Payment Server Design

Persistence

I chose to use Mongo for the persistence layer because I needed a schemaless solution to persist Json documents.

If I had requirements regarding how the database would be used in the future (transactional workloads or analytics for example) or more information regarding the data model itself (many-to-one and/or many-to-many relations) maybe another solution could have been a better fit. Yet in this context, Mongo is a good choice (in my humble opinion :).

Project Structure

The Mongo client (db.go) belongs to the `payment` package. This might be subject to discussions (compared to having it in a dedicated `mongo` or `db` package for example).

I chose this design because if the application evolves and that we need to handle new resources, it is not mandatory to have them persisted in the very same database than the one used for payments.

Moreover, having self-contained packages per resource can also be the first step towards a micro-services architecture. A modular application is easier to migrate to an autonomous micro-service.

HTTPS

The application provides only an HTTP endpoint as in modern architectures, it's not part of the application anymore to handle HTTPS.

The TLS termination can be done by a load-balancer or by a service mesh deployed as a sidecar.

Criticism

In the real world, I would maybe not create a direct coupling between the server and the database itself.

For example, with `POST` and `PUT` requests I would rather choose a design where the server publishes requests to a message broker and returns `202 Accepted` responses.

Having a message broker in our context can be a good way to buffer incoming requests and smooth eventual throughput peaks.

Something also worth mentioning, if the application is used by different consumer types (mobile, web application etc.) we may have to deal at some point with concurrent requests. Meanwhile, I saw that the `version` was an attribute of the `Payment` object to provide (I assume) an optimistic concurrency control.

In this regard, we may want to guarantee the ordering of the `POST` and `PUT` requests per payment requester. This has to be achieved by the load balancer in front of our application (using sticky sessions for example). Moreover, the message broker must also provide such guarantees. For example, Kafka could provide it if the payment topic was partitioned by the requester identifier. The only caveat would be to use a consistent hashing algorithm if we want to keep ordering per requester while scaling our application.

Well, let's not dig too much into the details here...

Conclusion

In any case, that was a cool exercise! Thanks for your time :)