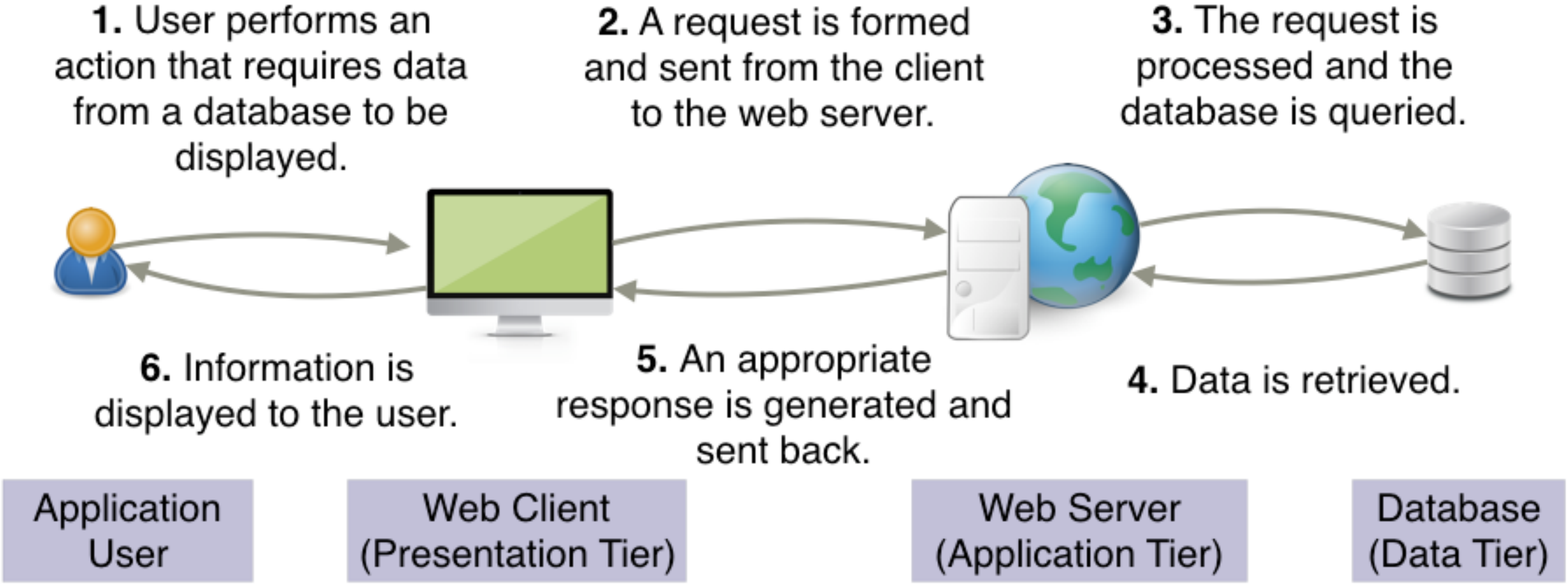
COMSE6998: Modern Serverless Cloud Applications

Lecture 1: Introduction and Concepts

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

Introduction

Web Application Basic Concepts



1. User performs an action that requires data from a database to be displayed.

2. A request is formed and sent from the client to the web server.

3. The request is processed and the database is queried.

4. Data is retrieved.

5. An appropriate response is generated and sent back.

6. Information is displayed to the user.

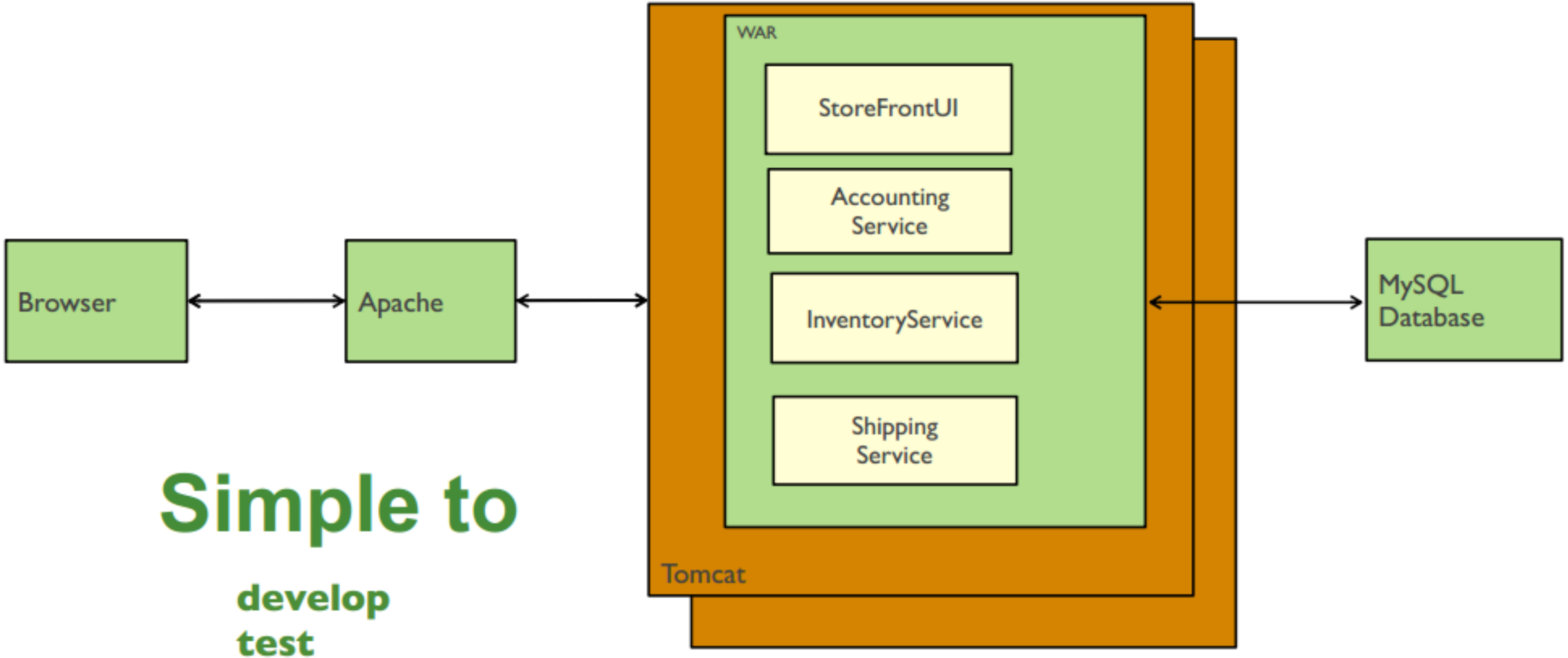
Application User

Web Client (Presentation Tier)

Web Server (Application Tier)

Database (Data Tier)

Traditional web application architecture



Simple to

develop
test
deploy
scale

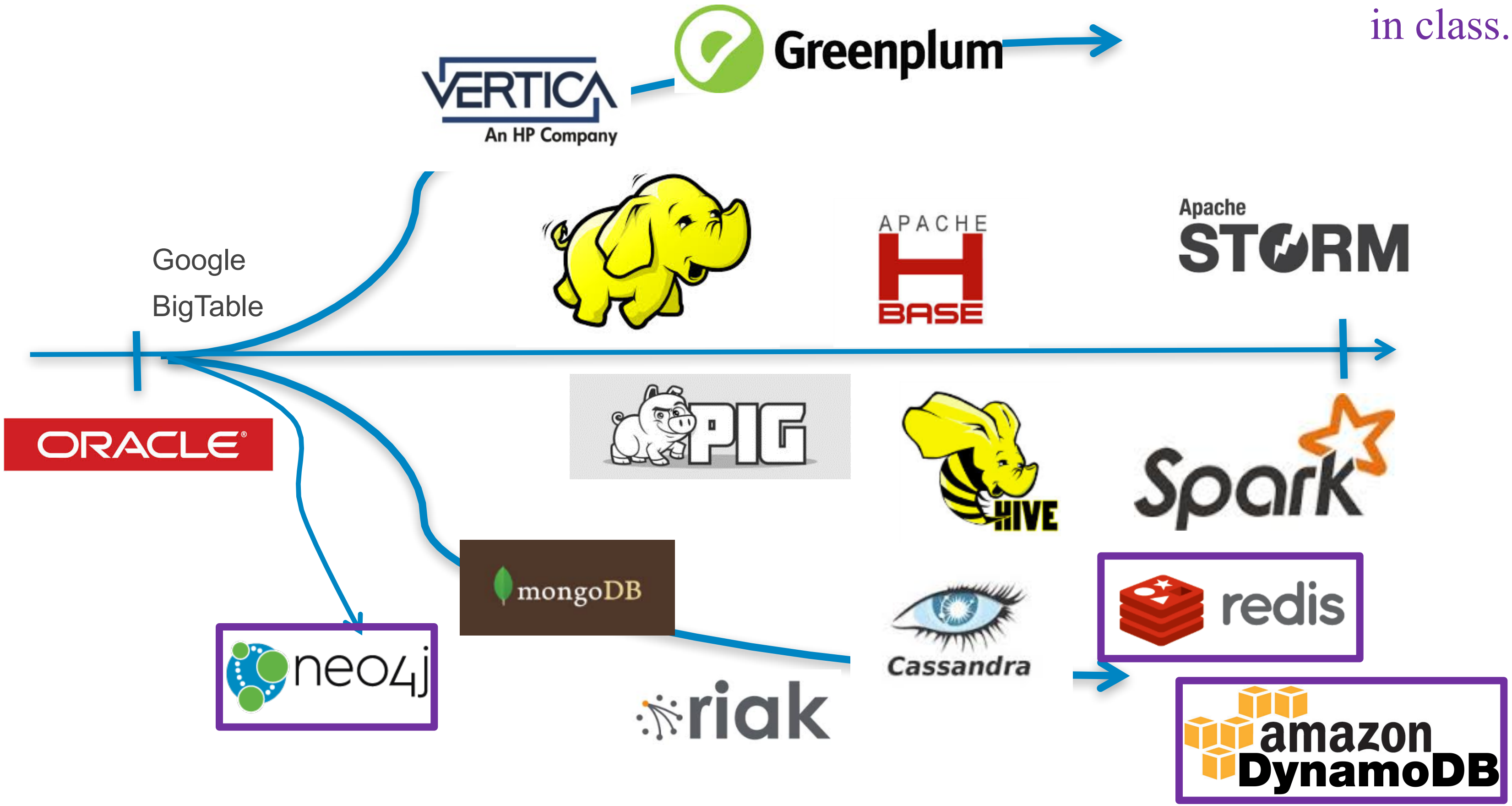
"Decomposing applications for deployability and scalability," Chris Richardson, http://plainoldobjects.com/

Evolution of Application Development

- (Web) application development went through a phase in which there were two dominant technologies:
  - J2EE: Java, JDBC, JMS, … …
  - .NET: C#, ADO.NET, SQL Server, … …
- Polyglot persistence emerged because
  - Use cases emerged
  - That were difficult to map to RDB semantics and optimizations.
  - Which drove the development of new, simple, problem focused DBs
- Polyglot programming emerged because
  - Solving some problems seems easier with specific, focused languages
  - Java and C# became powerful and complicated, and many scenarios needed much simpler and some different capabilities.
  - The browser document model is more dynamically typed than stricture languages →
    - Single language for {UI, business logic, data}.
    - More flexibly typed, dynamic languages.

