Payments API design

Goals

- API should be able to:
  - Fetch a payment resource
  - Create, update and delete a payment resource
  - List a collection of payment resources
  - Persist resource state (e.g. to a database)

Use cases

- Our internal system lists all Payments (can filter by Payment properties)
- Our internal system fetches a Payment
- Our internal system creates a Payment
- Our internal system updates a Payment
- Our internal system deletes a Payment

Scope of the development

Covered here

- Payment routes only
- all API routes are private (not accessible to external users yet, only to our internal systems)
- database migrations for the minimal resource types that we need in order to have a payments system (Payments, Currencies, Parties, Banks)
- tests to be run locally, packaged in docker containers and using the same database as in production, with the same database models and constraints

Not covered

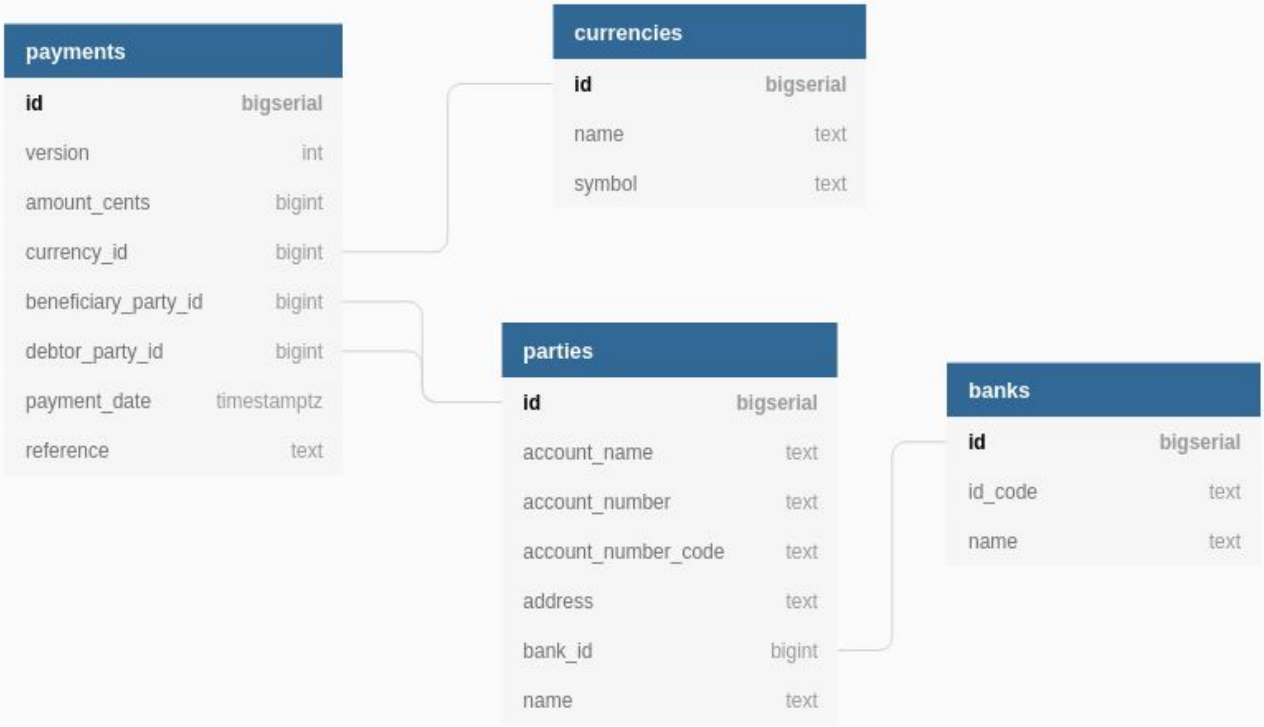
These are features we won't add in this exercise, but that would normally be part of a production API:

- Normally other types like Currencies, Parties, Banks would have associated API routes to
- Compared to the provided api mock response, the following fields are missing from the Payment resource definition:
    - fx
    - payment schemes
    - payment types
    - sponsor parties
- On a production API, if we expose the API directly to external users, we would need to restrict access for each user to their data. Which would mean implementing authentication and authorizations. We're not yet implementing those parts in the scope of this exercise.
- Even if we don't expose the API to our external users directly, we'd still need a way to make sure it's our internal system doing all calls (authorizing every call and restricting callers to an internal network). For this exercise we're assuming this will be done outside of the application code, at deploy time
- Tests are runnable locally, but for a production setting we'd have the tests run automatically in CI/CD pipelines. The pipeline definitions are not covered here, but the tests are packaged to be run in docker containers in order to make it easier to run as part of CI/CD pipelines if we deployed this API to production.

DB models

The DB models are somewhat simplified compared to the mock API response that was provided, in order to focus on the strict essential features that seem needed in order to design a basic payments API:

Payments, Parties, Currencies, Banks



holistics

API routes

- Create a Payment: POST /payments
- List Payments: GET /payments
- Get a Payment's details: GET /payments/{id}
- Update (part or all of) a Payment: PATCH /payments/{id}
- Delete a Payment: DELETE /payments/{id}

