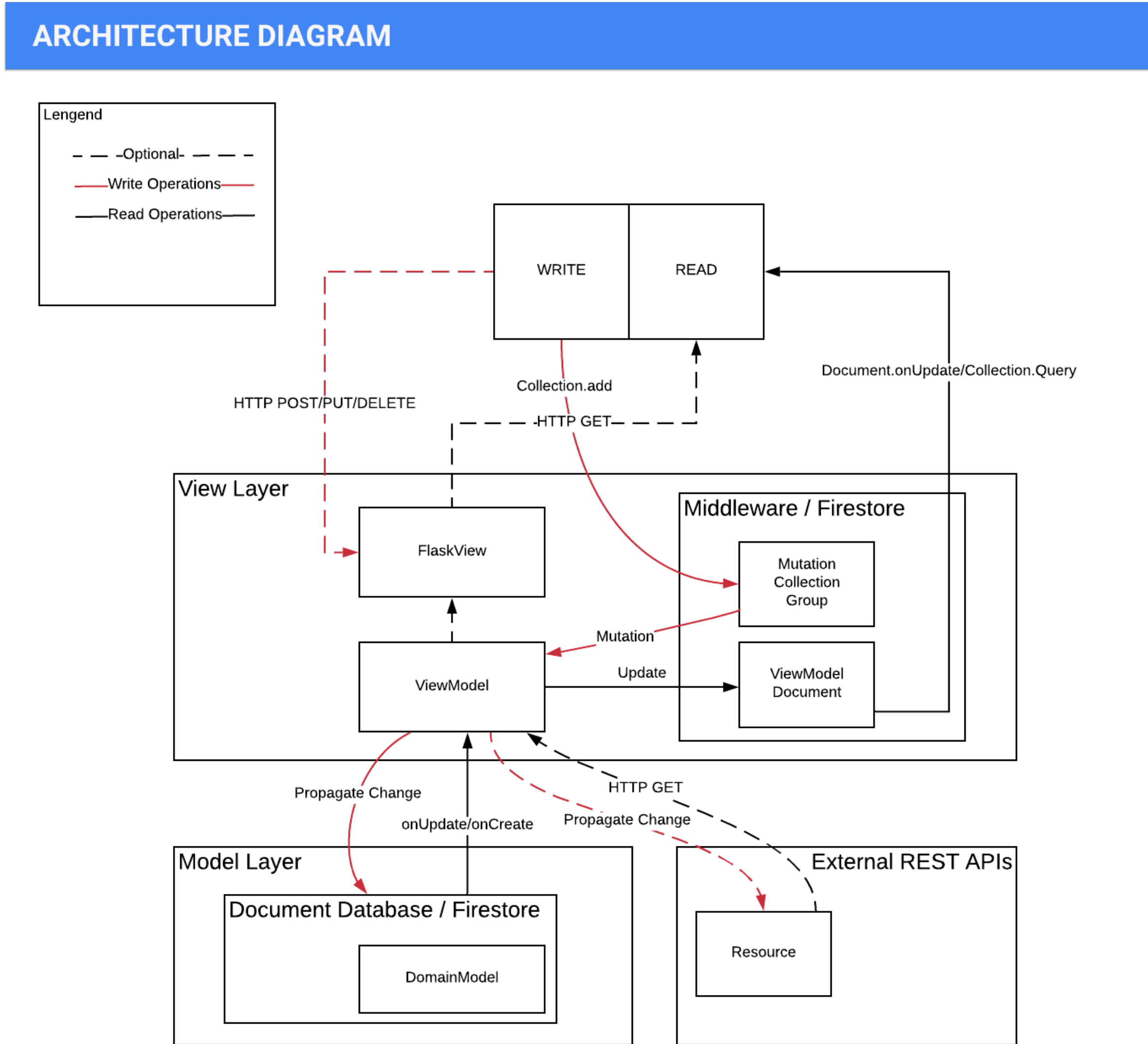
onto controls role-based access with security rules integrated with Firestore. REST API usually controls these access with a JWT token.

