Form3 Payments API

Getting started

Running the server

On your host

The makefile provides with two targets you can run in order to get dependencies and run the main program from your host machine:

```
make sqlite3
```

This will run the server at http://localhost:8080/ using an in-memory Sqlite3 store by default.

From Docker

There is a pre-packaged docker image you can run too:

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

This will download the docker image and start the the api server also against Sqlite3, by default.

Switching to Postgres

The provided `docker-compose.yml` file starts both postgres and our server in a single service:

```
docker-compose up
```

Running the tests

Once the server is running, you can easily run all BDD scenarios:

```
make bdd
```

Alternatively you can run only those BDD scenarios that are tagged with the `@wip` tag (useful during development):

```
make bdd-wip
```

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`.

Note:

- In the spec, the endpoints `/health` and `/metrics` are documented as part of the `v1` version, however in the implementation I finally decided to move them one level up (i.e. they don't have a version prefix).

Content types

All endpoints accept and return `application/json` content-type, except the `/metrics` endpoint, which only returns `text/plain`.

Application endpoints

	Path	Method	Description	Query parameters	Specific codes returned
1	/v1/payments/:id	GET	Retrieve an existing payment		200, 404, 500
2		PUT	Update an existing payment.		200, 404, 400, 409, 500
3		DELETE	Delete an existing payment	version	204, 404, 400, 409, 500
4	/v1/payments	GET	Retrieve a collection of payments	from, size	200, 400, 500
5		POST	Create a payment		201, 400, 409, 500

Admin endpoints

The admin endpoints are used in BDDs. They can be enabled/disabled using the `-admin` command line flag:

	Path	Method	Description
6	/admin/repo	GET	Get basic information about the payments repository
7	/admin/repo	DELETE	Delete all entries from the payments repository

Monitoring endpoints

	Path	Method	Description
8	/health	GET	Readiness probe
9	/metrics	GET	Prometheus metrics
10	/profiling/*		Runtime profiling data

Notes:

- Prometheus metrics can be enabled or disabled, via the `-metrics` command line flag
- Profiling data can be enabled or disabled via the `-profiling` command line flag

Return codes

The following table summarizes the HTTP status codes returned by the application:

Code	Description
200	OK
201	Created
204	No Content
400	Bad Request
404	Not Found
409	Conflict
429	Too Many requests
500	Server Error
503	Service unavailable

