COMS E6998: **Microservices and Cloud Applications**

Lecture 5: Idempotency, Conditional Execution, Orchestration, Pub/Sub, API GW

Dr. Donald F. Ferguson
dff9@columbia.edu

Contents

Introduction

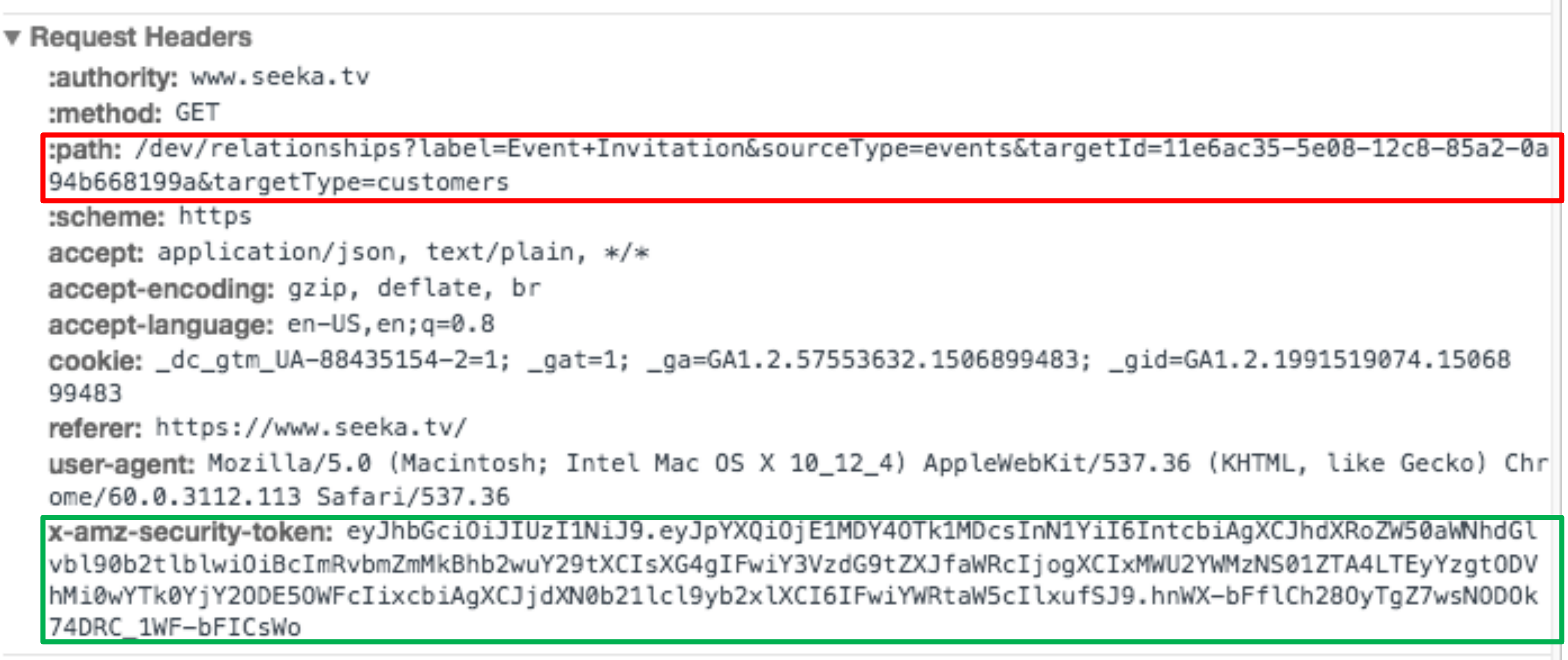
Questions?
Comments?

Project 1 Review Schedule

Implementing Routing Paths

Overview

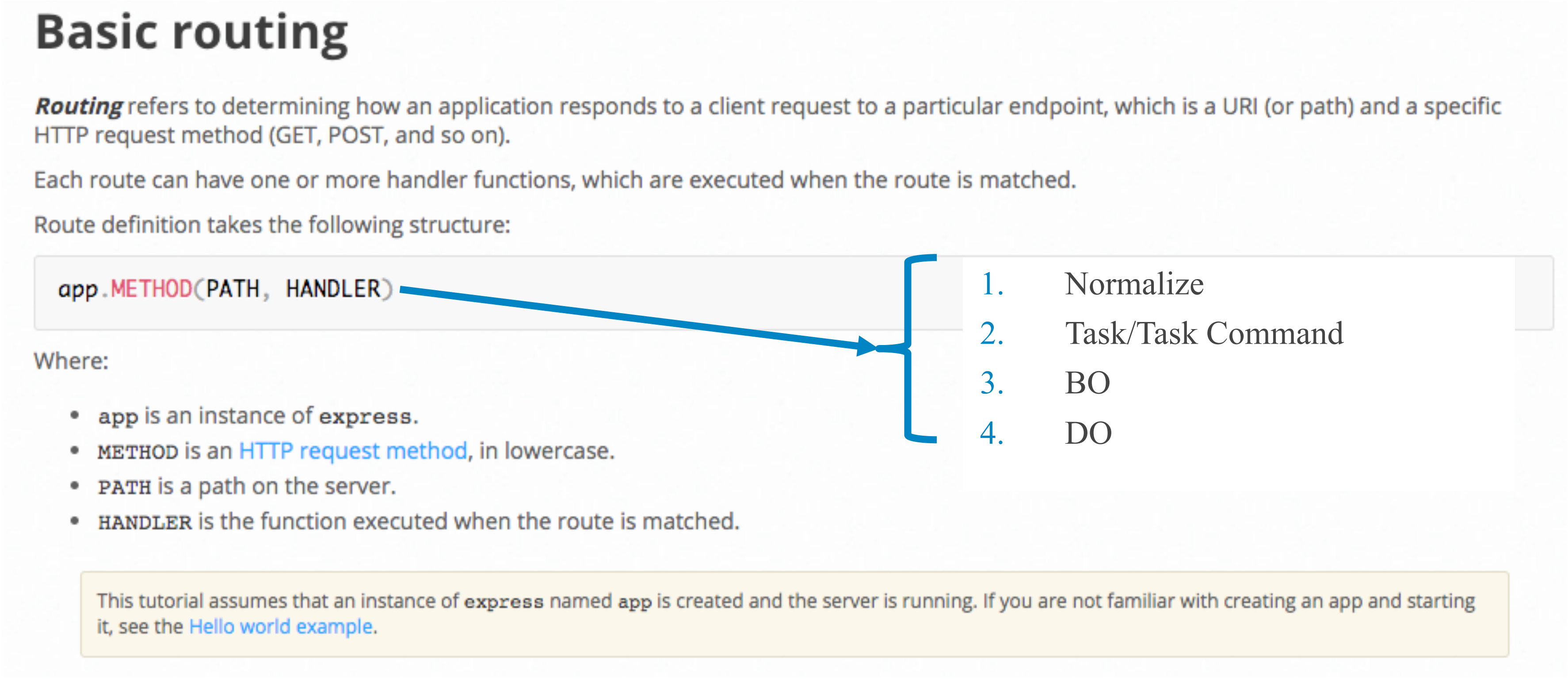
Headers (Path and JWT in Example)



```
▼ Request Headers
   :authority: www.seeka.tv
   :method: GET
   :path: /dev/relationships?label=Event+Invitation&sourceType=events&targetId=11e6ac35-5e08-12c8-85a2-0a
   94b668199a&targetType=customers
   :scheme: https
   accept: application/json, text/plain, */*
   accept-encoding: gzip, deflate, br
   accept-language: en-US,en;q=0.8
   cookie: _dc_gtm_UA-88435154-2=1; _gat=1; _ga=GA1.2.57553632.1506899483; _gid=GA1.2.199519074.15068
   99483
   referer: https://www.seeka.tv/
   user-agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_4) AppleWebKit/537.36 (KHTML, like Gecko) Chr
   ome/60.0.3112.113 Safari/537.36
   x-amz-security-token: eyJhbGciOiJIUzI1NiJ9.eyJpYXQiOjE1MDY4OTk1MDcsInN1YiI6IntcbiAgXCJhdXRoZW50aWNhdGl
   vbl90b2tlblwiOiBcImRvbmZmMkBhb2wuY29tXCIsXG4gIFwiY3VzdG9tZXJfaWRcIjogXCIxMWU2YWMzNS01ZTA4LTEyYzgtODV
   hMi0wYTk0YjY2ODE5OWFcIixcbiAgXCJjdXN0b21lcl9yb2xlXCI6IFwiYWRtaW5cIlxufSJ9.hnWX-bFflCh28OyTgZ7wsNODOk
   74DRC_1WF-bFICsWo
```

Microservices Maps Method/Paths to Handlers

(Example – Node + Express: https://expressjs.com/en/starter/basic-routing.html)



Basic routing

Routing refers to determining how an application responds to a client request to a particular endpoint, which is a URI (or path) and a specific HTTP request method (GET, POST, and so on).

Each route can have one or more handler functions, which are executed when the route is matched.

Route definition takes the following structure:

```
app.METHOD(PATH, HANDLER)
```

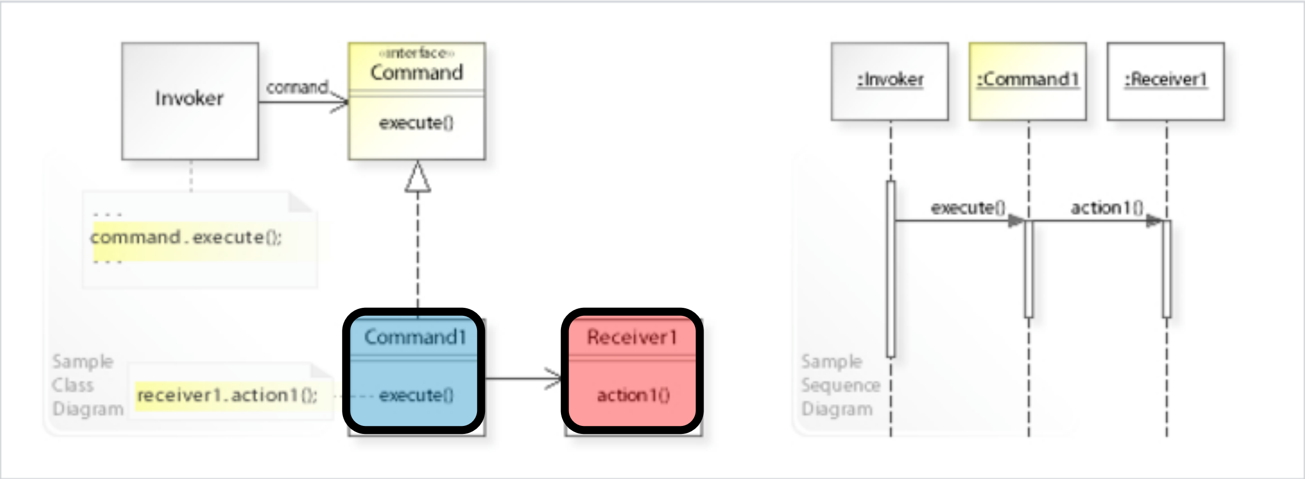
Where:

- `app` is an instance of `express`.
- `METHOD` is an HTTP request method, in lowercase.
- `PATH` is a path on the server.
- `HANDLER` is the function executed when the route is matched.

This tutorial assumes that an instance of `express` named `app` is created and the server is running. If you are not familiar with creating an app and starting it, see the Hello world example.

1. Normalize
2. Task/Task Command
3. BO
4. DO

Command/Task Pattern



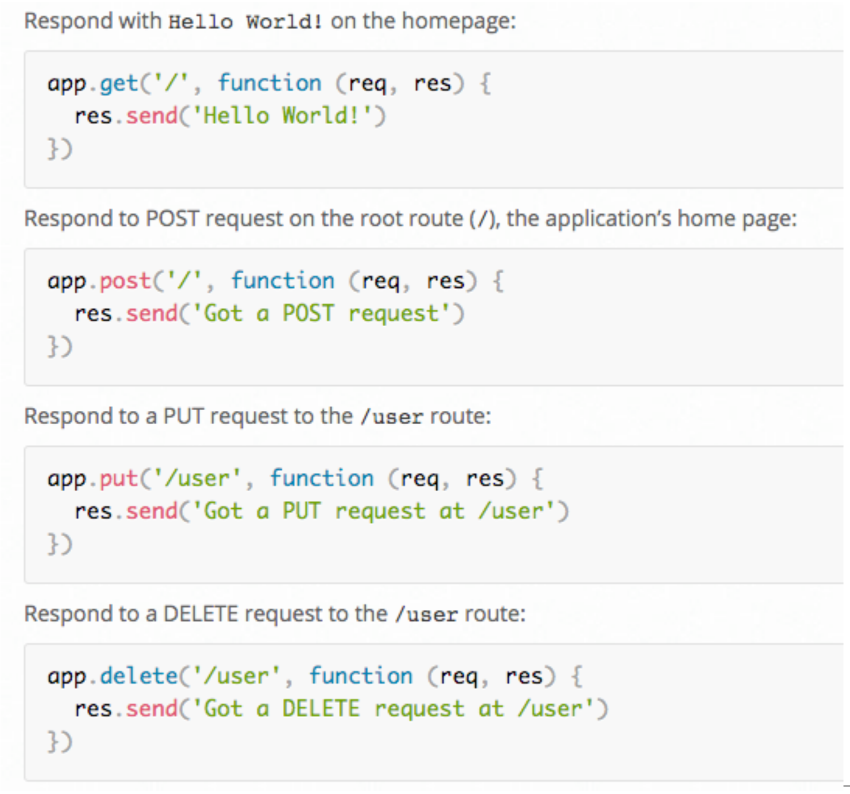
A sample UML class and sequence diagram for the Command design pattern. [3]

In the terminology we have been using,

- The Command is the Task connecting the composing/routing environment
- To the Receiver BO or Proxy for executing the composition/environment independent application logic.

Microservices Maps Method/Paths to Handlers

(Node + Express Example; https://expressjs.com/en/starter/basic-routing.html)



Respond with `Hello World!` on the homepage:

```
app.get('/', function (req, res) {
  res.send('Hello World!')
})
```

Respond to POST request on the root route (/), the application's home page:

```
app.post('/', function (req, res) {
  res.send('Got a POST request')
})
```

Respond to a PUT request to the `/user` route:

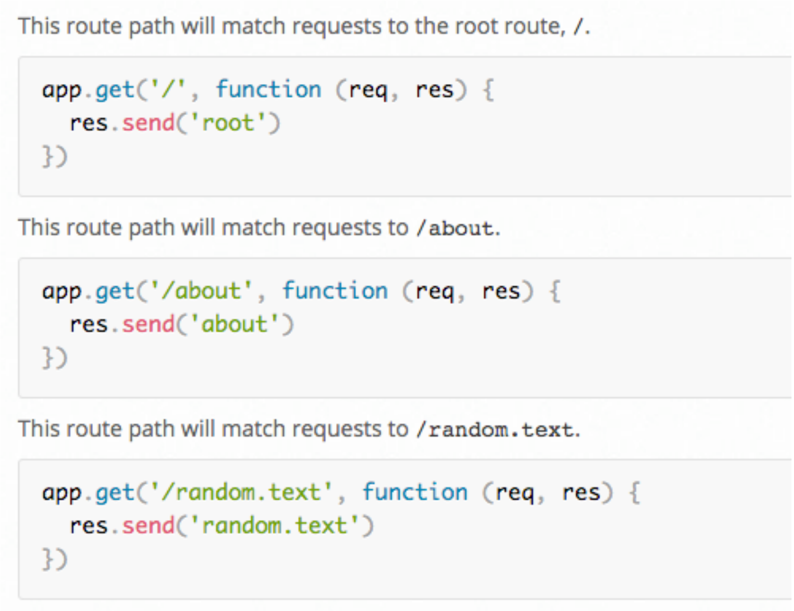
```
app.put('/user', function (req, res) {
  res.send('Got a PUT request at /user')
})
```

Respond to a DELETE request to the `/user` route:

```
app.delete('/user', function (req, res) {
  res.send('Got a DELETE request at /user')
})
```

- These are trivial examples
- Instead of inline implementation code, the route call
  - Normalize the input.
  - Call XXXBO module via
  - Some exported function.
- The XXXBO.method
  - Implements the application logic.
  - Interact with the DO layer.
  - Returns the normalized answer
- The route handler converts to correct response format and protocol.