OpenAPI definition

The OpenAPI definition can be copy-pasted in the preview tool of your choice or viewed live at :

https://kata-payments-api.docs.stoplight.io/


```
{"openapi":"3.0.2","info":{"title":"kata-payments-api","description":"kata-payments-api","version":"1.0.0"},"paths":{"/v1":{},"/v1/payments":{"get":{"summary":"listPayments","description":"listPayments","operationId":"listPayments","responses":{"200":{"description":"Success","content":{"application/json":{"schema":{"type":"array","items":{"$ref":"#/components/schemas/payment"}},"examples":{"payments":{"value":[{"amount_cents":84,"currency_id":17,"beneficiary_party_id":32,"debtor_party_id":61,"payment_date":"2018-02-10T09:30Z","reference":"some
text"},{"amount_cents":60,"currency_id":61,"beneficiary_party_id":34,"debtor_party_id":29,"payment_date":"2018-02-10T09:30Z","reference":"some
text"}]}}}}}}},"post":{"summary":"createPayment","description":"createPayment","operationId":"createPayment","requestBody":{"content":{"application/json":{"schema":{"$ref":"#/components/schemas/payment"},"examples":{"payment":{"value":{"amount_cents":37,"currency_id":36,"beneficiary_party_id":63,"debtor_party_id":82}}}}},"required":true},"responses":{"201":{"description":"Success","content":{"application/json":{"schema":{"$ref":"#/components/schemas/payment"}}}}}}},"/v1/payments/{id}":{"get":{"summary":"getPayment","description":"getPayment","operationId":"getPayment","parameters":[{"name":"id","in":"path","description":"Payment
ID","required":true,"schema":{"type":"integer"}}],"responses":{"200":{"description":"Success","content":{"application/json":{"schema":{"$ref":"#/components/schemas/payment"},"examples":{"payment":{"value":{"amount_cents":10,"currency_id":31,"beneficiary_party_id":90,"debtor_party_id":83,"payment_date":"2018-02-10T09:30Z","reference":"some
text"}}}}}}}},"delete":{"summary":"deletePayment","description":"deletePayment","operationId":"deletePayment","parameters":[{"name":"id","in":"path","description":"PaymentI
ID","required":true,"schema":{"type":"integer"}}],"responses":{"200":{"description":"Success"}}},"patch":{"summary":"updatePayment","description":"updatePayment","operationId":"updatePayment","parameters":[{"name":"id","in":"path","description":"Payment
ID","required":true,"schema":{"type":"integer"}}],"responses":{"202":{"description":"Success","content":{"application/json":{"schema":{"$ref":"#/compo
```

```
nents/schemas/payment"},"examples":{"payment":{"value":{"amount_cents":2,"currency_id":86,"beneficiary_party_id":3,"debtor_party_id":68,"payment_date":"2018-02-10T09:30Z","reference":"some text"}}}}}}}}},"components":{"schemas":{"payment":{"title":"Root Type for payment","description":"payment","type":"object","properties":{"amount_cents":{"format":"int32","type":"integer"},"currency_id":{"format":"int32","type":"integer"},"beneficiary_party_id":{"format":"int32","type":"integer"},"debtor_party_id":{"format":"int32","type":"integer"},"payment_date":{"format":"date-time","type":"string"},"reference":{"type":"string"},"currency":{"$ref":"#/components/schemas/currency"},"beneficiary_party":{"$ref":"#/components/schemas/party"},"debtor_party":{"$ref":"#/components/schemas/party"}},"example":{"amount_cents":63,"currency_id":69,"beneficiary_party_id":33,"debtor_party_id":15,"payment_date":"2018-02-10T09:30Z","reference":"some text","currency":{"id":6,"name":"some text","symbol":"some text"},"beneficiary_party":{"account_name":"some text","account_number":"some text","account_number_code":"some text","address":"some text","bank_id":"some text","bank_id_code":"some text","name":"some text"},"debtor_party":{"account_name":"some text","account_number":"some text","account_number_code":"some text","address":"some text","bank_id":"some text","bank_id_code":"some text","name":"some text"}}},"currency":{"title":"Root Type for currency","description":"The root of the currency type's schema.","type":"object","properties":{"id":{"format":"int32","type":"integer"},"name":{"type":"string"},"symbol":{"type":"string"}},"example":"{\n      \"id\": 876,\n    \"name\": \"Euro\",\n   \"symbol\": \"EUR\"\n}"},"party":{"title":"Root Type for party","description":"The root of the party type's schema.","type":"object","properties":{"account_name":{"type":"string"},"account_number":{"type":"string"},"account_number_code":{"type":"string"},"address":{"type":"string"},"bank_id":{"type":"string"},"bank_id_code":{"type":"string"},"name":{"type":"string"}},"example":"{\n   \"account_name\": \"EJ Brown Black\",\n  \"account_number\": \"GB29XABC10161234567801\",\n \"account_number_code\": \"IBAN\",\n      \"address\": \"10 Debtor Crescent Sourcetown NE1\",\n     \"bank_id\": \"203301\",\n    \"bank_id_code\": \"GBDSC\",\n      \"name\": \"Emelia Jane Brown\"\n}"},"bank":{"title":"Root Type for bank","description":"The root of the bank type's schema.","type":"object","properties":{"id":{"format":"int32","type":"integer"},"name":{"type":"string"}},"example":"{\n   \"id\": 45678,\n  \"name\": \"International Bank\"\n}"}}}}
```

Running the project

Usage

Before running the tests or the app, run `make gen` to generated the models code. (see ##How-this-works)

Run `make down up dev` to start server with docker (requires: docker, docker-compose).

(alternative: run `go run main.go` (requires go 1.11+, postgres installed locally and tweaking config files))

The API is accessible at http://localhost:36480

Requirements

- Go 1.11+
- docker
- docker-compose

Testing

Run tests with `make test`

How this works

You'll notice models are entirely absent from this repo. That's because we'll get them using code generation. Models (and their tests) are generated from sql migrations applied to the database, which means:

- code generation must happen in docker since the database runs in docker
- the generated code gets written back to the host via a docker volume
- a second `docker build` is necessary to compile the Go app including the newly generated code

So, before you can run `make test` or `make up dev`, you have to run `make gen` once after cloning the project. If you change the database migrations folder, you'll have to run `make gen` again to update the generated models.

Generated models are not committed to source control.