1.7.3 Redux

Redux is implemented mostly in front end. onto targets back end and is more scalable, since all data are communicated with Firestore, a infinitely scalable NoSQL datastore.

onto is declarative, and Redux is imperative. The design pattern of REDUX requires you to write functional programming in domain models, but onto favors a different approach: ViewModel reads and calculates data from domain models and exposes the attribute as a property getter. (When writing to DomainModel, the view model changes domain model and exposes the operation as a property setter). Nevertheless, you can still add function callbacks that are triggered after a domain model is updated, but this may introduce concurrency issues and is not perfectly supported due to the design tradeoff in onto.

1.7.4 Architecture Diagram:



Architecture Diagram

1.8 Contributing

Pull requests are welcome.

Please make sure to update tests as appropriate.

1.9 License

MIT

CHAPTER 2

Migrate from google.cloud.firestore

This document compares google-cloud-python-client firestore client library with flask-boiler ORM.

2.1 Add data

Original:

```
doc_ref = db.collection(u'users').document(u'alovelace')
doc_ref.set({
    u'first': u'Ada',
    u'last': u'Lovelace',
    u'born': 1815
})
```

New:

```
user = User.new(doc_id="alovelace", first='Ada')
user.last = 'Lovelace'
user.born = "1815"
user.save()
```

(Extra steps required to declare model. See quickstart for details.)

2.2 Read data

Original:

```
users_ref = db.collection(u'users')
docs = users_ref.stream()
```

(continues on next page)

(continued from previous page)

```
for doc in docs:
    print(u'{} => {}'.format(doc.id, doc.to_dict()))
```

New:

```
for user in User.all():
    print(user.to_dict())
```

2.3 Save data

Original:

```
class City(object):
    def __init__(self, name, state, country, capital=False, population=0,
                 regions=[]):
        self.name = name
        self.state = state
        self.country = country
        self.capital = capital
        self.population = population
        self.regions = regions

    @staticmethod
    def from_dict(source):
        # ...

    def to_dict(self):
        # ...

    def __repr__(self):
        return(
            u'City(name={}, country={}, population={}, capital={}, regions={})'
            .format(self.name, self.country, self.population, self.capital,
                    self.regions))

cities_ref = db.collection(u'cities')
cities_ref.document(u'SF').set(
    City(u'San Francisco', u'CA', u'USA', False, 860000,
         [u'west_coast', u'norcal']).to_dict())
#...
```

New:

```
def CityBase(DomainModel):
    _collection_name = "cities"

City = ClsFactory.create_customized(
        name="City",
        fieldnames=["name", "state", "country", "capital", "population", "regions"],
        auto_initialized=False,
        importable=False,
        exportable=True,
        additional_base=(CityBase,)
    )
```

(continues on next page)

(continued from previous page)

```
City.new(
        doc_id='SF',
        name='San Francisco',
        state='CA',
        country='USA',
        capital=False,
        populations=860000,
        regions=['west_coast', 'norcal']).save()

# ...
```

(fieldname kwarg in ClsFactory to be implemented soon)

2.4 Get data

Original:

```
doc_ref = db.collection(u'cities').document(u'SF')

try:
    doc = doc_ref.get()
    print(u'Document data: {}'.format(doc.to_dict()))
except google.cloud.exceptions.NotFound:
    print(u'No such document!')
```

New:

```
sf = City.get(doc_id='SF')
if sf is not None:  # To be implemented soon
    print(u'Document data: {}'.format(doc.to_dict()))
else:
    print("No such document")
```

2.5 Simple queries

Original:

```
docs = db.collection(u'cities').where(u'capital', u'==', True).stream()

for doc in docs:
    print(u'{} => {}'.format(doc.id, doc.to_dict()))
```

New:

```
for city in City.where(capital=True):
    print(city.to_dict())
```

2.6 Query operators

Original:

```
cities_ref = db.collection(u'cities')

cities_ref.where(u'state', u'==', u'CA')
cities_ref.where(u'population', u'<', 1000000)
cities_ref.where(u'name', u'>=', u'San Francisco'
```

with this,

```
City.where(state="CA")
City.where(City.population<1000000)
City.where(City.name>="San Francisco")
```

