Form3 Payments API

API overview

The following sections provide with a high level description of the API. For more detail, please refer to the OpenApi 3.0 schema located at api/openapi.yml.

Endpoints

	URL	Method	Description	Query parameters
1	/v1/payments/:id	GET	Retrieve an existing payment	
2		PUT	Update an existing payment.	version
3		DELETE	Delete an existing payment	version
4	/v1/payments	GET	Retrieve a collection of payments	from, size
5		POST	Create a payment	
6	/v1/health	GET	Return the service status	
7	/v1/metrics	GET	Return operational metrics	

Return codes

Code	Description	Usages
200	OK	1, 2, 4, 6, 7
201	Created	5
204	No Content	3
400	Bad Request	2, 3, 5
404	Not Found	1, 2, 3
409	Conflict	2, 3, 5
429	Too Many requests	All endpoints
500	Server Error	All endpoints

Content types

Mime	Usages
application/json	All but 7
text/plain	7

Architecture

General

The architecture described in this document follows **CQRS** principles, where write operations are decoupled from read operations. In order to keep the implementation simple, we **do not** implement Event Sourcing.

Components

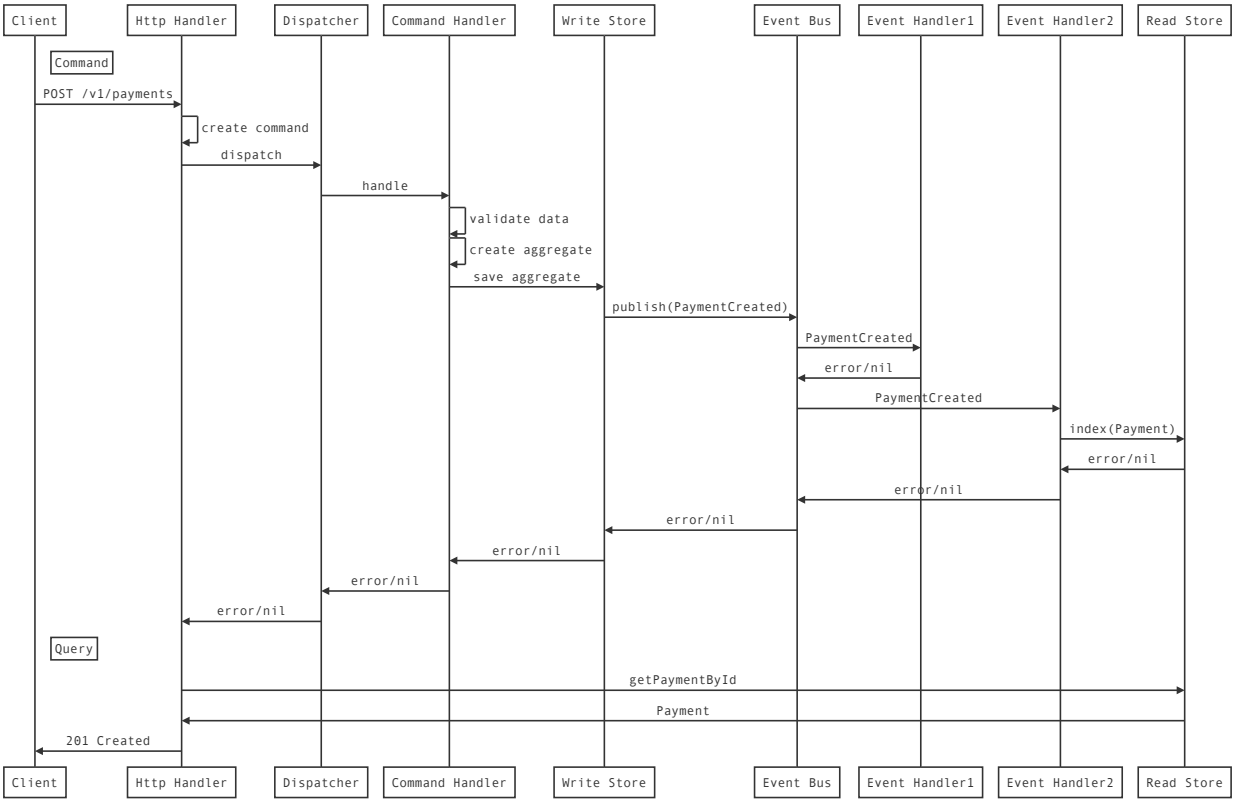
Our programming model involves the following concepts:

