COMSE6998: Modern Serverless Cloud Applications

Lecture 1: Introduction and Concepts

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

Agenda

Agenda

- Introduction
  - A little about me.
  - Class logistics.
  - Q&A.
- Course Content and Overview
  - Application we will build and technology we will learn/use.
  - Technology evolution: web apps → microservices → serverless.
  - Introduction to serverless (cloud) applications.
- Break
- Build our First, Very Simple, Serverless Application
  - Java example.
  - AWS Console.
  - NodeJS example.
  - Amazon API Gateway.
- Introduction to REST and REST APIs
  - Core concepts.
  - Anatomy of a URL: Our first best practice/design pattern.
- First project.

Introduction

A Little About Me …

- Career
  - Columbia
    - 11 years as a student: B.A., M.S., M.Phil and Ph.D from Columbia University.
    - Ph.D. Thesis – *The Application of Microeconomics to the Design of Resource Allocation and Control Algorithms.*
    - Previously taught 4 classes at Columbia.
  - IBM
    - IBM Research for 10 years; IBM Software Group for 10 years.
    - Foundational work on web applications, J2EE and Web Services.
    - IBM Fellow and Chief Architect for IBM Software Group.
  - Microsoft
    - Technical Fellow.
    - Technical strategy for future innovation in enterprise software.
    - Initial work on BizTalk.net, and Integration-Platform-as-a-Service (part of Azure).
  - CA technologies
    - Chief architect, Distinguished Engineer and CTO
    - Technical strategy and product architecture: product integration, cloud/SaaS enablement, next generation security, IT service management, collaborative IT management, …
  - Dell Software Group
    - Senior Fellow and CTO
    - Product architecture and technical strategy: cloud, IoT, security, system/application mgmt., system integration, …
  - SparqTV
    - Co-found and CTO
    - Architecture and writing code: AWS Lambda, RESTAngular, streaming video, social media integration, XMPP/chat, programmable web, OAuth2, graph data models, ...
- Interests
  - Languages: Speak Spanish well. Learning Arabic slowly. Interested in linguistics and language theory.
  - Amateur astronomer.
  - Road bicycling.
  - Martial arts: Black belt in Kenpo karate; Krav Maga.
  - 2LT New York Guard (state defense force).

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.

Course Content

Course Description

This course will cover core topics in designing, developing, deploying and delivering *serverless cloud applications*. Serverless is the natural, and perhaps inevitable, endpoint of cloud computing's evolution from IaaS, SaaS and PaaS. *The course will cover key serverless concepts and technologies in the context of a concrete, real application that a startup is developing and delivering. The solution provides interactive and collaborative video streaming.* Other features the solution provides and the course will cover include content management, integration with social media (e.g. FaceBook, Twitter), OAuth2 federated security, data analytics and insight into customer usage and behavior, programmable web, and integration-platform-as-a-service, mobile-backend-as-a-service (MBaaS).

Tentative Lecture Schedule:

1. Core Concepts: services, micro-services, serverless and serverless technology (AWS Lambda, AWS API Gateway), …
2. APIs: Design patterns for REST APIs, API Management, implementation design patterns and frameworks.
3. Data-as-a-Service: AWS Aurora/RDS, DynamoDB, S3, …
4. Interacting with Users: AWS SNS, Email, chat (XMPP)
5. Security and Authentication: HTTPS/PKI, Basic Auth, API Keys, OAuth2, … ...
6. Social Media Integration and APIs.
7. Service Integration: AWS SNS, AWS SQS, Google Pub/Sub, … ...
8. Cloud Based Integration and Integration-Platform-as-a-Service.
9. Content Management and Delivery: content management systems, digital media delivery, content distribution networks.
10. Cloud Data Insight: Google Analytics, data models, Apache Spark, Presto, …

Application design, application design patterns, and testing/continuous delivery will be a common topic woven through all lectures and projects.

