Payments API design

Platform Level Design (what other services should expect from the API)

Transport: HTTP

Most, if not all, clients will be familiar with HTTP allowing them to integrate with a minimum amount of friction. There are plenty of libraries and frameworks available for both clients and servers allowing for rapid development.

Trade-offs: Not the fastest protocol, but one almost every one of our users should be familiar with. When security becomes an issue, upgrading to HTTPS should be relatively straightforward.

Alternatives: Raw TCP sockets, RabbitMQ, ZeroMQ

Payload data format: JSON, encoded as UTF-8

Again, most, if not all, of our clients will be familiar with JSON and there are plenty of serialisation and deserialisation libraries in many languages available. It also has a compact and human readable form allowing rapid understanding and development. UTF-8 is required (as per the RFC) for open systems. Using UTF-8 will allow maximum compatibility with clients.

Trade-offs: Not a descriptive as XML or as extensible as EDN. Not as compressed as a binary format, somewhat mitigated by HTTP compression.

Alternatives: XML, EDN, Protocol Buffers

While the API should accept and publish JSON, this should not necessarily be the default choice for internal serialisation. It should be simple to extend the API to accept and publish other data formats.

Architecture

At a minimum a RESTful architecture should be implemented. A more complete approach imposes the HATEOAS constraint, which should be applied by default. Document any areas where this is not possible. See Appendix A for the Swagger specification.

To keep things simple, only full resource replacement should be implemented at this time.

Simultaneous updates to the same payment should be gracefully handled, by rejecting updates based on versions older than the current one. Etag and If-Match headers should be helpful here. The API should be synchronous to start with as this gives the easiest end user experience.

If the data fails to be stored this should result in an error, rather than aiming for eventual consistency. At this stage not losing data is preferable to performance.

Payment data structure

Since there's no processing to be done on the payment data structure itself and no schema available, all that's required is a globally unique ID.

```
{
  "$id": "http://example.com/example.json",
  "type": "object",
  "properties": {
    "id": {
      "$id": "/properties/id",
      "type": "string",
      "title": "ID of the payment",
      "default": "",
      "examples": [
        "e7119659-f1bc-4a84-8cd2-c892f2c56e22"
      ]
    }
  },
  "required": [
    "id"
  ]
}
```

This does mean the API is very liberal with what it accepts, but I think that that is better than locking it down too much. Determining whether we will need to process the payments and getting a better idea of what the shape of that data we can expect should be top of our priority list.

Authentication and Authorisation

Deferred for the time being until the API has been verified as meeting the customers needs and the service is definitely being deployed to production.