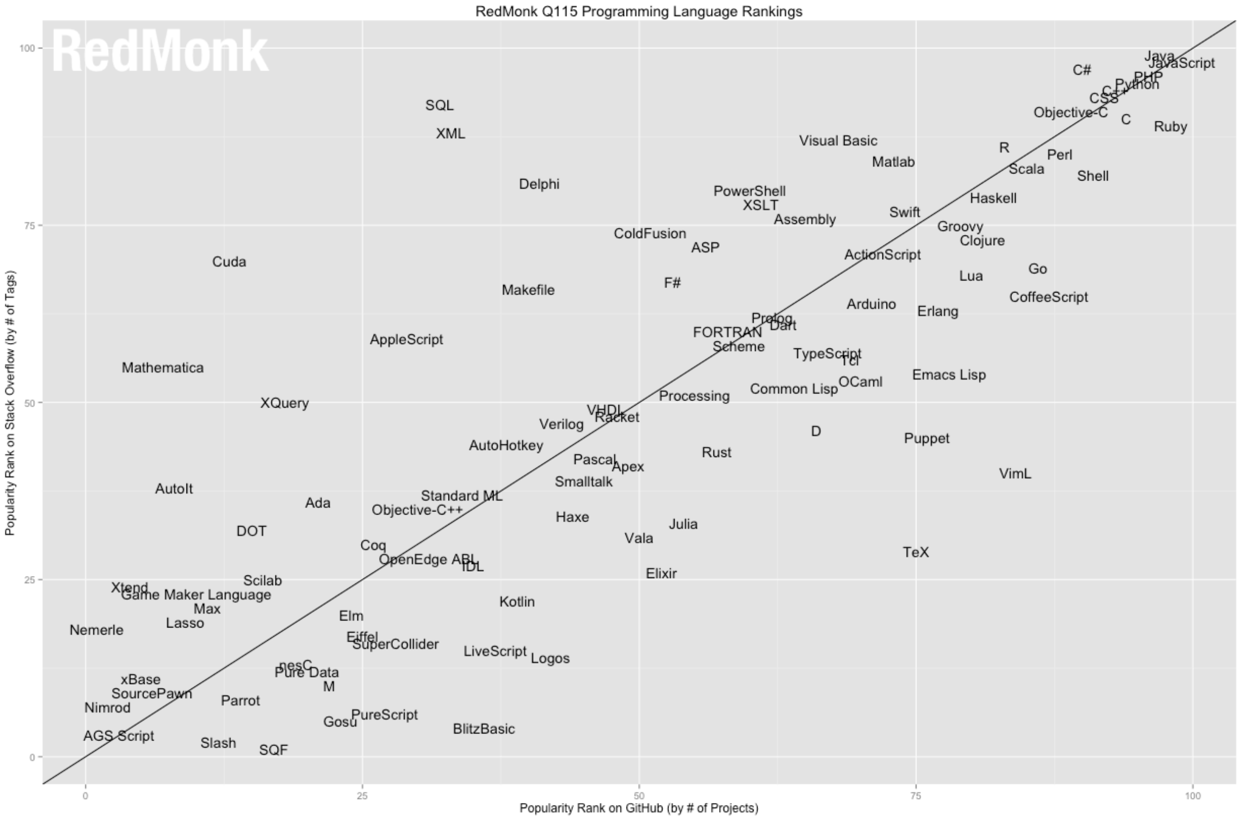
Polyglot Persistence (Very Incomplete List)

Will overview in class.

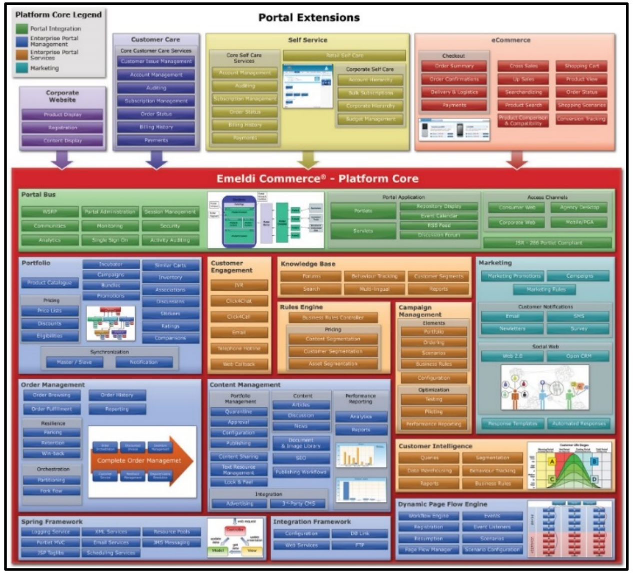


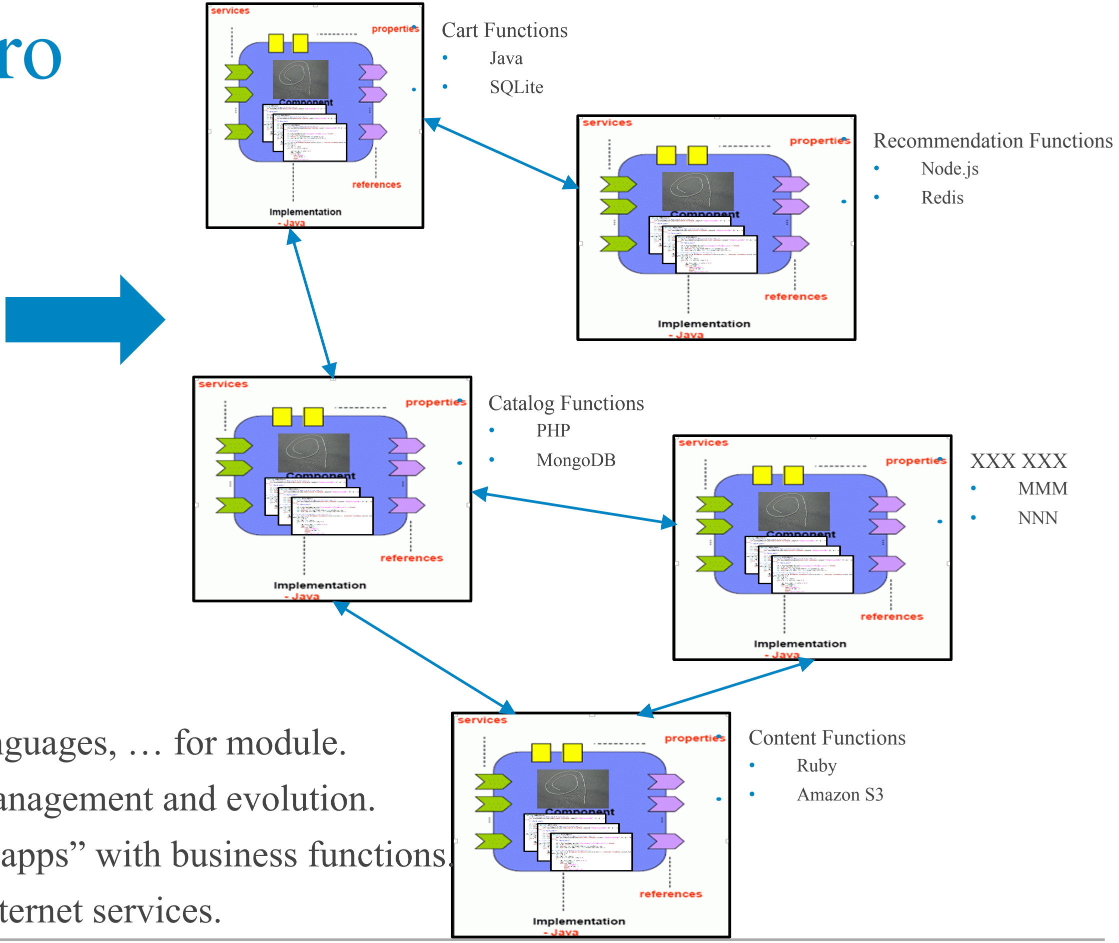
Polyglot Programming

Polyglot Programming



Monolithic to Micro





Cart Functions

- Java
- SQLite

Recommendation Functions

- Node.js
- Redis

Catalog Functions

- PHP
- MongoDB

XXX XXX

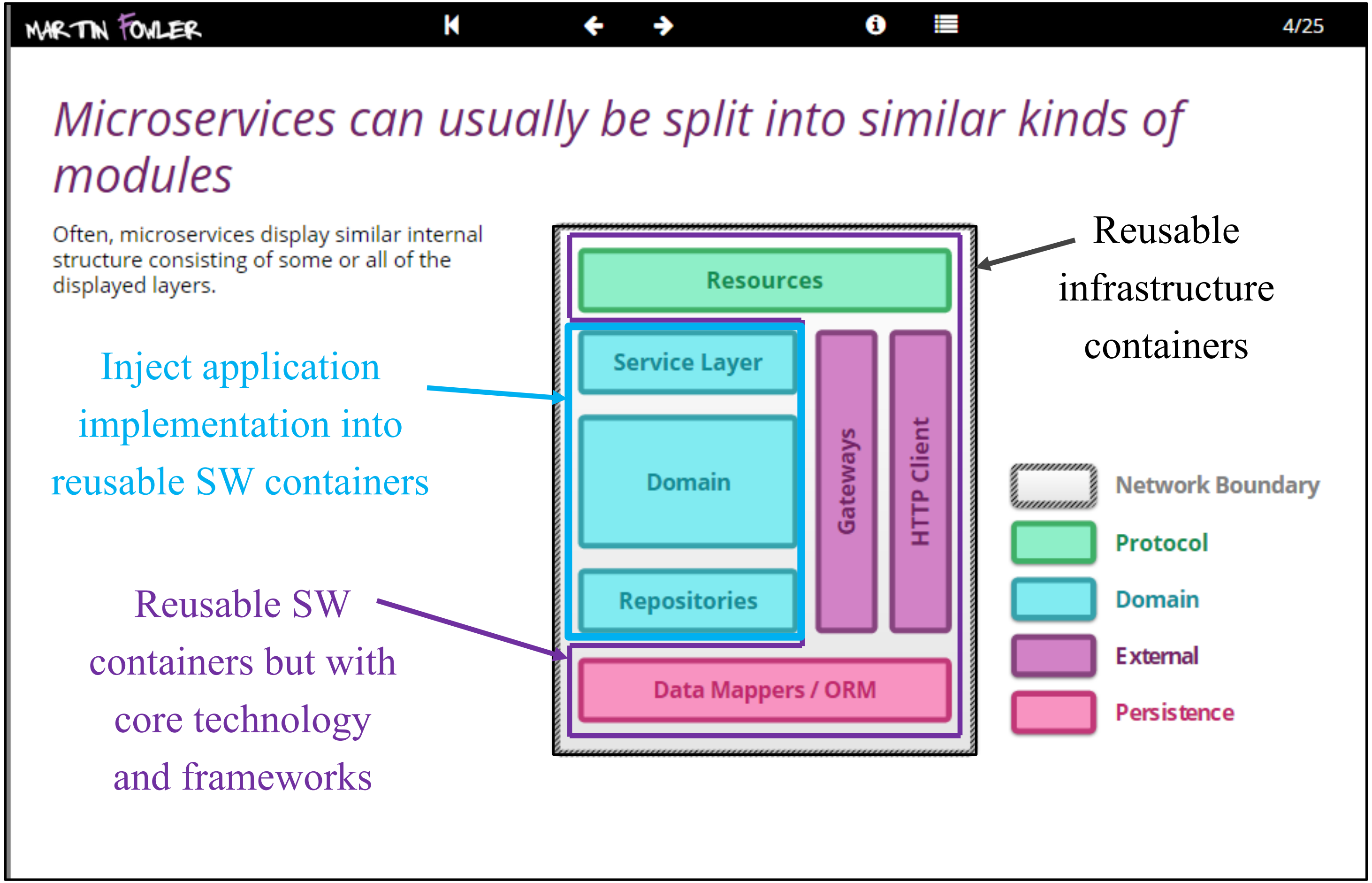
- MMM
- NNN

Content Functions

- Ruby
- Amazon S3

Motivations

- Enable best tools, languages, … for module.
- Simplifies change management and evolution.
- Better alignment of "apps" with business functions.
- Reuse of code and internet services.



Microservices can usually be split into similar kinds of modules

Often, microservices display similar internal structure consisting of some or all of the displayed layers.