SparqTV Solution Overview

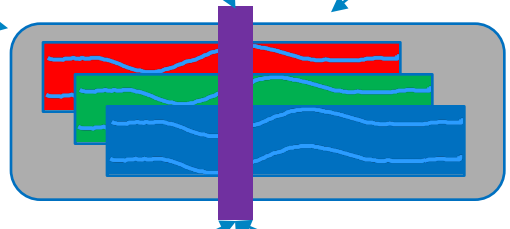
- Content Developers
  - Produce content.
  - Integrate and link with “cloud” content.
  - Interact/collaborate with consumers.

- Content Deliverers
  - Reuse the platform.
  - Using various models, e.g. SaaS, APIs.

- 3rd Parties
  - Use APIs to gain data insight.
  - Advertise and interact.

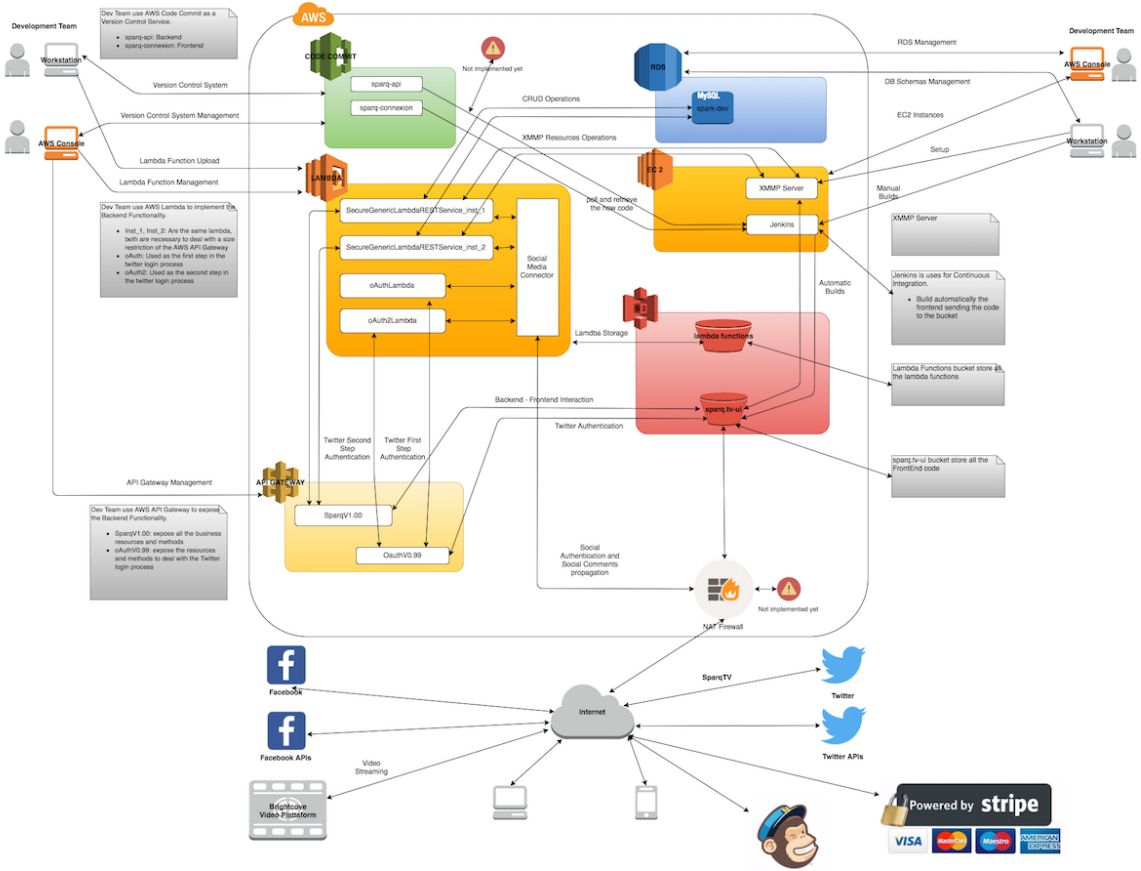
Connector/Integration with

- Digital asset/content management.
- Internal, private collaboration.
- Public social media.
- Payment.
- Email.
- … …