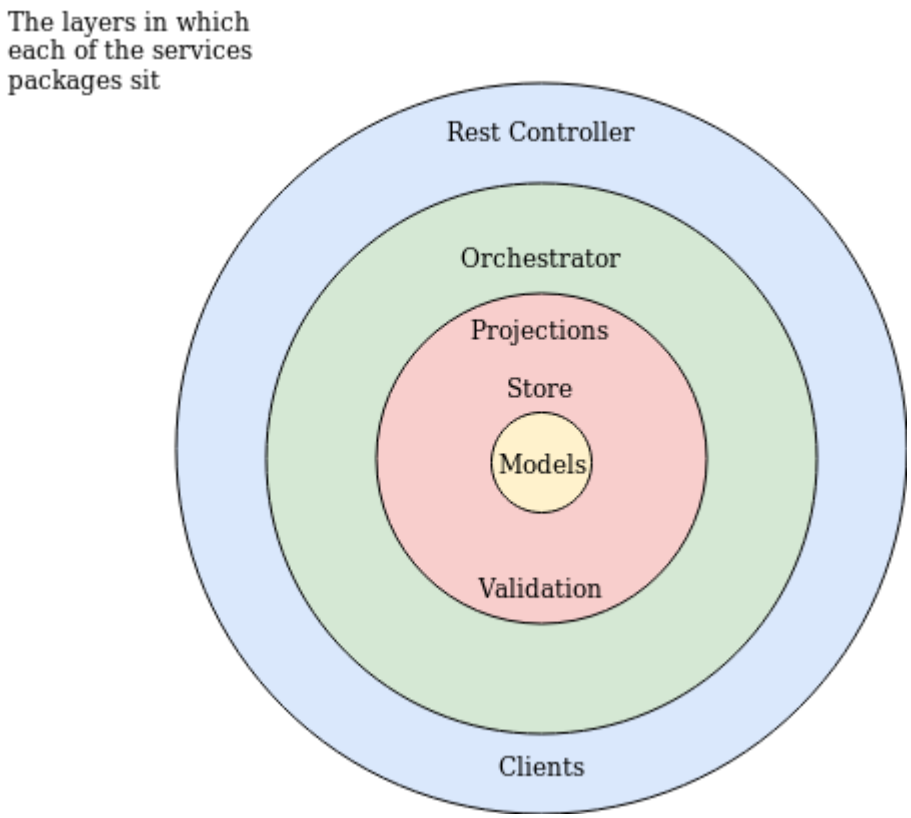
Performance

Although we don't yet have performance requirements, keep performance in mind where it minimally impacts on extensibility, robustness and correctness.

Internal Service Design (driven out through the use of TDD/BDD)

Clean Architecture

The architecture of the service is based on Bob Martin's Clean Architecture. The defining features in this case are the storage client and web server being on the outside of the design while the models and application logic are towards the centre. This has allowed an approach to automated testing that is behaviour focused and requires minimal mocking.