Reusable infrastructure containers

Inject application implementation into reusable SW containers

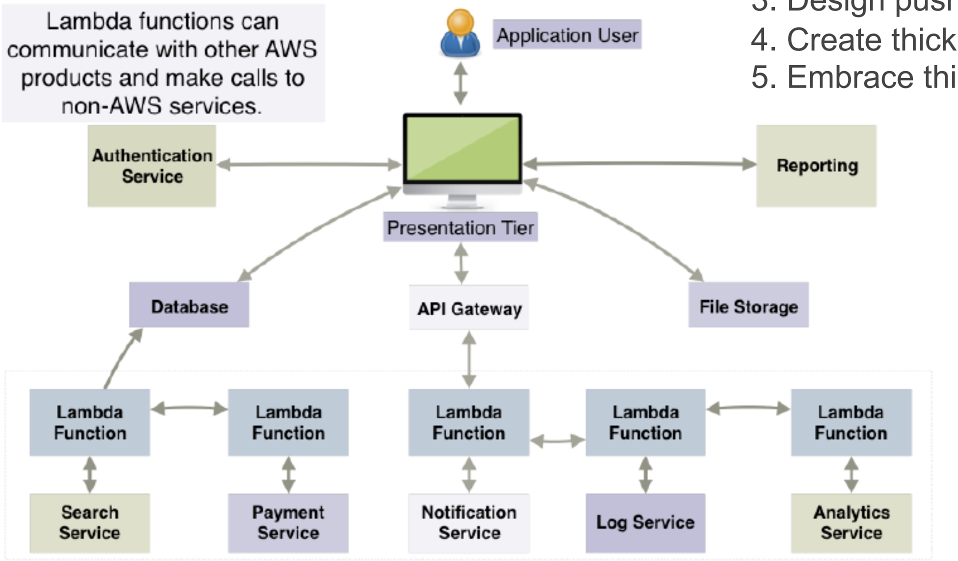
Reusable SW containers but with core technology and frameworks

Micro-services Characteristics

- Componentization via Services
- Organized around Business Capabilities
- Products not Projects
- Smart endpoints and dumb pipes
- Decentralized Governance
- Decentralized Data Management
- Infrastructure Automation
- Design for failure
- Evolutionary Design

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



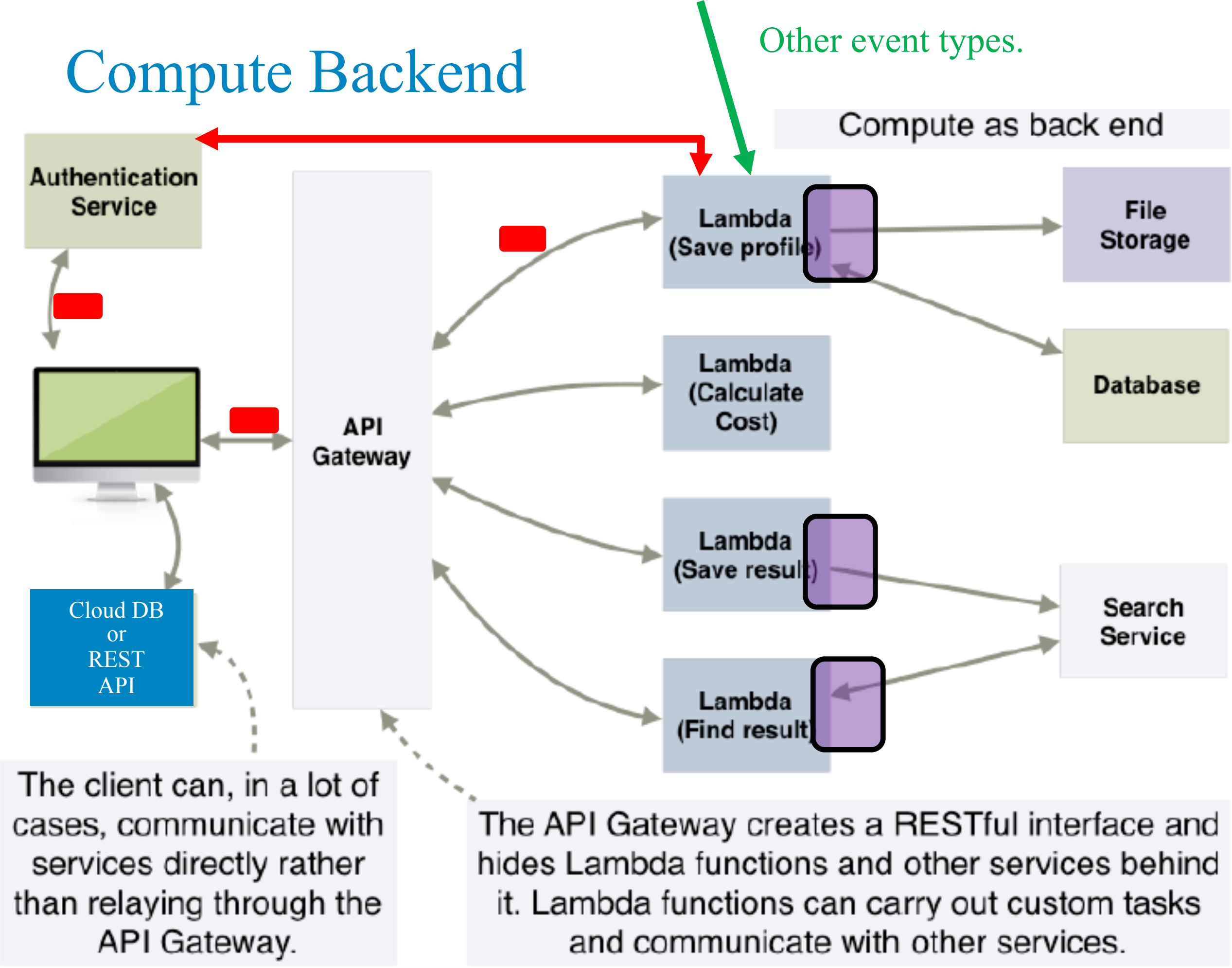
Lambda functions can communicate with other AWS products and make calls to non-AWS services.

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.”

Compute Backend



The client can, in a lot of cases, communicate with services directly rather than relaying through the API Gateway.

The API Gateway creates a RESTful interface and hides Lambda functions and other services behind it. Lambda functions can carry out custom tasks and communicate with other services.

Observations

- Front end and backend both communicate with authentication service.
- Context flows on calls.
- Well-designed code, even a single function, uses a service abstraction for accessing data.
- Multiple event types drive "business function" → API cannot be coupled to specific event format.

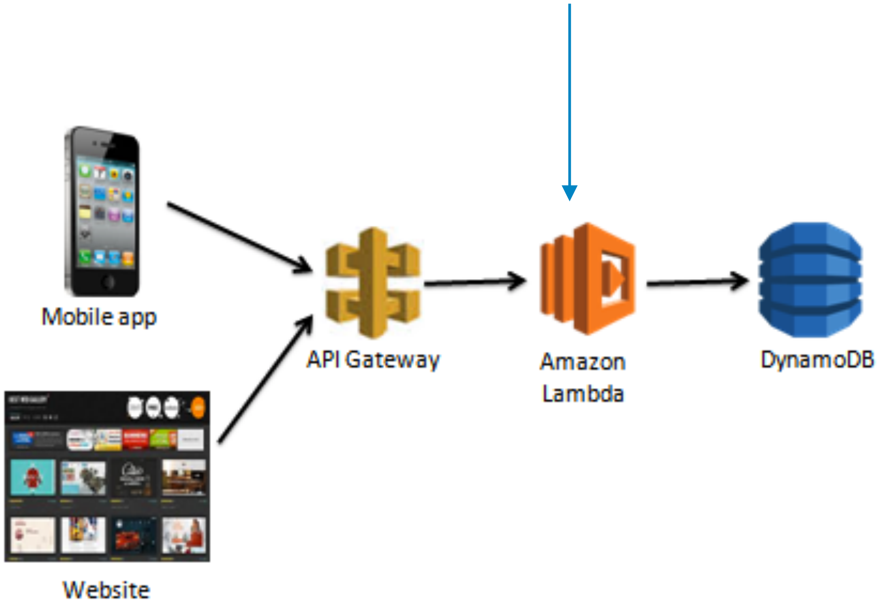
Serverless from Microservices

- 1. Use a compute service to execute code on demand (no servers)
  - No need to define and maintain a runtime engine and app server.
  - Eliminates managing, monitoring, … app server runtime instances.
- 2. Write single-purpose stateless functions
  - More flexible and dynamic lifecycle → agility
  - Evolves to an HTML/wiki like model from a stop-deploy module-restart, especial where a lot of the module has not changed.
- 3. Design push-based, event-driven pipelines
  - Microservices implies invocation only by HTTP/REST.
  - Multiple event types trigger serverless: {event, condition, action, event} model.
- 4. Create thicker, more powerful front ends
  - No "module" → code that assembles multiple data sources and functions
  - Moves from microservice to front-end.
- 5. Embrace third-party services
  - No local libraries and server runtimes →
  - All calls are inherently "web" calls.

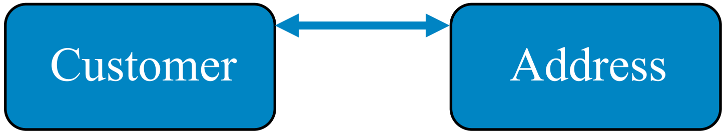
I do not totally agree with this observation.

Let's Start to Build Another Lambda Function

API Gateway – Lambda Function -- DynamoDB



Other types of events.



Data Model

- Name in *contained*.
- Address is *referenced*.

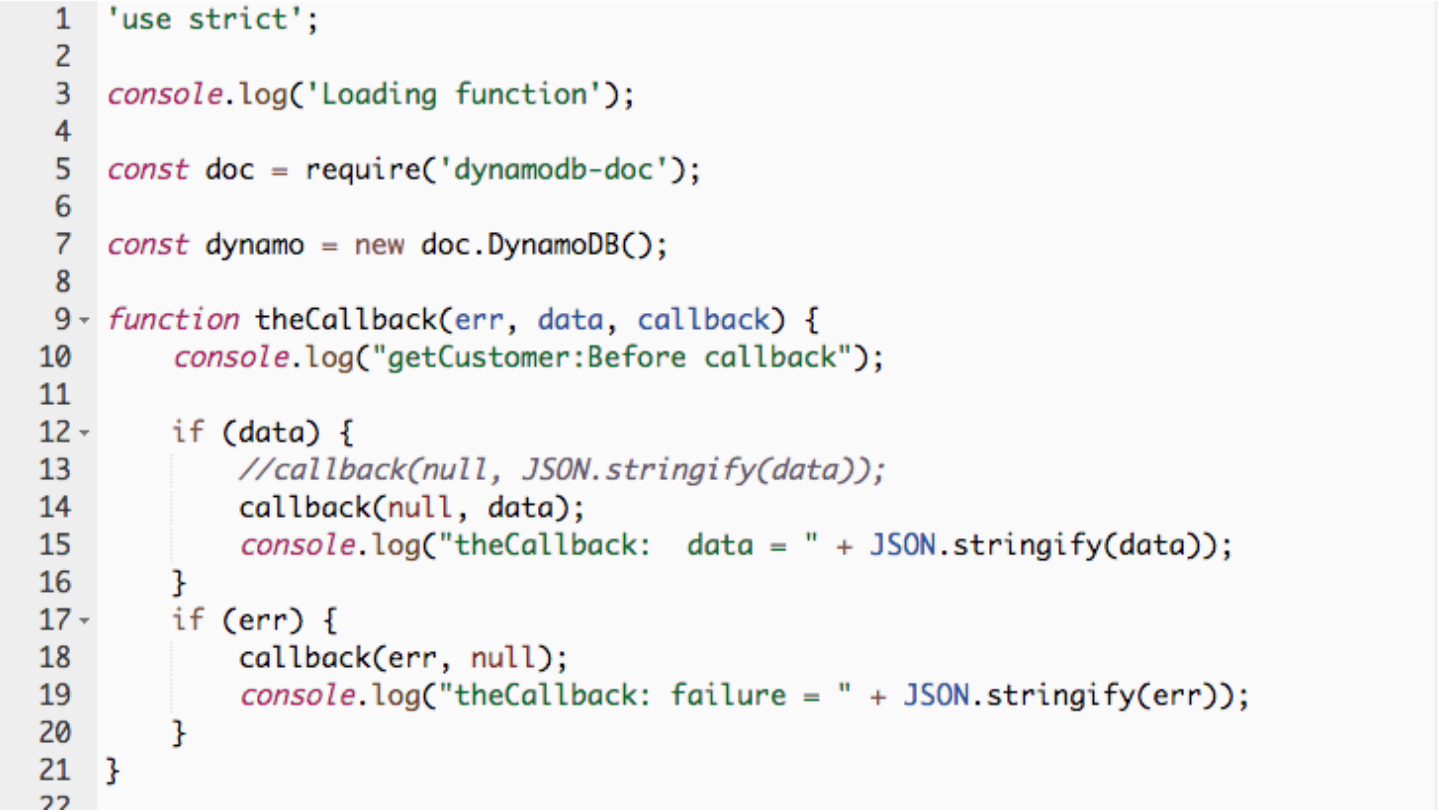
Why?

- Correct an address error w/out scanning all customers for addr.
- Deleting a customer does not vaporize the house.