2.7 Declare Models

Method 1: onto.attrs

```
class City(DomainModel):
    city_name = attrs.bproperty()
    country = attrs.bproperty()
    capital = attrs.bproperty()

    class Meta:
        collection_name = "City"


class Municipality(City):
    pass


class StandardCity(City):
    city_state = attrs.bproperty()
    regions = attrs.bproperty()
```

Method 2: onto.mapper.schema

```
class CitySchema(Schema):
    city_name = fields.Raw()

    country = fields.Raw()
    capital = fields.Raw()


class MunicipalitySchema(CitySchema):
    pass


class StandardCitySchema(CitySchema):
    city_state = fields.Raw()
    regions = fields.Raw(many=True)

```

(continues on next page)

(continued from previous page)

```
class City(DomainModel):
    _collection_name = "City"
```

2.8 Field name conversion

Sometimes, you want to have object attributes in “snake_case” and Firestore Document field name in “camelCase”. This is by default for flask-boiler. You may customize this conversion also.

```
sf = StandardCity.create(doc_id="SF")
sf.city_name, sf.city_state, sf.country, sf.capital, sf.regions = \
    'San Francisco', 'CA', 'USA', False, ['west_coast', 'norcal']
sf.save()

la = StandardCity.create(doc_id="LA")
la.city_name, la.city_state, la.country, la.capital, la.regions = \
    'Los Angeles', 'CA', 'USA', False, ['west_coast', 'socal']
la.save()

dc = Municipality.create(doc_id="DC")
dc.city_name, dc.country, dc.capital = 'Washington D.C.', 'USA', True
dc.save()

tok = Municipality.create(doc_id="TOK")
tok.city_name, tok.country, tok.capital = 'Tokyo', 'Japan', True
tok.save()

beijing = Municipality.create(doc_id="BJ")
beijing.city_name, beijing.country, beijing.capital = \
    'Beijing', 'China', True
beijing.save()
```

object `la` saves to a document in firestore with “camelCase” field names,

```
{
    'cityName': 'Los Angeles',
    'cityState': 'CA',
    'country': 'USA',
    'capital': False,
    'regions': ['west_coast', 'socal'],
    'obj_type': "StandardCity",
    'doc_id': 'LA',
    'doc_ref': 'City/LA'
}
```

Similarly, you can query the objects with your local object attribute or firestore field name.

```
for obj in City.where(city_state="CA"):
    print(obj.city_name)
```

Or equivalently

```
# Currently broken
for obj in City.where("cityState", "==", "CA"):
    print(obj.city_name)
```

CHAPTER 3

Context Management

3.1 Auto Configuration with boiler.yaml

Provide authentication credentials to flask-boiler by moving the json certificate file to your project directory and specify the path in `boiler.yaml` in your current working directory.

```
app_name: "<Your Firebase App Name>"
debug: True
testing: True
certificate_filename: "<File Name of Certificate JSON>"
```

In `__init__` of your project source root:

```
from onto.context import Context as CTX

CTX.load()
```

3.2 Manual Configuration

In `__init__` of your project source root:

```
import os

from onto import context
from onto import config

Config = config.Config

testing_config = Config(app_name="your_app_name",
                        debug=True,
                        testing=True,
```

(continues on next page)

(continued from previous page)

```
                              certificate_path=os.path.curdir + "/../your_project/config_
↪jsons/your_certificate.json")

CTX = context.Context
CTX.read(testing_config)
```

Note that initializing `Config` with `certificate_path` is unstable and may be changed later.

In your project code,

```
from onto import context

CTX = context.Context

# Retrieves firestore database instance
CTX.db

# Retrieves firebase app instance
CTX.firebase_app
```

CHAPTER 4

Quickstart

This page is adapted from Quickstart using a server client library released under Creative Commons Attribution 4.0 License. Some code samples are also adapted from the source, which are released under the Apache 2.0 License. This page is part of a repository under MIT License, but does not override some licensing conditions of the Google's quickstart guide. Please refer to these license for more information.