Architecture

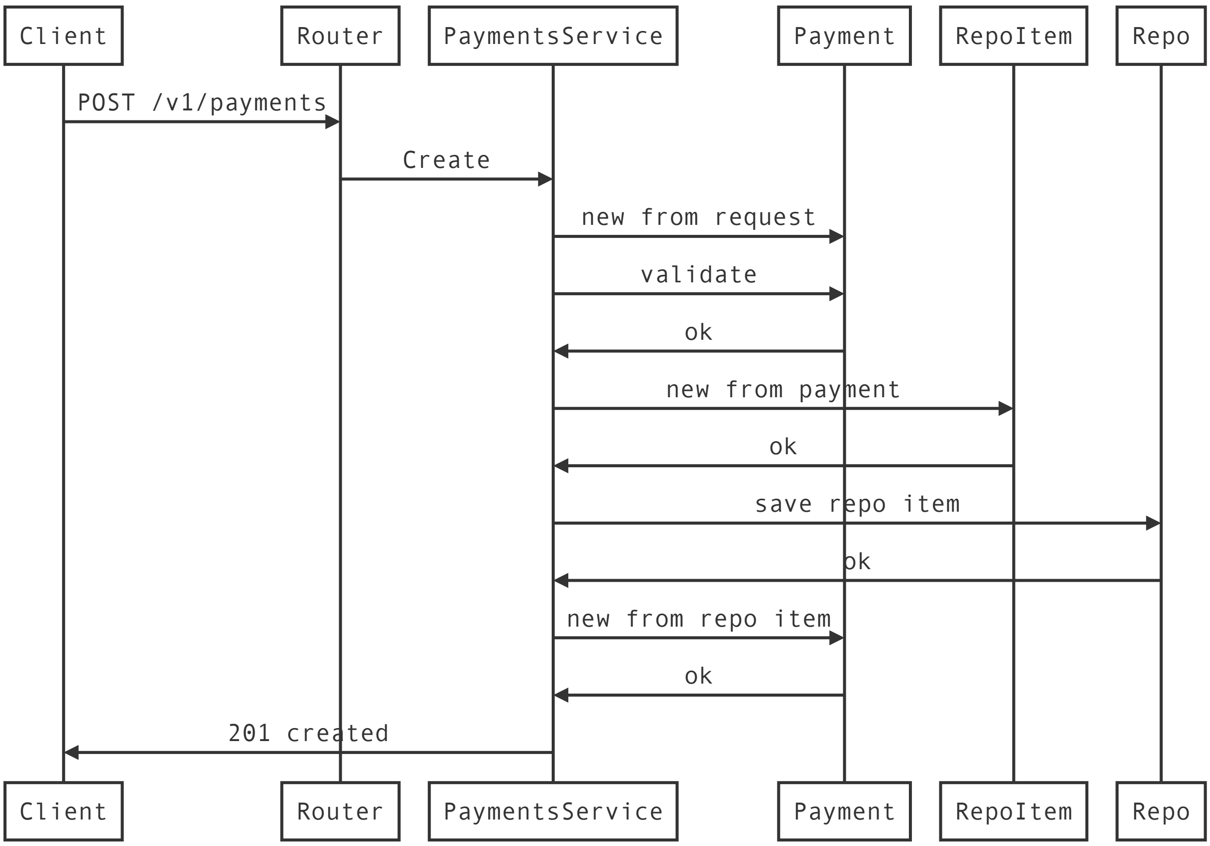
Overview

The application server follows a conventional layered architecture with a:

- A web layer, with the following features:
  - A pluggable, hierarchical **router** so that we can add new api versions in the future, and still keep backwards compatibility with old clients
  - A set of **middleware** that perform common concerns such as **logging**, **rate-limiting**, **supervision** against panics, content-type validation, cache management, response headers customizations, etc..
- A persistence layer, abstracted by the concept of a **Repo**, that decouples our application behaviour from a particular storage technology.

Request-Response cycle

The following diagrams depics a basic request-response cycle:



In this diagram:

- A HTTP POST request comes in, it is routed to the PaymentsService' `Create` handler function.
- The request body JSON payload is parsed and a new `Payment` struct is created
- The payment is validated.
- If success, the payment is converted into a `RepoItem`, in order to be saved into the `Repo`.
- If success, the RepoItem is converted back into a Payment and sent back to the client as JSON payload.

Data model

In this version, we manage a very simple data model, in which a Payment has the following properties:

Property	Type	Constraints
Id	String	Globally unique, non-empty
Version	Int	Positive integer
Type	String	Constant, hardcoded to `Payment`
Organisation	String	Non-empty
Attributes	PaymentAttributes	Non-null

The PaymentAttributes type defines the additional data we manage about a payment:

Property	Type	Constraints
Amount	String	Must represent a number strictly greater than zero

Notes:

- I am **intentionally skipping** any other validations or parsing on the internal structure of the attributes payload.
- Implementation details are in package `gitHub.com/pedro-gutierrez/form3/pkg/payments`.

Authentication

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

Admin endpoints are also anonymous, however they can be disabled via the `-admin` command line flag.

Logging

Package `github.com/pedro-gutierrez/form3/pkg/logger` introduces a structured logger that outputs JSON formatted log entries to the standard output. This way:

- We support distributed, elastic deployments, such as Kubernetes, where we could have many replicas of our service running side to side. We leave the orchestrator the task to collect logs from all pods and centralize their management.
- By being JSON, it will be easier to consume log entries and aggregate them, using tools such as Elasticsearch/Kibana.

Testing

We use BDDs in order to specify the functional behavior to be implemented by this service, and drive our development.