Operations (and 1^{St} Assignment)

- CRUD
  - Independently CRUD Customer or Address
  - C Customer and Address in one operation.
- Customer – Address is many-to-one.
- Some example "finds"
  - Customer by email
  - Customers by phone number or last name.
  - Customer's Address
  - All Customers living at an Address
  - All Addresses in a zipcode.
  - … ...
- Ensure consistency, even though database is not typed nor has referential integrity, e.g.
  - "Yellow" is not a valid String value for phone number.
  - { "pet" : "Canary"} is not a valid *property* of Address.
  - Cannot "create" a Customer with email="xxx" if one already exists.

Beginning of JavaScript Impl.

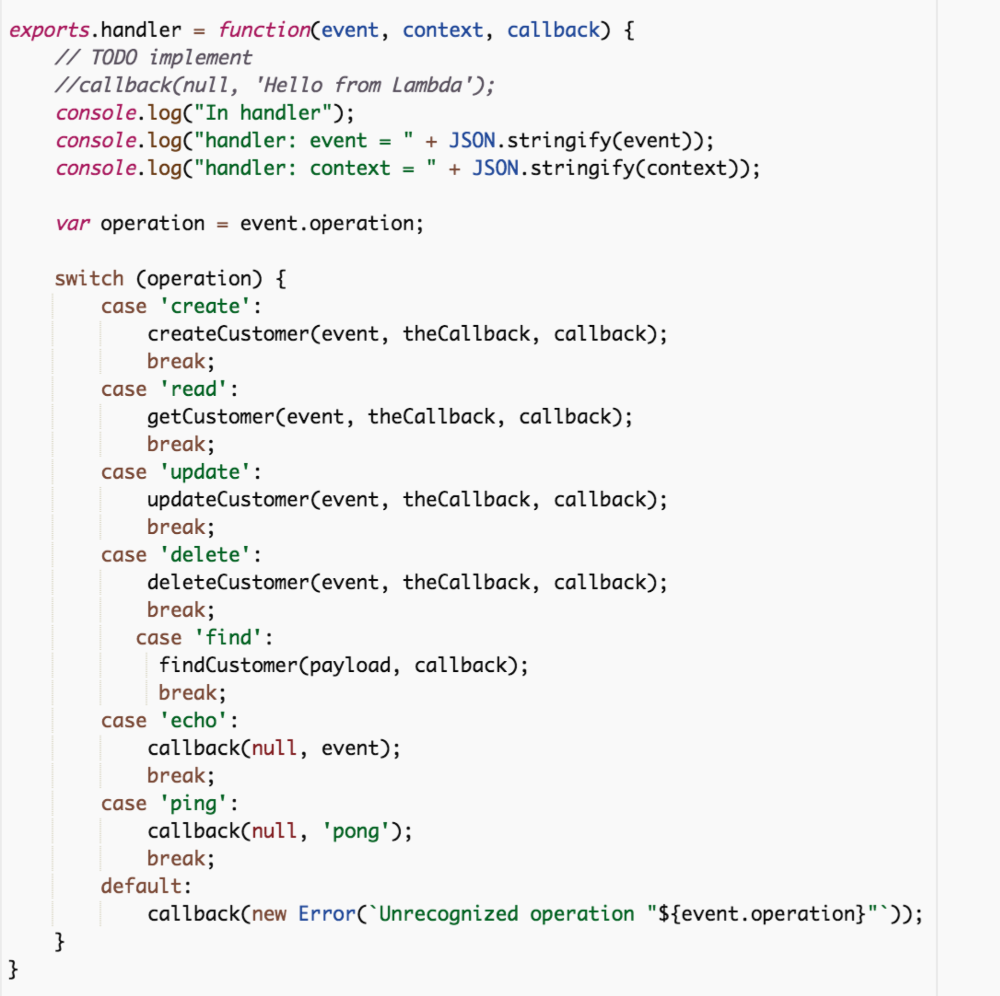


```
'use strict';

console.log('Loading function');

const doc = require('dynamodb-doc');

const dynamo = new doc.DynamoDB();

function theCallback(err, data, callback) {
    console.log("getCustomer:Before callback");

    if (data) {
        //callback(null, JSON.stringify(data));
        callback(null, data);
        console.log("theCallback:  data = " + JSON.stringify(data));
    }
    if (err) {
        callback(err, null);
        console.log("theCallback: failure = " + JSON.stringify(err));
    }
}
```

The Core

- Input
  - Event (the data, from GW in our example)
  - Context (see http://docs.aws.amazon.com/lambda/latest/dg/nodejs-prog-model-context.html)
  - Callback(err, data) – Where to “return.”
- Application Event

```
{
      "operation" : "read",
      "email" : "dff9@columbia.edu"
}
Or
{
      "operation" : "create",
      "item”: {
            "email":"don@foo.edu",
            "lastname":"Ferguson",
            "Firstname":"Donald”
      }
}
```

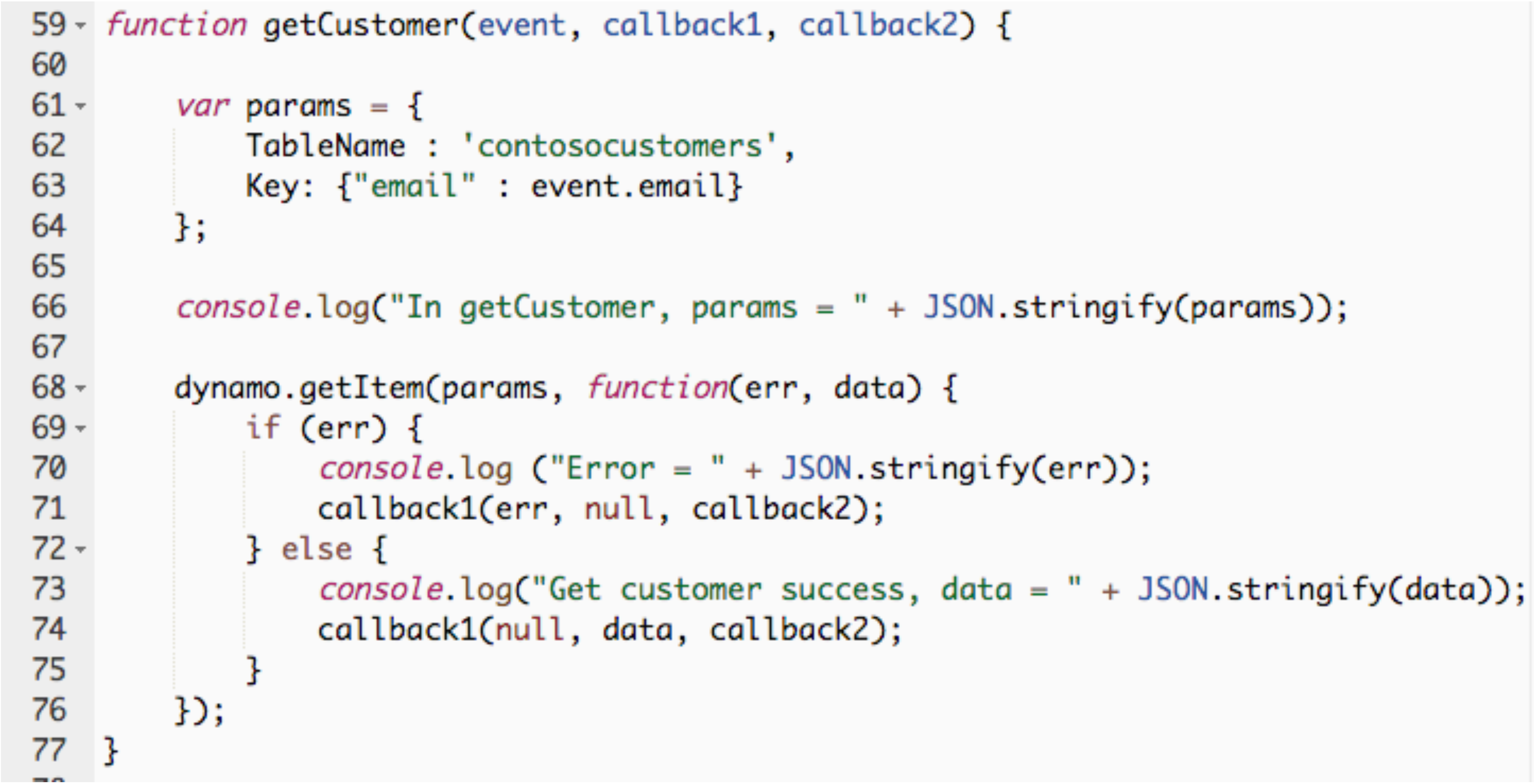


```
exports.handler = function(event, context, callback) {
    // TODO implement
    //callback(null, 'Hello from Lambda');
    console.log("In handler");
    console.log("handler: event = " + JSON.stringify(event));
    console.log("handler: context = " + JSON.stringify(context));

    var operation = event.operation;

    switch (operation) {
        case 'create':
            createCustomer(event, theCallback, callback);
            break;
        case 'read':
            getCustomer(event, theCallback, callback);
            break;
        case 'update':
            updateCustomer(event, theCallback, callback);
            break;
        case 'delete':
            deleteCustomer(event, theCallback, callback);
            break;
           case 'find':
             findCustomer(payload, callback);
             break;
        case 'echo':
            callback(null, event);
            break;
        case 'ping':
            callback(null, 'pong');
            break;
        default:
            callback(new Error(`Unrecognized operation "${event.operation}"`));
    }
}
```

Context Object

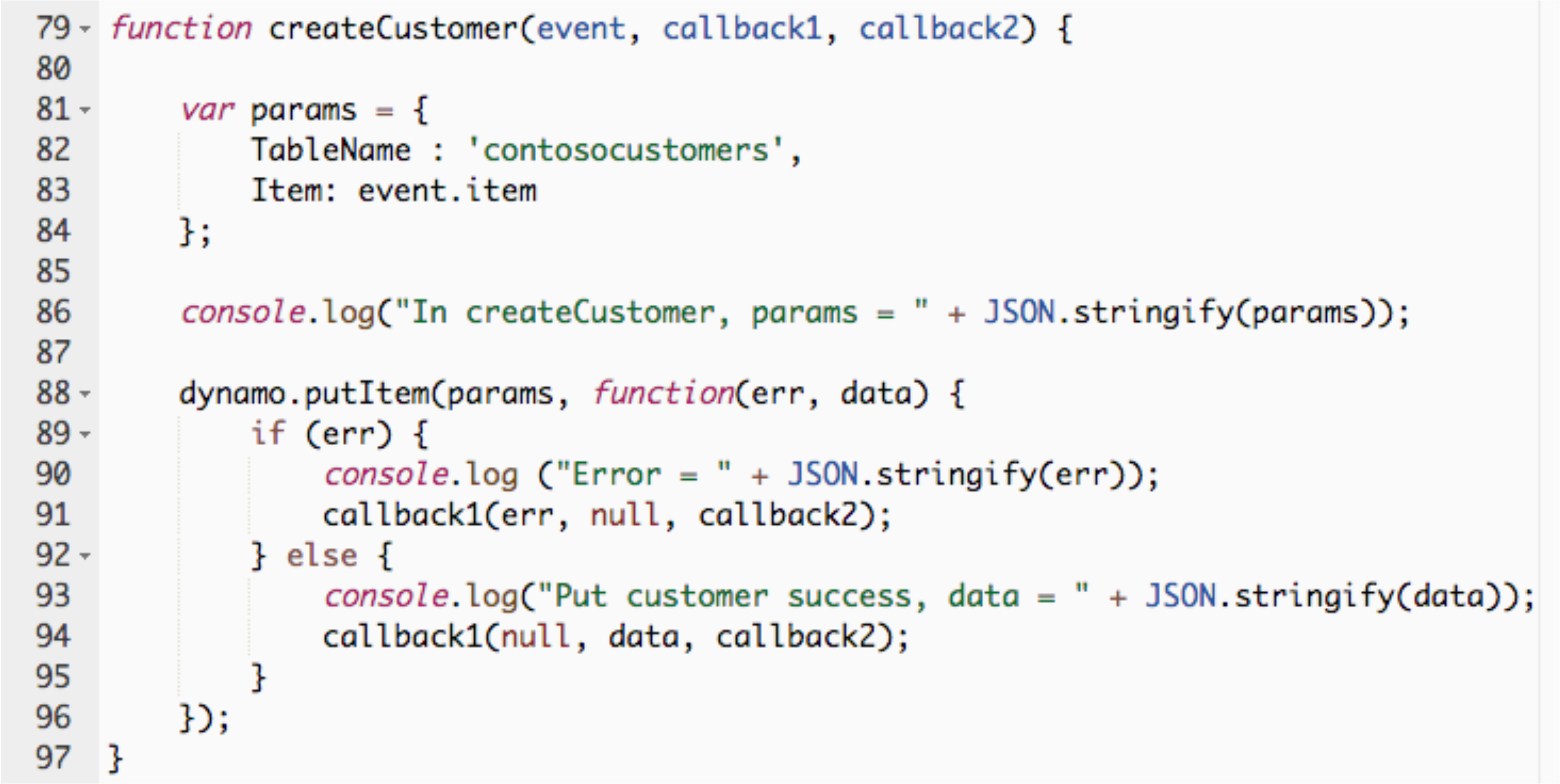
```
{
        "callbackWaitsForEmptyEventLoop":true,
        "logGroupName":"/aws/lambda/SimpleDynamoDB",
        "logStreamName":"2016/09/14/[$LATEST]df3cc187a8be44ada8ada02a85f8f4bd",
        "functionName":"SimpleDynamoDB",
        "memoryLimitInMB":"128",
        "functionVersion":"$LATEST","invokeid":"a4dbd4ff-7aad-11e6-aef6-97fdb5206df9",
        "awsRequestId":"a4dbd4ff-7aad-11e6-aef6-97fdb5206df9",
        "invokedFunctionArn":"arn:aws:lambda:us-east-
1:832720255830:function:SimpleDynamoDB”
}
```