4.1 Retrieve credentials

1. In the GCP Console, go to the **Create service account** key page.

   Go to Create Service Account Key Page

2. From the **Service account** list, select **New service account**.
3. In the **Service account name** field, enter a name.
4. From the **Role** list, select **Project > Owner**.

   > Note: The **Role** field authorizes your service account to access resources. You can view and change this field later by using the GCP Console. If you are developing a production app, specify more granular permissions than **Project > Owner**. For more information, see granting roles to service accounts.

5. Click Create. A JSON file that contains your key downloads to your computer.

4.2 Configure service account

Grant `roles/cloudfunctions.admin` role if you will use flask-boiler to deploy cloud functions. (Replace with your own service account information where applicable)

```
gcloud projects add-iam-policy-binding flask-boiler-testing --
↪member=serviceAccount:firebase-adminsdk-4m0ec@flask-boiler-testing.iam.
↪gserviceaccount.com --role roles/cloudfunctions.admin
```

Also run, You may need to run

```
gcloud iam service-accounts add-iam-policy-binding firebase-adminsdk-nztgj@gravitate-
↪backend-testing.iam.gserviceaccount.com --member=$MEMBER --role=roles/iam.
↪serviceAccountUser
```

(Effect unclear)

4.3 Add flask-boiler and the server client library to your app

Add the required dependencies and client libraries to your app.

In your project's requirements.txt,

```
# Append to requirements, unless repeating existing requirements

google-cloud-firestore
flask-boiler  # Not released to pypi yet
```

Configure virtual environment

```
pip install virtualenv
virtualenv env
source env/bin/activate
```

In your project directory,

`pip install -r requirements.txt`

4.4 Create a Document As View

In this example, we will build a mediator that forwards domain models (eg. `City/TOK`) to view models (eg. `cityView/TOK`). Both data models are stored in a NoSQL datastore, but only the view model is intended to be shown to the user. This example is similar to a stream converter, but you may build something more advanced by leveraging ViewModel.store to query multiple domain models across the datastore. The example is located in `examples/city`

4.4.1 Configure Project

Provide authentication credentials to flask-boiler by moving the json certificate file to your project directory and specify the path in `boiler.yaml` in your current working directory.

```
app_name: "<Your Firebase App Name>"
debug: True
testing: True
certificate_filename: "<File Name of Certificate JSON>"
```

In `__init__` of your project source root:

```
from onto.context import Context as CTX

CTX.load()
```

4.5 Declare a Domain Model

In `models.py`, create a model,

```
from onto.domain_model import DomainModel
from onto import attrs

class City(DomainModel):

    city_name = attrs.bproperty()
    country = attrs.bproperty()
    capital = attrs.bproperty()

    class Meta:
        collection_name = "City"
```

Create Attribute objects for your domain model. These will be converted to a Marshmallow Schema for serialization and deserialization.

```
class Municipality(City):
    pass


class StandardCity(City):
    city_state = attrs.bproperty()
    regions = attrs.bproperty()
```

You can create subclasses of `City`. By default, they will be stored in the same collection as `City`. Running a query on `City.where` will query all objects that are of subclass of `City`: `City, Municipality, StandardCity`. A query on `Municipality.where` will query all objects of subclass of `Municipality`: `Municipality`.

4.5.1 Declare View Model

Declare a subclass of `Store` first. This object helps you reference domain models by calling `self.store.<domain_model_name>`. In this example, you should initialize the store with a snapshot you may receive from the View Mediator.

```
class CityStore(Store):
    city = reference(many=False)
```

Next, declare a View Model. A View Model has attributes that converts inner data models to presentable data models for front end. The `doc_ref` attribute chooses where the view model will save to.

```
class CityView(ViewModel):

    name = attrs.bproperty()
    country = attrs.bproperty()

    @classmethod
    def new(cls, snapshot):
        store = CityStore()
        store.add_snapshot("city", dm_cls=City, snapshot=snapshot)
        store.refresh()
        return cls(store=store)
```