Some Examples



This route path will match requests to the root route, /.

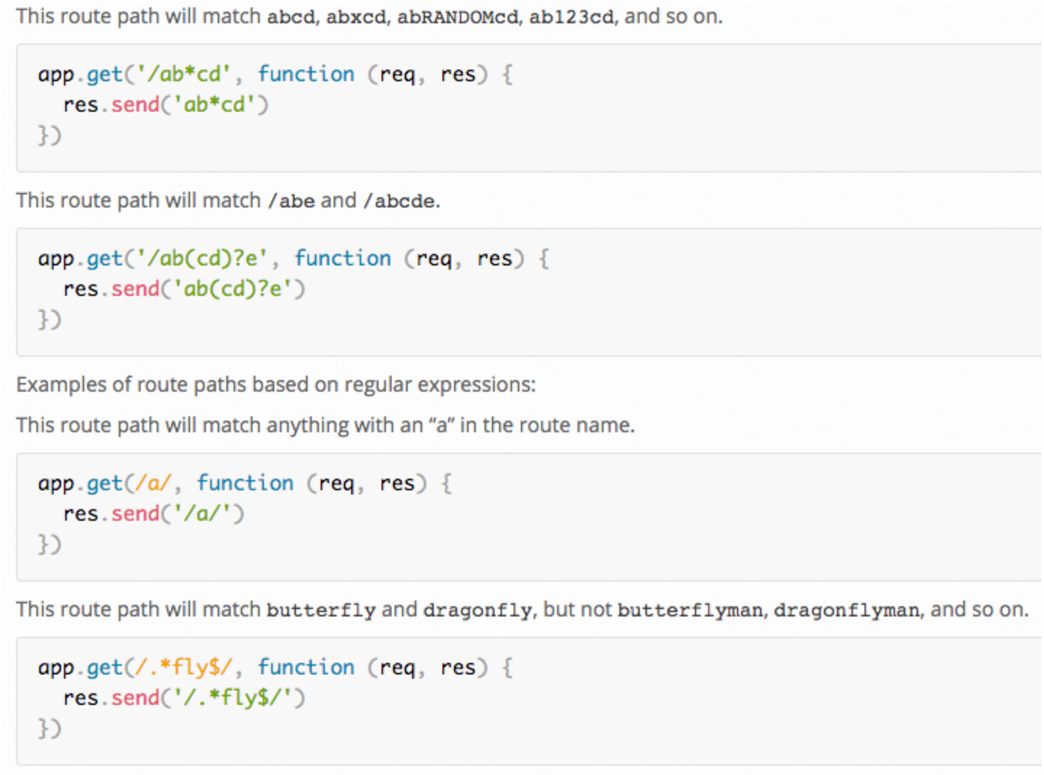
```
app.get('/', function (req, res) {
  res.send('root')
})
```

This route path will match requests to /about.

```
app.get('/about', function (req, res) {
  res.send('about')
})
```

This route path will match requests to /random.text.

```
app.get('/random.text', function (req, res) {
  res.send('random.text')
})
```



This route path will match abcd, abxcd, abRANDOMcd, ab123cd, and so on.

```
app.get('/ab*cd', function (req, res) {
  res.send('ab*cd')
})
```

This route path will match /abe and /abcde.

```
app.get('/ab(cd)?e', function (req, res) {
  res.send('ab(cd)?e')
})
```

Examples of route paths based on regular expressions:

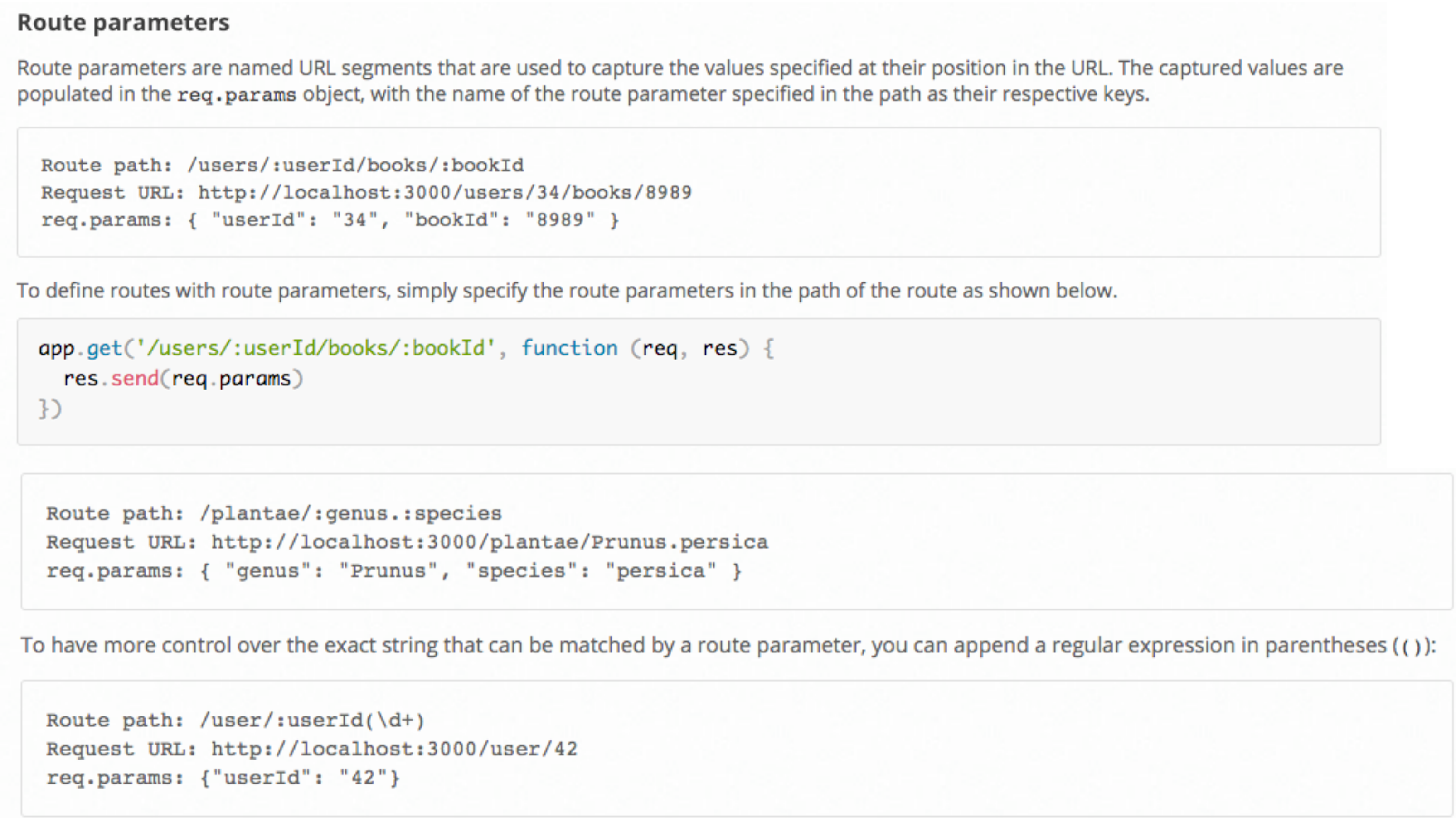
This route path will match anything with an “a” in the route name.

```
app.get(/a/, function (req, res) {
  res.send('/a/')
})
```

This route path will match butterfly and dragonfly, but not butterflyman, dragonflyman, and so on.

```
app.get(/.*fly$/, function (req, res) {
  res.send('/.*fly$/')
})
```

Route/Path Parameters



Route parameters

Route parameters are named URL segments that are used to capture the values specified at their position in the URL. The captured values are populated in the `req.params` object, with the name of the route parameter specified in the path as their respective keys.

```
Route path: /users/:userId/books/:bookId
Request URL: http://localhost:3000/users/34/books/8989
req.params: { "userId": "34", "bookId": "8989" }
```

To define routes with route parameters, simply specify the route parameters in the path of the route as shown below.

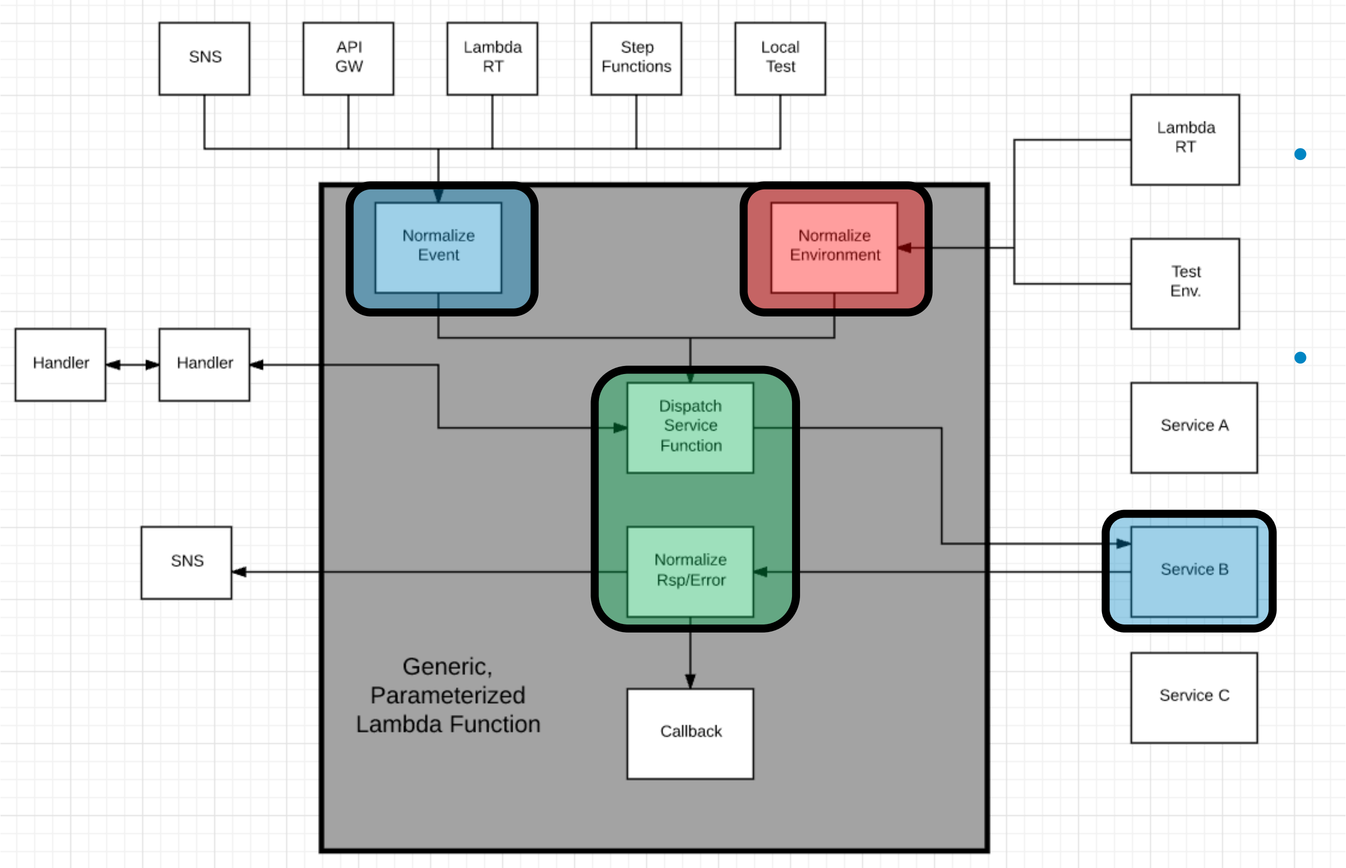
```
app.get('/users/:userId/books/:bookId', function (req, res) {
  res.send(req.params)
})
```

```
Route path: /plantae/:genus.:species
Request URL: http://localhost:3000/plantae/Prunus.persica
req.params: { "genus": "Prunus", "species": "persica" }
```

To have more control over the exact string that can be matched by a route parameter, you can append a regular expression in parentheses (`()`):

```
Route path: /user/:userId(\d+)
Request URL: http://localhost:3000/user/42
req.params: {"userId": "42"}
```

Design Pattern – Generic Lambda Function



- Options
  - In zip file
  - Separate Lambda
- Config info
  - Added to Lambda
  - Env Variable
  - From S3

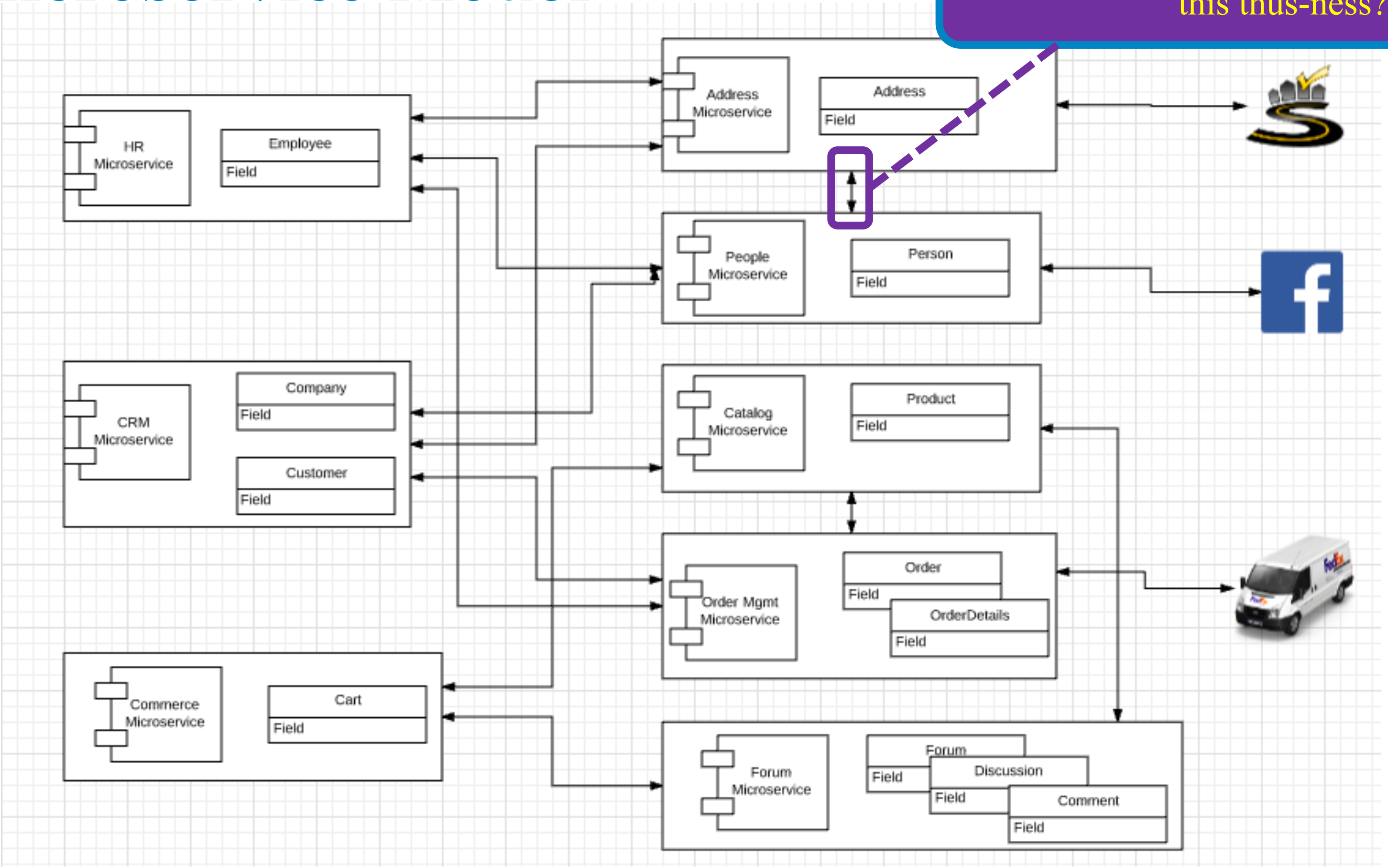
Project 1 Implications

1st Project – Part 1

- **Implement two distinct microservices**
  - **Person**
  - **Address**
- Tasks
  - Use Swagger Editor to define and document
  - Implement an Elastic Beanstalk application the relevant REST API.
  - Each microservice should support
    - GET and POST on resource, e.g. /Person
    - GET, PUT, DELETE on resource/id, e.g. /Person/dff9
    - Simple query, e.g. /Person?lastName=Ferguson
    - Pagination
    - **Relationship paths: /Person/dff9/address and /Addresses/someID/persons**
    - HATEOAS links where appropriate.
  - Simple HTML/Angular demo UI.
- Due: 11:59 PM on 26-Sep-2017