getCustomer



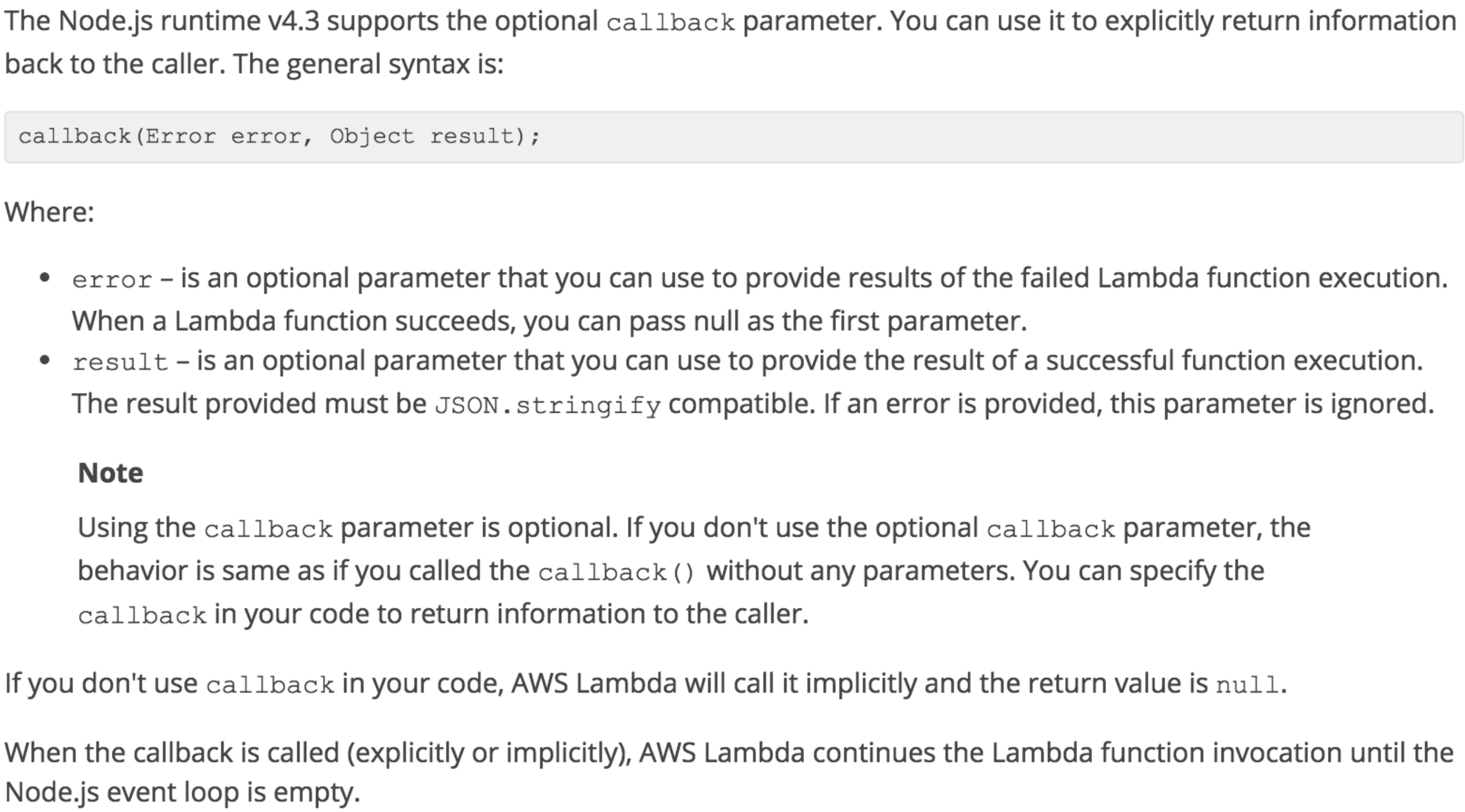
```
function getCustomer(event, callback1, callback2) {

    var params = {
        TableName : 'contosocustomers',
        Key: {"email" : event.email}
    };

    console.log("In getCustomer, params = " + JSON.stringify(params));

    dynamo.getItem(params, function(err, data) {
        if (err) {
            console.log ("Error = " + JSON.stringify(err));
            callback1(err, null, callback2);
        } else {
            console.log("Get customer success, data = " + JSON.stringify(data));
            callback1(null, data, callback2);
        }
    });
}
```

createCustomer



```
function createCustomer(event, callback1, callback2) {

    var params = {
        TableName : 'contosocustomers',
        Item: event.item
    };

    console.log("In createCustomer, params = " + JSON.stringify(params));

    dynamo.putItem(params, function(err, data) {
        if (err) {
            console.log ("Error = " + JSON.stringify(err));
            callback1(err, null, callback2);
        } else {
            console.log("Put customer success, data = " + JSON.stringify(data));
            callback1(null, data, callback2);
        }
    });
}
```

Using the Callback Parameter



The Node.js runtime v4.3 supports the optional `callback` parameter. You can use it to explicitly return information back to the caller. The general syntax is:

```
callback(Error error, Object result);
```

Where:

- `error` – is an optional parameter that you can use to provide results of the failed Lambda function execution. When a Lambda function succeeds, you can pass null as the first parameter.
- `result` – is an optional parameter that you can use to provide the result of a successful function execution. The result provided must be `JSON.stringify` compatible. If an error is provided, this parameter is ignored.

Note

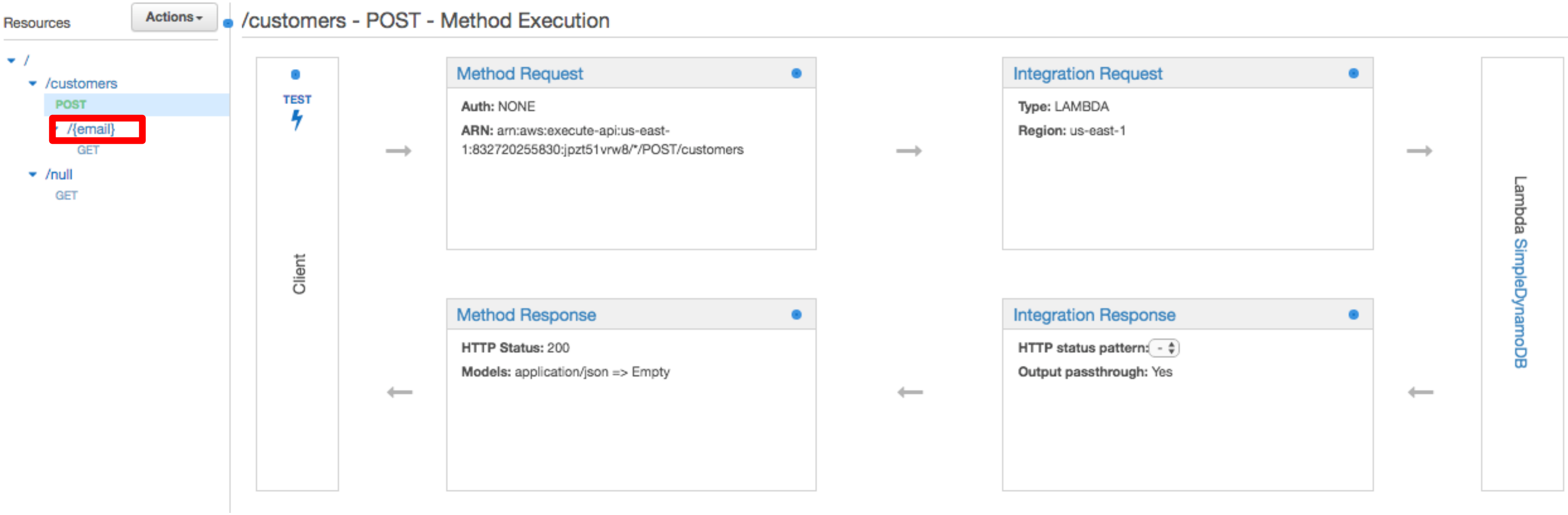
Using the `callback` parameter is optional. If you don't use the optional `callback` parameter, the behavior is same as if you called the `callback()` without any parameters. You can specify the `callback` in your code to return information to the caller.

If you don't use `callback` in your code, AWS Lambda will call it implicitly and the return value is `null`.

When the callback is called (explicitly or implicitly), AWS Lambda continues the Lambda function invocation until the Node.js event loop is empty.

Demo

API Gateway (Subtlety 1)