(continues on next page)

(continued from previous page)

```
    @name.getter
    def name(self):
        return self.store.city.city_name

    @country.getter
    def country(self):
        return self.store.city.country

    @property
    def doc_ref(self):
        return CTX.db.document(f"cityView/{self.store.city.doc_id}")
```

4.5.2 Declare Mediator Class

`Protocol.on_create` will be called every time a new document (domain model) is created in the `City/` collection. When you start the server, `on_create` will be invoked once for all existing documents.

```
class CityViewMediator(ViewMediatorDeltaDAV):

    def notify(self, obj):
        obj.save()

    class Protocol(ProtocolBase):

        @staticmethod
        def on_create(snapshot: DocumentSnapshot, mediator: ViewMediatorBase):
            view = CityView.new(snapshot=snapshot)
            mediator.notify(obj=view)
```

4.6 Add Entrypoint

In `main.py`,

```
city_view_mediator = CityViewMediator(
    query=City.get_query()
)

if __name__ == "__main__":
    city_view_mediator.start()
```

When you create a domain model in `City/TOK`,

```
obj = Municipality.new(
        doc_id="TOK", city_name='Tokyo', country='Japan', capital=True)
obj.save()
```

The framework will generate a view document in `cityView/TOK`,

```
{
    'doc_ref': 'cityView/TOK',
    'obj_type': 'CityView',
    'country': 'Japan',
```

(continues on next page)

(continued from previous page)

```
    'name': 'Tokyo'
}
```

Now, you have the basic app set up.

4.7 Create a form service

In this example, the user can post a form to `/users/<user_id>/cityForms/<city_id>` and create a new city.

4.7.1 Create Form Class

Declare a `CityForm` used for user to create a new city.

The function decorated with `city.init` will be called to initialize `city` attribute to a blank `City` Domain Model in the default location for property reads: `obj._attrs`. The fields of the blank Domain Model are set through the property setters. `propagate_change` will be called by the mediator to save the newly created city Domain Model to datastore.

```
class CityForm(ViewModel):

    name = attrs.bproperty()
    country = attrs.bproperty()
    city_id = attrs.bproperty()

    city = attrs.bproperty(initialize=True)

    @city.init
    def city(self):
        self._attrs.city = City.new(doc_id=self.doc_ref.id)

    @name.setter
    def name(self, val):
        self.city.city_name = val

    @country.setter
    def country(self, val):
        self.city.country = val

    def propagate_change(self):
        self.city.save()
```

4.7.2 Declare Form Mediator

```
class CityFormMediator(ViewMediatorDeltaDAV):

    def notify(self, obj):
        obj.propagate_change()

    class Protocol(ProtocolBase):
```

(continues on next page)

(continued from previous page)

```
        @staticmethod
        def on_create(snapshot: DocumentSnapshot, mediator: ViewMediatorBase):
            obj = CityForm.new(doc_ref=snapshot.reference)
            obj.update_vals(with_raw=snapshot.to_dict())
            mediator.notify(obj=obj)
```

4.7.3 Add Security Rule

In your Firestore console, add the following security rule:

```
    match /users/{userId}/{documents=**} {
      allow read, write: if request.auth.uid == userId
    }
```

This restricts the ability to post a city to only registered users.

4.7.4 Add Service

Now, your `main.py` should be,

```
city_view_mediator = CityViewMediator(
    query=City.get_query()
)

city_form_mediator = CityFormMediator(
    query=CTX.db.collection_group("cityForms")
)

if __name__ == "__main__":
    city_view_mediator.start()
    city_form_mediator.start()
```

When the user creates a document in `users/uid1/cityForms/LA`,

```
{
    'country': 'USA',
    'name': 'Los Angeles'
}
```

you should be able to receive the domain model in `City/LA`,

```
{
    'cityName': 'Los Angeles',
    'country': 'USA',
    'doc_id': 'LA',
    'doc_ref': 'City/LA',
    'obj_type': 'City'
}
```

and the view model in `cityView/LA`,

```

{
```

(continues on next page)

(continued from previous page)

```
    'name': 'Los Angeles',
    'country': 'USA',
    'doc_ref': 'cityView/LA',
    'obj_type': 'CityView',
}
```

This completes the setup for a simple CQRS read model / form set up for flask-boiler. The user may create new cities by posting a collection they own, and view cities by reading `cityView`.