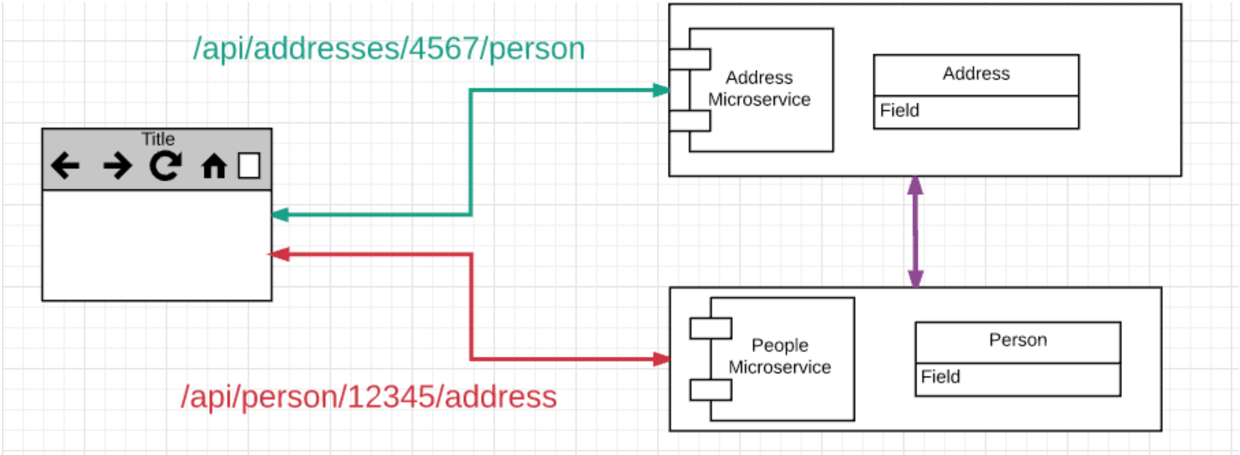
- How could this work?
- There are two distinct microservices, but these URLs reference both.

Microservice Model



"Why is this thus, and what is the reason for this thus-ness?"

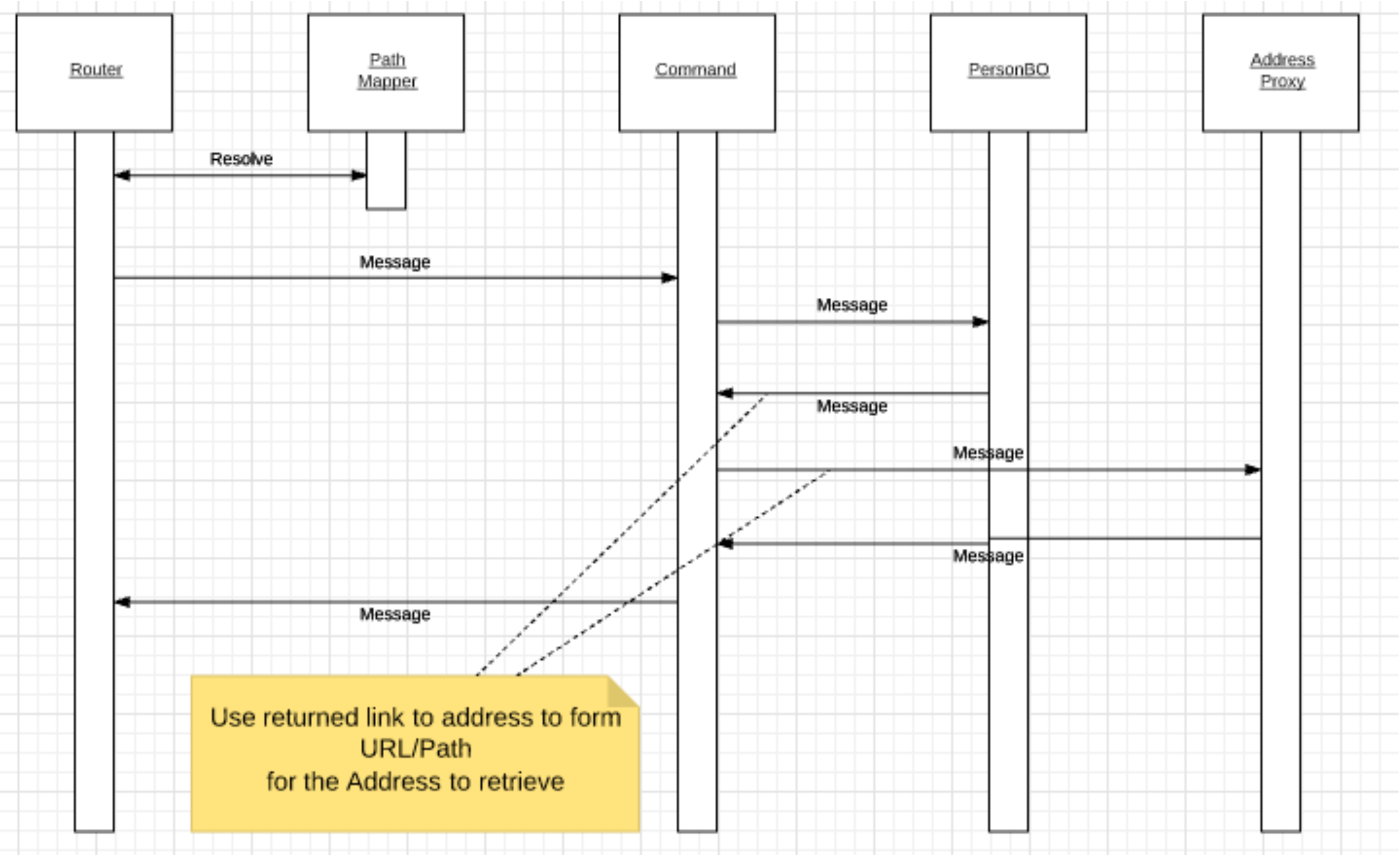
Microservice – Microservice Interactions



- Microservices interact to process methods/operations.
- These two examples are relatively simple. For example, GET /person/1234/address
  - Command/call to retrieve /person/1234 via local BO/DO.
  - Access address link.
  - Command/call to "GET" on address link via Proxy
  - Return requested information

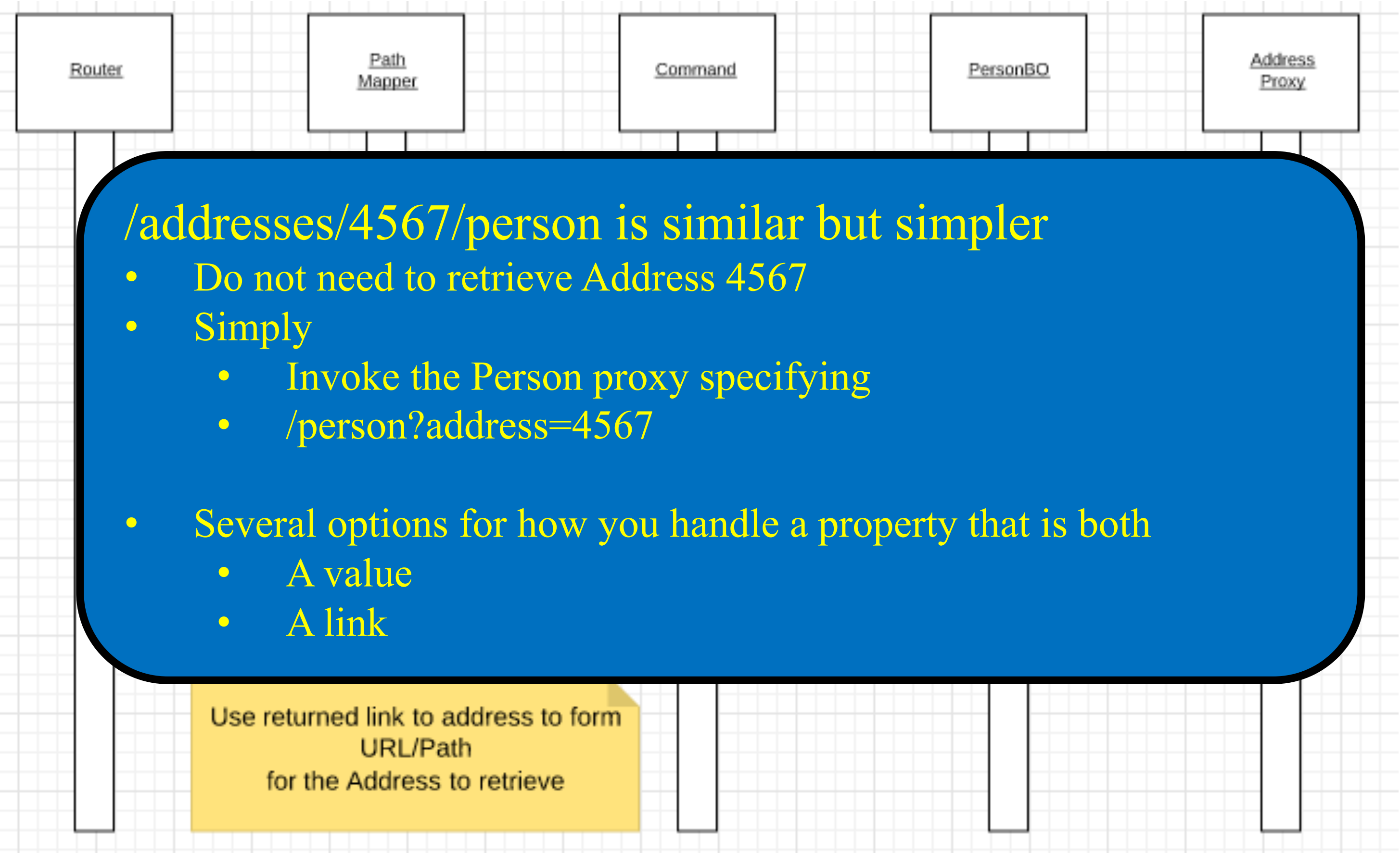
Which is basically the same approach as implementing in the UI.

GET /Person/1234/address



Use returned link to address to form URL/Path for the Address to retrieve

GET /Person/1234/address



Use returned link to address to form URL/Path for the Address to retrieve

/addresses/4567/person is similar but simpler

- Do not need to retrieve Address 4567
- Simply
  - Invoke the Person proxy specifying
  - /person?address=4567

- Several options for how you handle a property that is both
  - A value
  - A link

Path Summary

Path Summary

- The slides express the concepts in very systematic terms
  - Path Router
  - Command/Task
  - BO, DO
  - etc.
- There are frameworks and models that are very rigorous about the concepts, but much/most of the time you
  - Think with the concept in your mind.
  - Implement in relatively simple code.
- The key takeaways are:
  - REST API users expect to be able to follow paths.
  - Microservices interact to implement operations.
  - Microservice implementation needs to implement mapping paths to specific code.

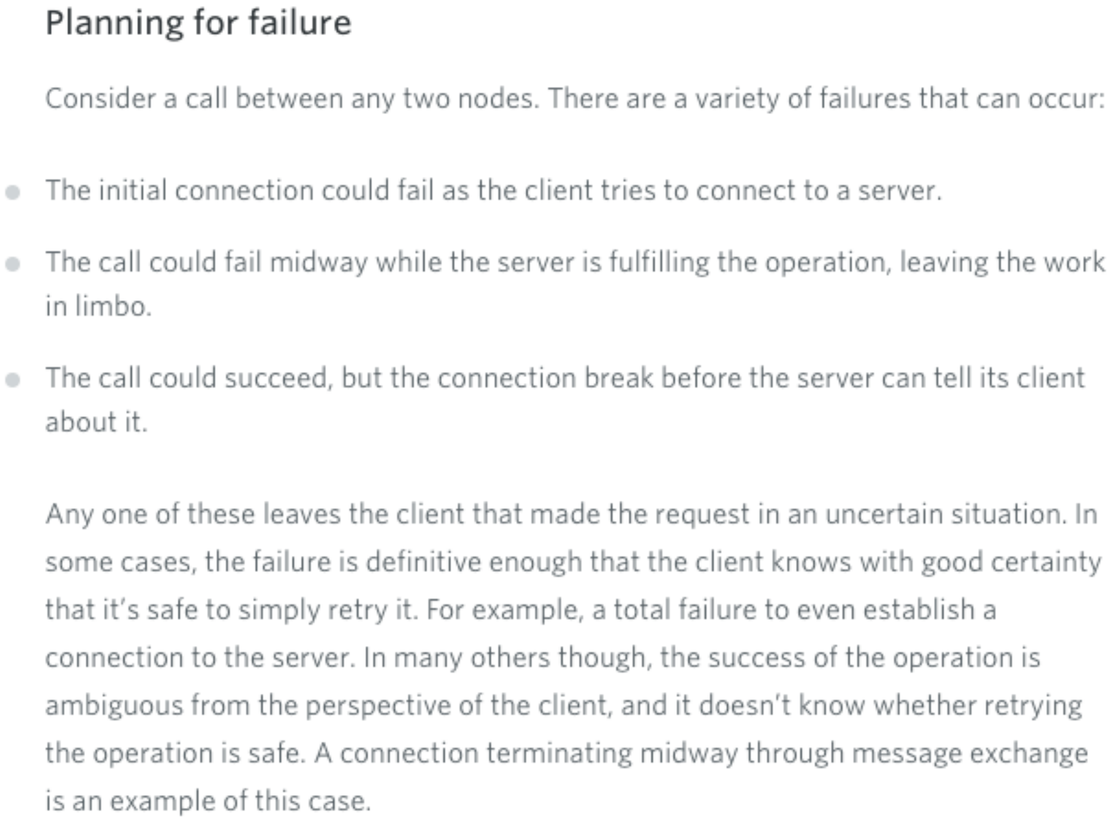
Idempotency

Safe Methods

Conditional Execution

Idempotent
Safe

Client and Server Must Plan for Call Failure



Planning for failure

Consider a call between any two nodes. There are a variety of failures that can occur:

- The initial connection could fail as the client tries to connect to a server.
- The call could fail midway while the server is fulfilling the operation, leaving the work in limbo.
- The call could succeed, but the connection break before the server can tell its client about it.