/customers - POST - Method Execution

Method Request

Auth: NONE

ARN: arn:aws:execute-api:us-east-1:832720255830:jpzt51vrw8/*/POST/customers

Integration Request

Type: LAMBDA

Region: us-east-1

Method Response

HTTP Status: 200

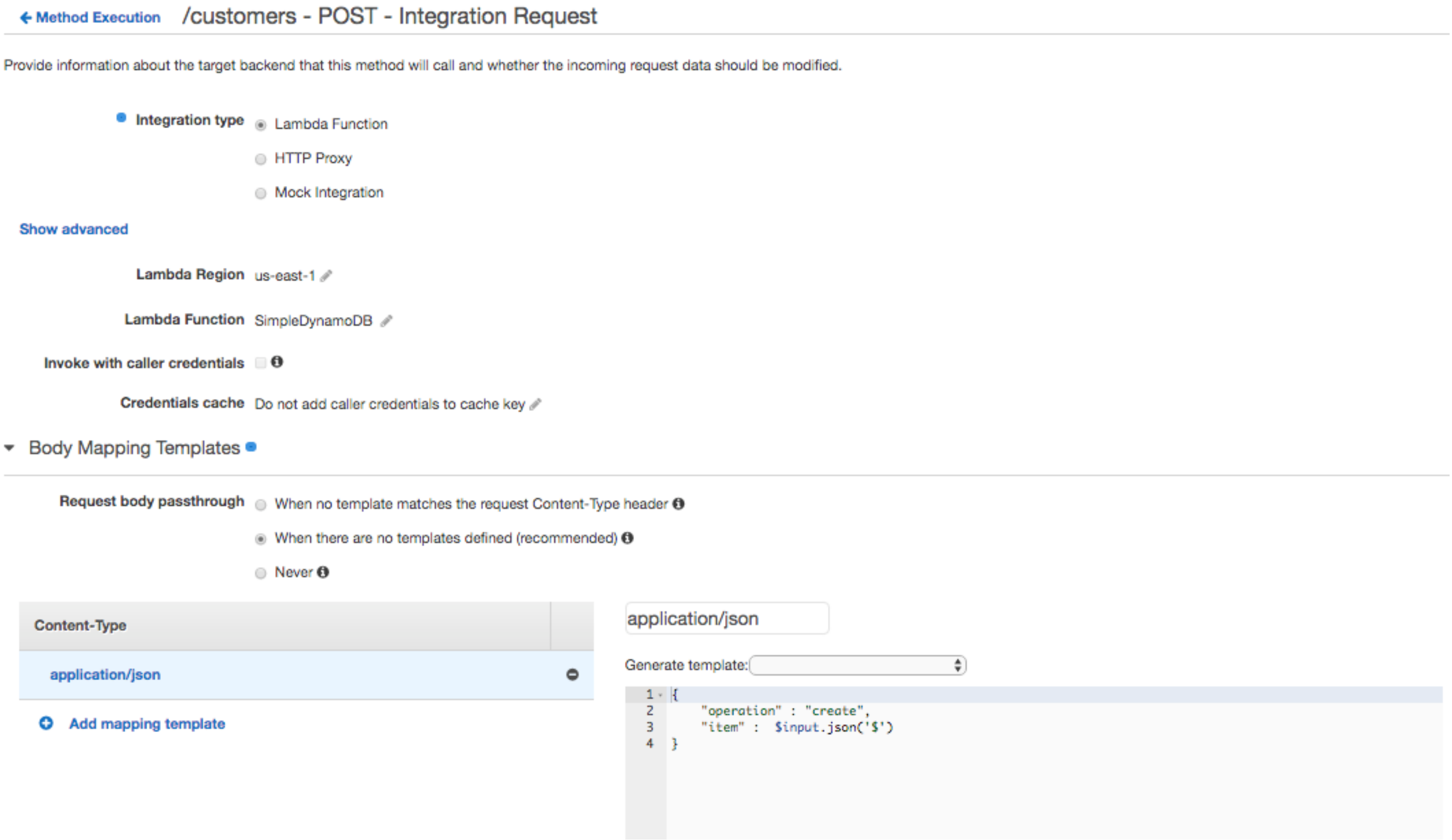
Models: application/json => Empty

Integration Response

HTTP status pattern: -

Output passthrough: Yes

Integration Request (Complexity 2)



← Method Execution   /customers - POST - Integration Request

Provide information about the target backend that this method will call and whether the incoming request data should be modified.

Integration type
● Lambda Function
○ HTTP Proxy
○ Mock Integration

Show advanced

Lambda Region: us-east-1

Lambda Function: SimpleDynamoDB

Invoke with caller credentials: □

Credentials cache: Do not add caller credentials to cache key

▾ Body Mapping Templates

Request body passthrough
○ When no template matches the request Content-Type header
● When there are no templates defined (recommended)
○ Never

Content-Type
application/json

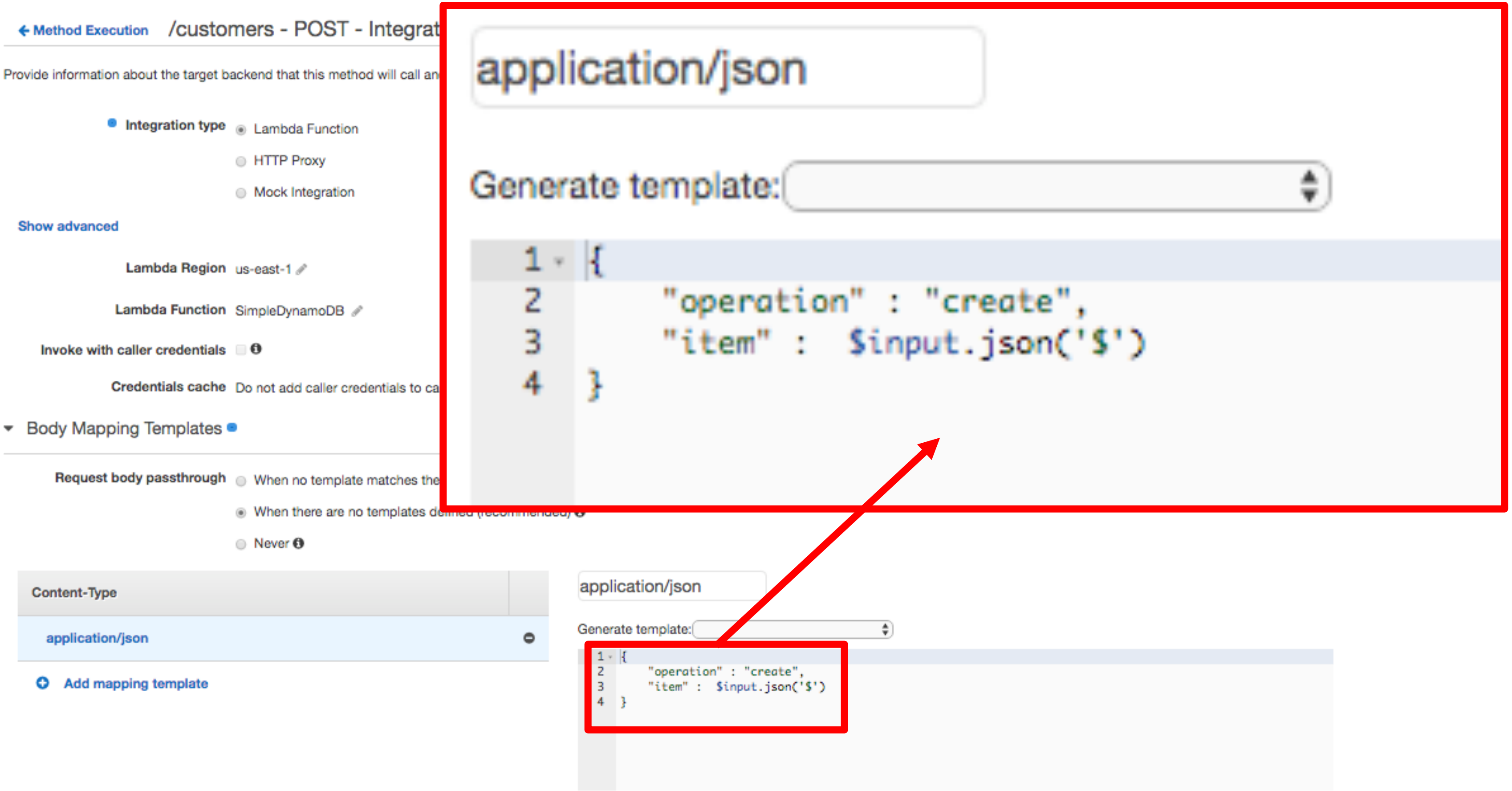
⊕ Add mapping template

application/json

Generate template:

```
{
    "operation" : "create",
    "item" :  $input.json('$')
}
```

Integration Request (Complexity 2)



← Method Execution /customers - POST - Integrat

Provide information about the target backend that this method will call an

Integration type ● Lambda Function
○ HTTP Proxy
○ Mock Integration

Show advanced

Lambda Region us-east-1

Lambda Function SimpleDynamoDB

Invoke with caller credentials □

Credentials cache Do not add caller credentials to ca

Body Mapping Templates

Request body passthrough ○ When no template matches the
● When there are no templates defined (recommended)
○ Never

Content-Type
application/json

Add mapping template

application/json

Generate template:

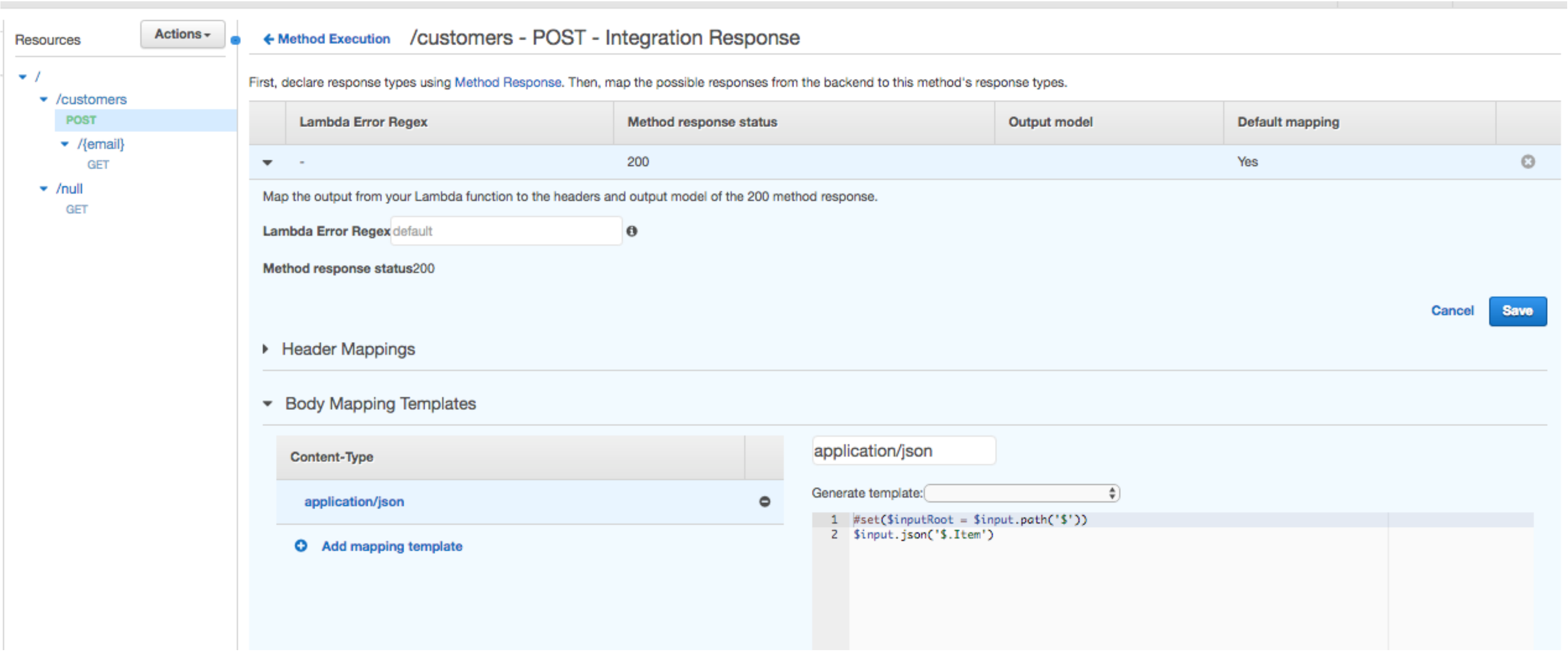
```
{
    "operation" : "create",
    "item" :  $input.json('$')
}
```

application/json

Generate template:

```
{
    "operation" : "create",
    "item" :  $input.json('$')
}
```

Integration Response (Complexity 3)



/customers - POST - Integration Response

First, declare response types using Method Response. Then, map the possible responses from the backend to this method's response types.

	Lambda Error Regex	Method response status	Output model	Default mapping	
▾	-	200		Yes	⊗

Map the output from your Lambda function to the headers and output model of the 200 method response.