Component	Description
Client	A client application such as a web browser, curl program or client side sdk performing HTTP requests according to the API defined above.
Http Handler	A registered callback function that handles a HTTP request, and returns a HTTP response back to the client. The API is defined by existing go libraries (net/http, gin, etc.)
Command	A simple datastructure that describes the request being performed on the write model.
Command Dispatcher	Transports commands and routes them to command handlers
Command Handler	Reacts to an incoming command, and implements the logic necessary to modify and persist an aggregate. A single command will have only and only one Command Handler.
Event	A simple datastructure that describes something that took place.
Event Dispatcher	Transports events and routes them to event handlers
Event Handler	An event handler reacts to an incoming event, and implements a projection by updating the read model. A single event may trigger multiple event handlers
Store	A store for domain model instances. CommandHandlers may read from it. EvenHandlers may write into it.

Happy path callflow

The following diagram illustrates at a high level how a succesfull HTTP create request can be mapped to:

- A **command**, to create a new payment Aggregate and persist it into the Write store. The Payment aggregate produces the *PaymentCreated* event thats triggers a projection that indexes a view of the new payment into the Read Store.
- A **query**, to return a view of that payment, from the Read store.



Notes:

- The terms introduced here are just concepts. The implementation might then introduce extra types, depending on the third party software libraries used, but roughly the ideas should be the same.

Validation rules

Request data is encapsulated into commands. Commands are validated and processed by Command Handlers.

The following table summarizes the different validation rules will need to implement:

Type	Property	Validation
Payment	Id	Non empty
	Version	Positive
	Type	Equal to `Payment`
	Organisation	Non empty
	Attributes	Present and not empty (*)

(*) I am not doing further validations on the internal structure of the attributes payload, unless it is explicity stated.

Datatypes

The following table summarizes the main data types we will implement:

Golang Type	Property Names	Property Types
Payment	PaymentVersion	PaymentVersion
	Organisation	String
	Attributes	Interface{}
PaymentVersion	Id	String
	Version	Int

The following table summarizes the list of commands to be implemented. Each command will have a single handler associated to it.

Golang Type	Property Names	Property Types
CreatePayment	Payment	Payment
UpdatePayment	Payment	Payment
DeletePayment	PaymentVersion	PaymentVersion

The following table summarizes the list of events to be implemented:

Golang Type	Property Names	Property Types
PaymentCreated	PaymentVersion	PaymentVersion
	DateTime	Time
	Elapsed	Int
PaymentUpdated	PaymentVersion	PaymentVersion
	DateTime	Time
	Elapsed	Int
PaymentDeleted	PaymentVersion	PaymentVersion
	Datetime	Time
	Elapsed	Int
PaymentCreateError	PaymentVersion	PaymentVersion
	DateTime	Time
PaymentUpdateError	PaymentVersion	PaymentVersion
	DateTime	Time
PaymentDeleteError	PaymentVersion	PaymentVersion
	DateTime	Time

The following table summarizes the list of event handlers to be implemented:

Golang Type	Events listened to	Purpose
ListView	PaymentCreated, PaymentUpdated, PaymentDeleted	Read model projection
DetailView	PaymentCreated, PaymentUpdated, PaymentDeleted,	Read model projection
ErrorsLog	PaymentCreateError, PaymentUpdateError, PaymentDeleteError	Log errors

Limitations

Our design has the following important considerations/limitations:

- Comands and Events are dispatched, and handled, **synchronously**, in context with the http request. This confines the execution to the same Golang node (we sacrifice scalability) but provides with **strong consistency** semantics.