CHAPTER 5

Features

Here are some reasons for choosing `flask-boiler` over the protocols and frameworks it is employing.

5.1 Native to Distributed Systems

5.1.1 Load Balancing

`flask-boiler` is designed to run on clusters such as `kubernetes`. By limiting the source range, you may route an event to its dedicated podwhich, for example, may already hold the states it requires.

5.1.2 Consistency

`flask-boiler` has timepoint check built into the system. You may avoid overwriting earlier changes made by other nodes by setting conditions on the sink.

5.2 Exchangable Components

5.2.1 Ease of Change

As your requirements evolve, for example, you may have a new group of users that uses your service less frequently. It would cause unnecessary amount of space to render and store MeetingSession. You may make minimal modifications to change the sink of the operation from NoSQL datastore to websocket, for example, a MeetingSession ViewModel is only refreshed when the user is active, and release space when the user logs out. The client can communicate with the server via WebSocket.

5.2.2 Abundance of Options

You may write different source/sink classes when you switch the database or services.

5.2.3 Scalability

As your requirements evolve, you may need a higher throughput, and flask-boiler's worker-queue based on multi-threading, and "pull every dependent for every push" may induce a higher cost. In this case, you can declare a mediator to be hosted as a Flink UDF (User Defined Function), and Flink will handle the invocation and logics for triggering. `flask-boiler` will compile/transform your code to be runnable as UDF.

5.3 Object-Oriented Code

5.3.1 Type hinting

- Write queries more easily and more accurately
- Refactor more easily

5.3.2 Easy to see the overall picture

The code to write with `flask-boiler` reflects the use case better. You may obtain inspirations on your business, since classes are domain-driven.

5.3.3 Better than Functional Reactive Programming (FRP)

If FRP is buttom-up, then `flask-boiler` is relatively top-down.

Example: suppose that you want to make an object MeetingSession View which requires an update to the number of people attending the meeting whenever a user changes their Ticket. Functional reactive programming would require you to modify an order when this happens. With `flask-boiler`, you would write how MeetingSession is composed of individual Tickets, and write logic on the side of MeetingSession.

5.4 N+1 Considerations

Consider the documents involved in a building a domain model or view model as a graph. We want to retrieve relevant documents when making an evaluation. This may lead to `N+1 problem` if not handled properly. We already know that most NoSQL usually does not support server-side join, or subqueries, so it is natural to fetch the dependent nodes (documents) to local storage. Even though this framework does all the fetching for you, you may still want to limit the reference between your data. For example, you want to avoid fetching every User document in your database when your User document has Friend's, and each User in Friend contains other User's. Beware that built-in verifications or warning systems have been implemented yet, but it is planned in the future.

5.4.1 Shallow Reference

We want to limit the Tree Height of the said graph. Make sure that your models reference up to finitely many layers (and fewer layers when applicable).

5.5 Product Roadmap

5.5.1 Future

Support SQL (Maybe)

Support Flink Stateful Functions

Support Flink: We want to offer a higher level of abstraction to Flink, rather than defining Schema's and operations with a pipelined API, we can define DomainModel's and ViewModel's that are object oriented.

- Statically compile mediator functions to flink operators
- Dynamically compile mediator functions to flink UDF
- Statically compile "Store" attrs and adapt to flink (use ast)

CHAPTER 6

flask-boiler

6.1 Domain Model

6.2 View Model

6.3 Flask REST API View Mediator

6.4 Document View Mediator

6.5 Query Delta View Mediator

6.6 Context

6.7 Auth

CHAPTER 7

Indices and tables

- genindex
- modindex
- search