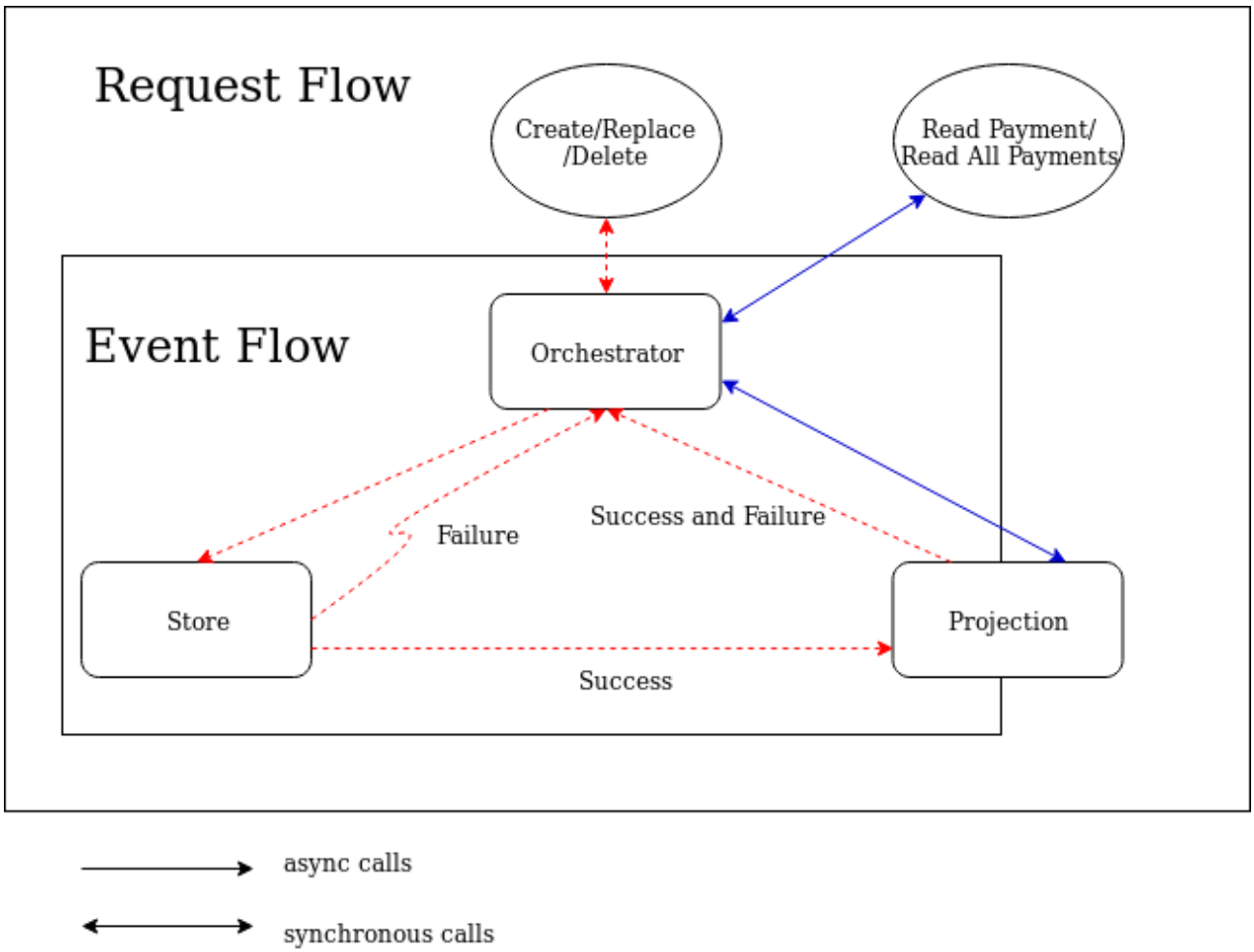
Event Drive Architecture

Each of the mutating actions (create/replace/delete) creates an event. These events are stored in an append only data store before being passed to the projection via a concurrent blocking queue. A single worker within the projection takes events from the queue and updates its internal model which is exposed as an immutable data structure. This data structure can then be queried by the safe actions (read payment/read all payments).

Each event contains a pipe which the orchestrator listens to on which the result of the action is placed. This is to place a synchronous layer on what is an asynchronous architecture.

Most databases can be used in an append only fashion, although investing specific databases (such as Datomic) would probably be worthwhile.

Storing the events as they come in before marshalling them into a projection gives us maximum flexibility when expanding going forward. We would be able to replay all of the events, in order, to build projections based on data going back to the launch of day 1 of the API.



Points of note

No particular database is required, just an interface that needs to be implemented. An in memory version is provided. My experience is that database provision is not part of my remit.

The data passed back to the orchestrator is wrapped in a Try object. This contains either the event or an exception, if something went wrong. If storing the event to persistent storage fails the pipeline is short circuited, an error is returned to the orchestrator and no event is sent to the projection.

Appendix A

Swagger Specification

```
swagger: '2.0'
info:
  description: An API for managing payments
  version: 0.0.1
  title: Payments API
host: https://github.com/HughPowell/payments
tags:
  - name: payments
    description: All payment actions
schemes:
  - http
```

```
paths:
  /payments:
    get:
      tags:
        - payments
      summary: Get a list of all payments
      consumes:
        - application/json
      produces:
        - application/json
      responses:
        '200':
          description: 'A list of URLs, one for each available Payment'
    post:
      tags:
        - payments
      summary: Add a new payment
      consumes:
        - application/json
      produces:
        - application/json
      parameters:
        - in: body
          name: body
          description: Payment to be created
          required: true
      responses:
        '201':
          description: Payment successfully created
          headers:
            eTag:
              description: Digest of the payment
              type: string
        '400':
          description: Invalid payment object supplied
        '409':
          description: A payment with this id has already been created
  '/payments/{paymentId}':
    get:
      tags:
        - payments
      summary: Get this version of this payment
      consumes:
        - application/json
      produces:
        - application/json
      parameters:
        - name: paymentId
          in: path
          description: The ID of the payment
          required: true
          type: string
      responses:
        '200':
          description: The payment data
          headers:
            eTag:
              description: Digest of the payment
              type: string
        '404':
          description: The payment does not exist
    put:
      tags:
        - payments
      summary: Update this payment with new payment data
      consumes:
        - application/json
      produces:
```

```
    - application/json
  parameters:
    - name: paymentId
      in: path
      description: The ID of the payment
      required: true
      type: string
    - name: If-Match
      in: header
      description: Digest of the payment to be updated
      required: true
      type: string
    - in: body
      name: body
      description: Payment to be created
      required: true
  responses:
    '204':
      description: Payment successfully updated
      headers:
        eTag:
          description: Digest of the payment
          type: string
    '404':
      description: The payment data was not properly formatted
    '409':
      description: The Id of the Payment does not match the paymentId
    '412':
      description: Payment has been updated since you last retrieved it
    '428':
      description: Required precondition headers missing
delete:
  tags:
    - payments
  summary: Deletes this payment
  consumes:
    - application/json
  produces:
    - application/json
  parameters:
    - name: paymentId
      in: path
      description: The ID of the payment
      required: true
      type: string
  responses:
    '204':
      description: The payment was successfully deleted
```