Package `github.com/pedro-gutierrez/form3/pkg/test` provides with:

- A customized HTTP client, specifically designed to execute HTTP requests and capture reponses and errors, for further assertions in step definitions.
- A shared scenario context, defined by the concept of `World`.
- A fluent expectation and assertions api, that relies on `smartystreets/assertions` and `mdaverde/jsonpath`.
- A rich collection of compact, composable step definitions, designed so they can be easily reused, in order to design more advanced and refined feature scenarios quickly, and with very little extra coding effort.

Persistence

Abstract API

We define the abstract concept of a `Repo` that manages `RepoItems`. This way we abstract our Web layer from the actual persistence tecnlogy. A Repo defines basic CRUD operations on RepoItems.

We then define an abstract **SQLRepo**, which relies on the standard sql Go package.

We then provide two implementations:

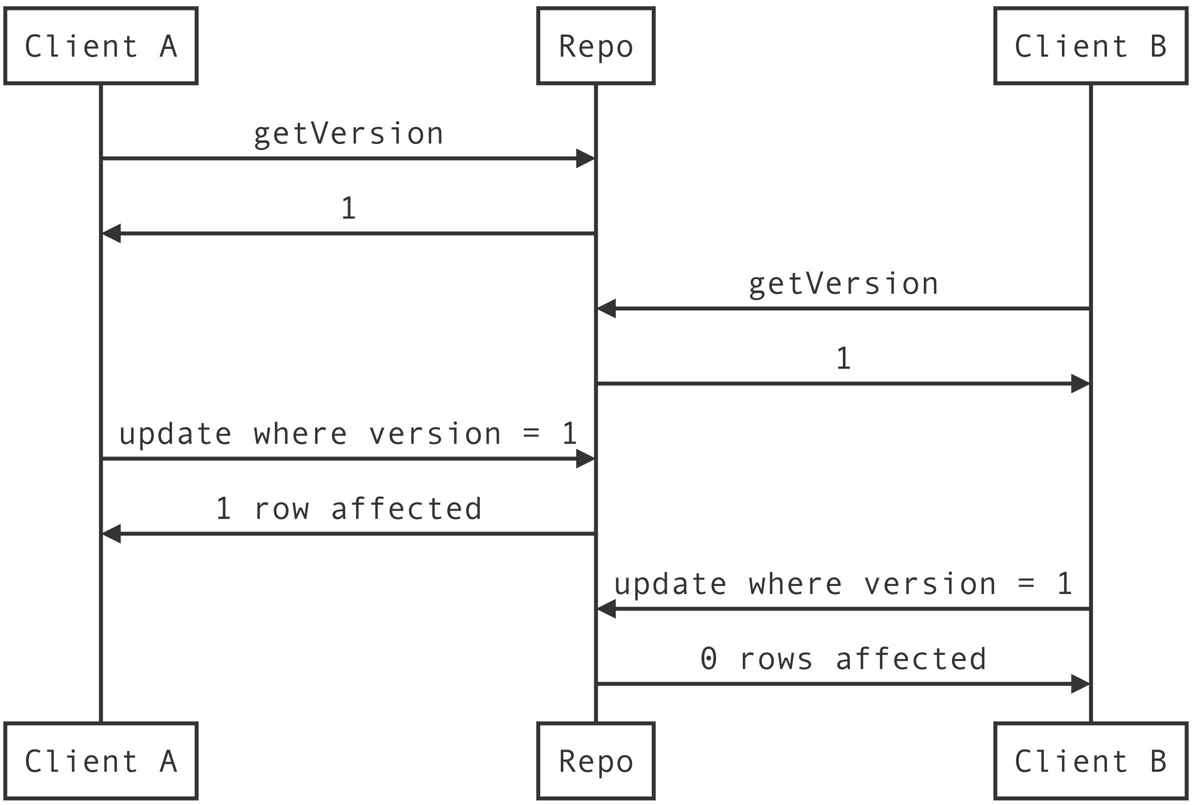
- **Sqlite3Repo**, with both memory and file-based backends.
- **PostgresRepo**

With this design, it is easy to switch, out of the box, from Sqlite3 to Postgres (please see the `–repo-xxx` command line flags and the provided Makefile for more info and usage examples).

It should straightforward to extend the system with alternative NoSQL implementations (eg. MongoRepo, RedisRepo).