Any one of these leaves the client that made the request in an uncertain situation. In some cases, the failure is definitive enough that the client knows with good certainty that it's safe to simply retry it. For example, a total failure to even establish a connection to the server. In many others though, the success of the operation is ambiguous from the perspective of the client, and it doesn't know whether retrying the operation is safe. A connection terminating midway through message exchange is an example of this case.

https://www.safaribooksonline.com/library/view/restful-web-services/9780596809140/ch01s04.html

Idempotent and Safe

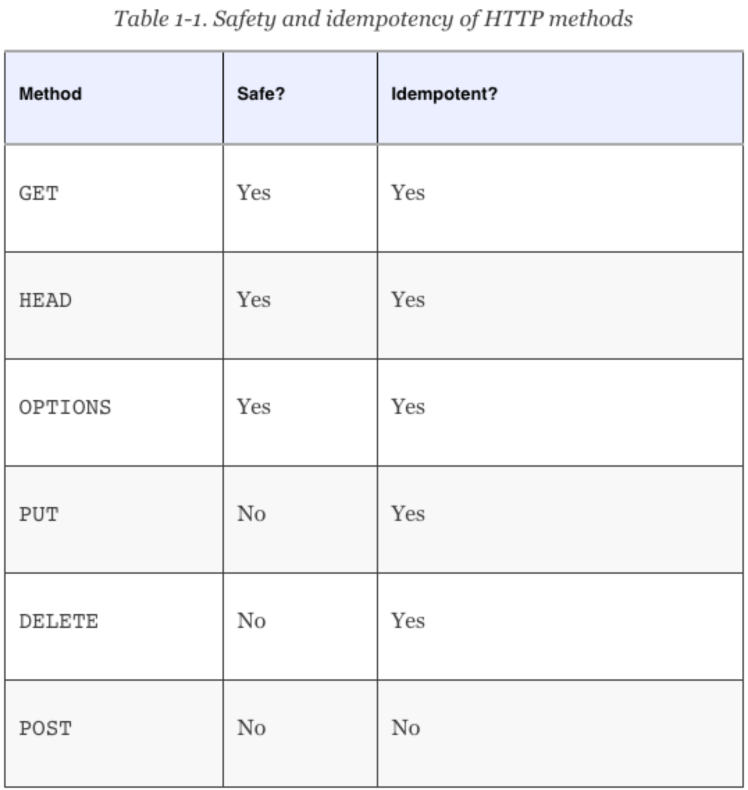


Table 1-1. Safety and idempotency of HTTP methods

Method	Safe?	Idempotent?
GET	Yes	Yes
HEAD	Yes	Yes
OPTIONS	Yes	Yes
PUT	No	Yes
DELETE	No	Yes
POST	No	No

https://www.safaribooksonline.com/library/view/restful-web-services/9780596809140/ch01s04.html

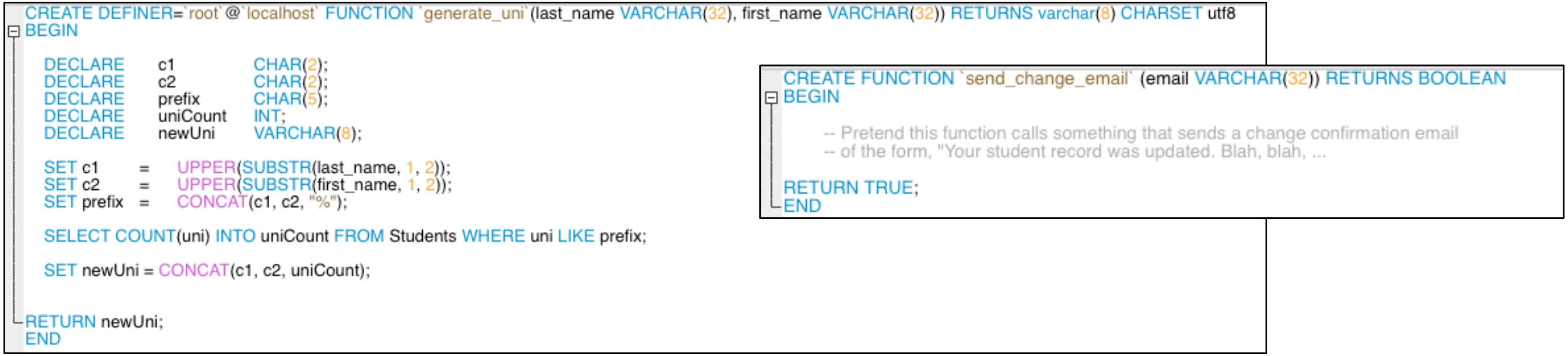
"**Idempotence** is the property of certain operations in mathematics and computer science that they can be applied multiple times without changing the result beyond the initial application." (Wikipedia)

"In particular, the convention has been established that the GET and HEAD methods SHOULD NOT have the significance of taking an action other than retrieval. These methods ought to be considered "safe". This allows user agents to represent other methods, such as POST, PUT and DELETE, in a special way, so that the user is made aware of the fact that a possibly unsafe action is being requested.

Naturally, it is not possible to ensure that the server does not generate side-effects as a result of performing a GET request; in fact, some dynamic resources consider that a feature. The important distinction here is that the user did not request the side-effects, so therefore cannot be held accountable for them."

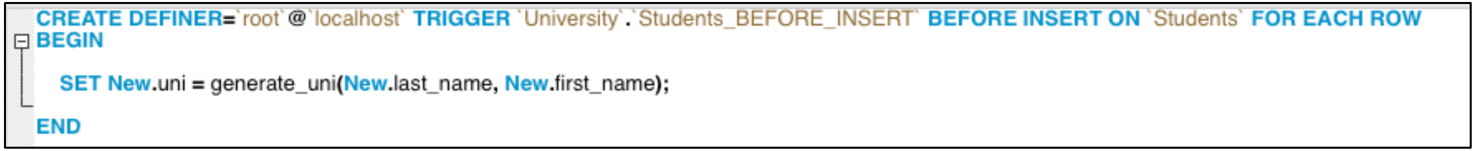
https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html

Well, That was Vaguely Alarming
Well, Consider a Student Table

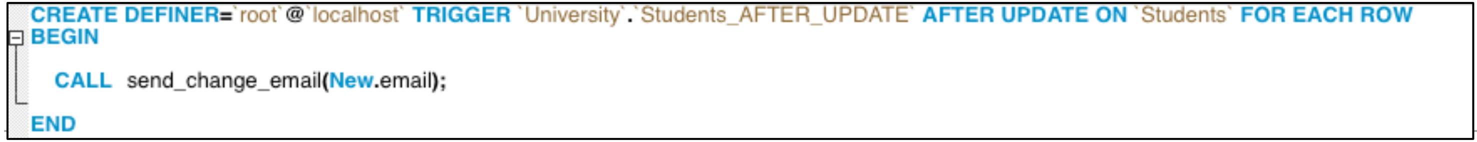


```
CREATE DEFINER=`root`@`localhost` FUNCTION `generate_uni`(last_name VARCHAR(32), first_name VARCHAR(32)) RETURNS varchar(8) CHARSET utf8
BEGIN

    DECLARE     c1          CHAR(2);
    DECLARE     c2          CHAR(2);
    DECLARE     prefix      CHAR(5);
    DECLARE     uniCount    INT;
    DECLARE     newUni      VARCHAR(8);

    SET c1      =    UPPER(SUBSTR(last_name, 1, 2));
    SET c2      =    UPPER(SUBSTR(first_name, 1, 2));
    SET prefix  =    CONCAT(c1, c2, "%");

    SELECT COUNT(uni) INTO uniCount FROM Students WHERE uni LIKE prefix;

    SET newUni = CONCAT(c1, c2, uniCount);


RETURN newUni;
END
```

```
CREATE FUNCTION `send_change_email` (email VARCHAR(32)) RETURNS BOOLEAN
BEGIN

    -- Pretend this function calls something that sends a change confirmation email
    -- of the form, "Your student record was updated. Blah, blah, ...

RETURN TRUE;
END
```



```
CREATE DEFINER=`root`@`localhost` TRIGGER `University`.`Students_BEFORE_INSERT` BEFORE INSERT ON `Students` FOR EACH ROW
BEGIN

    SET New.uni = generate_uni(New.last_name, New.first_name);

END
```



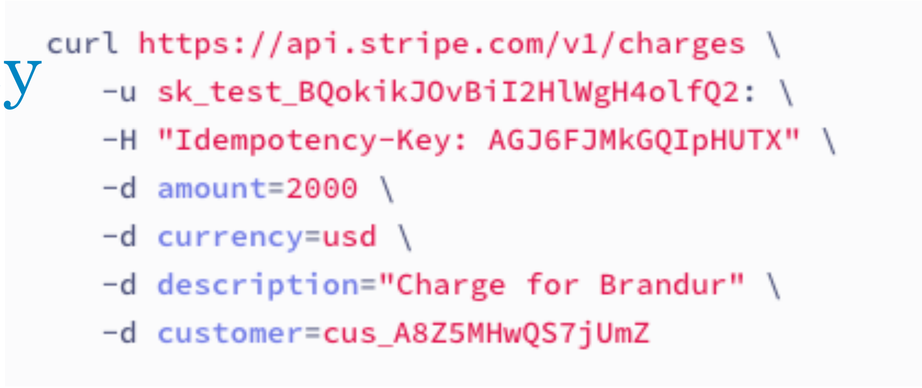
```
CREATE DEFINER=`root`@`localhost` TRIGGER `University`.`Students_AFTER_UPDATE` AFTER UPDATE ON `Students` FOR EACH ROW
BEGIN

    CALL  send_change_email(New.email);

END
```

Well, That was Vaguely Alarming
Well, Consider a Student Table

- INSERT is not idempotent.
  - Calling "INSERT INTO Students (last_name, first_name, email)
    VALUES ("Smith", "John", "js@contoso.com");
  - 3 times results in 3 tuples for the same person
- UPDATE is idempotent but not "safe."
  - Calling "UPDATE Students SET "email=js23@columbia.edu"
    WHERE uni="JS01";
  - 3 times has the same end state for the tuple as one call
  - BUT sends 3 annoying emails (side effects).
- SELECT
  - Does not change the data and does not send an email, and is idempotent and safe (probably).
  - Multiple SELECTs *may* not return the same result. Someone else may have changed the tuple.
- DELETE
  - Multiple deletes have the same result (record deleted), but 2, 3, … probably return a different result.
  - May or may not have side effects.

Implementing Idempotency

(Consider Stripe.com)



```
curl https://api.stripe.com/v1/charges \
   -u sk_test_BQokikJOvBiI2HlWgH4olfQ2: \
   -H "Idempotency-Key: AGJ6FJMkGQIpHUTX" \
   -d amount=2000 \
   -d currency=usd \
   -d description="Charge for Brandur" \
   -d customer=cus_A8Z5MHwQS7jUmZ
```

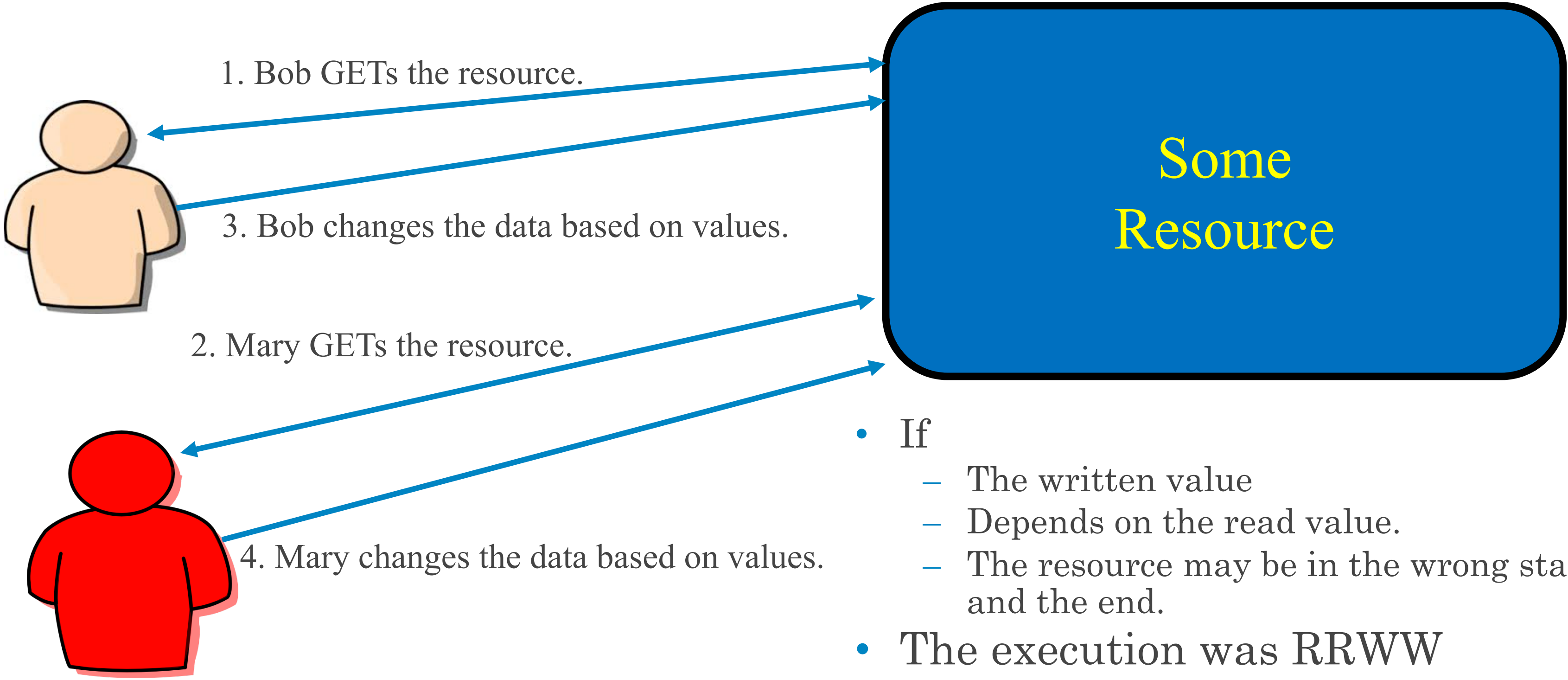
- Requests have a caller generated
  - UUID
  - Stored in a header.
- The Microservice
  - Check if the message is “too old.”
  - Checks a ledger for the header.
  - Return the previously recorded response if found.
  - Otherwise
    - BEGIN TRANSACTION
      - Record idempotency header.
      - Change resource.
      - Generate response.
      - Record response.
    - COMMIT
  - Return response.

Periodically purge the ledger of old headers.

This is a conceptual explanation.

Many implementations do not use transactions and a implement a best effort, except when extremely critical.

Read then Update Conflict



1. Bob GETs the resource.

3. Bob changes the data based on values.

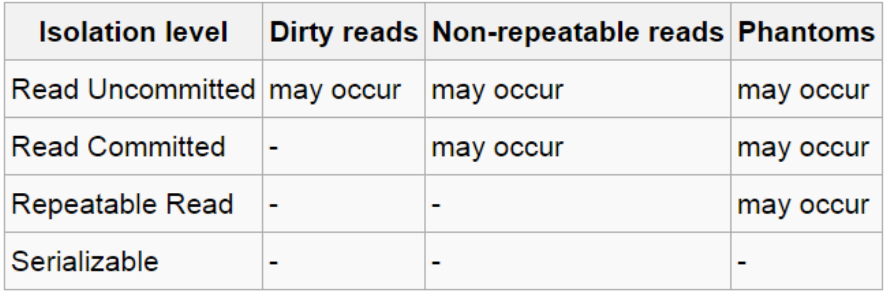
2. Mary GETs the resource.

4. Mary changes the data based on values.

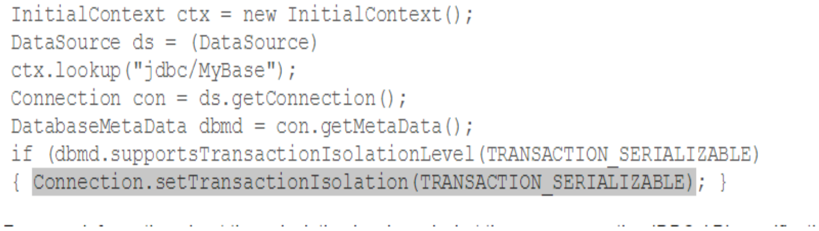
Some Resource

- If
  - The written value
  - Depends on the read value.
  - The resource may be in the wrong state and the end.
- The execution was RRWW
- But the intent was RWRW.

In Databases, Cursors Define *Isolation*



Isolation level	Dirty reads	Non-repeatable reads	Phantoms
Read Uncommitted	may occur	may occur	may occur
Read Committed	-	may occur	may occur
Repeatable Read	-	-	may occur
Serializable	-	-	-



```
InitialContext ctx = new InitialContext();
DataSource ds = (DataSource)
ctx.lookup("jdbc/MyBase");
Connection con = ds.getConnection();
DatabaseMetaData dbmd = con.getMetaData();
if (dbmd.supportsTransactionIsolationLevel(TRANSACTION_SERIALIZABLE)
{ Connection.setTransactionIsolation(TRANSACTION_SERIALIZABLE); }
```

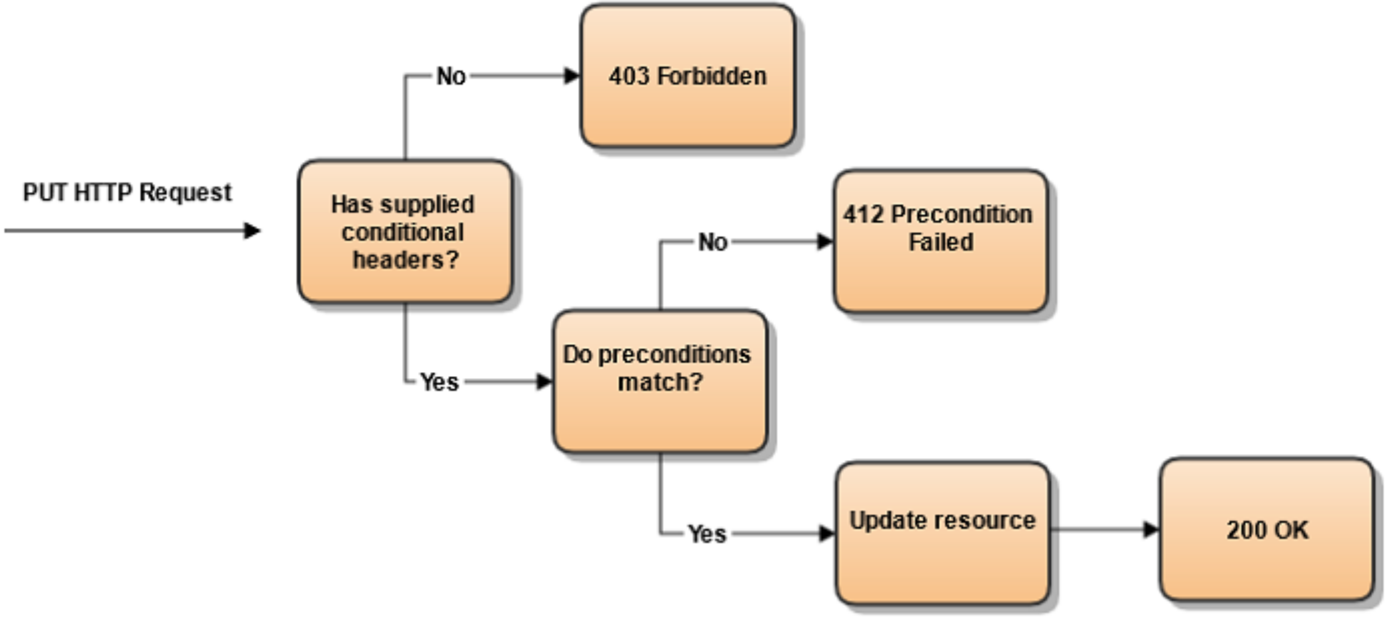
- We have talked about ACID transactions
  - Atomic: A set of writes all occur or none occur.
  - Durable: The data "does not disappear," e.g. write to disk.
  - Consistent: My applications move the database between consistent states, e.g. the transfer function works correctly.
- Isolation
  - Determines what happens when two or more threads are manipulating the data at the same time.
  - And is defined relative to where cursors are and what they have touched.
  - Because the cursor movement determines *what you are reading or have read.*
- *But, … Cursors are client conversation state and cannot be used in REST.*

Isolation/Concurrency Control

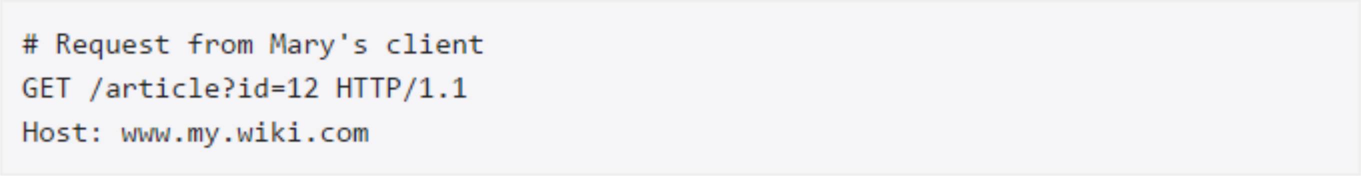
- There are two basic approaches to implementing isolation
  - Locking/Pessimistic, e.g. cursor isolation
  - Optimistic: Before committing, each transaction verifies that no other transaction has modified the data it has read.
- How does this work in REST?
  - The server maintains an ETag (Entity Tag) for each resource.
  - Every time a resource's state changes, the server computes a new ETag.
  - The server includes the ETag in the header when returning data to the client.
  - The server may *optionally* maintain and return "Last Modified" time for resources.
- Semantics on updates
  - If-Match – value of a previous calls ETag response used with a PUT or DELETE. Server should only act if nobody else has modified the resource since you last fetched it. Otherwise provides a 412.
  - If-Modified-Since – value of a previous Last-Modified response used with a GET. Server should only provide a response if the resource was modified since the timestamp submitted. Use in conjunction with If-None-Match in case the change occurred within the same second. Otherwise provide a 304.
  - If-None-Match – value of a previous calls ETag response used with a GET. Server should only provide a response if the ETag doesn't match, i.e. the resource has been altered. Otherwise provide a 304.
  - If-Unmodified-Since – value of a previous Last-Modified response used with a PUT or DELETE. Server should only act if nobody else has modified the resource since you last fetched it. Otherwise provides a 412

Conditional Processing

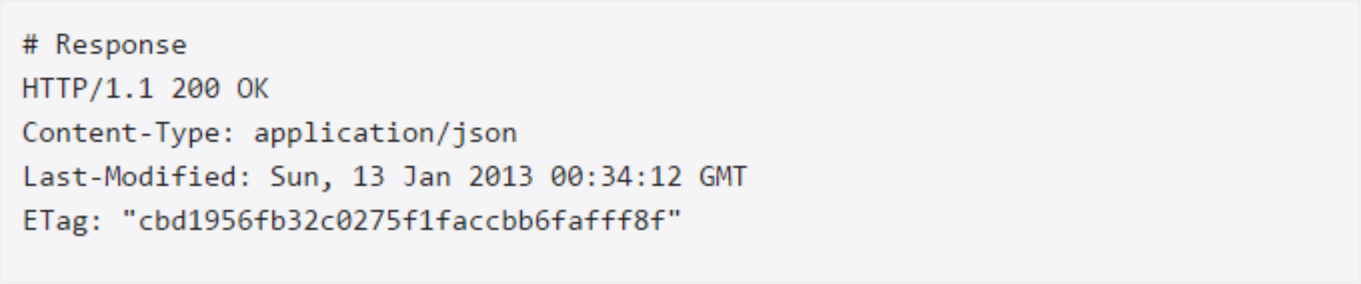
(https://blog.4psa.com/rest-best-practices-managing-concurrent-updates/)



Request/Response

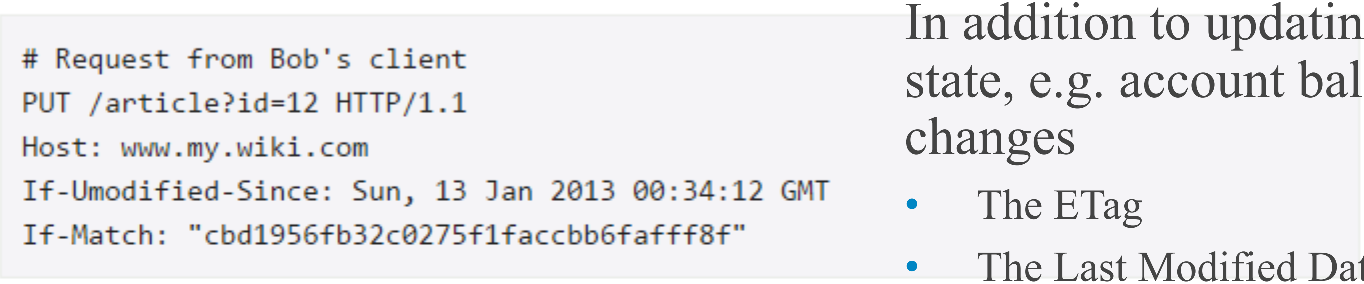


```
# Request from Mary's client
GET /article?id=12 HTTP/1.1
Host: www.my.wiki.com
```



```
# Response
HTTP/1.1 200 OK
Content-Type: application/json
Last-Modified: Sun, 13 Jan 2013 00:34:12 GMT
ETag: "cbd1956fb32c0275f1faccbb6fafff8f"
```

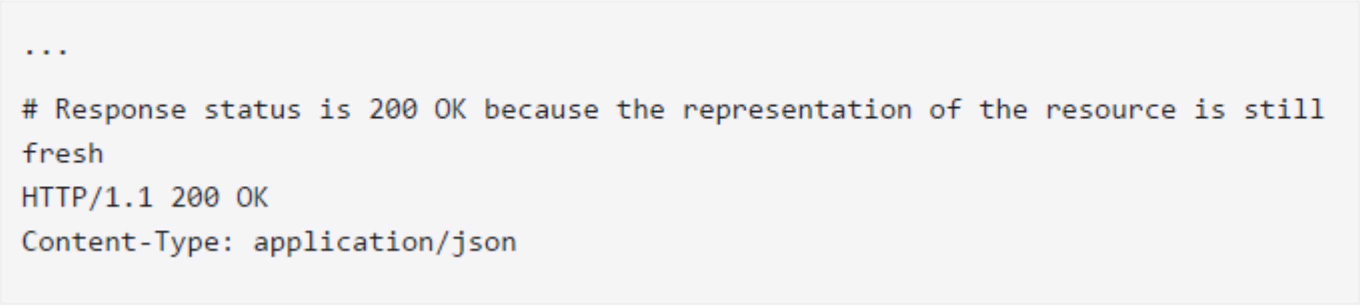
Bob Already Read and now Updates



```
# Request from Bob's client
PUT /article?id=12 HTTP/1.1
Host: www.my.wiki.com
If-Umodified-Since: Sun, 13 Jan 2013 00:34:12 GMT
If-Match: "cbd1956fb32c0275f1faccbb6fafff8f"
```

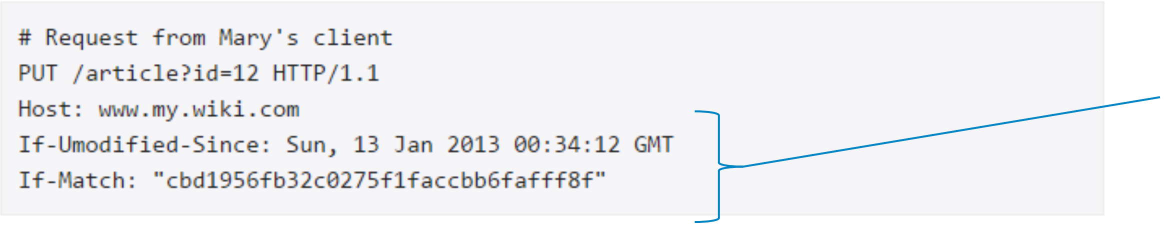
In addition to updating the resource state, e.g. account balance, this also changes

- The ETag
- The Last Modified Date



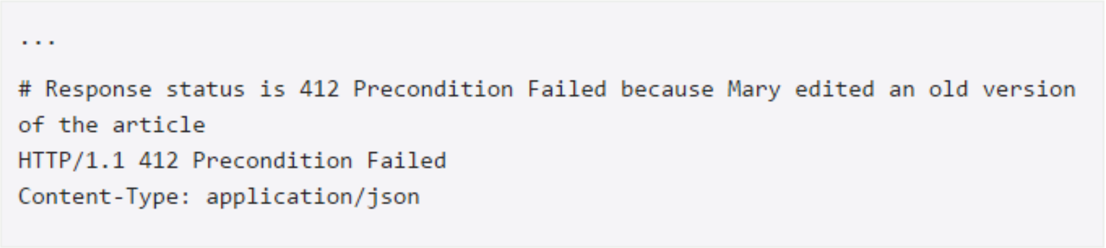
```
...

# Response status is 200 OK because the representation of the resource is still
fresh
HTTP/1.1 200 OK
Content-Type: application/json
```

Now Mary Tries to Update

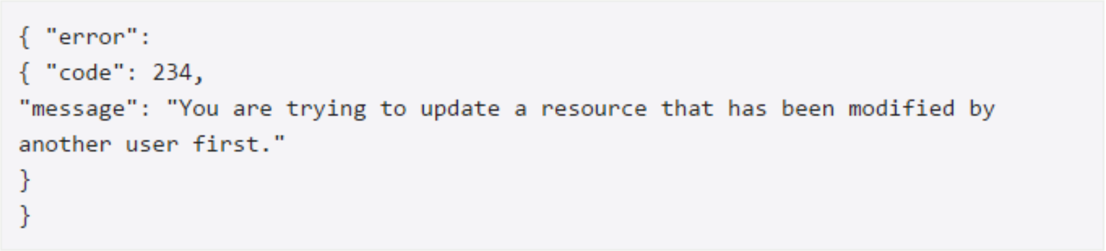


```
# Request from Mary's client
PUT /article?id=12 HTTP/1.1
Host: www.my.wiki.com
If-Umodified-Since: Sun, 13 Jan 2013 00:34:12 GMT
If-Match: "cbd1956fb32c0275f1faccbb6faff8f"
```

Bob's update changed the data, and changed this "metadata."



```
...

# Response status is 412 Precondition Failed because Mary edited an old version
of the article
HTTP/1.1 412 Precondition Failed
Content-Type: application/json
```



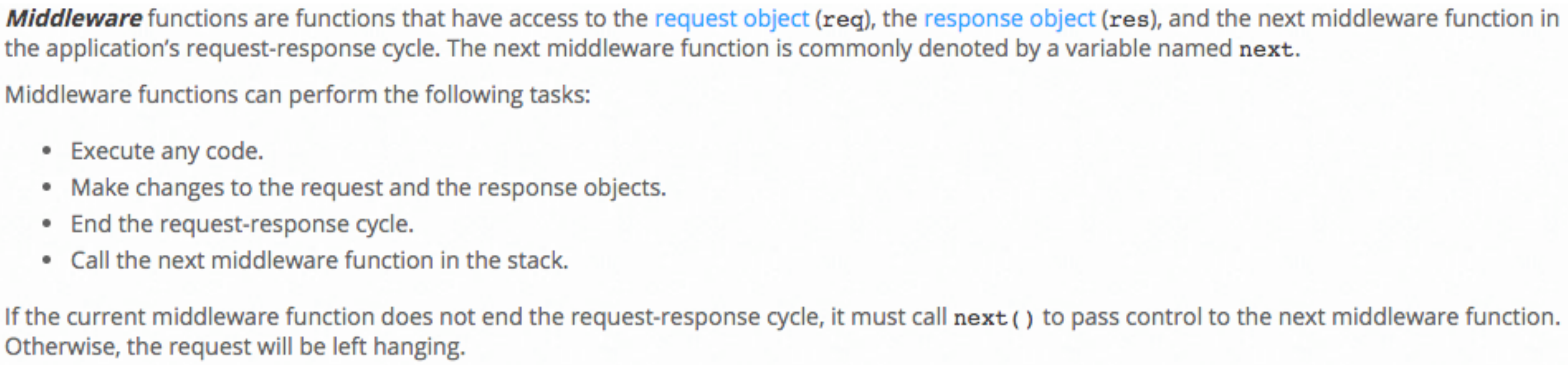
```
{ "error":
{ "code": 234,
"message": "You are trying to update a resource that has been modified by
another user first."
}
}
```

Middleware

Middleware

"**Middleware** is computer software that provides services to software applications beyond those available from the operating system. It can be described as "software glue".[1]

Middleware makes it easier for software developers to implement communication and input/output, so they can focus on the specific purpose of their application." (Wikipedia)



Middleware functions are functions that have access to the request object (`req`), the response object (`res`), and the next middleware function in the application's request-response cycle. The next middleware function is commonly denoted by a variable named `next`.

Middleware functions can perform the following tasks:

- Execute any code.
- Make changes to the request and the response objects.
- End the request-response cycle.
- Call the next middleware function in the stack.

If the current middleware function does not end the request-response cycle, it must call `next()` to pass control to the next middleware function. Otherwise, the request will be left hanging.

Middleware

“Mi

a

"

M

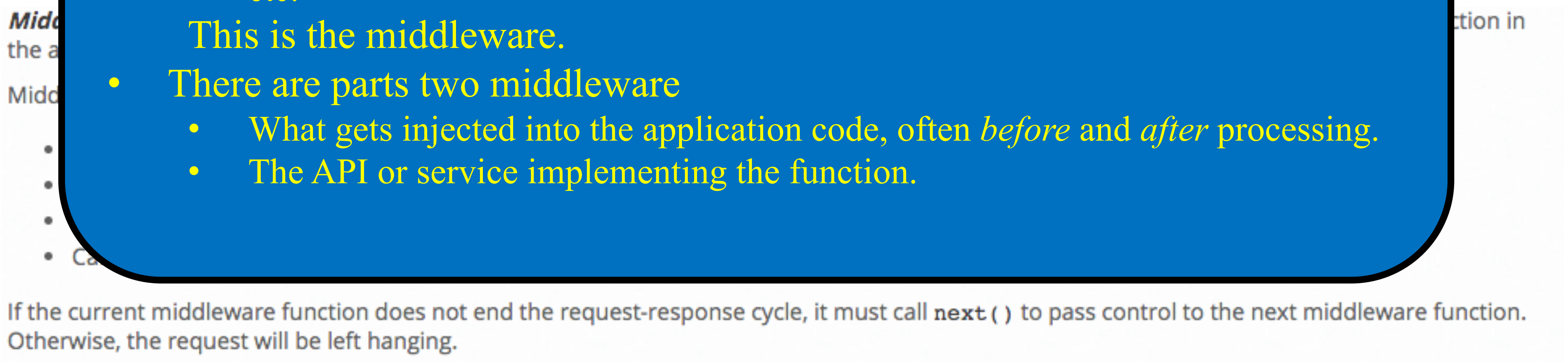
a

(

Don’s definition

- The is application specific logic.
- Icky, complex stuff that all applications seems to need
  - Transactions
  - Security
  - etc.

  This is the middleware.
- There are parts two middleware
  - What gets injected into the application code, often *before* and *after* processing.
  - The API or service implementing the function.



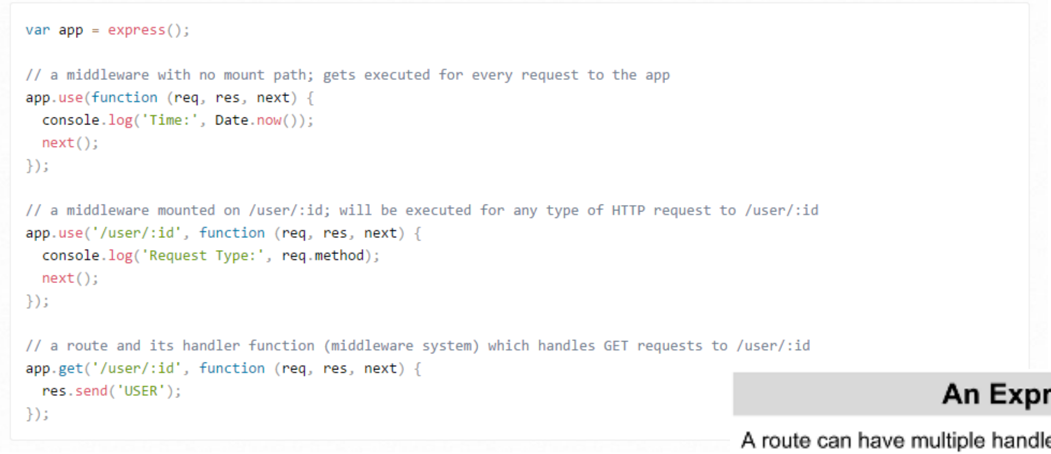
Midd ction in the a

Midd

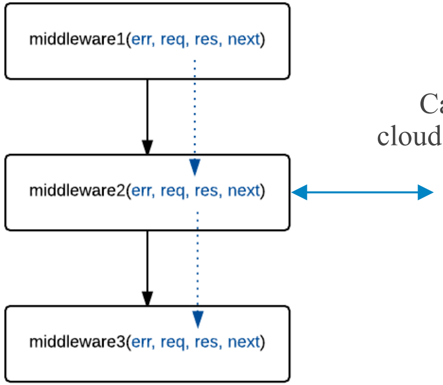
-
-
-
- Ca

If the current middleware function does not end the request-response cycle, it must call `next()` to pass control to the next middleware function. Otherwise, the request will be left hanging.

Node Express Express Example



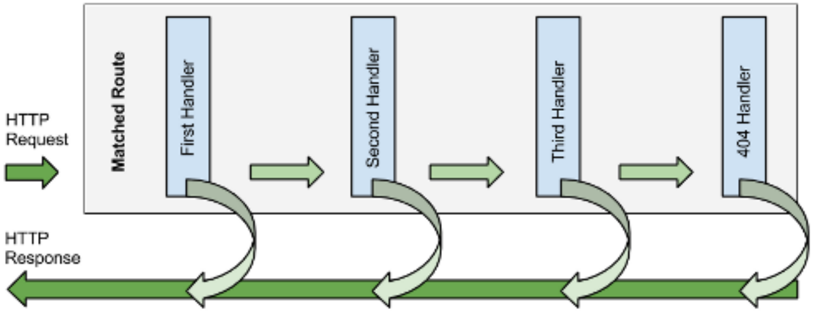
```
var app = express();

// a middleware with no mount path; gets executed for every request to the app
app.use(function (req, res, next) {
  console.log('Time:', Date.now());
  next();
});

// a middleware mounted on /user/:id; will be executed for any type of HTTP request to /user/:id
app.use('/user/:id', function (req, res, next) {
  console.log('Request Type:', req.method);
  next();
});

// a route and its handler function (middleware system) which handles GET requests to /user/:id
app.get('/user/:id', function (req, res, next) {
  res.send('USER');
});
```


middleware1(err, req, res, next)
middleware2(err, req, res, next)
middleware3(err, req, res, next)

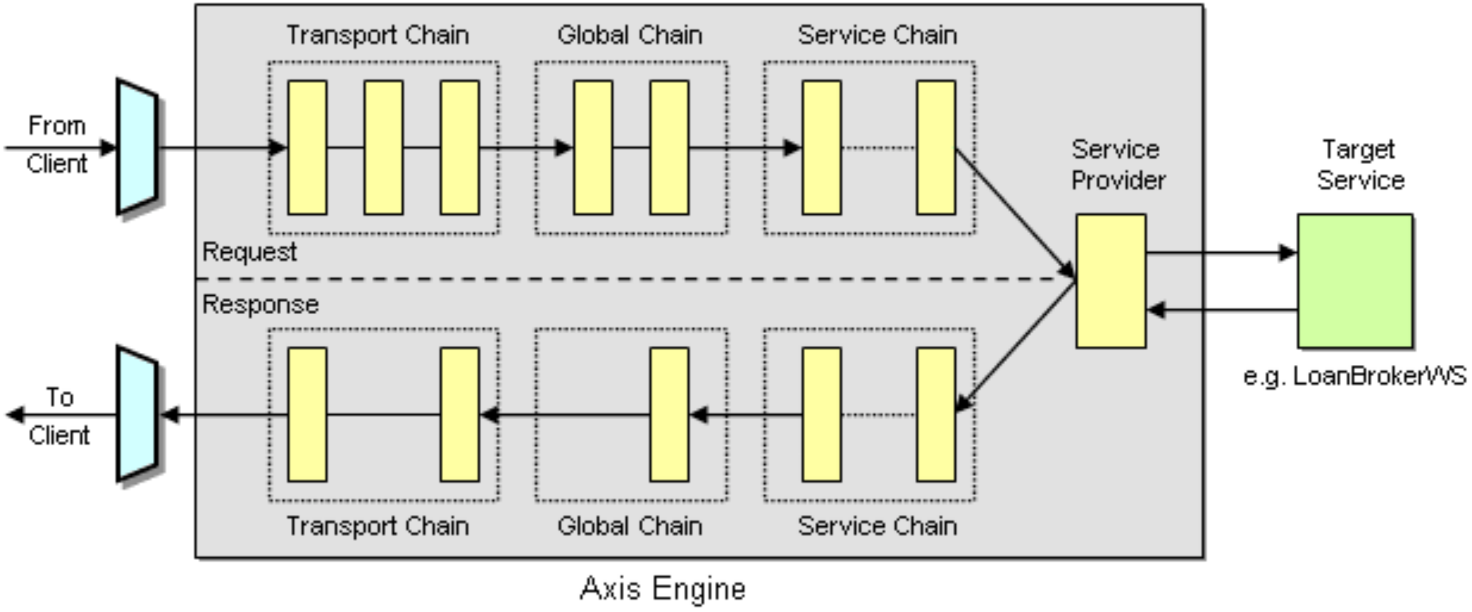
Call local or cloud infrastructure APIs

An Express Route and its Handlers

A route can have multiple handlers, each with the potential of sending back a HTTP response or passing on the request to the next handler in the route stack.



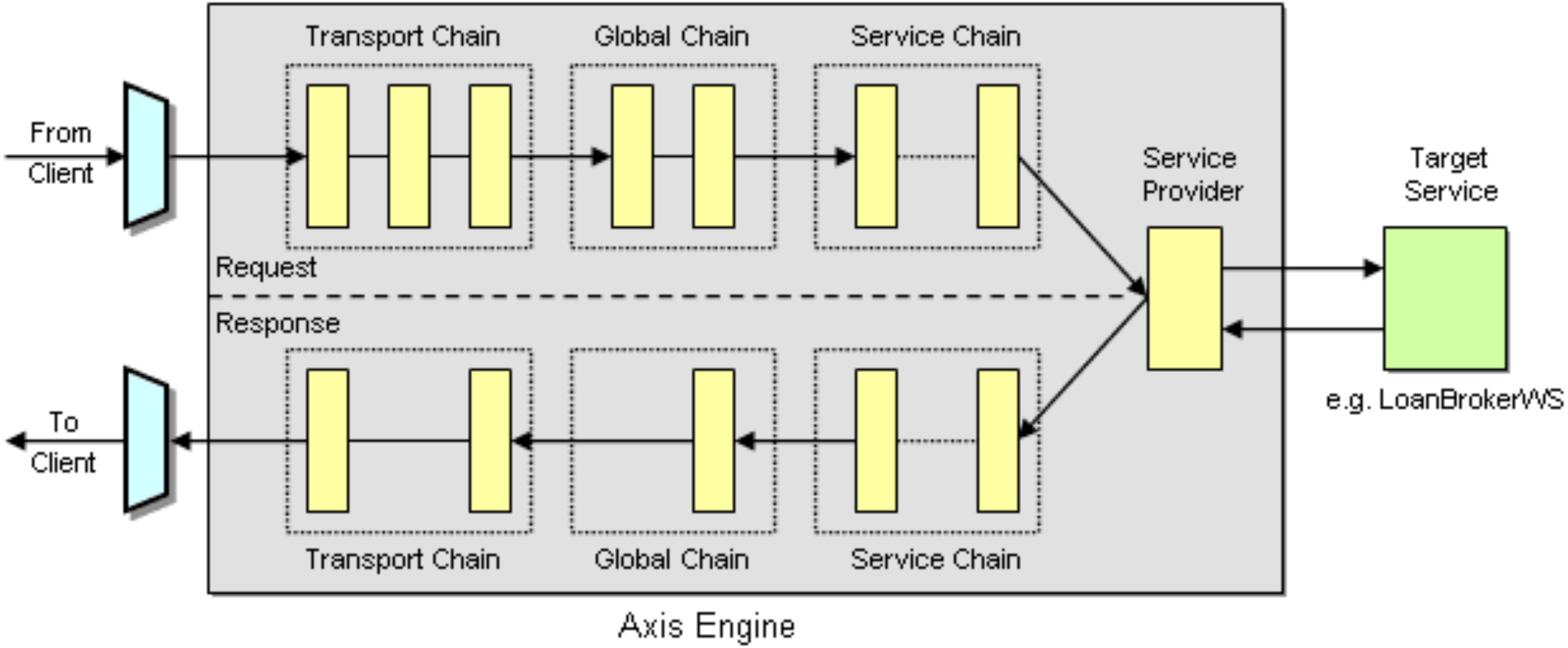
Apache Axis Example



The *before* and *after* handlers/middleware interceptors

- *May* perform inline processing of request, e.g. validate signature and/or
- *May* call an external service API, e.g. logging, check an ETag in a ledger, etc.