Testing

We use BDD in order to specify the functional behavior to be implemented by this service, and drive our development. Tests will be configured to use the in-memory store.

Authentication

We are not covering authentication/authorization. All requests are anonymous.

Logging

We use the standard logging facility provided by the Golang standard library. Logs are written to standard output so that they can be collected by Docker and then managed and aggregated by the plateform where this software will be deployed.

Persistence

Database schema

Our framework provides with a high level contract to be implemented by Store implementations. Here we provide a local, basic one based on Sqlite3 (memory and disk based).

The intial database schema for both the write and the read store will reduced to:

Table	Column	Type	Description
Payments	uuid	varchar(255)	
	version	Int	
	organisation	varchar(255)	
	attributes	Blob	
	created	Int	Unix Time, the number of seconds since 1970-01-01 00:00:00 UTC

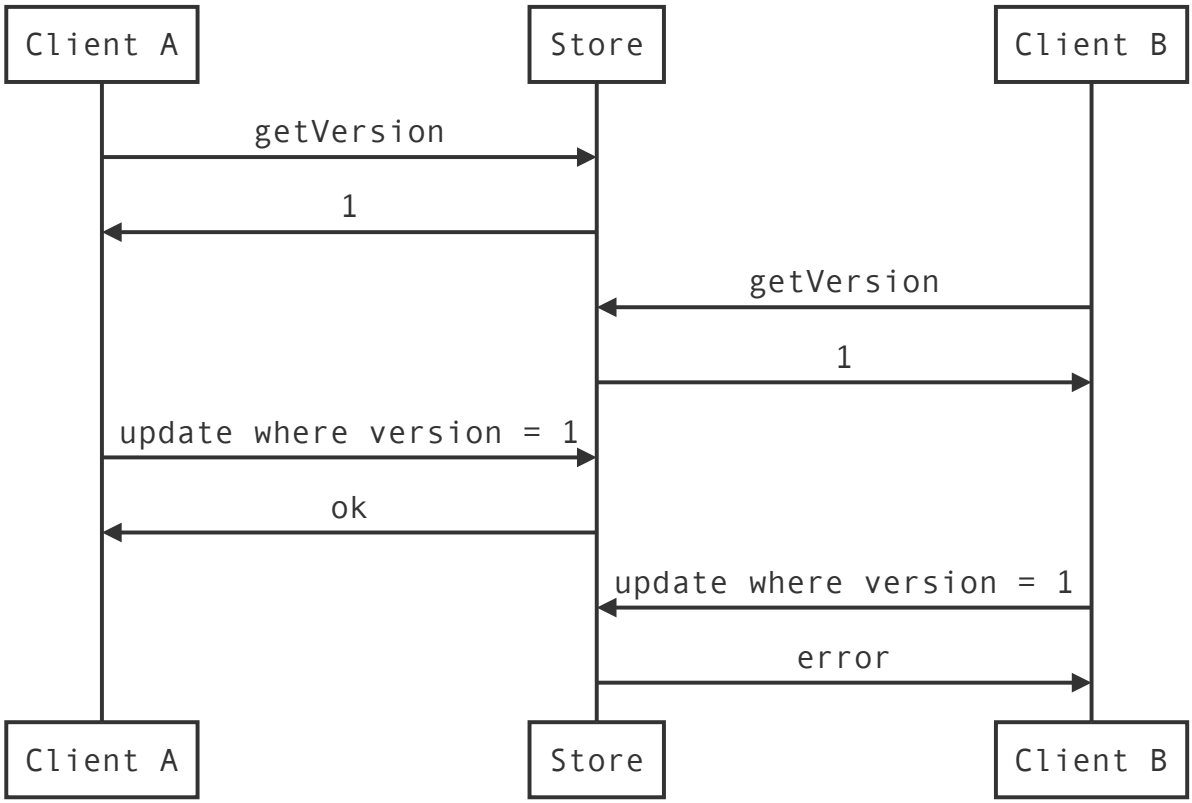
Index	Columns	Unique
Payments_UK	uuid, version	Yes

Notes:

- By using the standard Golang sql interface, we allow for compatibility with proper, distributed SQL RDMBS such as MySQL, Postgres, or CockroachDB in the future.
- It should be also possible to connect to NoSQL stores by implementing the provided Store interfaces and registering them accordingly.