Concurrency

In the **SQLRepo**, 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:
- 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:
- 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, version 1 no longer exists. 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 fetch the most recent version of the payment and retry the update.

Monitoring

We provide the ability to turn on, and expose Prometheus based metrics. This will give useful information about Go's runtime performance and also will give HTTP request/reponse statistics (method, paths, return codes, etc..).

Resiliency

- We provide a simple liveliness probe that checks the connectivity to the repo and returns a `503 Service unavailable` status code as soon it can no longer be reached. This should instruct the orchestrator (eg. Kubernetes) to stop sending traffic to the offending pod or even to shut it down if necessary.
- Our SQLRepo implementation relies on Go's standard sql package. This allows us to rely on the default connection pooling and automatically recover from connection loss from the database.

Scalability

Vertical

We rely on Go's net/http package which leverages go routines in order to handle HTTP requests. This should maximize usage of all available cores.

Horizontal

The application server is stateless, so it should be very straightforward to add more replicas to the service using a Kubernetes deployment resource.

The database itself has state. With the current implementation, can easily scale the repo (we can go from a local Sqlite3, to a full Postgres distributed cluster), as long as we keep the same **strong consistency** semantics.

Fault tolerance

Mechanisms in place

- All errors resulting from our interaction with the database are caught and logged in the web layer, using our structure logger.
- Panics are recovered by Chi's standard Recoverer middleware.
- Long requests will timeout according to a configurable settings (`–timeout` command line flag)

Mechanisms not yet in place

- Using a circuit breaker, such as Netflix's Hystrix will certainly let us fail fast, in case the database becomes a bottleneck and starts to repond slowly. This could be added to our microservice or we can also manage this a higher level using a service mesh architecture.

Capacity

Resource limits such as CPU, memory can be set on the Docker containers. Also, using a service mesh architecture can help control traffic between services.

For simplicity, we include `ulule/limiter` which makes it easy to implement a rate limit mechanism, by configuration (see the `–liimit` command line flag).

Once the configured rate is reached, a 429 status code will be returned.

Configuration

Our microservice only supports configuration via standard Go command line flags.

The following settings are supported:

```
  -admin
        enable admin endpoints
  -api-version string
        api version to expose our services at (default "v1")
  -compress
        gzip responses
  -cors
        enable cors
  -external-url string
        url to access our microservice from the outside (default
"http://localhost:8080")
```

```
  -limit string
        rate limit (eg. 5-S for 5 reqs/second)
  -listen string
        the http interface to listen at (default ":8080")
  -max-results int
        Maximum number of results when listing items (eg. payments) (default 
20)
  -metrics
        expose prometheus metrics
  -profiling
        enable profiling
  -repo string
        type of persistence repository to use, eg. sqlite3, postgres (default 
"sqlite3")
  -repo-migrations string
        path to database migrations (default "./schema")
  -repo-schema-payments string
        the table or schema where we store payments (default "payments")
  -repo-uri string
        repo specific connection string
  -timeout int
        request timeout (default 60)
```

Thirparty libraries

The following table summarizes the main third-party libraries used in this project:

Url	Description
https://github.com/DATA-DOG/godog	Cucumber for Golang
https://github.com/ulule/limiter	Dead simple rate limit middleware for Go.
https://github.com/mattn/go-sqlite3	sqlite3 driver for go using database/sql
https://github.com/rubenv/sql-migrate	SQL schema migration tool for Go.
https://github.com/prometheus/client_golang/promhttp	Prometheus instrumentation library for Go applications
https://github.com/smartystreets/assertions	Fluent assertion-style functions used by goconvey and gunit. Can also be used in any test or application.
https://github.com/mdaverde/jsonpath	jsonpath golang library to help with getting and setting values on paths (even nonexistent paths)
https://github.com/go-chi/chi	lightweight, idiomatic and composable router for building Go HTTP services
https://github.com/pkg/errors	Simple error handling primitives
https://github.com/lib/pq	Pure Go Postgres driver for database/sql
https://github.com/ddliu/go-httpclient	Advanced HTTP client for golang