Summary

Design and build a CRUD system to receive payments. The format of the messages is given and the following actions must be implemented:

- Create a new payment
- Show a payment
- Update a payment
- Delete a payment
- List all payments

Implementations details

Application packages

Database

Package responsible for accessing the database. Should provide an abstraction over the selected DB to allow changing to another DB in the future.

HTTP api

Package responsible for providing http interface. Each resource (payments, users, etc) should have it's own subpackage for easy maintainability .

Also we want to version it. There are multiple ways:

- Providing a http header to target the desired version of the api.
- Using the beginning of the path like, "/v1/payments".
- Using different subdomains like "v1.domain.com"

For simplicity I have chosen to version the api with the path of the url. So we can access different versions in this way:

/v1/payments

/v2/payments

Each version will be in it's own package, and the package must provide everything needed to mount that resource in the overall schema.

Selected database

As a payment provider platform we want to receive information for several clients and each one much have it's own format to encode the information. A NoSQL database seems to be a good option because:

- As it don't enforces a fixed schema we can easily receive information with formats. A minimum common schema might be needed, but we can allow schema-free fields.
- Usually NoSQL databases scales horizontally better than SQL databases. If aim to integrate with big banks for sure they will have huge volumes of data.
- If this were to be a real project I would have used some SAAS database from amazon or google, like amazon DynamoDB

Endpoints

The endpoints to implement following REST rules are (already including api versioning)):

GET /v1/payments - List all payments

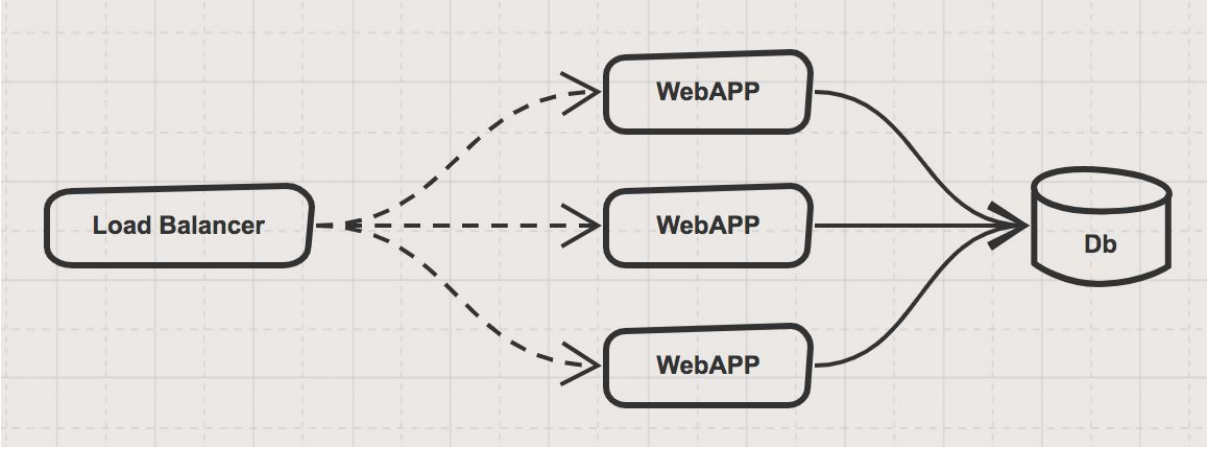
POST /v1/payments - Create a new payment

GET /v1/payments/<some-id> - Show a payment

PUT /v1/payments/<some-id> - Update a payment

DELETE /v1/payments/<some-id> Delete a payment

Architecture



As the web application will be stateless we can deploy more than one to ensure high availability (although as we are using go it shouldn't be a problem initially due to each http request running on it's own gorouting).

To balance the http requests between the different instances we need a load balancer like nginx, haproxy or if we migrate into kubernetes we can rely on the already present ingress.

Setting resource IDs

As I propose to use mongodb as DB it's important to take a look on how to uniquely identify each document. We can't rely on the uniqueness of some field coming on the payload as different clients might use different way to identify a resource, they might be even using a compound identifier. Therefore we have to create our own identifier. MongoDB don't provide any auto increment key feature out of the box, I will be using a UUID generator.

The generation of the resource identifier is explained in the following sequence diagram: