COMSE6998: Modern Serverless Cloud Applications

Lecture 10: Asynchronous APIs, Message Queueing, Pub/Sub

Dr. Donald F. Ferguson
Donald.F.Ferguson@gmail.com

Contents

Contents

- Course status
  - Lecture schedule and topics.
  - Project schedule and grading.
- New technical topics
  - Configuring credentials for code.
  - CAP Theorem and eventual consistency.
  - Event driven
    - Asynchronous REST responses.
    - Callbacks.
    - Webhooks.
    - Message queuing and SQS.
    - Pub/Sub and SNS.
- Final project

Course Status

Course Status

- Lecture schedule
  - 01-Dec-2016: Final project definition. Some new topics.
  - 08-Dec-2016: General Q&Q, project questions, “chalk talk” on security.

- Projects:
  - Complete reviews of projects 2 and 3 by 08-Dec. Will publish in-person and web times.
  - Project 4:
    - I have to submit grades by 02-Jan-2017.
    - Would like to complete all reviews by 23-Dec-2016, but will do some after Christmas.
  - Each project will count for 25% of grade.

- Next spring
  - I think I am teaching next Spring, if dept. approves class proposal. Will post topics if/when department approves.
  - Continuation of topics in serverless, *aaS and micro-services.
  - Will cover **new content**. The current class is **not** a prerequisite.

New Technical Topics

Configuring Credentials

Credentials for Calling APIs

- Server/Function "calls" other servers/functions, e.g.
  - MySql
  - Smartystreets.com
  - Digital asset

```
var mysql      =   require("mysql");


//First you need to create a connection to the db
var con = mysql.createConnection({
    host: "sparq-[redacted]",
    user: "spa[redacted]",
    password: [redacted]"
});

const episode_comment_count =   'SELECT episode_name, count FROM reporting.fpse_names_comment_count order by count';

var     root_callback;

var     root_context;

var     query_callback  = function(err, data) {
    if (err) {
        root_context.fail(err);
    }
    else {
        console.log("Query result = " + JSON.stringify(data));
        root_context.succeed(data);
    }
};
```

- Calling code
  - Needs credentials.
  - But no one is logged on and app is running unattended.

- I had you put creds in code for simplicity, but this is
  - Security exposure.
  - Config. exposure.

```
/*
 * These are Brightcove supplied/generated API key and API secret key.
 * The Brightcove API expects this information in the HTTP
 * authentication header in a specific format. Using the API requires
 * exchanging these credentials to retrieve shorter-lived Access Tokens.
 * (https://docs.brightcove.com/en/video-cloud/media-management/guides/
 * authentication.html)
 */
public static String username = "[redacted]";
public static String password = "[redacted]";
```

Credentials in Code

- Security exposure
  - In theory, the remote server or function is “protected.”
    - Firewall.
    - “Compiled code.”
    - Logon to access code, config files, etc.
  - But, the code is *also* in/on
    - Your development laptop.
    - Source code control system.
    - Part of collaborative development processes.
- Configuration exposure –
  - May to access credentials from
    - Several different functions.
    - Modules of several different micro-services.
    - etc.
  - Must be kept in synch with code lifecycle changes, even when not all modules are changed.

AWS Solution

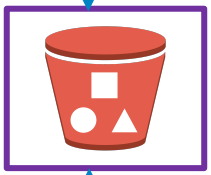
Which implements a general pattern, that you do not have to worry about.

Admin

- Logs on in IAM role.
- Edits info “in situ.”

Configure “run as” IAM Role

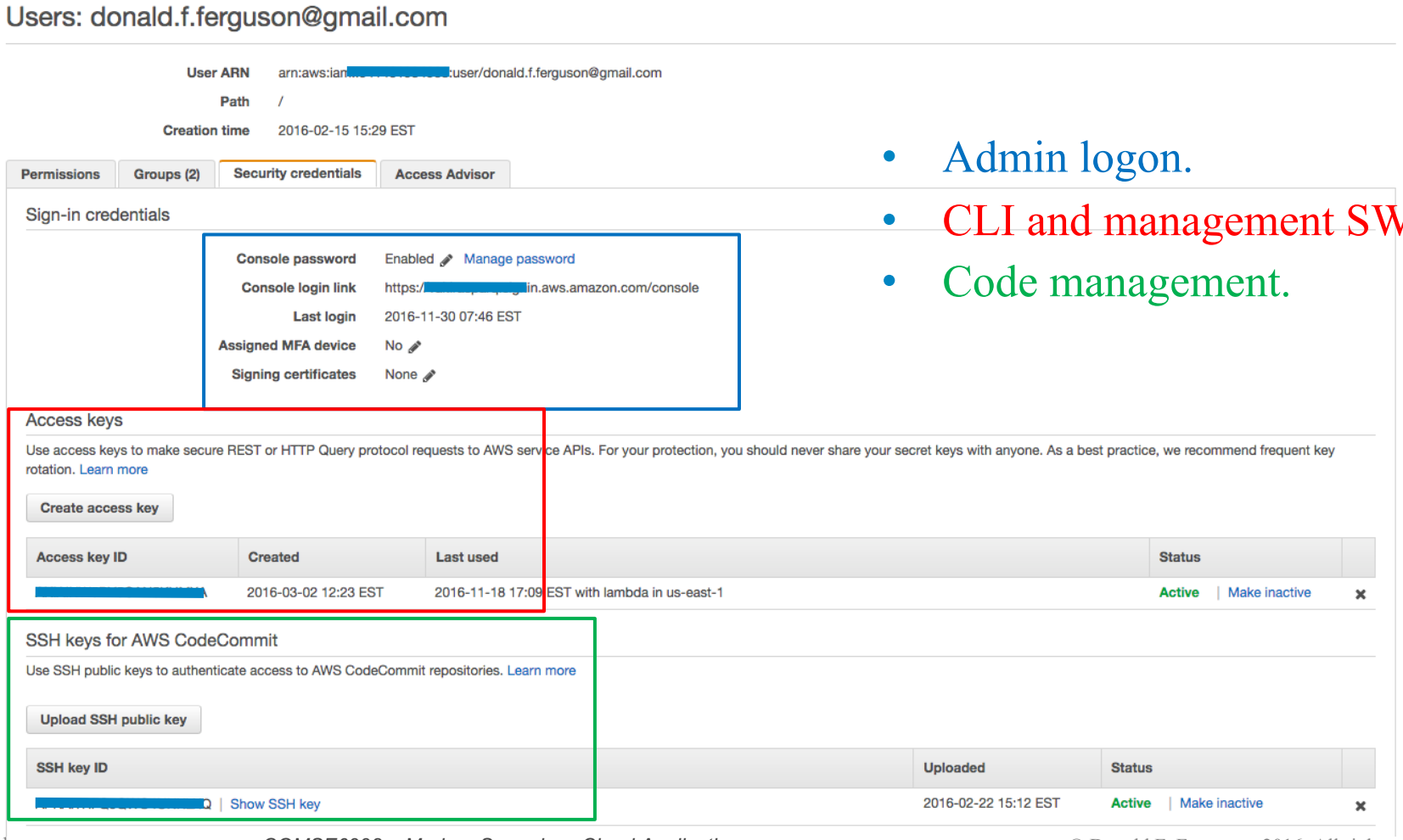
Configure IAM for access.



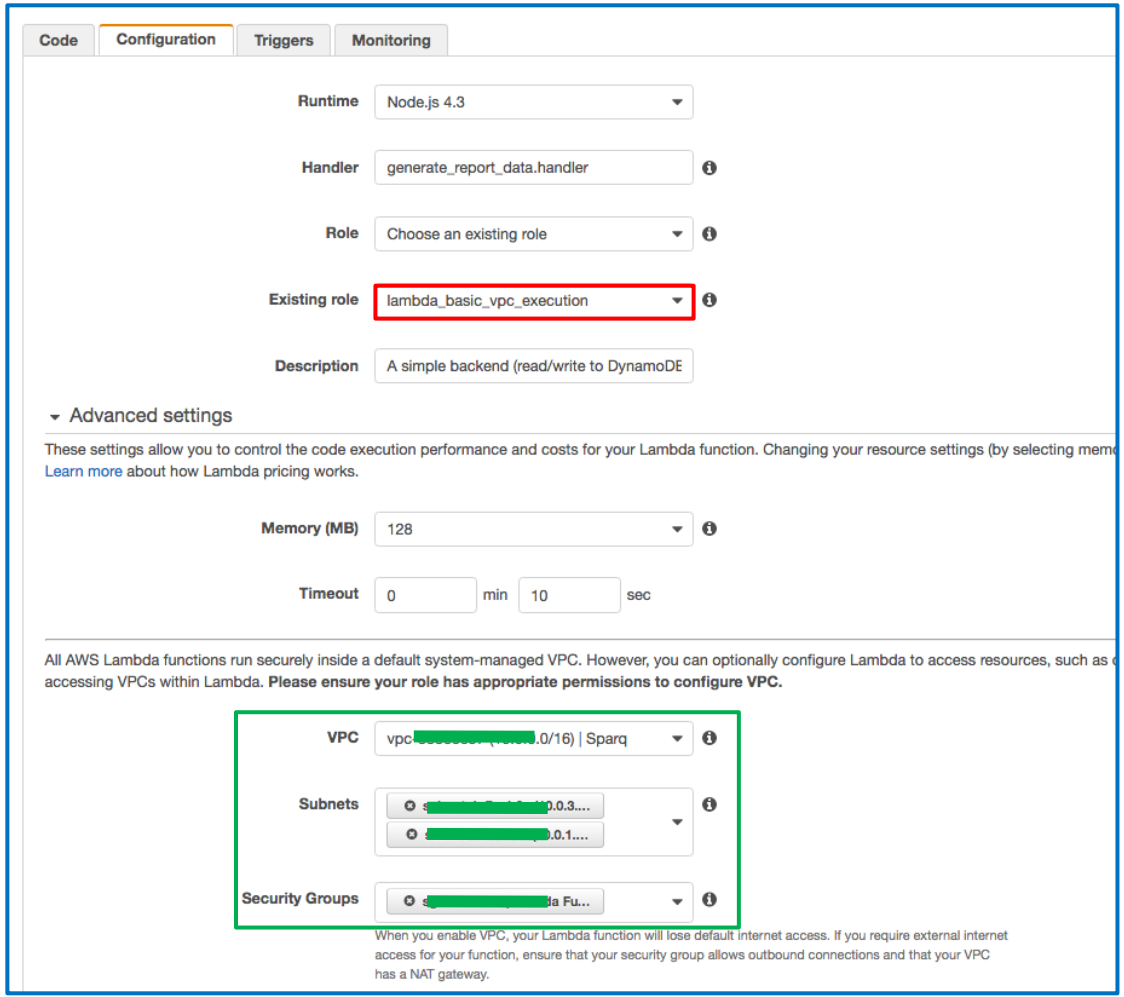
Read S3 to retrieve credentials.

AWS propagates IAM role on access.

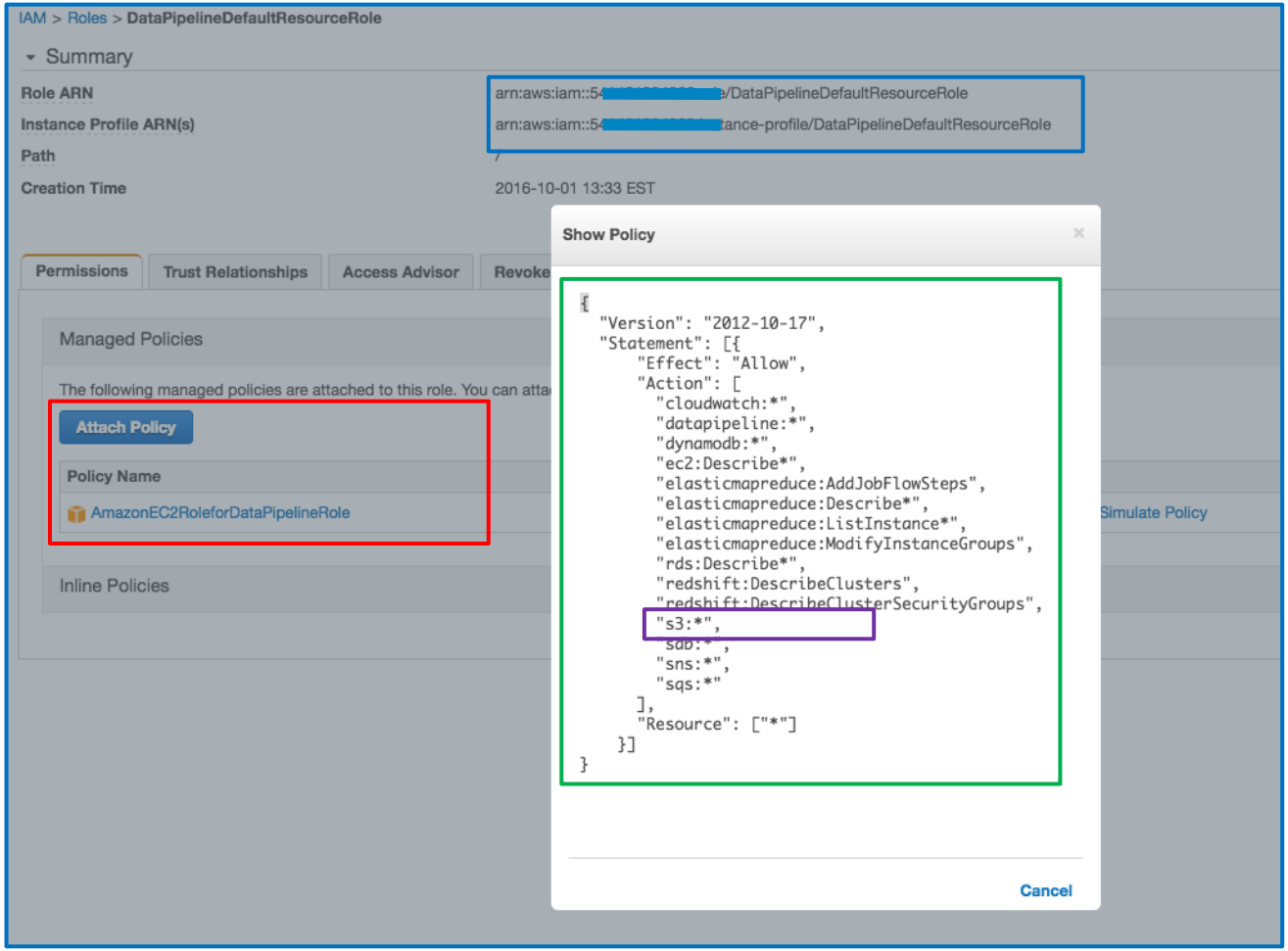
Use credentials for access.



- Admin logon.
- CLI and management SW.
- Code management.



- Configure Lambda
- Assign Roles
- Assign
  - VPC subnets
  - Security groups



- IAM Role
- Enabling Policies
- Policy
- Specific
  - Operations
  - On resources
    - Type
    - Group
    - Patterns
    - Instances

CAP “Theorem” Eventual Consistency

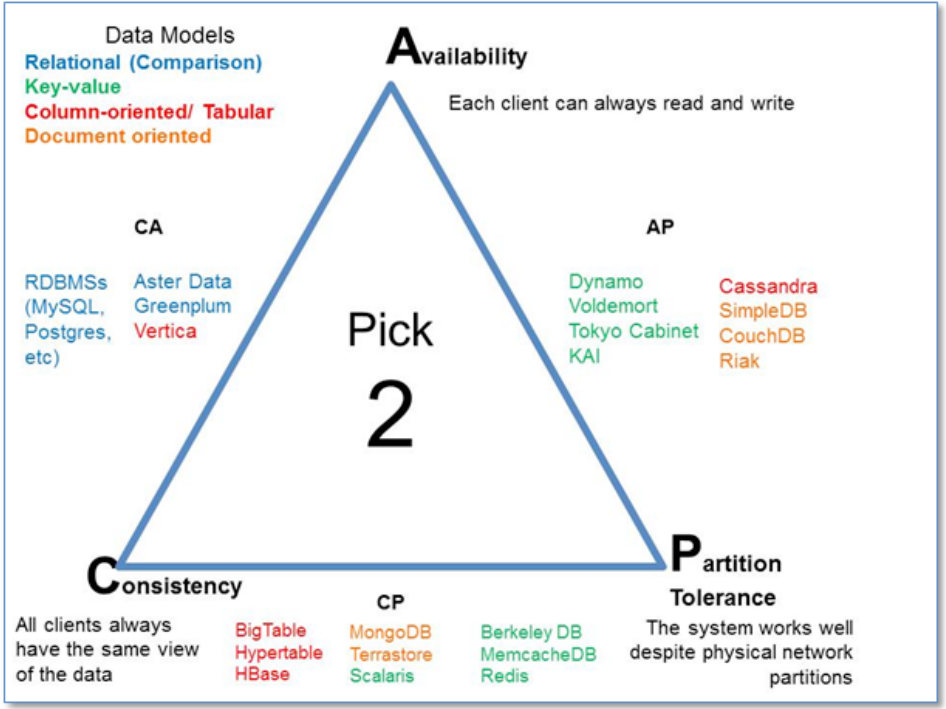
CAP "Theorem"

https://en.wikipedia.org/wiki/**CAP_theorem**

Any database supports **at most** 2 of

- Availability
- Consistency
- Partition Tolerance

- Choose the specific database based on
  - Functionality
  - Required elements of CAP

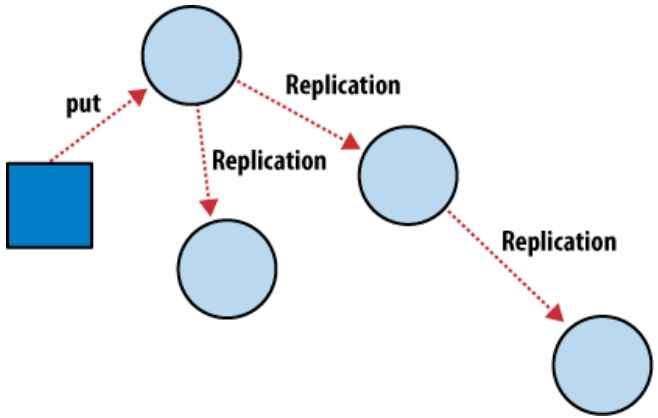


Guarantees	Definition
Consistency	Every read receives the most recent write or an error
Availability	Every request receives a response, without guarantee that it contains the most recent version of the information
Partition tolerance	The system continues to operate despite an arbitrary number of messages being dropped by the network between nodes

Eventual Consistency https://en.wikipedia.org/wiki/Eventual_consistency

Eventual consistency is a **consistency** model used in distributed computing to achieve high availability that informally guarantees that, if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value.

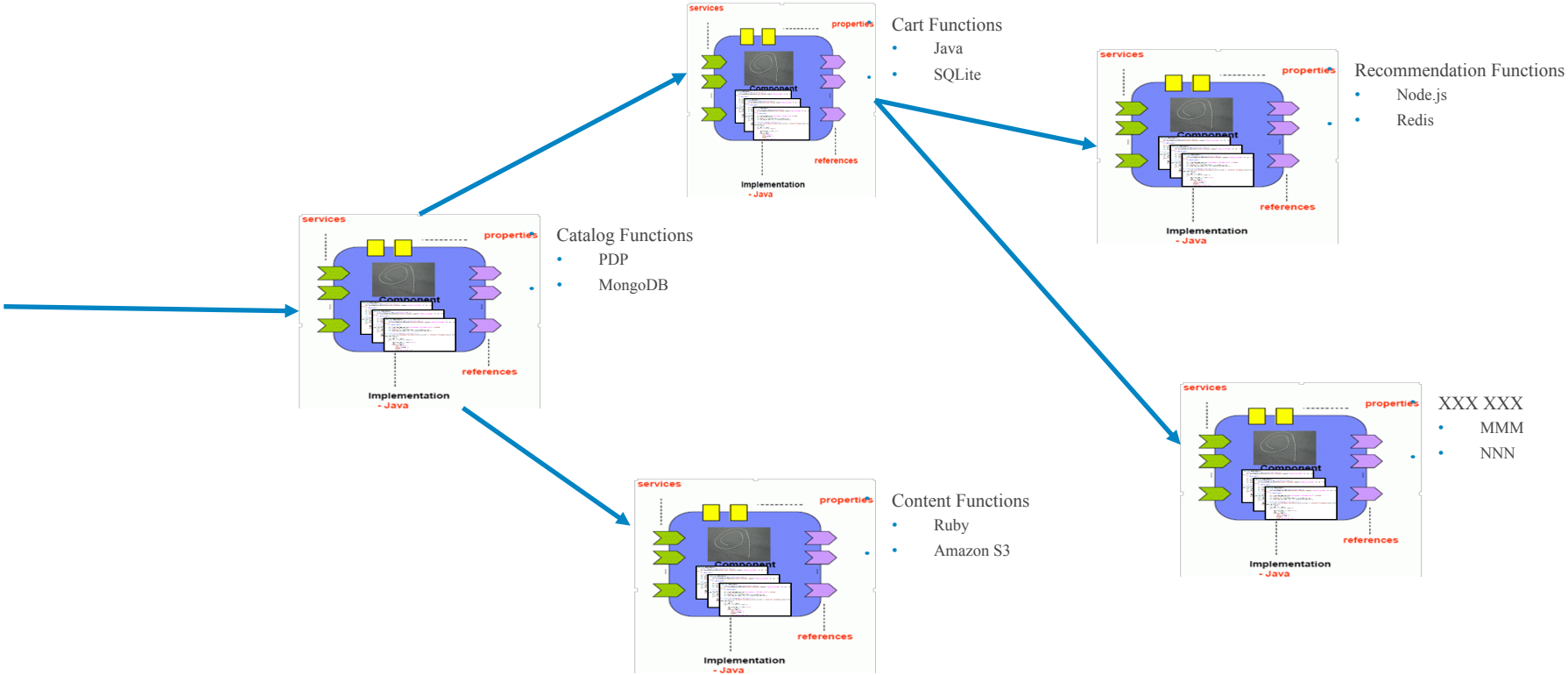
- Traditional databases are are “instantly consistent,” e.g. RDBMS by using
  - Locking
  - Transactions
  - Isolation
- Instantly, immediately consistent is “impossible to achieve” for
  - Massive scale.
  - Asynchronous, event and message driven applications even if the DB supports instant consistency.
  - And is often unnecessary because applications needs to handle consistency errors.



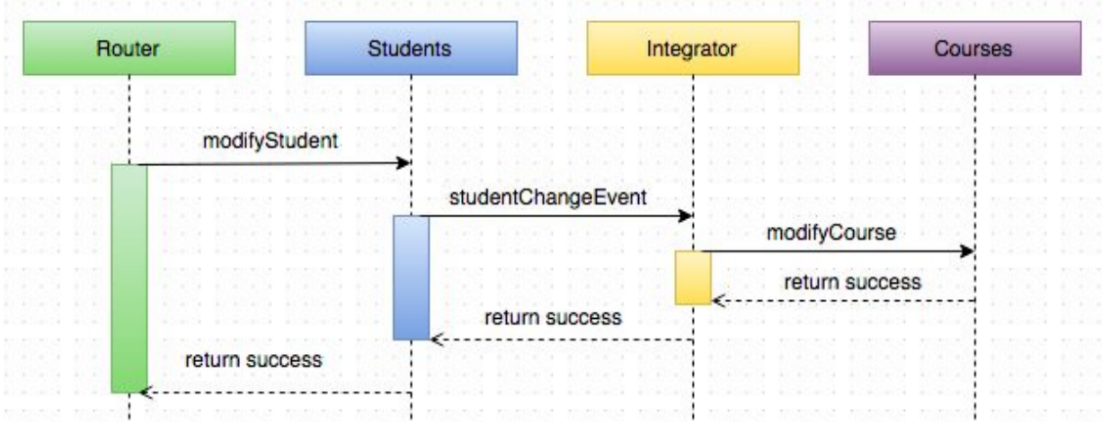
Asynchronous Operations and Message Queues

Asynchronous Operations

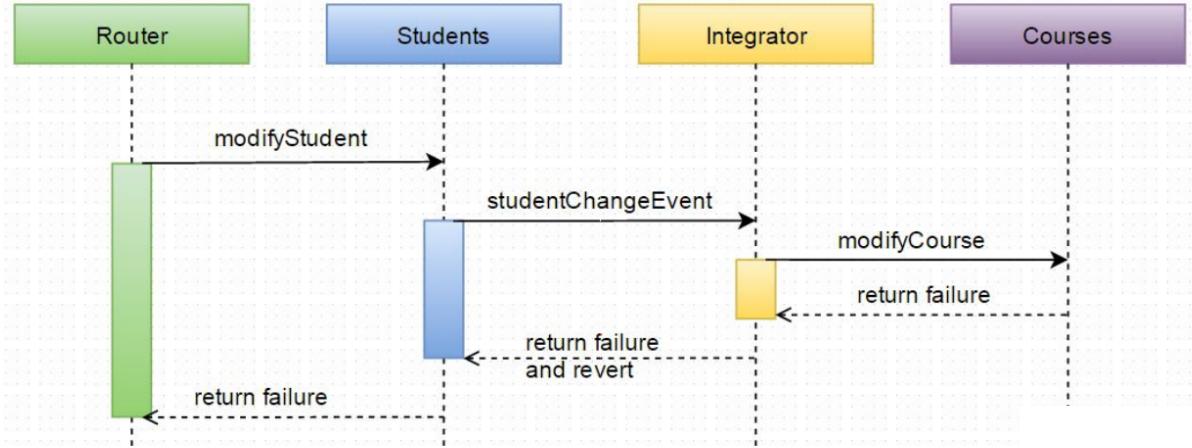
REST Implementations Call REST Services



Synchronous Integration



However, if that request fails when updating a course in Courses, the sequence looks as follows:



Synchronous and Asynchronous

- HTTP operations are inherently *synchronous*
  1. Connect
  2. Verb
  3. Response
  4. Disconnect

  Single Request
- The implementation of the operation may "take a while,"
  - A calls B before returning. B calls C and D before returning to A. …
  - The implementation may be computationally or data intensive, e.g. DB query, image transcoding, ...
  - Something in the implementation may operate at "people speed," e.g.
    - Creating an Account may require
    - Manual approval by "Dave."
- Holding open connections can cause problems
  - A server typically has an upper limit on open connections. Holding a connection for long requests will cause some client calls to "get a busy signal."
  - A connection is more likely to break the longer it is open. A call graph of "long" connections is inherently fragile. One breaks and the whole thing breaks.
  - In "in flight" data access may prevent other operations and applications from accessing data.

In the worst scenario, for large multi-service applications, EVERY end-to-end request fails.

Asynchronous Operation

Request:

This is going to "take a while."

```
POST /blogs HTTP/1.1
<xml>
    blogdata
</xml>
```

Response:

```
HTTP/1.1 202 Accepted
Location: /queue/12345
```

So, I am going to

- Tell you that your request looks OK.
- Acknowledge that I am working on it.
- Give you a URL on which you can poll with GET for the answer

Basic Implementation Approach

- Define a *collection* /QueuedResponses
  - A client can call …/QueuedResponses/21 to get a specific response.
  - You already know how to do this for …/customer, …/address
  - The data format in the table is {id, status, JSONString}, where
    - JSONString is the response you would have received for a synchronous request.
    - Status is one of "in-progress" or "complete."
- A simple implementation would be writing a *façade*
  - Accept request
  - Create new table entry with status = "in progress"
  - Return 202 and URL
  - Call the *actual* implementation
  - Update the database table entry with the JSON result
- Most application platforms have middleware approaches to support registering callbacks, threads, etc. The implementation would typically
  - Invoke some long running action, e.g. DB query, workflow process and register a callback
  - The callback implementation updates the entry in the response table.

Callbacks
Webhooks

Register Callback on Request

```
public static String ingestVideo(String videoId, String contentInstanceId, String kind, String url) {

    String result = null;

    try {

        IngestRequest r = new IngestRequest(url, null);
        r.callbacks = new String[1];
        r.callbacks[0] =
        "https://[redacted].com/test/brightcove";

        r.callbacks[0] = Connector.getCallbackURL(videoId, contentInstanceId, kind, url);
        String requestUrl = getVideoIngestUrl(videoId);
        String response = Connector.post(r, requestUrl);

        JsonReader jrd = Json.createReader(new StringReader(response));
        javax.json.JsonObject jo = jrd.readObject();
        result = jo.getString("id");
    } catch (Exception e) {
        e.printStackTrace();
    }

    return result;

}
```

URL for POST to a Lambda Function

- "Ingestion and Rendering"
  - Takes a raw video
  - and produces additional, useful formats
    - Screen sizes
    - Thumbnails
- But can take 15 to 20 minutes.
- I need to know when completed before "marking" video active in UI/catalog.

Callback

Remote Service


```
public class BrightcoveHandler extends Handler {

    private static Logger logger = LoggerFactory.getLogger(HandlerRESTService.class);
    public static Gson prettyGson = new GsonBuilder().setPrettyPrinting().create();
    public static String resourceType = RESTServiceContext.authenticationResourceTypeName

    public  static  String  HandlerName     =   "BrightcoveHandler";

    @Override
    public void handle(RESTMessageProcessorResult result) {
        logger.error("In BrightcoveHandler, request Body = " +
                prettyGson.toJson(result.getRequest().getBody()));
        GenericTransfer inGTO;

        List<GenericTransfer>   l = result.getRequest().getBody().getTheValues();
        GenericTransfer t = l.get(0);
        java.util.LinkedHashMap<String, Object> map;
        map =   (java.util.LinkedHashMap<String, Object>) t.theData;

        logger.error("Generic Transfer = " + prettyGson.toJson(t));
        logger.error("Data Class is " + t.theData.getClass().getName());

        BrightcoveHandlerResponseTO tt = new BrightcoveHandlerResponseTO();
        tt.answer = "Hurray!";
        t.theData = tt;
//      Object o    =   t.theData;

//      JSONObject  j   =   new JSONObject(o);

            Connector.handleIgestCallback(
                map,
                result.getRequest().getContext().getQueryParameter("key"));

        return;
    }
```

Performs the action, e.g. mark video active.

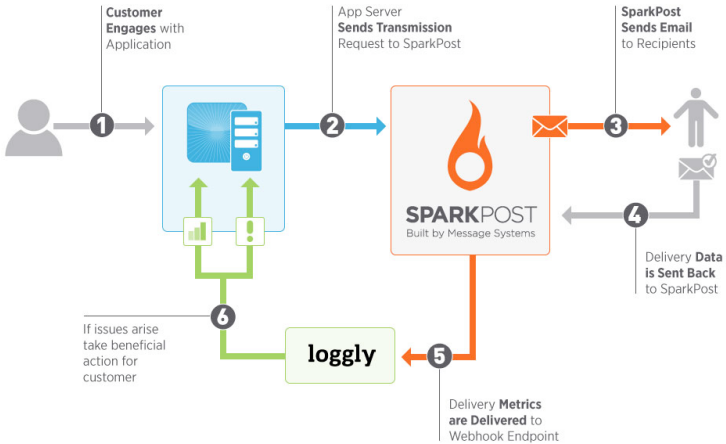
Webhooks -- https://webhooks.pbworks.com

What is a WebHook?

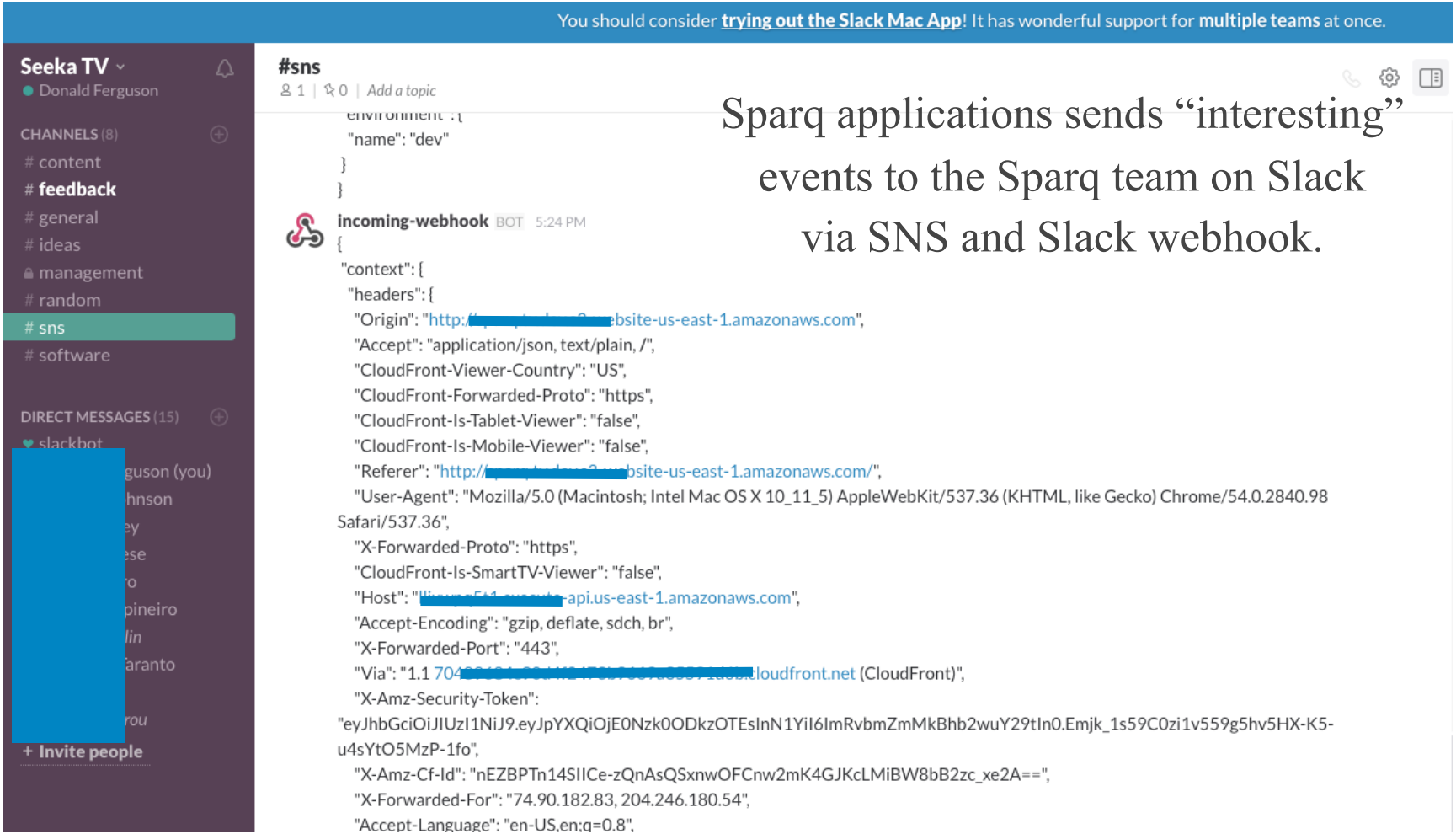
The concept of a WebHook is simple. A WebHook is an HTTP callback: an HTTP POST that occurs when something happens; a simple event-notification via HTTP POST.

A web application implementing WebHooks will POST a message to a URL when certain things happen. When a web application enables users to register their own URLs, the users can then extend, customize, and integrate that application with their own custom extensions or even with other applications around the web. For the user, WebHooks are a way to receive valuable information *when it happens*, rather than continually polling for that data and receiving nothing valuable most of the time. WebHooks have enormous potential and are limited only by your imagination! (No, it can't wash the dishes. Yet.)

Example Engine

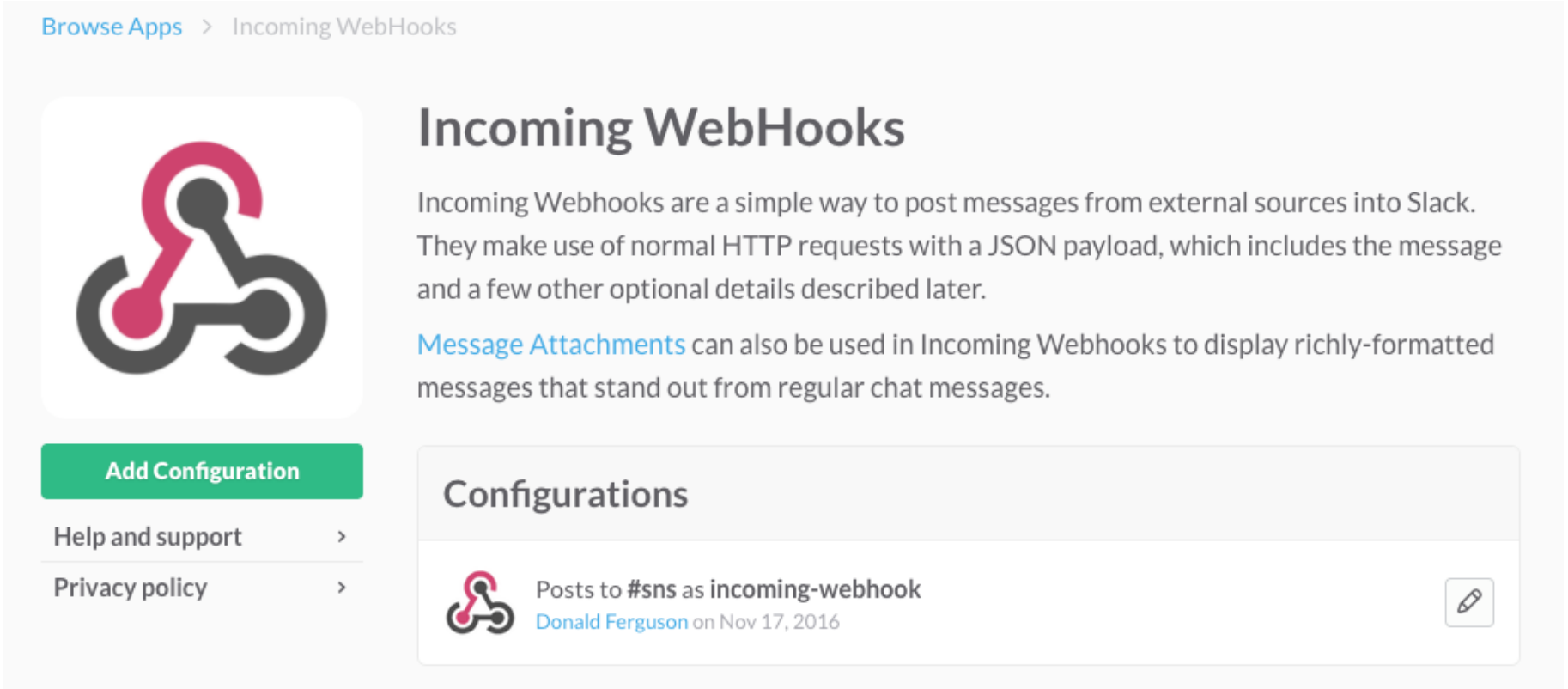


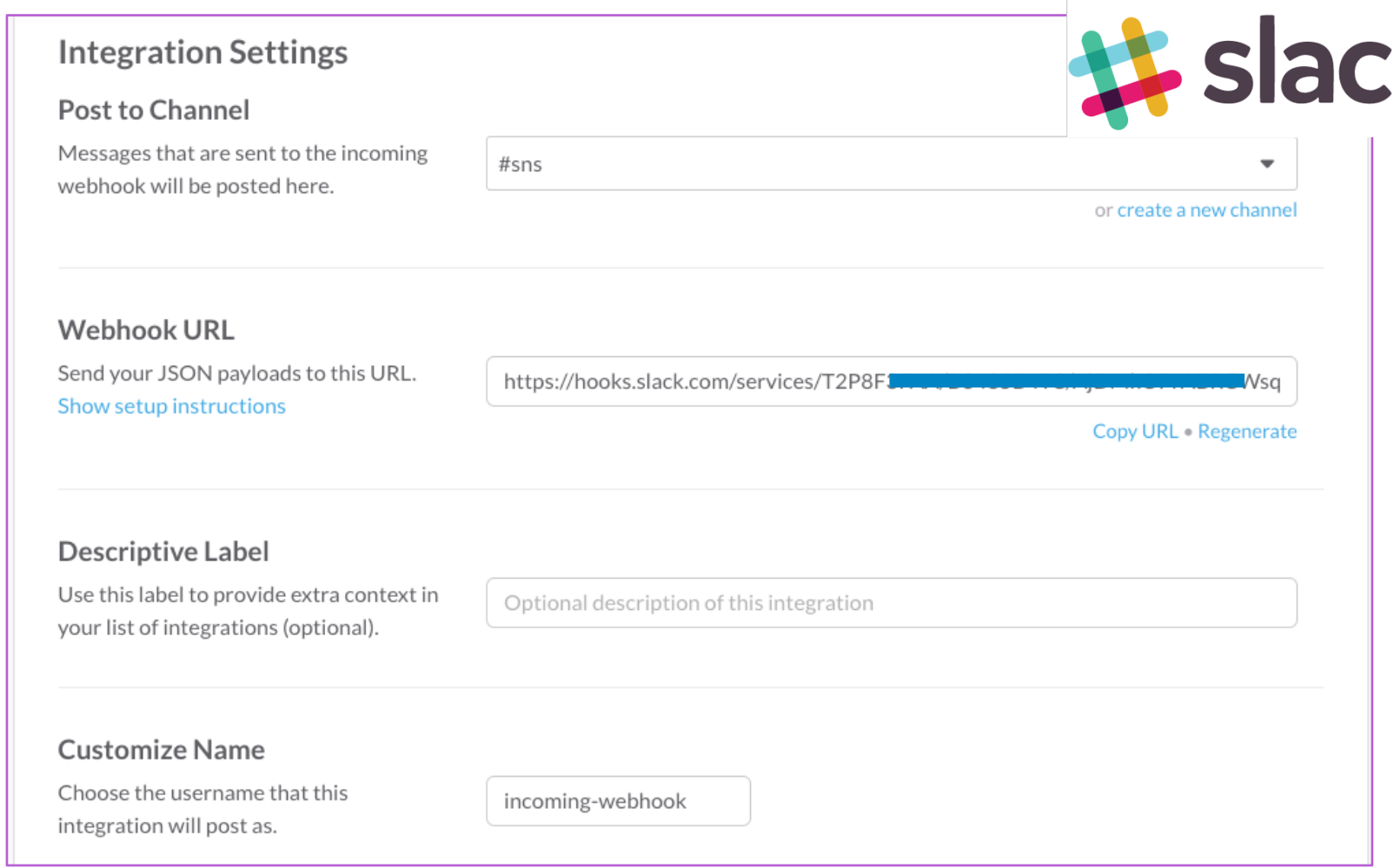
Pervasive Model



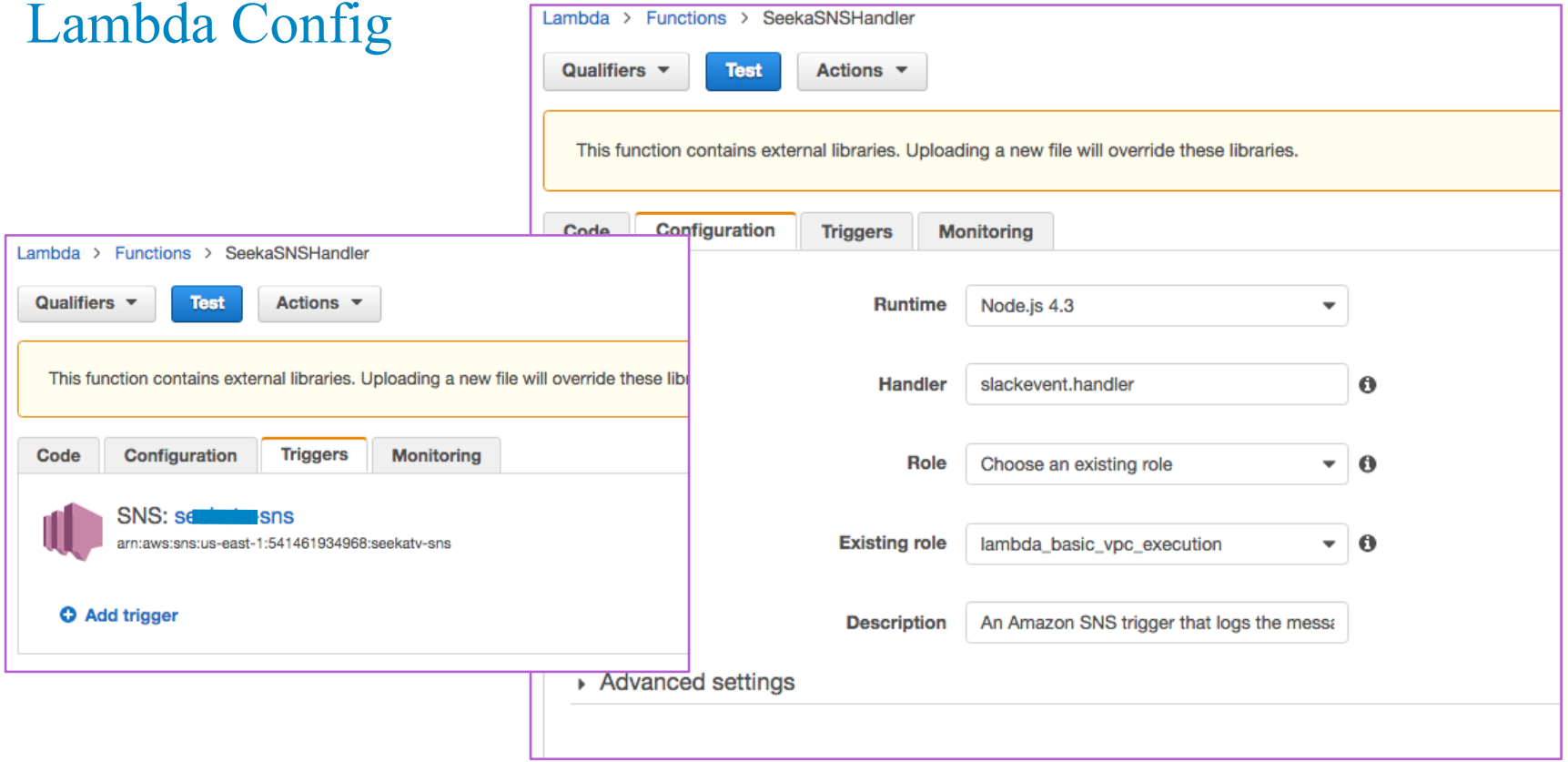
Sparq applications sends “interesting” events to the Sparq team on Slack via SNS and Slack webhook.

Slack Webhook Integration





Lambda Config



Service “XXX” shouts, “anyone interested?”


```
exports.handler = function(event, context, callback) {
    console.log('Received event:', JSON.stringify(event, null, 2));
    var message = event.Records[0].Sns.Message;
    console.log('From SNS:', message);

   // message = JSON.stringify(big_message, null, 2);

    var slack_msg = {
        text : message
    };
    send_it(slack_msg, callback);
    //callback(null, message);
};
```

Code | Configuration | Triggers | Monitoring

Code entry type: Edit code inline

```
'use strict';

console.log('Loading function');
var request = require('request');

var slack_sns = 'https://hooks.slack.com/services/T[redacted]rAE

var     send_it =   function(message, callback) {

    var data = {
            url: slack_sns,
            method: "POST",
            json: true,   // <--Very important!!!
            body: message
        };

    console.log("body = " + JSON.stringify(data.body));

    request(data, function (error, response, body){
        //console.log(response);
        console.log("Status code = " + response.statusCode);
        callback(null, message);
    });
};
```

Message Driven Processing

There are 5 principles of serverless architecture that describe how an ideal serverless system should be built. Use these principles to help guide your decisions when you create serverless architecture.

1. Use a compute service to execute code on demand (no servers)
2. Write single-purpose stateless functions
3. Design push-based, event-driven pipelines
4. Create thicker, more powerful front ends
5. Embrace third-party services

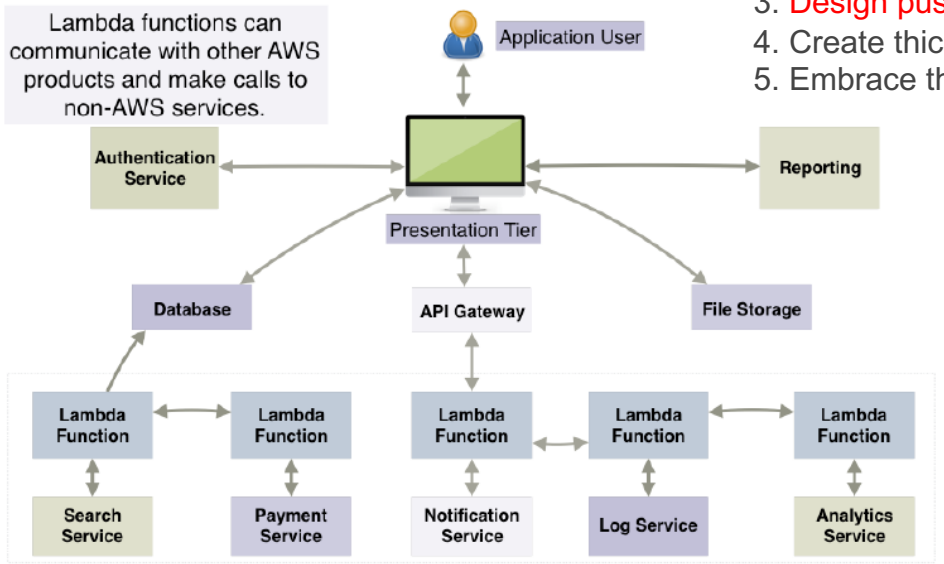


Figure 1.3: In a serverless architecture there is no single traditional back end. The front end of the application communicates directly with services, the database, or compute functions via an API gateway. Some services, however, must be hidden behind compute service functions where additional security measures and validation can take place.

Some observations on serverless

- There is running code → "some server somewhere."
- In IaaS,
  - You get the virtual sever from the cloud.
  - But know it is there, and manage and config it.
- In PaaS/microservices
  - You are aware of/build the "application sever."
  - And supporting frameworks.
  - And bundle/tarball it all together.
- In serverless,
  - You write a function based on a template.
  - Upload to an internet "event" endpoint.
  - Anything you call is a "cloud service."

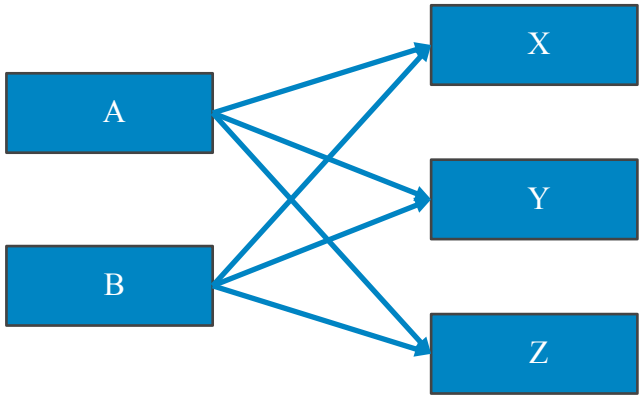
SOA vs Microservices

http://www.pwc.com/us/en/technology-forecast/2014/cloud-computing/features/microservices.jhtml

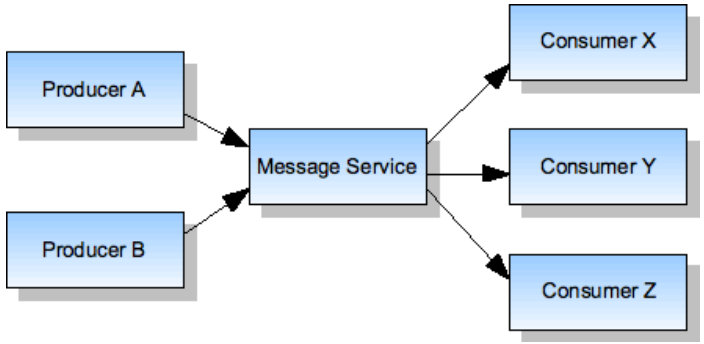
	Traditional SOA	Microservices
Messaging type	Smart, but dependency-laden ESB	Dumb, fast messaging (as with Apache Kafka)
Programming style	Imperative model	Reactive actor programming model that echoes agent-based systems
Lines of code per service	Hundreds or thousands of lines of code	100 or fewer lines of code
State	Stateful	Stateless
Messaging type	Synchronous: wait to connect	Asynchronous: publish and subscribe
Databases	Large relational databases	NoSQL or micro-SQL databases blended with conventional databases
Code type	Procedural	Functional
Means of evolution	Each big service evolves	Each small service is immutable and can be abandoned or ignored
Means of systemic change	Modify the monolith	Create a new service
Means of scaling	Optimize the monolith	Add more powerful services and cluster by activity
System-level awareness	Less aware and event driven	More aware and event driven

Message Queue Service Pattern

Anti-Pattern



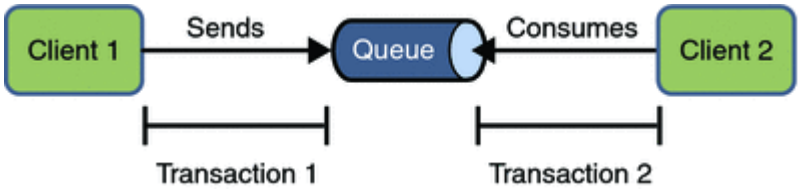
Best Practice Pattern

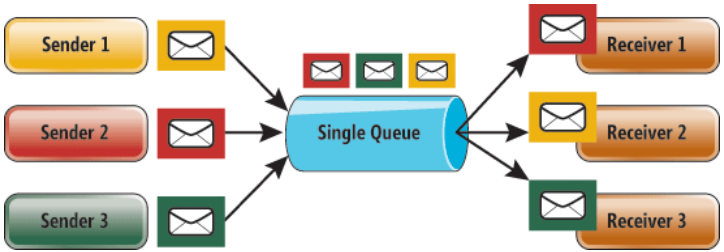


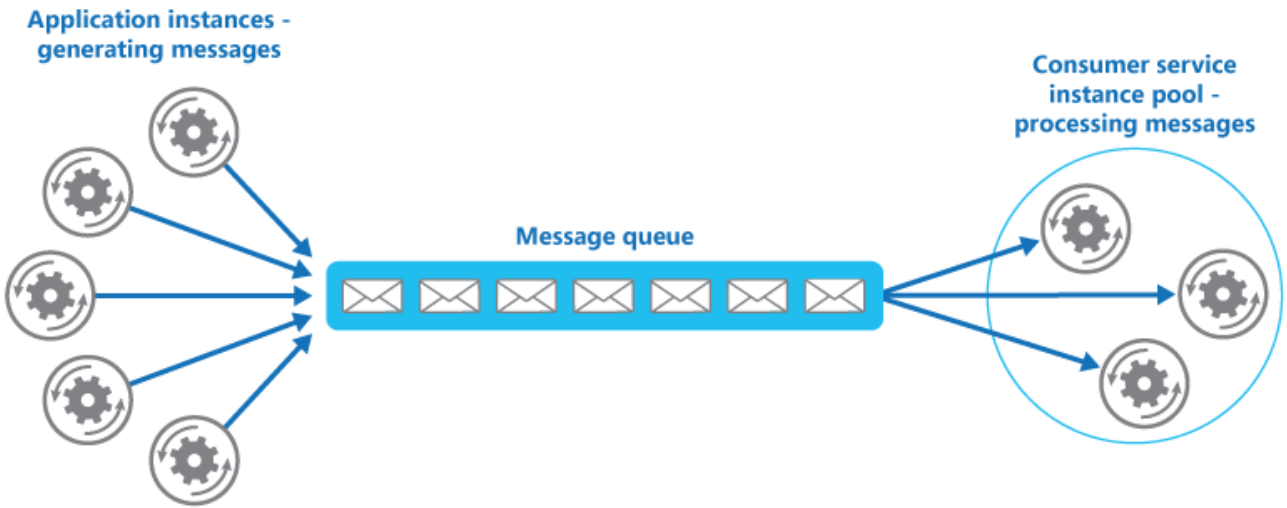
Point-to-Point communication between modules becomes fragile at scale

- 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.

Message Exchange Patterns







Why Would I Use a Queue

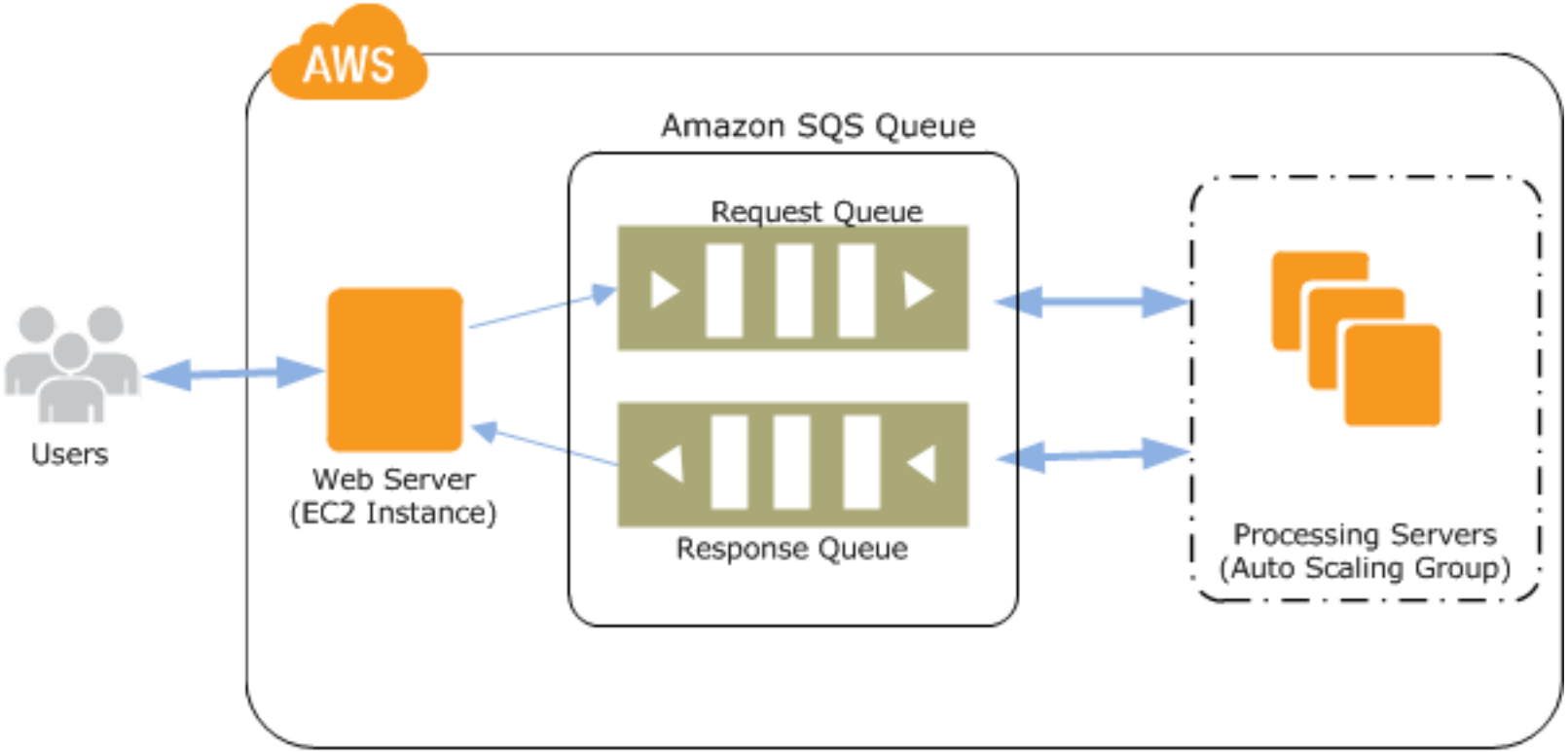
http://www.javacodegeeks.com/2013/06/working-with-amazon-simple-queue-service-using-java.html is one example)

- 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 service replica 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

Many use cases
and patterns for
Message driven processing.

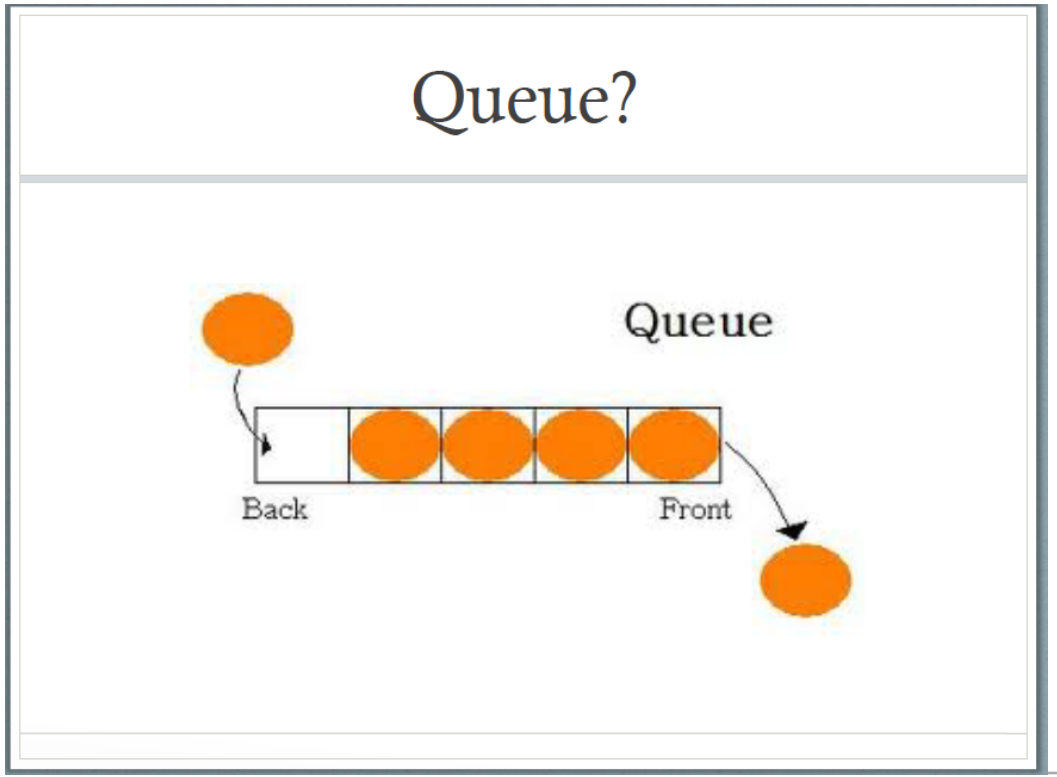
Amazon SQS

Amazon SQS – Example



SQS

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



Queue Operations

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

FIFO Semantics

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

What is SQS?

- Amazon ***S***imple ***Q***ueue ***S***ervice is a **fast, reliable, scalable and fully managed** queue service

- Amazon SQS enables **asynchronous message-based communication** between **distributed components** of an application

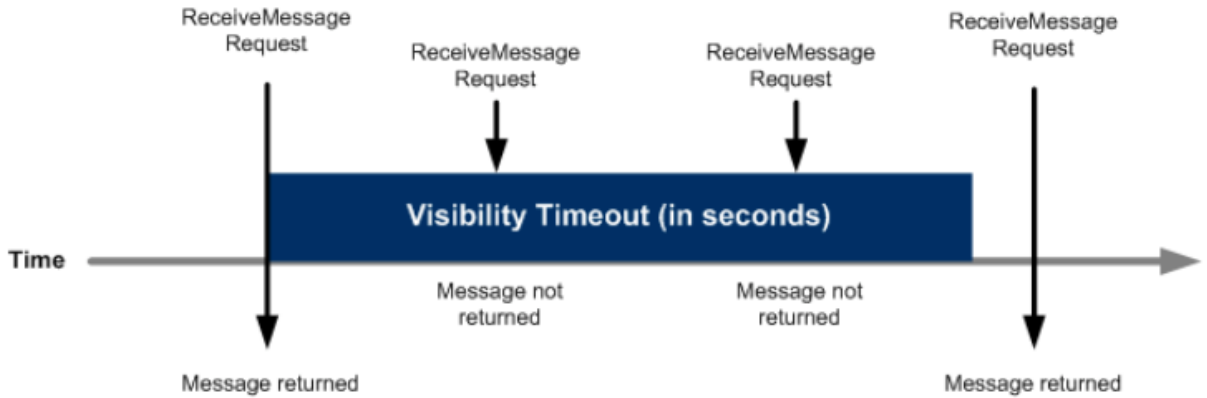
SQS – What it can do?

- Amazon AWS SQS page:
  - SQS makes it simple and cost-effective to ***decouple the components*** of a cloud application
  - You can use SQS to transmit any volume of data, at any level of throughput, ***without losing messages*** or requiring other services to be always available
  - With SQS, you can offload the administrative burden of ***operating and scaling*** a highly available messaging cluster, while paying a low price for only what you use.

SQS – Major Features

- Redundant Infrastructure
- Multiple writers and readers
- Configurable settings per queue
- Variable message size (up to 256 KB)
- Access Control
- Delay Queues

SQS – Message Visibility Timeout



SQS – Message Visibility Timeout

- The visibility timeout clock starts ticking once Amazon SQS returns the message
- During that time, the component processes and deletes the message
- But what happens if the component fails before deleting the message?
- If your system doesn't call *DeleteMessage* for that message before the visibility timeout expires, the message again becomes visible to the ReceiveMessage calls placed by the components in your system and it will be received again
- If a message should only be received once, your system should delete it within the duration of the visibility timeout
- Each queue starts with a default setting of 30 seconds for the visibility timeout

SQS Message Lifecycle

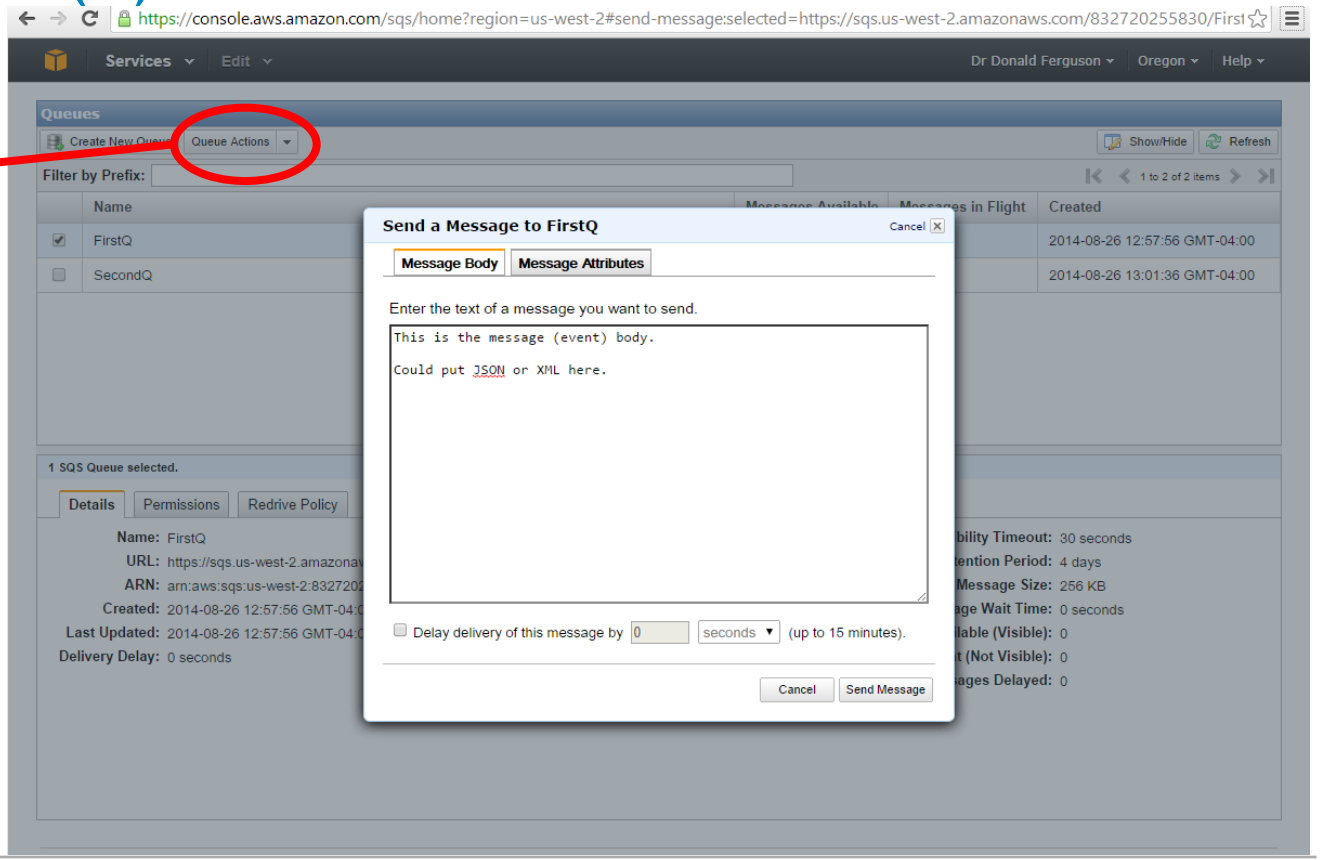
- Messages that are stored in Amazon SQS have a lifecycle that is easy to manage but ensures that all messages are processed.

1. A system that needs to send a message will find an Amazon SQS queue, and use *SendMessage* to add a new message to it
2. A different system that processes messages needs more messages to process, so it calls *ReceiveMessage*, and this message is returned
3. Once a message has been returned by *ReceiveMessage*, it will **not** be returned by any other *ReceiveMessage* until the **visibility timeout** has passed. This keeps multiple computers from processing the same message at once.
4. If the system that processes messages successfully finishes working with this message, it calls *DeleteMessage*, which removes the message from the queue so no one else will ever process it. If this system fails to process the message, then it will be read by another *ReceiveMessage* call as soon as the **visibility timeout passes**

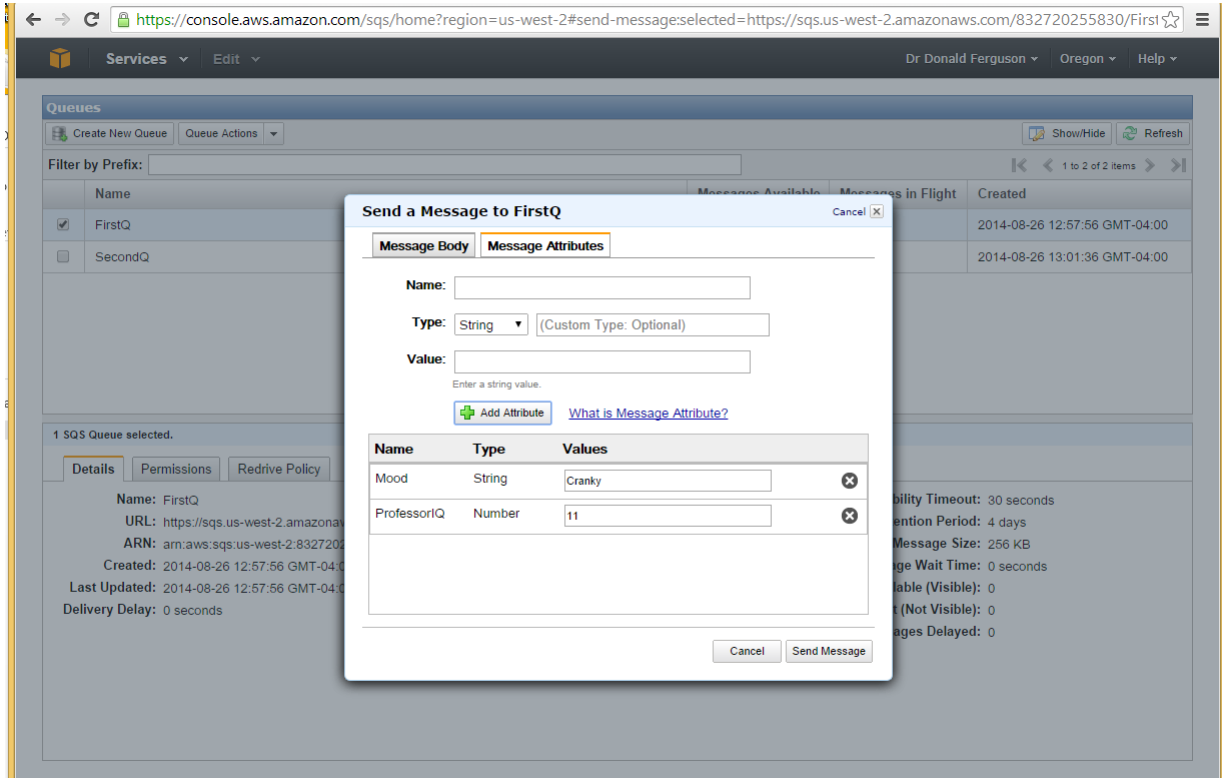
Amazon SQS (I)

Selected

“Send Msg”



Amazon SQS (I)



SQS API

(AWS.Request) **addPermission**(params = {}, callback)

Adds a permission to a queue for a specific principal.

(AWS.Request) **changeMessageVisibility**(params = {}, callback)

Changes the visibility timeout of a specified message in a queue to a new value.

(AWS.Request) **changeMessageVisibilityBatch**(params = {}, callback)

Changes the visibility timeout of multiple messages.

(AWS.Request) **createQueue**(params = {}, callback)

Creates a new queue, or returns the URL of an existing one.

(AWS.Request) **deleteMessage**(params = {}, callback)

Deletes the specified message from the specified queue.

(AWS.Request) **deleteMessageBatch**(params = {}, callback)

Deletes up to ten messages from the specified queue.

(AWS.Request) **deleteQueue**(params = {}, callback)

Deletes the queue specified by the **queue URL**, regardless of whether the queue is empty.

(AWS.Request) **getQueueAttributes**(params = {}, callback)

(AWS.Request) **getQueueUrl**(params = {}, callback)

Returns the URL of an existing queue.

(AWS.Request) **listDeadLetterSourceQueues**(params = {}, callback)

Returns a list of your queues that have the RedrivePolicy queue attribute configured with a dead letter queue.

For more information about using dead letter queues, see Using Amazon SQS Dead Letter Queues.

.

(AWS.Request) **listQueues**(params = {}, callback)

Returns a list of your queues.

(AWS.Request) **purgeQueue**(params = {}, callback)

Deletes the messages in a queue specified by the **queue URL**.

When you use the `PurgeQueue` API, the deleted messages in the queue cannot be retrieved.

When you purge a queue, the message deletion process takes up to 60 seconds.

(AWS.Request) **receiveMessage**(params = {}, callback)

Retrieves one or more messages, with a maximum limit of 10 messages, from the specified queue.

(AWS.Request) **removePermission**(params = {}, callback)

Revokes any permissions in the queue policy that matches the specified `Label` parameter.

(AWS.Request) **sendMessage**(params = {}, callback)

Delivers a message to the specified queue.

(AWS.Request) **sendMessageBatch**(params = {}, callback)

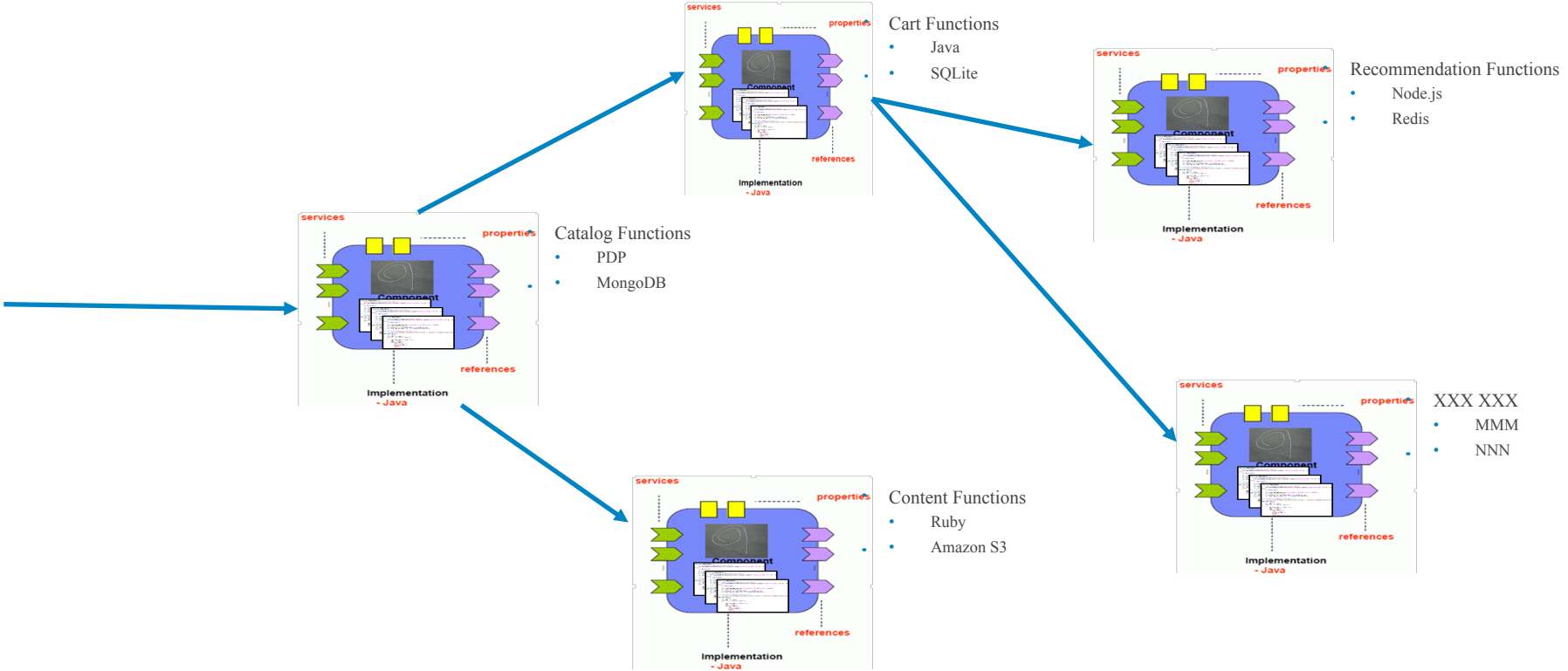
Delivers up to ten messages to the specified queue.

(AWS.Request) **setQueueAttributes**(params = {}, callback)

Sets the value of one or more queue attributes.

Pub/Sub

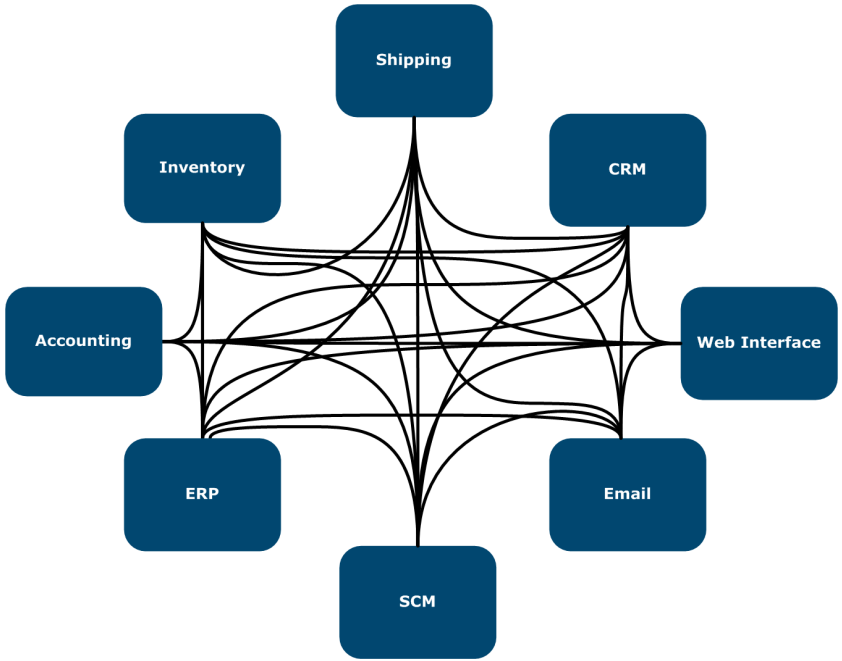
REST Implementations Call REST Services



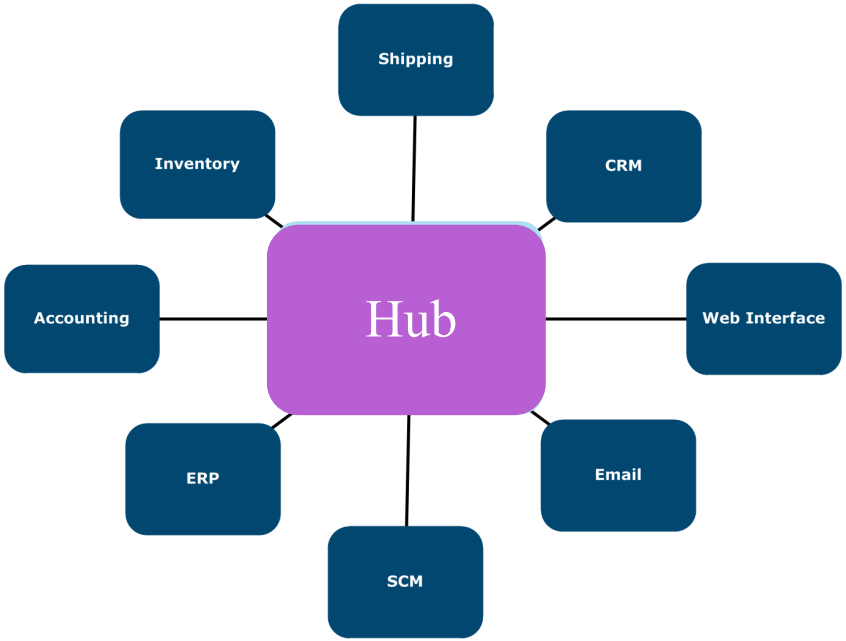
Serverless Function and Micro-Service Integration

(images from https://cdn.axway.com/u/documentation)

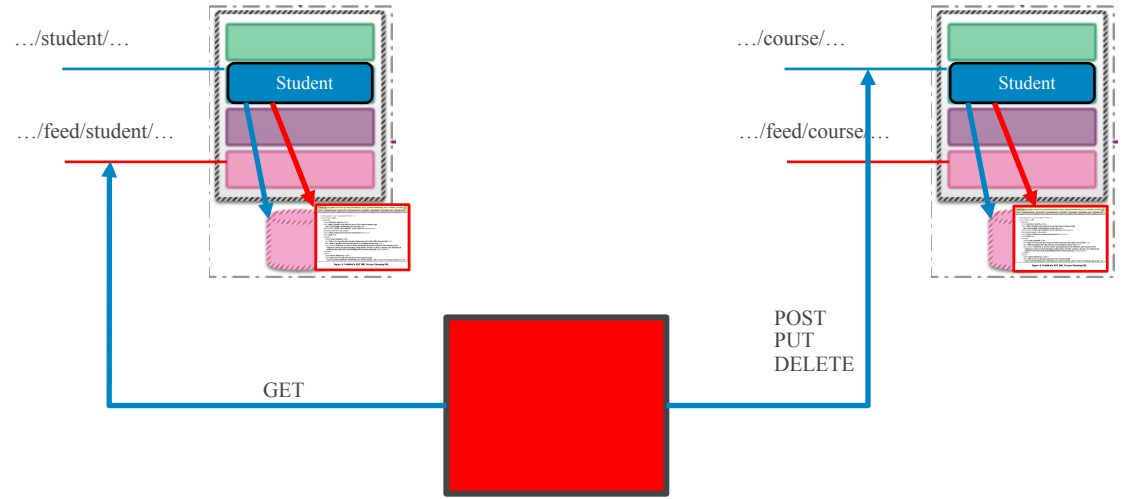
Bad, Bad, Bad, AAAARGH!



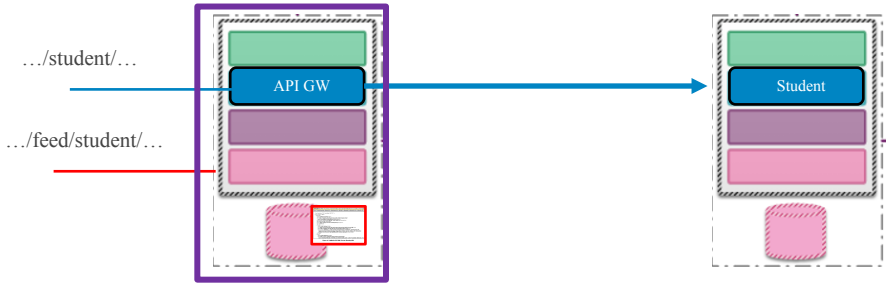
Pub/Sub or Message Hub



Event Driven Integration/Composition

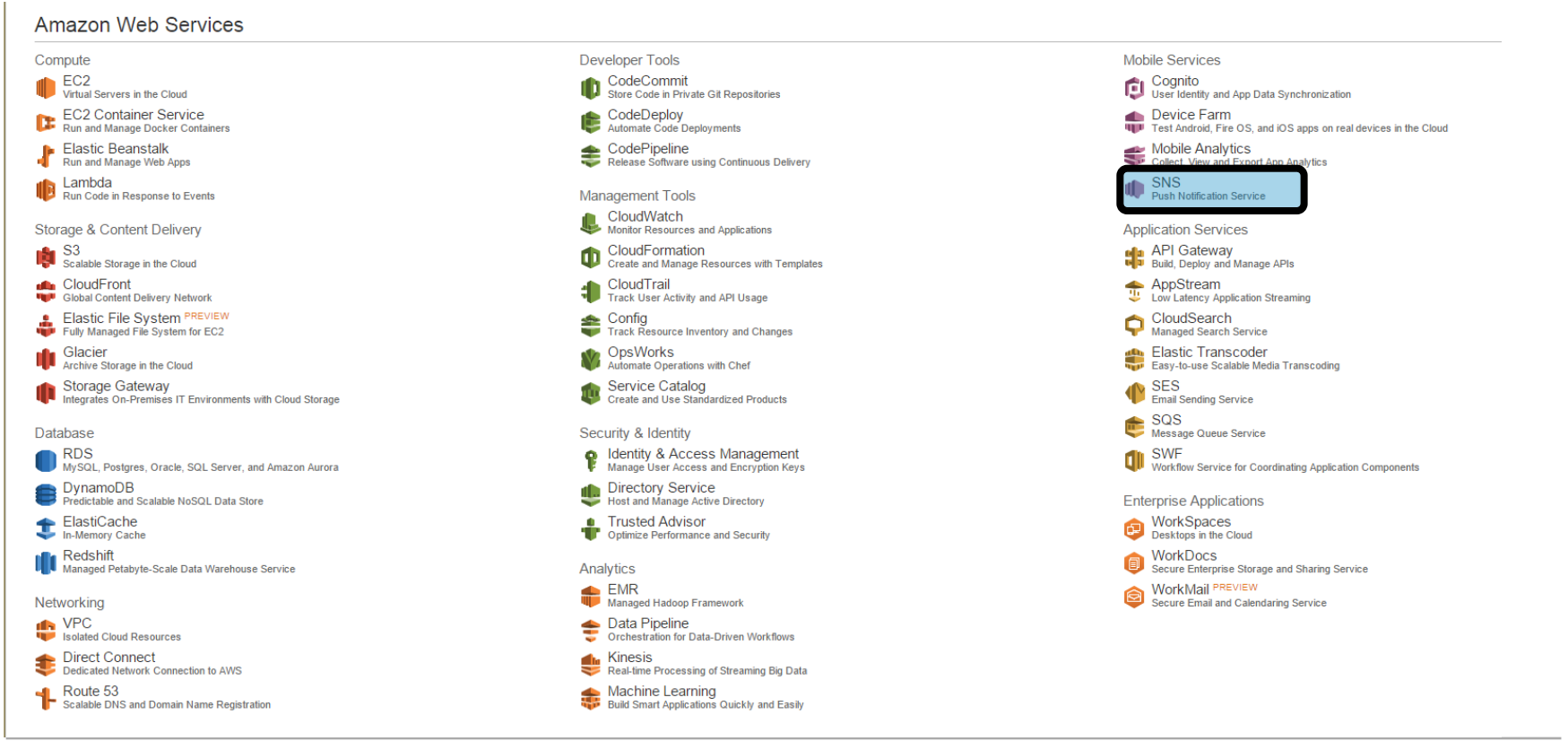


- If you have a bunch of microservices that do not emit events/feeds.
- Front with an API GW that
  - Delegates onto actual service
  - Looks at response
  - Writes a feed

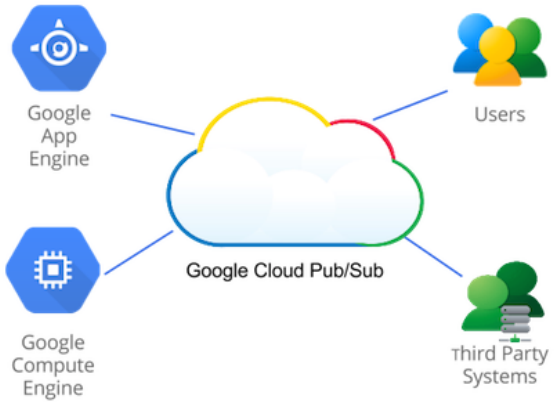


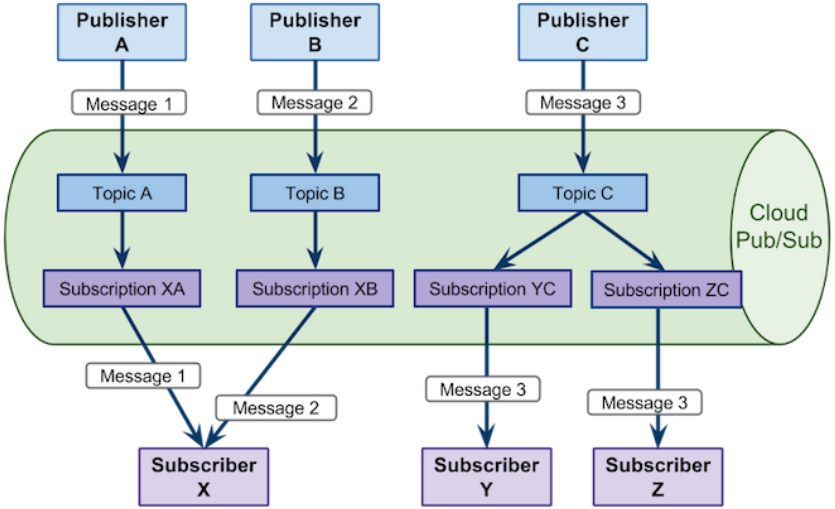
- If the microservice emits a feed/log
  - Add a new microservice
  - Reads feeds/logs
  - Calls appropriate REST API
  - Has an API for configuring the feeds/logs to read and mapping to REST APIs

Amazon Web Services



Google Cloud Pub/Sub

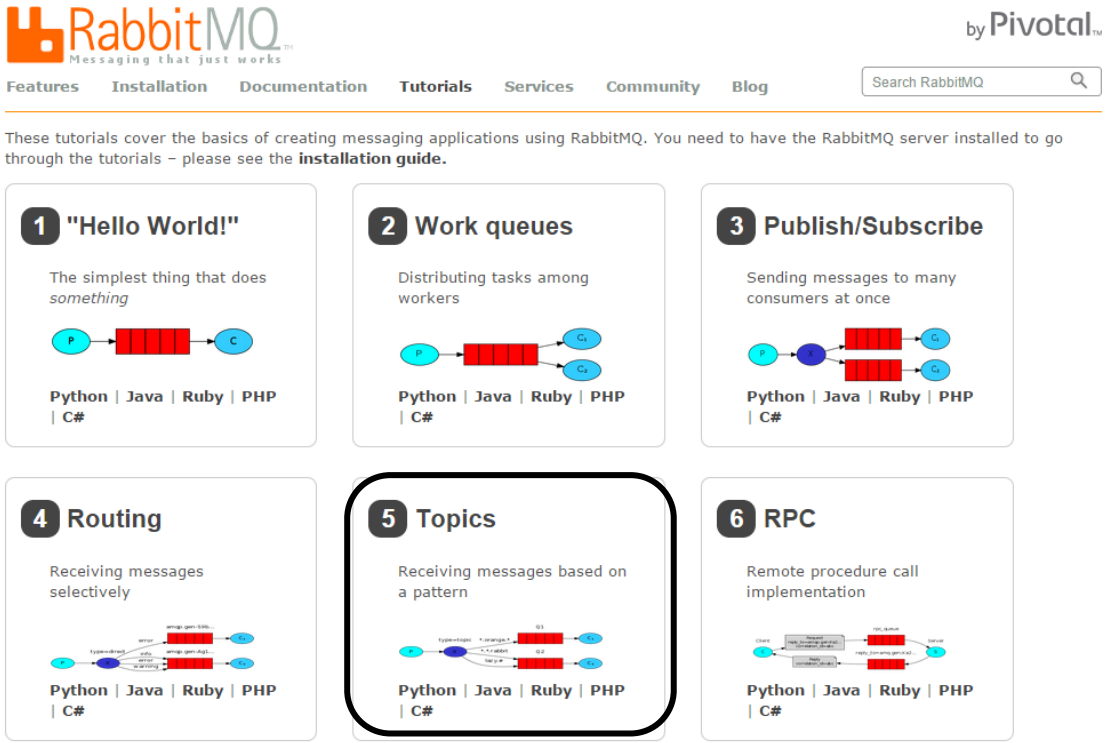




https://cloud.google.com/pubsub/overview

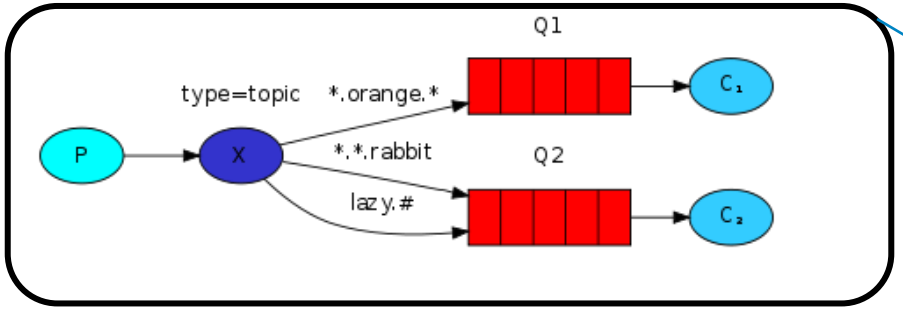
RabbitMQ

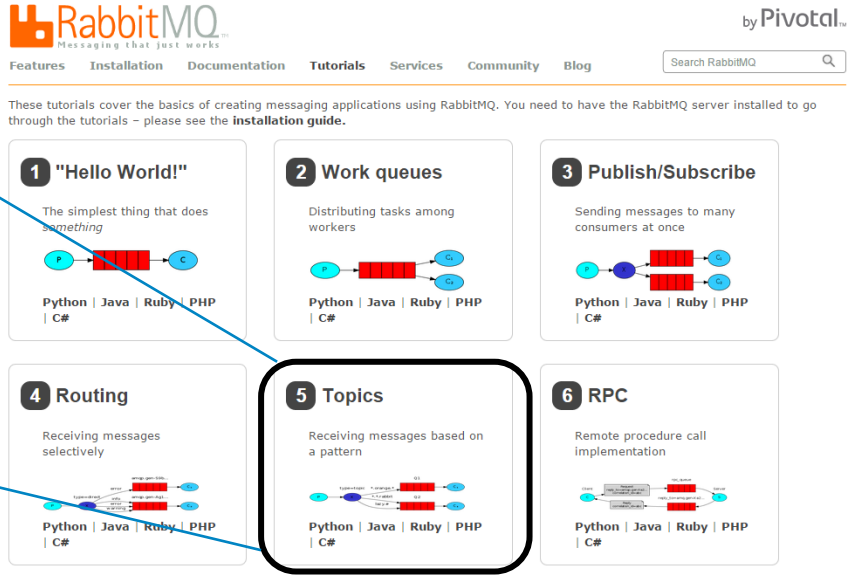
https://www.rabbitmq.com/getstarted.html



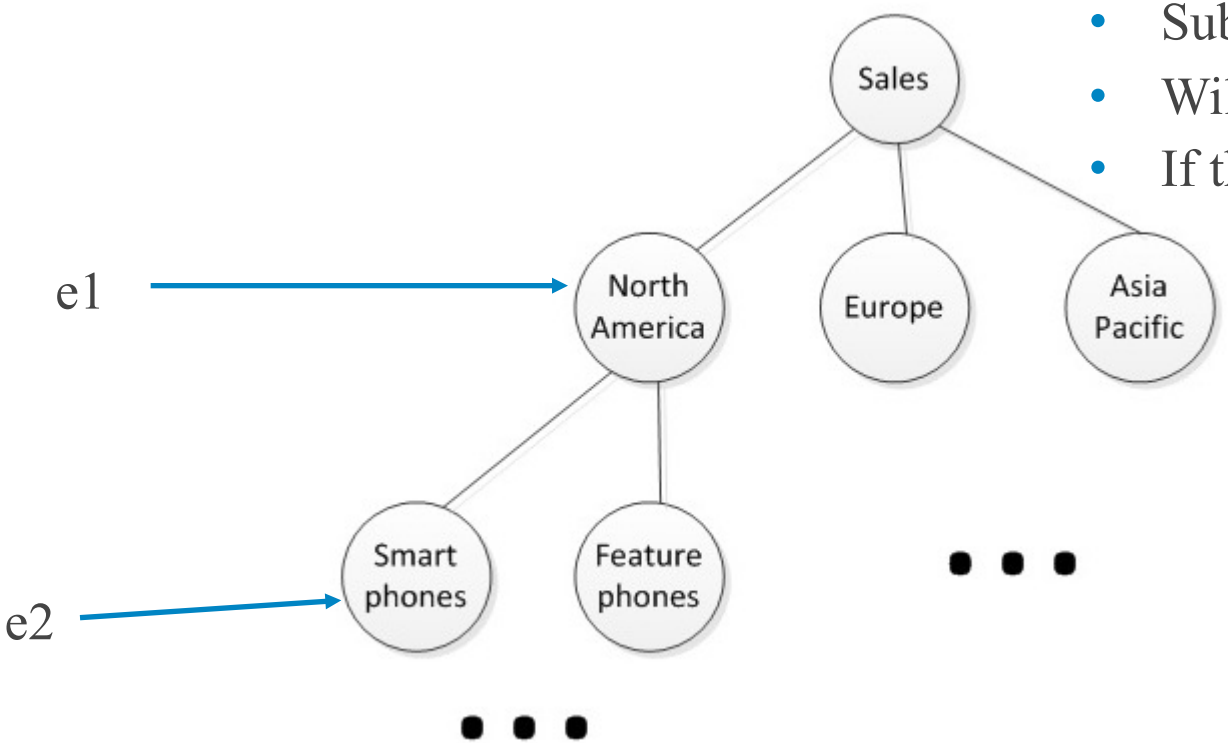
RabbitMQ

https://www.rabbitmq.com/tutorials/tutorial-five-python.html





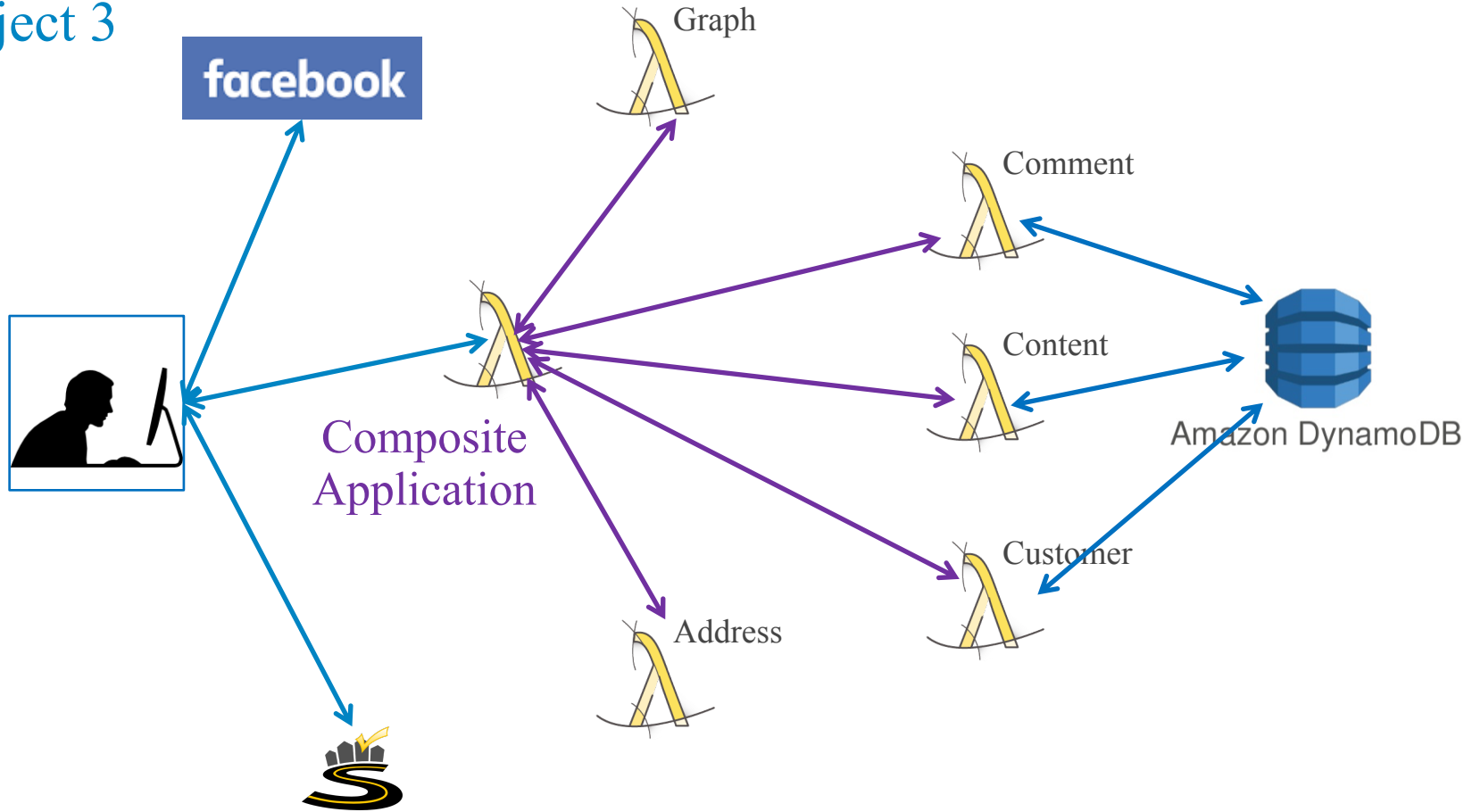
Topic Trees



- Subscribe to /sales
- Will see e1 and e2
- If they match the filter

Final Project (YAY!)

Project 3



3rd Project (Part 1)

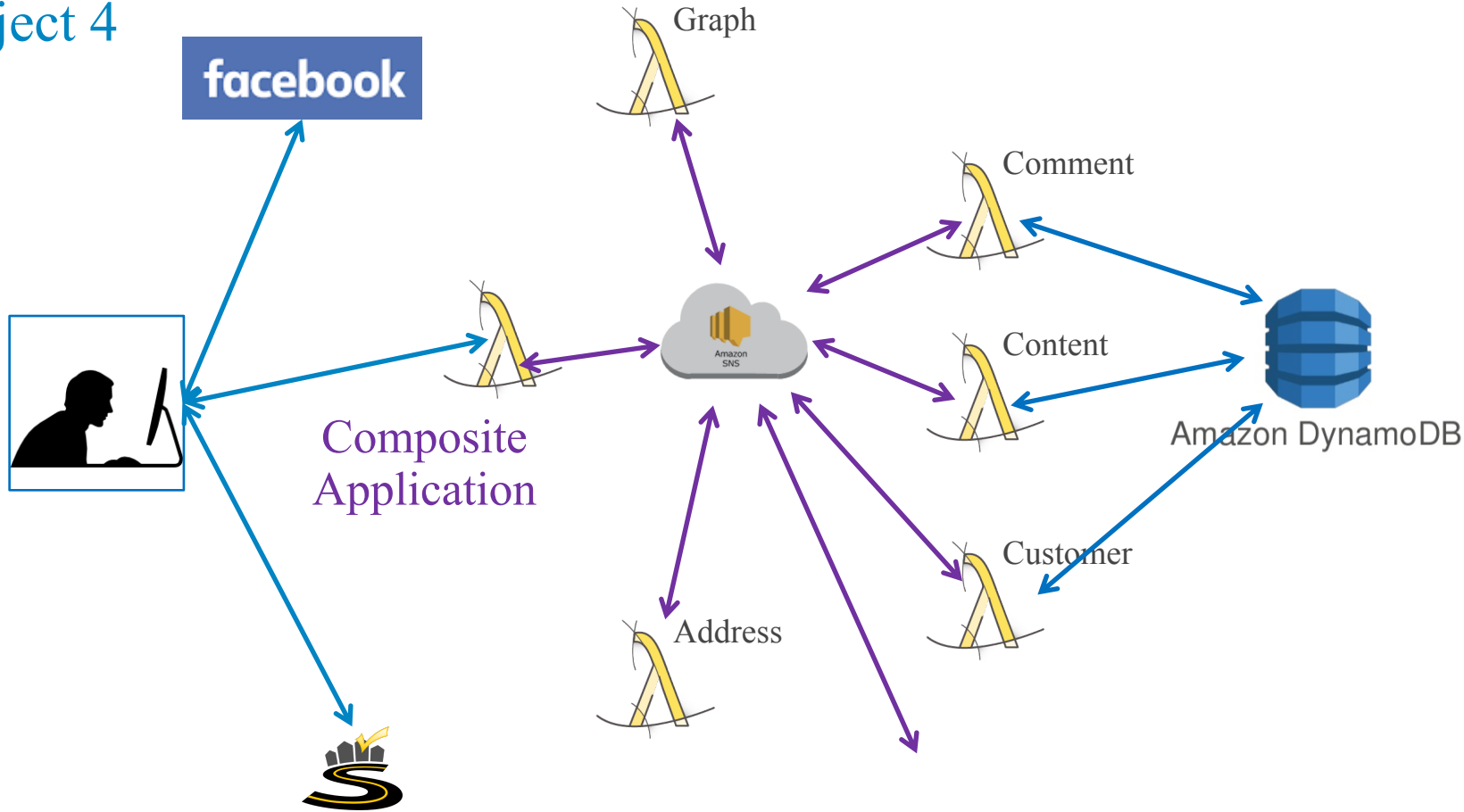
- Define and provide a Lambda/REST interface to a "content catalog" in DynamoDB.
  - Model: Content Instance
    - Property (e.g. HBO, CBS) is a set of one or more Franchises.
    - A Franchise (e.g. Game of Thrones, Bing Bang Theory) is a set of one or more Series.
    - A Series (e.g. Season 1, Christmas Specials) is a set of one or more Episodes.
    - An Episode is an instance of a "TV show."
    - Data for each type is just (ID, name).
  - Comment (ID, user, content instance, comment)
- Graph database
  - Signup for/begin to use REST callable Neo4J database (e.g. graphenedb.com)
  - Node type are Person, Content (one of Property, Franchise, …) and Comment.
  - Relationships between people (Follows, Friends)
  - Relationships between people and comments (commented, responded, liked, ...)
  - Relationships between people and content (Likes, Rates, ...)
- Expand REST API and Lambda functions to provide integrated API for accessing data and relationships
  - Customer/person, Comment, Content Instance, ... in DynamoDB including property links.
  - CRUD on Customer, Content Instance, Comment create nodes and links in DynamoDB.
- Login and Register with Facebook
  - Register creates name, email, etc. From Facebook data.
  - User can change address, name, preferred email, etc.

3rd Project (Part 1)

- Define and pro
  - Model
    -
    -
    -
    -
    -
  - Con
- Graph da
  - Sign
  - Nod
  - Rela
  - Rela
  - Rela
- Expand I ips
  - Cust
  - CRU
- Login and Re
  - Register creates name, email, etc. From Facebook data.
  - User can change address, name, preferred email, etc.

- Welcome to the "Big Leagues,"or H*ll, depending on your point-of-view.
- This project is going to "force you" to
  - Think in terms of user stories, scenarios, …
  - Do some basic high-level design diagrams.
  - Carefully think about API model
  - etc.
- Three weeks completion, with interim reviews.

Project 4



4th Project

- All of your Lambda Functions can be invoked by
  - HTTP/API Gateway
  - SNS events.
- Lambda functions
  - No longer "call each other."
  - React to SNS events.
  - Emit SNS events, e.g. "customer created," "episode created," …
- Define a set of SNS topics for events
  - Lambda functions publish to correct SNS topics.
  - React to correct SNS topics.
- And publish to a Slack channel on a Slack group for you project when something "interesting happens," specifically "customer created."

Defining a "topic space" and connecting "code" is part of application design just like defining datamodels and connecting applications.

4th Proj

- All of yo
  - HTTP
  - SNS
- Lambda
  - No lo
  - React
  - Emit
- Define a
  - Lambd
  - React to correct SNS topics.
- And publish to a Slack channel on a Slack group for you project when something "interesting happens," specifically "customer created."

pace" and
is part of
sign
tamodels
lications.

- Holy Cr*P!
- Do you hate us?
- Just do as much as you can.
- Define User Stories!