Project


Transport Chain
Global Chain
Service Chain
From Client
Request
Service Provider
Target Service
Response
To Client
e.g. LoanBrokerWS
Transport Chain
Global Chain
Service Chain
Axis Engine

Publish Subscribe

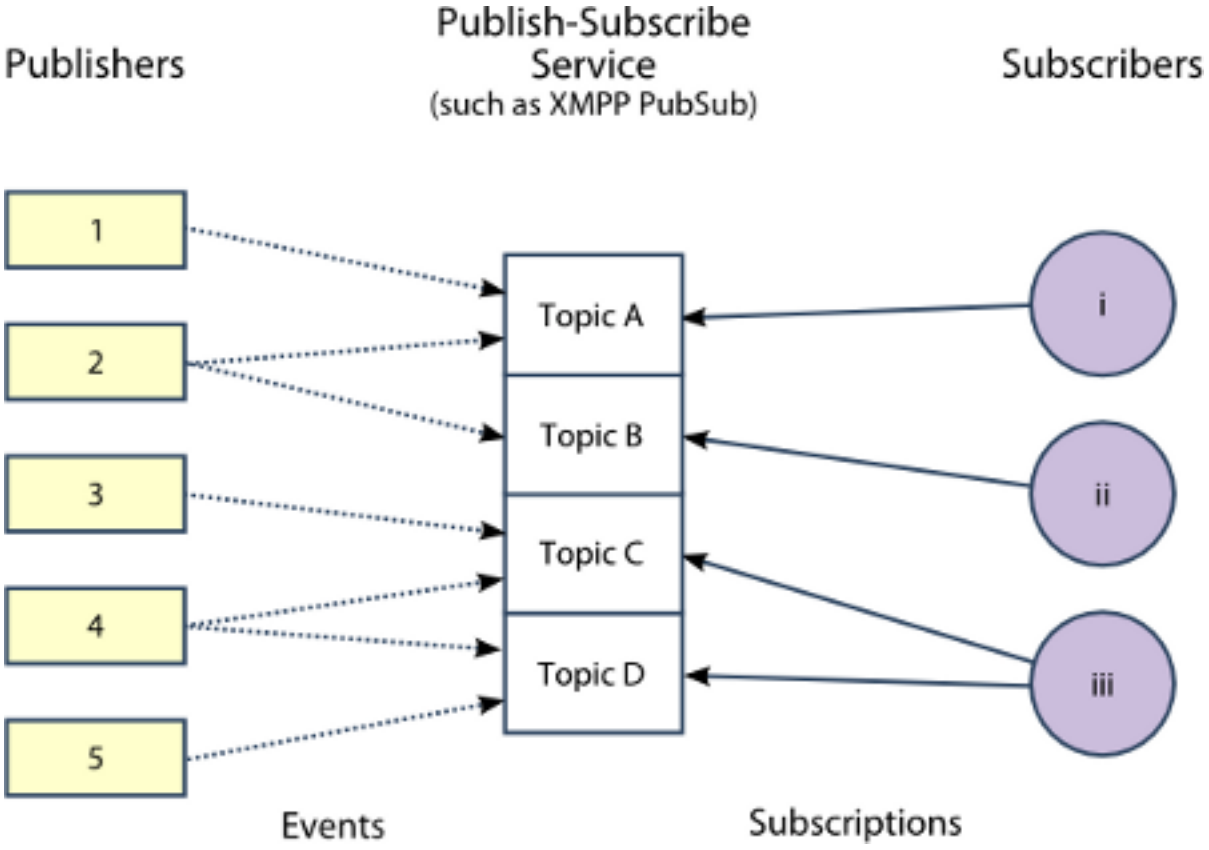
Publish/Subscribe (Wikipedia)

"In software architecture, **publish–subscribe** is a messaging pattern where senders of messages, called publishers, do not program the messages to be sent directly to specific receivers, called subscribers, but instead categorize published messages into classes without knowledge of which subscribers, if any, there may be. Similarly, subscribers express interest in one or more classes and only receive messages that are of interest, without knowledge of which publishers, if any, there are.

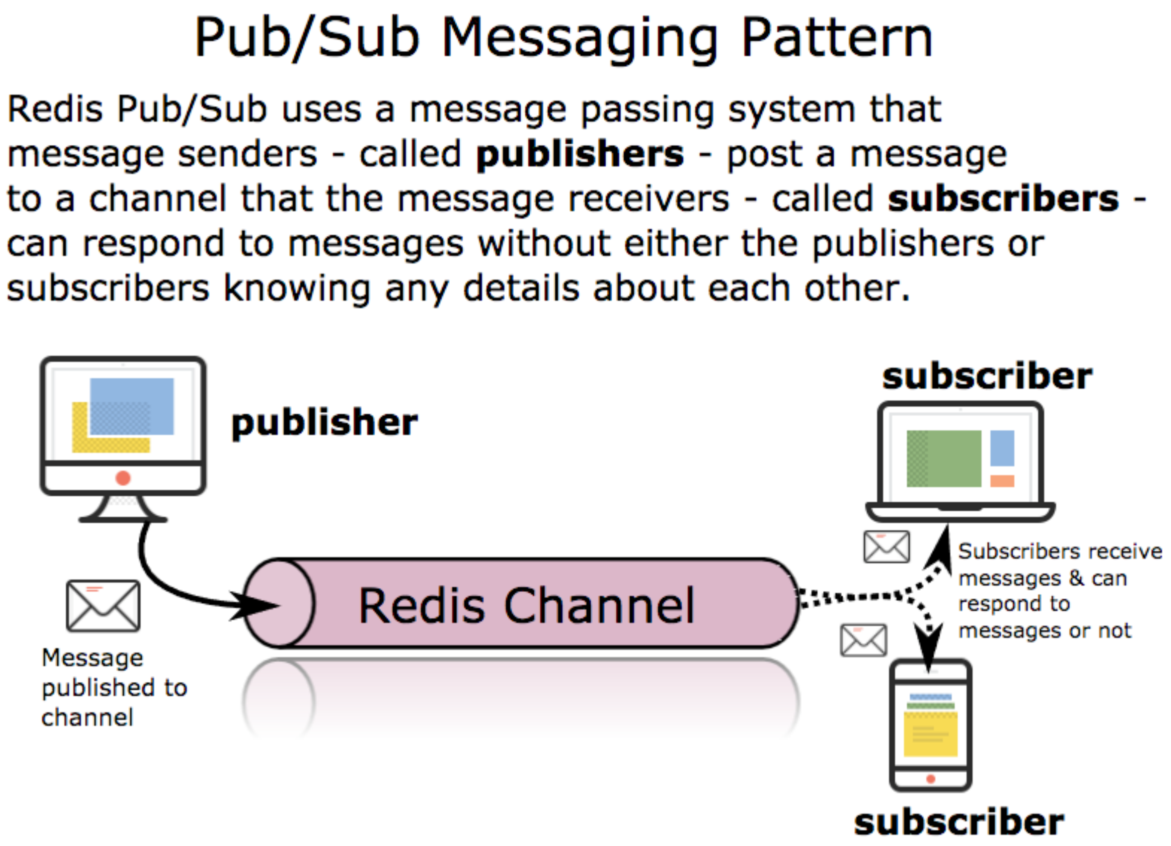
Publish–subscribe is a sibling of the message queue paradigm, and is typically one part of a larger message-oriented middleware system. Most messaging systems support both the pub/sub and message queue models in their API, e.g. Java Message Service (JMS).

This pattern provides greater network scalability and a more dynamic network topology, with a resulting decreased flexibility to modify the publisher and the structure of the published data."

Publish/Subscribe



Redis Example

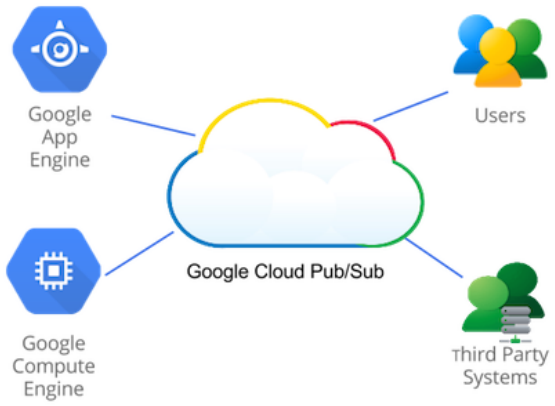


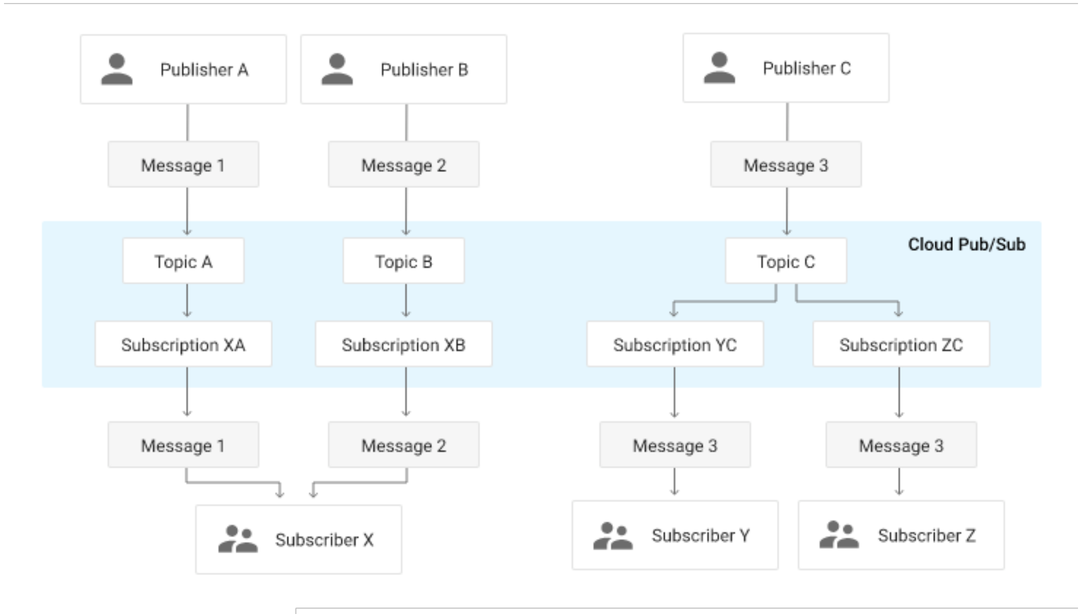
Pub/Sub Messaging Pattern

Redis Pub/Sub uses a message passing system that message senders - called **publishers** - post a message to a channel that the message receivers - called **subscribers** - can respond to messages without either the publishers or subscribers knowing any details about each other.

publisher
Message published to channel
Redis Channel
subscriber
Subscribers receive messages & can respond to messages or not
subscriber

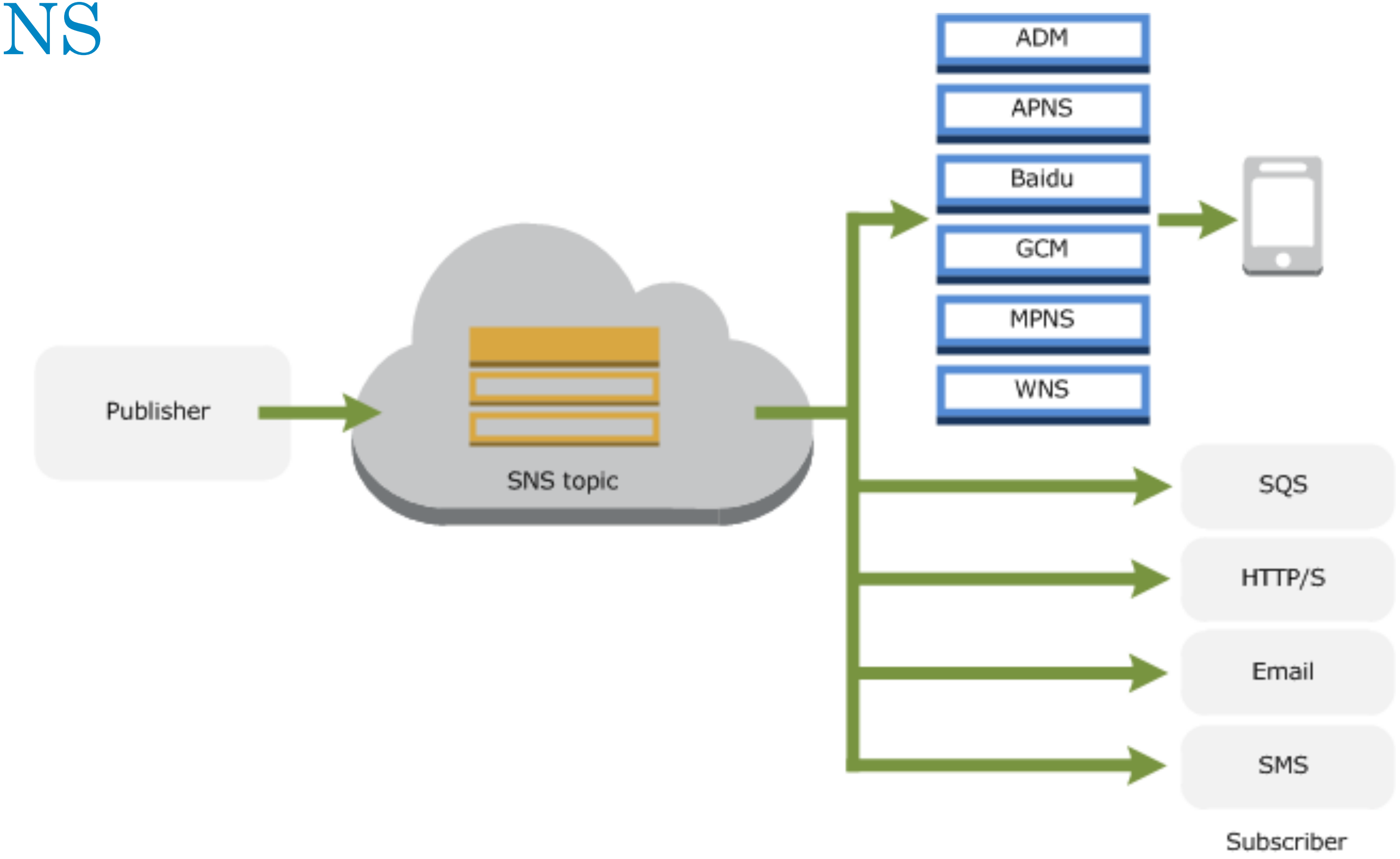
Google Cloud Pub/Sub





https://cloud.google.com/pubsub/overview

SNS


ADM
APNS
Baidu
GCM
MPNS
WNS
Publisher
SNS topic
SQS
HTTP/S
Email
SMS
Subscriber

API and Demo

Walkthrough and Demo

Message Queue

Message Queueing (Wikipedia)

“Message queues provide an asynchronous communications protocol, meaning that the sender and receiver of the message do not need to interact with the message queue at the same time. Messages placed onto the queue are stored until the recipient retrieves them. Message queues have implicit or explicit limits on the size of data that may be transmitted in a single message and the number of messages that may remain outstanding on the queue.