Lambda Error Regex: default

Method response status: 200

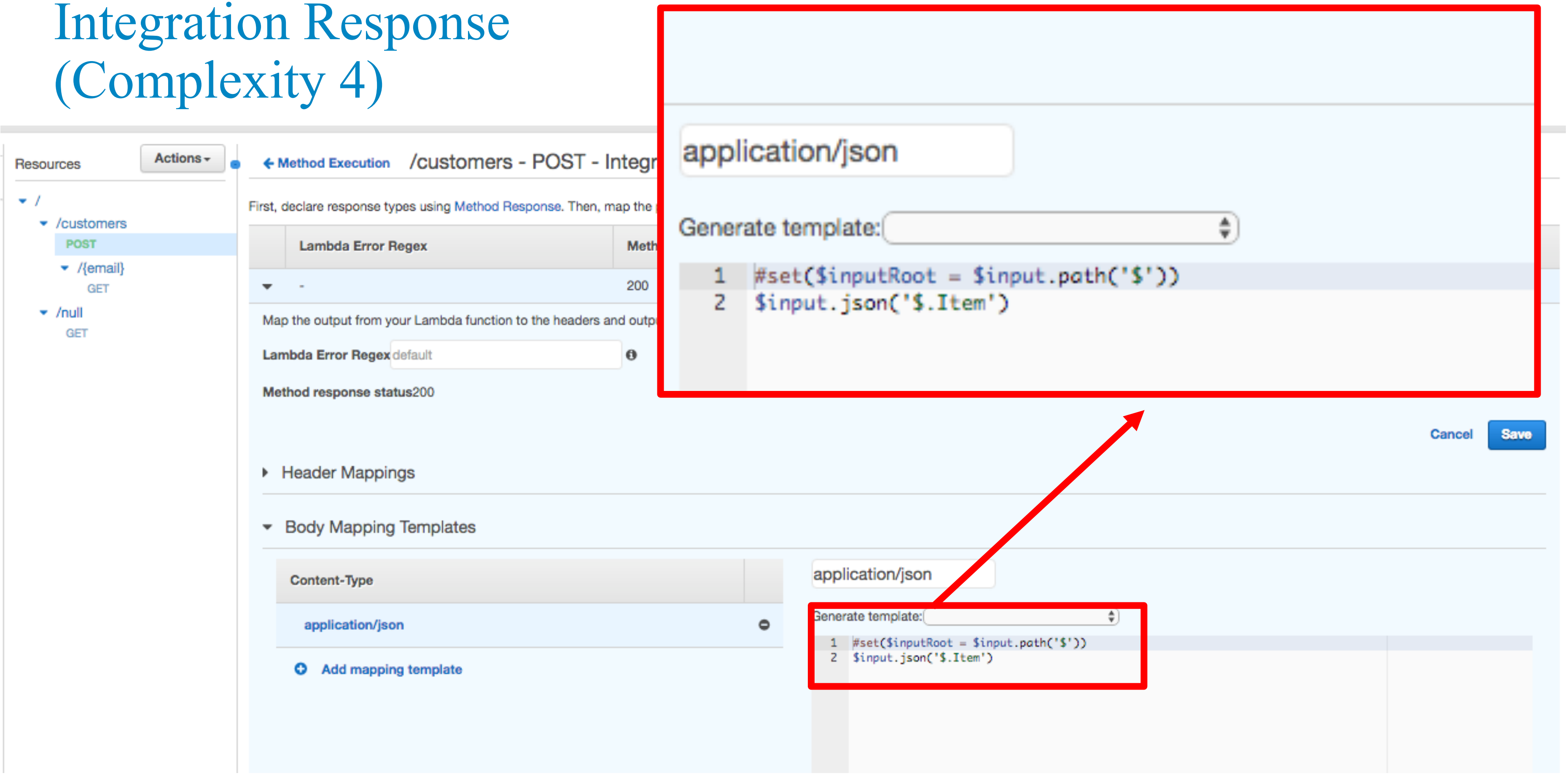
Content-Type	
application/json	⊖

application/json

Generate template:

```
#set($inputRoot = $input.path('$'))
$input.json('$.Item')
```

Integration Response (Complexity 4)



/customers - POST - Integr

First, declare response types using Method Response. Then, map the

	Lambda Error Regex	Meth
▾	-	200

Map the output from your Lambda function to the headers and outp

Lambda Error Regex default

Method response status 200

Content-Type	
application/json	⊖

application/json

Generate template:

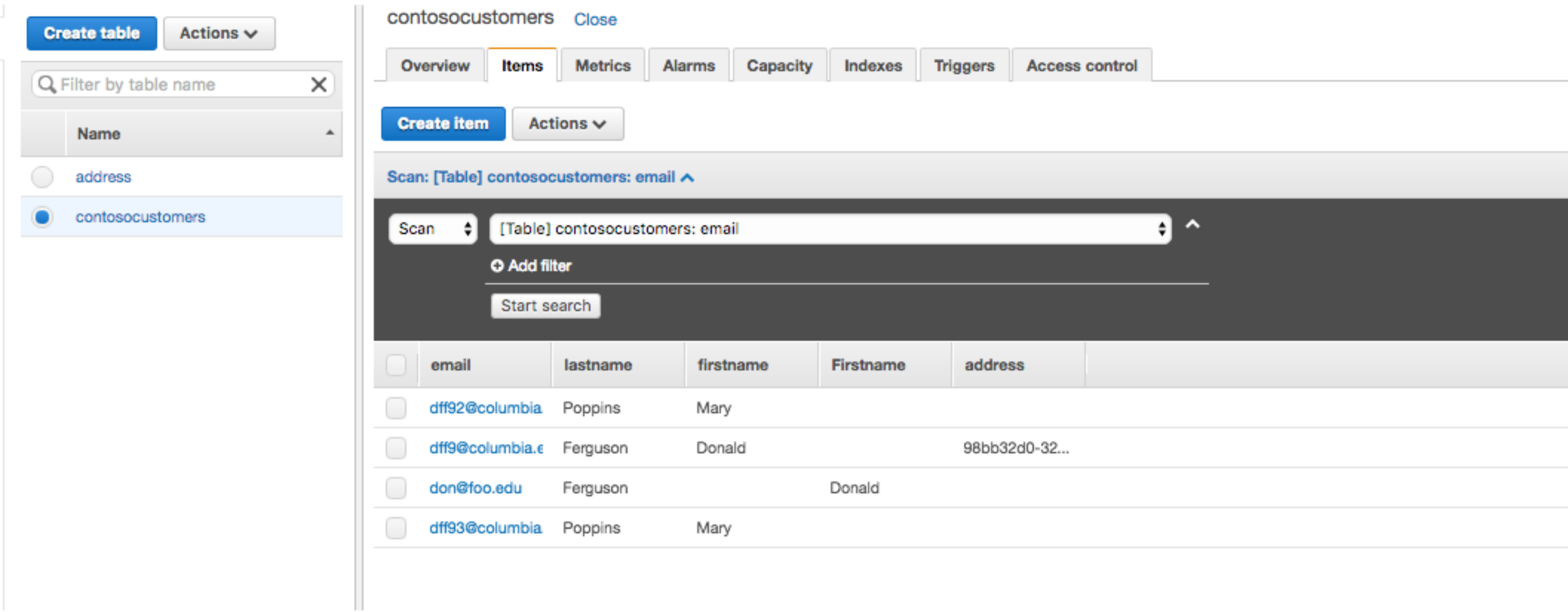
```
#set($inputRoot = $input.path('$'))
$input.json('$.Item')
```

application/json

Generate template:

```
#set($inputRoot = $input.path('$'))
$input.json('$.Item')
```

DynamoDB -- Customers



Name
○ address
● contosocustomers

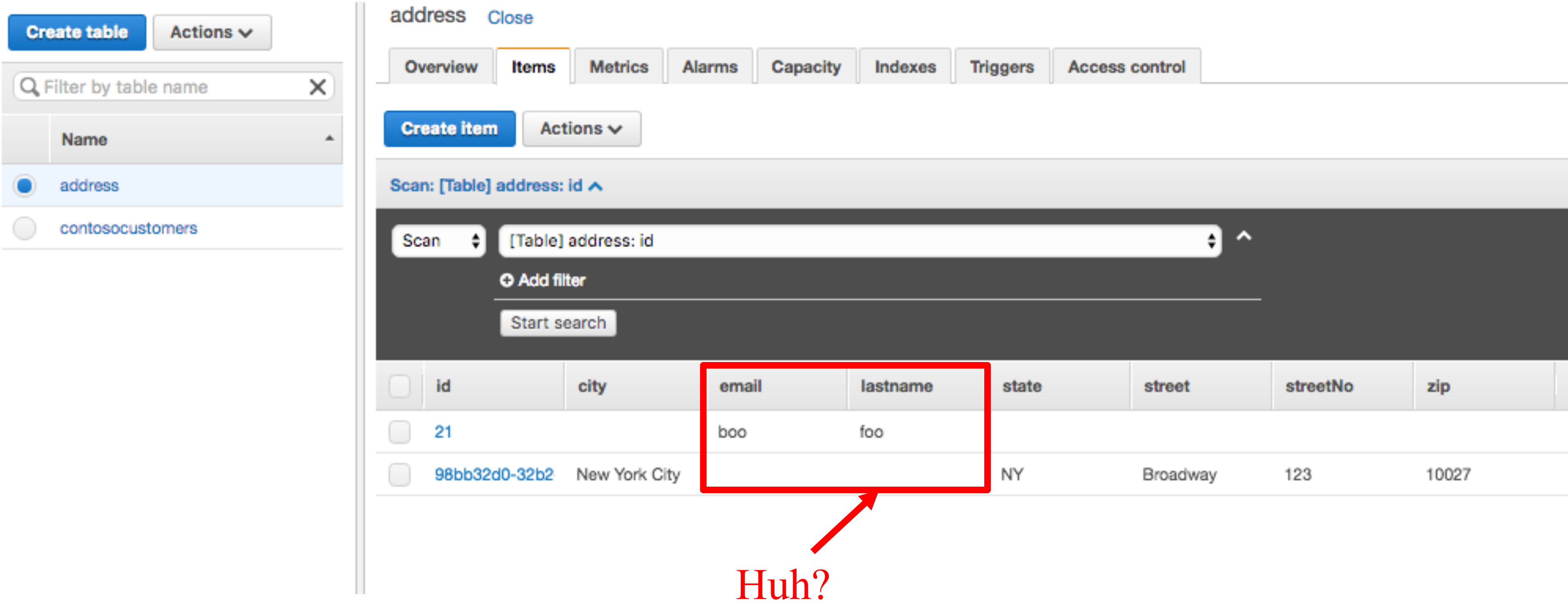
contosocustomers

Scan: [Table] contosocustomers: email

Scan | [Table] contosocustomers: email

	email	lastname	firstname	Firstname	address
□	dff92@columbia	Poppins	Mary		
□	dff9@columbia.e	Ferguson	Donald		98bb32d0-32...
□	don@foo.edu	Ferguson		Donald	
□	dff93@columbia	Poppins	Mary		

DynamoDB -- Address



Scan: [Table] address: id

Scan: [Table] address: id

id	city	email	lastname	state	street	streetNo	zip
21		boo	foo				
98bb32d0-32b2	New York City			NY	Broadway	123	10027

First Real Project

Anatomy of a URL

Basic model is

http://something.edu/someapplication/extent/p1/collection/p2/anothercollection?queryparameters.

where

- Extent is all resources of a type, e.g. “Students.”
- P1 is the “ID” of a specific student.
- Collection is a set of resources related to student p1, e.g. “classes.”
- Query parameters selects from the collection, e.g. “registration=closed”

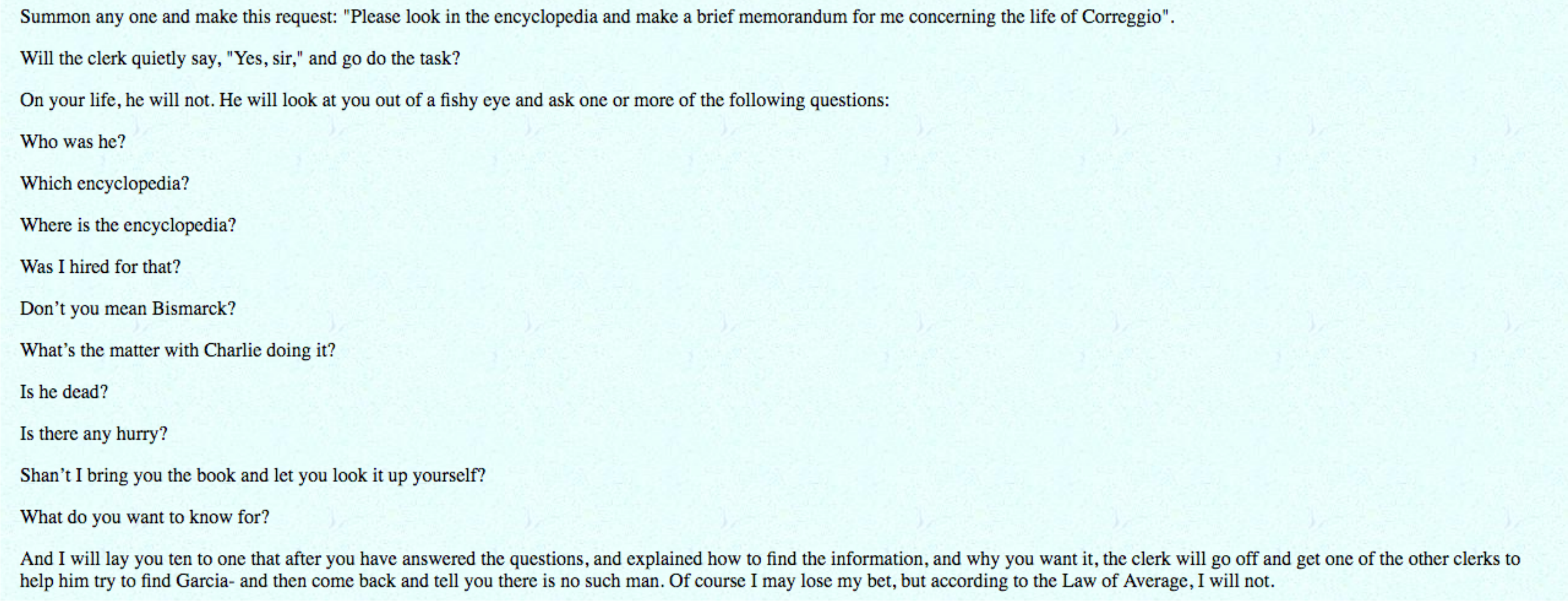
And each step is optional, e.g.