Concurrency

We implement a basic optimistic locking scheme in order to support concurrent updates to the same payment:



In the diagram above:

- Clients A and B try to update the same document, concurrently, from different goroutines. They both fetch the current version in the store. In this example, they both obtain version equal to 1:

```
SELECT version FROM payments WHERE id = ?
```

- In order to update the document's data, they both increase their version number on their side, as well as the payload, then issue a transaction similar to:

```
UPDATE payments SET version=2, data=... WHERE id=? AND version=1
```

- From client A's perspective, one row was affected by the update and the operation is considered to be successful. We rely on the data store's **ACID semantics** in order to ensure one of the concurrent updates succeeds.
- From client B's perspective, by the time its update statement is applied, no version 2 exists for that document. The number of affected rows is zero, therefore the update is discarded and treated as an error. The error will be reported back to the client, who will need to retry (if pertinent) with the most up to date version from the read store.

Performance

Monitoring

On top of Golang's exported metrics by default, we will expose the following custom metrics, labelled by status (success, error):

Name	Type
form3_payments_created	Gauge
form3_payments_updated	Gauge
form3_payments_deleted	Gauge
form3_payments_create_elapsed_time_ms	Histogram
form3_payments_update_elapsed_time_ms	Histogram
form3_payments_delete_elapsed_time_ms	Histogram

Benchmarks

TODO ...

Scalability

The default implementation relies on in-memory or local based stores and dispatchers. This maximizes consistency, but does not scale horizontally.

Implementations based on distributed stores and messaging queues can be plugged in order to distribute load into several nodes. However this will introduce network latency and eventual consistency that will need to be accounted for in the design and mapped to/validated against business requirements and user experience.

Fault tolerance

In order to simply our error handling code and account for potential lack of service responsiveness (eg. remote database) or progressive performance degradation, we use Netflix's Hystrix latency and fault tolerance library.

Capacity

We use a simple and configurable mechanism in order to rate limit user http requests.

Configuration

We support configuration in YAML format. The following options with default values are supported:

Section	Setting	Default value	Description
http	host	""	The HTTP interface to listen on. Empty string means listen on all network interfaces
	port	8080	The HTTP server port to listen to
	tracing	False	Enable basic http request tracing
	apiVersion	"v1"	Api version for all routes

TODO...

Deployment

This project is packaged as a docker container. In order to run it with its default configuration, you can type:

```
docker run -p 8080:8080 pedrogutierrez/form3:latest
```

Thirparty libraries

The following table summarizes the libraries used in this project:

Name	Url	Description
Godog	https://github.com/DATA-DOG/godog	Cucumber for Golang
go.cqrs	https://github.com/jetbasrawi/go.cqrs	A Golang reference implementation of the CQRS pattern.
hystrix-go	https://github.com/afex/hystrix-go	Netflix's Hystrix latency and fault tolerance library, for Go
limiter	https://github.com/ulule/limiter	Dead simple rate limit middleware for Go.
go-sqlite3	https://github.com/mattn/go-sqlite3	sqlite3 driver for go using database/sql
sql-migrate	https://github.com/rubenv/sql-migrate	SQL schema migration tool for Go.
client_golang	https://github.com/prometheus/client_golang/promhttp	Prometheus instrumentation library for Go applications
validate	https://github.com/go-openapi/validate	openapi toolkit validation helpers
Go-config	https://github.com/micro/go-config	A dynamic config framework
httprouter	https://github.com/julienschmidt/httprouter	A high performance HTTP request router that scales well
assertions	https://github.com/smartystreets/assertions	Fluent assertion-style functions used by goconvey and gunit. Can also be used in any test or application.
Jsonpath	https://github.com/mdaverde/jsonpath	jsonpath golang library to help with getting and setting values on paths (even nonexistent paths)