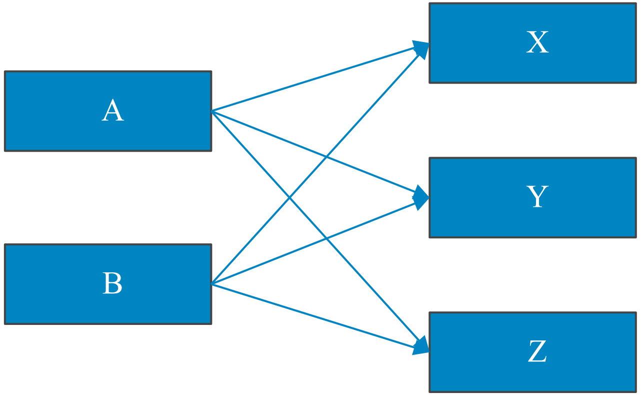
… …

There are often numerous options as to the exact semantics of message passing, including:

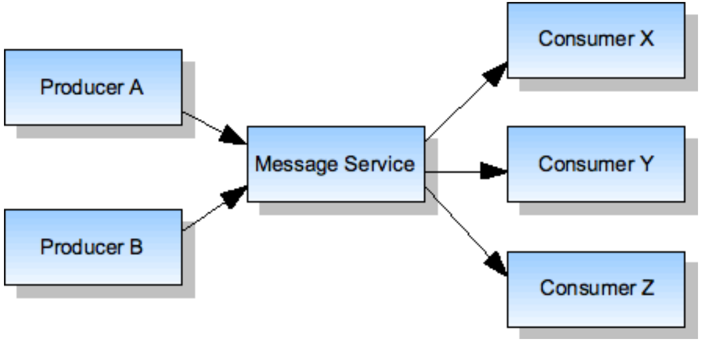
- Durability - messages may be kept in memory, written to disk, or even committed to a DBMS if the need for reliability indicates a more resource-intensive solution.
- Security policies - which applications should have access to these messages?
- Message purging policies - queues or messages may have a "time to live"
- Message filtering - some systems support filtering data so that a subscriber may only see messages matching some pre-specified criteria of interest
- Delivery policies - do we need to guarantee that a message is delivered at least once, or no more than once?
- Routing policies - in a system with many queue servers, what servers should receive a message or a queue's messages?
- Batching policies - should messages be delivered immediately? Or should the system wait a bit and try to deliver many messages at once?
- Queuing criteria - when should a message be considered "enqueued"? When one queue has it? Or when it has been forwarded to at least one remote queue? Or to all queues?
- Receipt notification - A publisher may need to know when some or all subscribers have received a message.

Message Queue Service Pattern

Anti-Pattern



Best Practice Pattern

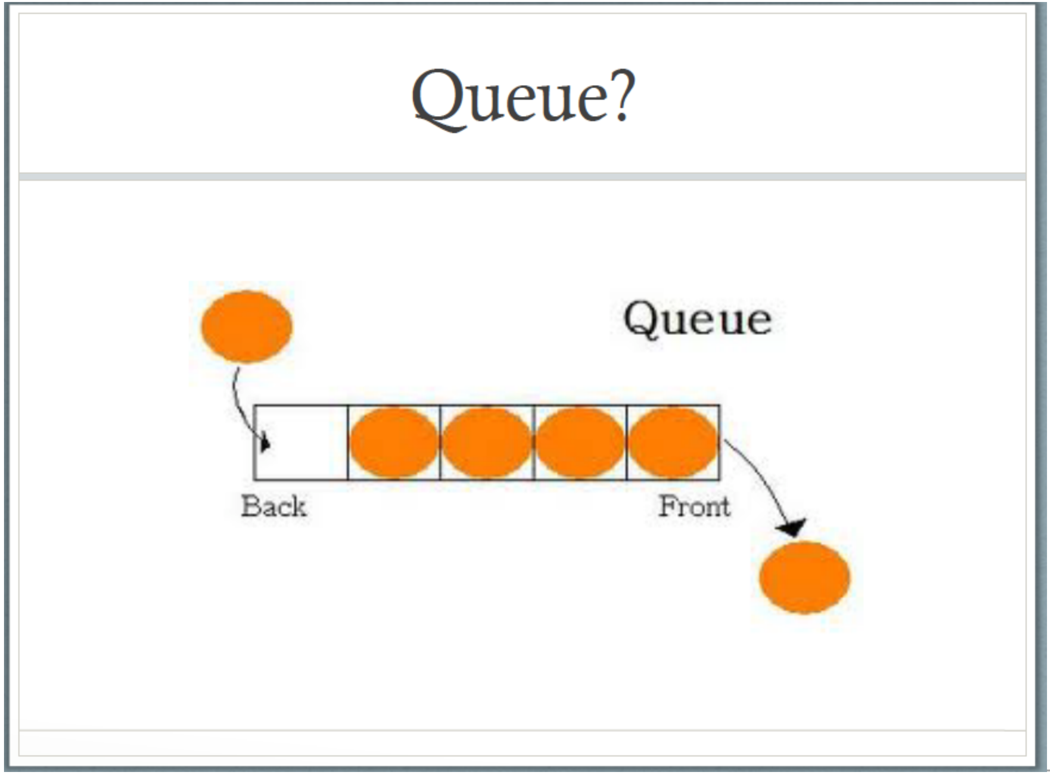


Point-to-Point communication between modules becomes fragile at scale

- Adding modules P and Q requires finding and configuring all senders.
- I want to send M to any one of X1, X2 and X3, but only one.
- I want to make sure that someone processes M but do not want to hold the transaction until I get a response (I may not even need the response).
- My destinations are not "up" at the same times I am.

SQS

http://www.slideshare.net/sandipkchaudhari/sqs-and-loose-coupling-presentation



Queue?

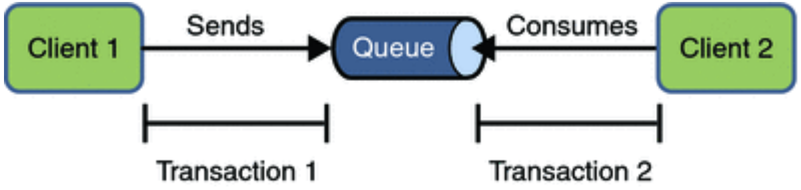
Queue Operations

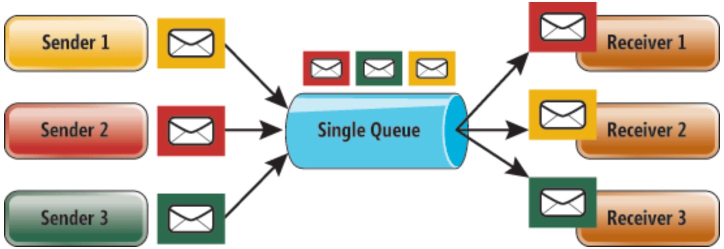
- Create Q
- Delete Q
- Send Message
- Receive Message
- Delete Message

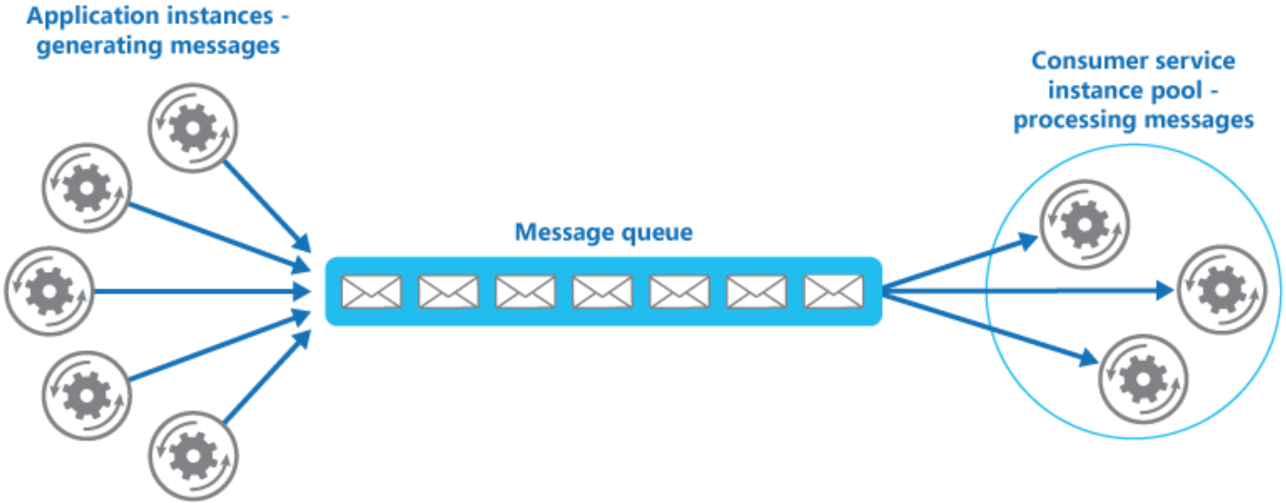
FIFO Semantics

- Strict, limited scalability.
- Best effort, highly scalable.
- Transactional

Message Exchange Patterns





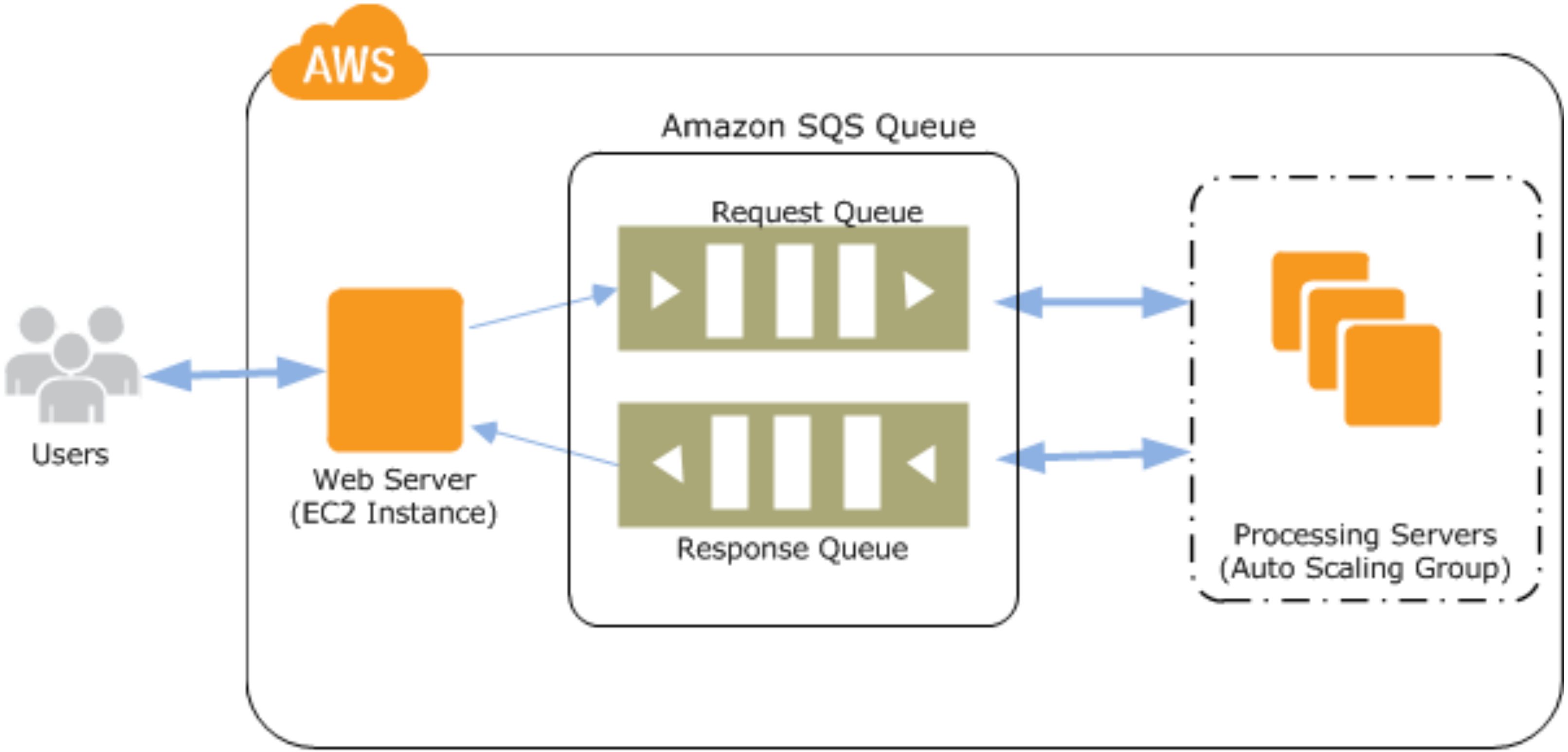

Application instances - generating messages
Message queue
Consumer service instance pool - processing messages

Why Would I Use a Queue

http://www.javacodegeeks.com/2013/06/working-with-amazon-simple-queue-service-using-java.html is one example)

- Flow control
  - My server may be able to process 100 reqs/second
  - There are thousands of clients
  - The aggregate rate could be 10 req/s or 300 req/s
  - Without a queue, clients would get connect failures under load
- The request is going to take a long time
  - Image reformatting and tagging
  - I cannot have my code "wait" for a response
  - And I do not want to write a lot of "Is it done yet?" calls
- I do not know (or care) which one implements the logic
  - I have 5 – 10 image processors for GIFs
  - I have 7 – 10 processors for .wav files
  - I do not want to know which ones are active, when and processing what.
  - I just want to put the "thing in a queue" knowing that someone will eventually pick it up.
- Think "email" for API calls

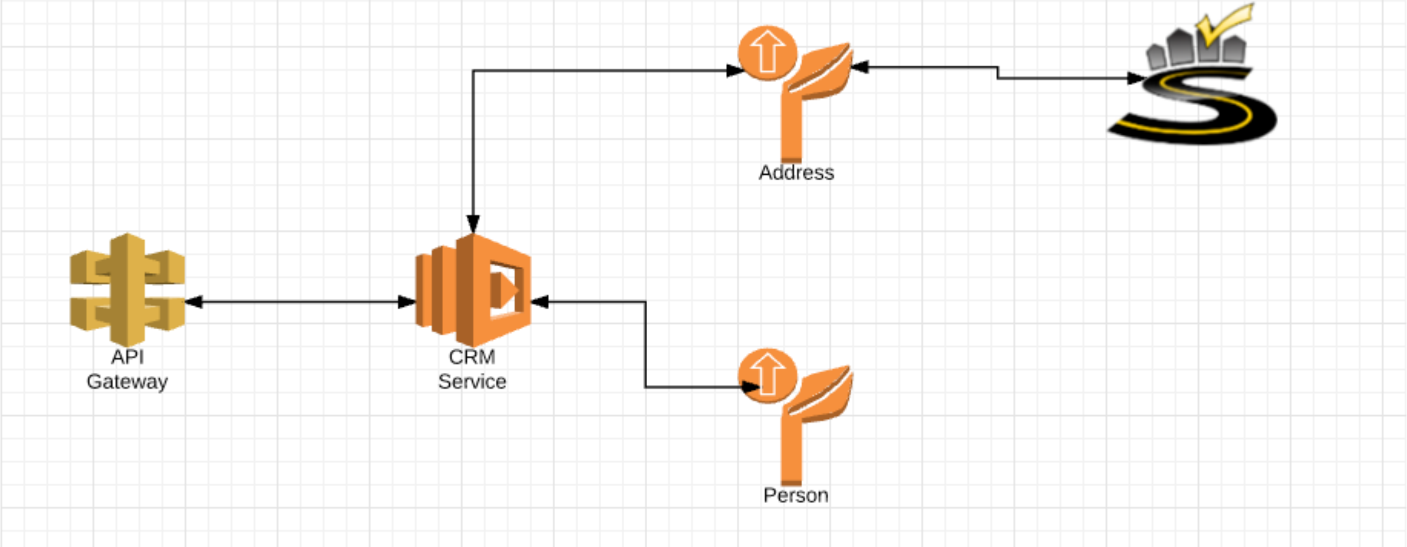
Amazon SQS – Example



API and Demo

Project 2

Project 2 – Part 1



Build on 1st to microservices

- Implement the CRM Service (and HR Service) using Lambda functions, and orchestration approach from previous lecture.
- Integrate with SmartyStreets
- Deploy all 3 microservices via API Gateway.

- Deliver web content via CloudFront and S3.

Project 2 – Part 2

- Implement middleware plugins and microservices implementing
  - ETag generation and processing.
  - Idempotent functions.
  - SNS Event generation on PUT, POST and DELETE.
- Plug/deploy the plugin layer in
  - Each of the Beanstalk Microservices
  - The CRM Lambda function.
- Write an empty, placeholder Lambda function that reacts to the SNS events. We will do some interesting things with this later.