- http://something.edu/someapplication/extent?college=SEAS
- http://something.edu/someapplication/faculty/ferguson/courses/e6998

Assignment Requirements

- Define DynamoDB Tables
  - Customer (lastname, firstname, email, phone number, address_ref)
  - Address (UUID, city, street, number, zip code)
- Implement a Lambda function for Customer and Address, e.g. for Customer
  - Methods
    - GET – key is "email."
    - POST (Create)
      - Body is the data, but …
      - Cannot create (POST) if there is already a customer with that email.
    - PUT (Update)
      - Body is a subset of the JSON fields.
      - Update only those fields.
      - Cannot update an object that does not exist.
    - DELETE – key is email.
  - For all function, implement validation checks, e.g. no ”new fields,” zipcode is a number with 5 digits, …
- API Gateway
  - Define resources /Customers and /Addresses
  - POST on /Customers and /Addresses
  - GET, PUT and DELETE on /Customers/{email} and /Addresses/{id}
  - Navigation works /Customers/{email}/address returns the address.

Think, and “Carry a Message to Garcia” (https://en.wikipedia.org/wiki/A_Message_to_Garcia)



Summon any one and make this request: "Please look in the encyclopedia and make a brief memorandum for me concerning the life of Correggio".

Will the clerk quietly say, "Yes, sir," and go do the task?

On your life, he will not. He will look at you out of a fishy eye and ask one or more of the following questions:

Who was he?

Which encyclopedia?

Where is the encyclopedia?

Was I hired for that?

Don’t you mean Bismarck?

What’s the matter with Charlie doing it?

Is he dead?

Is there any hurry?

Shan’t I bring you the book and let you look it up yourself?

What do you want to know for?

And I will lay you ten to one that after you have answered the questions, and explained how to find the information, and why you want it, the clerk will go off and get one of the other clerks to help him try to find Garcia- and then come back and tell you there is no such man. Of course I may lose my bet, but according to the Law of Average, I will not.

Think. Try. You will make mistakes. Correcting mistakes is how we learn.

About This Course … …

- *Five* equally weighted projects form the basis of grading.
  - Please form project teams of 4 to 5 students.
  - Each project is approximately two weeks, and extends previous projects.
  - Submission
    - "Top-Level Design Specification" document.
    - Demo/presentation.
    - Code and code review.
- Regular office hours are before the lecture (0830 – 1000). Location 415 CEPSR.
- I can lecture much, much faster than you can absorb, document, code, test, …
  - We will occaisionally not hold a lecture and I will hold extended office hours to include normal lecture hours.
    - Q&A
    - Project reviews
    - ... ...
  - This will happen two or three times at most.
  - I am travelling on 27-Oct-2016. There will not be a lecture or office hours. I will schedule a makeup, extended office hour session.
- Course material/textbooks
  - Recommend: *Serverless Architectures on AWS,* P. Sbarski and P.Kroonenburg. Prepublication review, and I am getting you copies.
  - I will suggest several links to papers, tutorials, … on the web.
- Most of the project work and material will be on Amazon Web Services.
  - You will need an AWS "free" developer acount. Please let me know if this is an issue for you.
  - I will try to cover other clouds and APIs, which may also require "free" accounts.

REST API Design (Intro)

Representational State Transfer (REST)

- People confuse
  - Various forms of RPC/messaging over HTTP
  - With REST
- REST has six core tenets
  - Client/server
  - Stateless
  - Caching
  - Uniform Interface
  - Layered System
  - Code on Demand

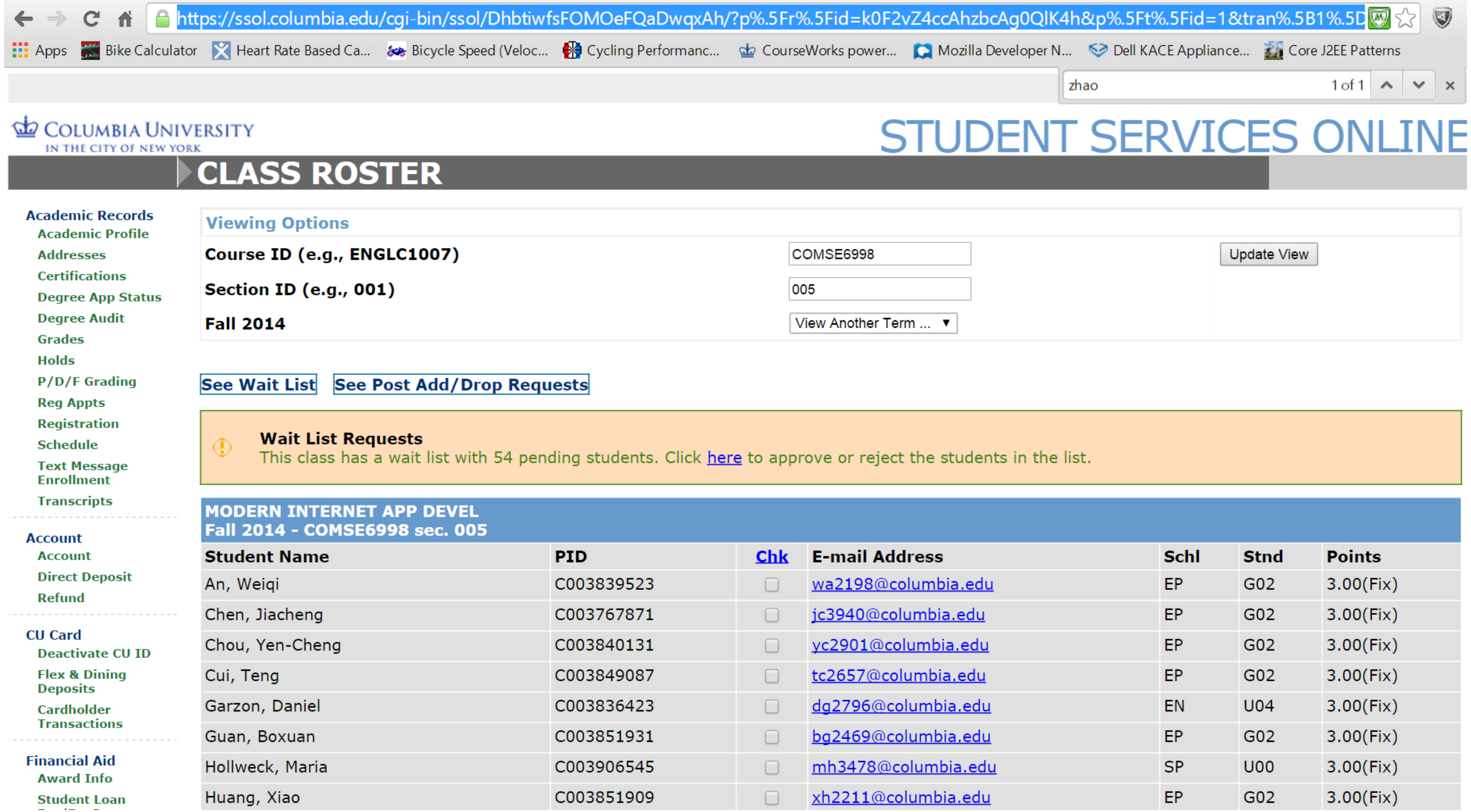
REST Tenets

- Client/Server (Obvious)
- Stateless is a bit confusing
  - The server/service maintains *resource* state, e.g. Customer and Agent info.
  - The *conversation* is stateless. The client provides all conversation state needed for an API invocation. For example,
    - customerCursor.next(10) requires the *server* to remember the client's position in the iteration through the set.
    - A *stateless* call is customerCollection.next("Bob", 10). Basically, the client passes the cursor position to the server.
- Caching
  - The web has significant caching (in browser, CDNs, …)
  - The resource provider must
    - Consider caching policies in application design.
    - Explicitly set control fields to tell clients and intermediaries what to cache/when.

REST Tenets

- Uniform Interface
  - Identify/locate resources using URIs/URLs.
  - A fixed set of “methods” on resources.
    - myResource.deposit(21.13) is not allowed.
    - The calls are
      - Get
      - Post
      - Put
      - Delete
  - Self-defining MIME types (Text, JSON, XML, …).
  - Default web application for using the API.
  - URL/URI for relationship/association.
- Layered System: Client cannot tell if connected to the server or an intermediary performing value added functions, e.g.
  - Load balancing.
  - Security.
  - Idempotency.
- Code on Demand (optional): Resource Get can deliver helper code, e.g.
  - JavaScript
  - Applets

SSOL Page



CLASS ROSTER

Viewing Options

Course ID (e.g., ENGLC1007)	COMSE6998	Update View
Section ID (e.g., 001)	005	
Fall 2014	View Another Term ... ▼	

See Wait List See Post Add/Drop Requests

Wait List Requests
This class has a wait list with 54 pending students. Click here to approve or reject the students in the list.

MODERN INTERNET APP DEVEL
Fall 2014 - COMSE6998 sec. 005

Student Name	PID	Chk	E-mail Address	Schl	Stnd	Points
An, Weiqi	C003839523	□	wa2198@columbia.edu	EP	G02	3.00(Fix)
Chen, Jiacheng	C003767871	□	jc3940@columbia.edu	EP	G02	3.00(Fix)
Chou, Yen-Cheng	C003840131	□	yc2901@columbia.edu	EP	G02	3.00(Fix)
Cui, Teng	C003849087	□	tc2657@columbia.edu	EP	G02	3.00(Fix)
Garzon, Daniel	C003836423	□	dg2796@columbia.edu	EN	U04	3.00(Fix)
Guan, Boxuan	C003851931	□	bg2469@columbia.edu	EP	G02	3.00(Fix)
Hollweck, Maria	C003906545	□	mh3478@columbia.edu	SP	U00	3.00(Fix)
Huang, Xiao	C003851909	□	xh2211@columbia.edu	EP	G02	3.00(Fix)

Anatomy of a URL

- SSOL for the Classlist

  https://ssol.columbia.edu/cgi-bin/ssol/DhbtiwfsFOMOeFQaDwqxAh/?p%.5Fr%.5Fid=k0F2vZ4ccAhzbcAg0QlK4h&p%.5Ft%.5Fid=1&tran%.5B1%.5D%.5Fentry=student&tran%.5B1%.5D%.5Fterm%.5Fid=20143&tran%.5B1%.5D%.5Fcid=COMSE6998&tran%.5B1%.5D%.5Fsecid=005&tran%.5B1%.5D%.5Fsch=&tran%.5B1%.5D%.5Fdpt=&tran%.5B1%.5D%.5Fback=&tran%.5B1%.5D%.5Ftran%.5Fname=scrs

- This is
  - Not REST
  - This is some form of Hogwarts spell
  - This is even *bad* for a web page

Anatomy of a URL

Basic model is

http://something.edu/someapplication/extent/p1/collection/p2/anothercollection?queryparameters.

where

- Extent is all resources of a type, e.g. “Students.”
- P1 is the “ID” of a specific student.
- Collection is a set of resources related to student p1, e.g. “classes.”
- Query parameters selects from the collection, e.g. “registration=closed”

And each step is optional, e.g.

- http://something.edu/someapplication/extent?college=SEAS
- http://something.edu/someapplication/faculty/ferguson/courses/e6998

Swagger

Swagger is a model/language for designing and thinking about good REST APIs.

And is well-integrated with AWS, API Gateway and other development tools.

(Swagger walkthrough here).

First Assignment

Simple

1. Signup for AWS
2. Define a simple URL model
   - …/customer
     - GET
     - POST
     - customer/id
       - GET
       - PUT
       - DELETE
3. Implement with Lambda functions.
4. Dummy data; no need for DB access.