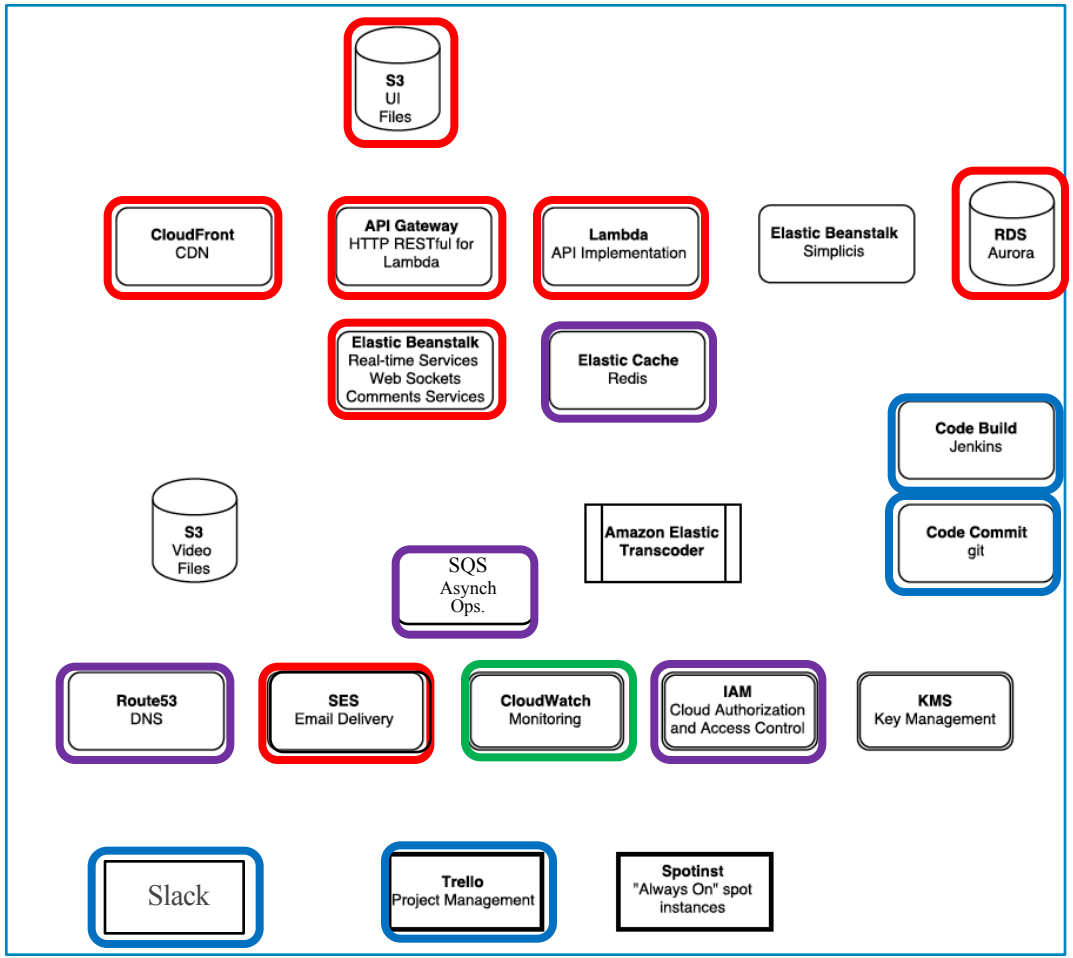


- End-Users
  - Interact with integrated “streams” and “moments.”
  - Chat and participate in communities.

SparqTV Application (Some) Technology

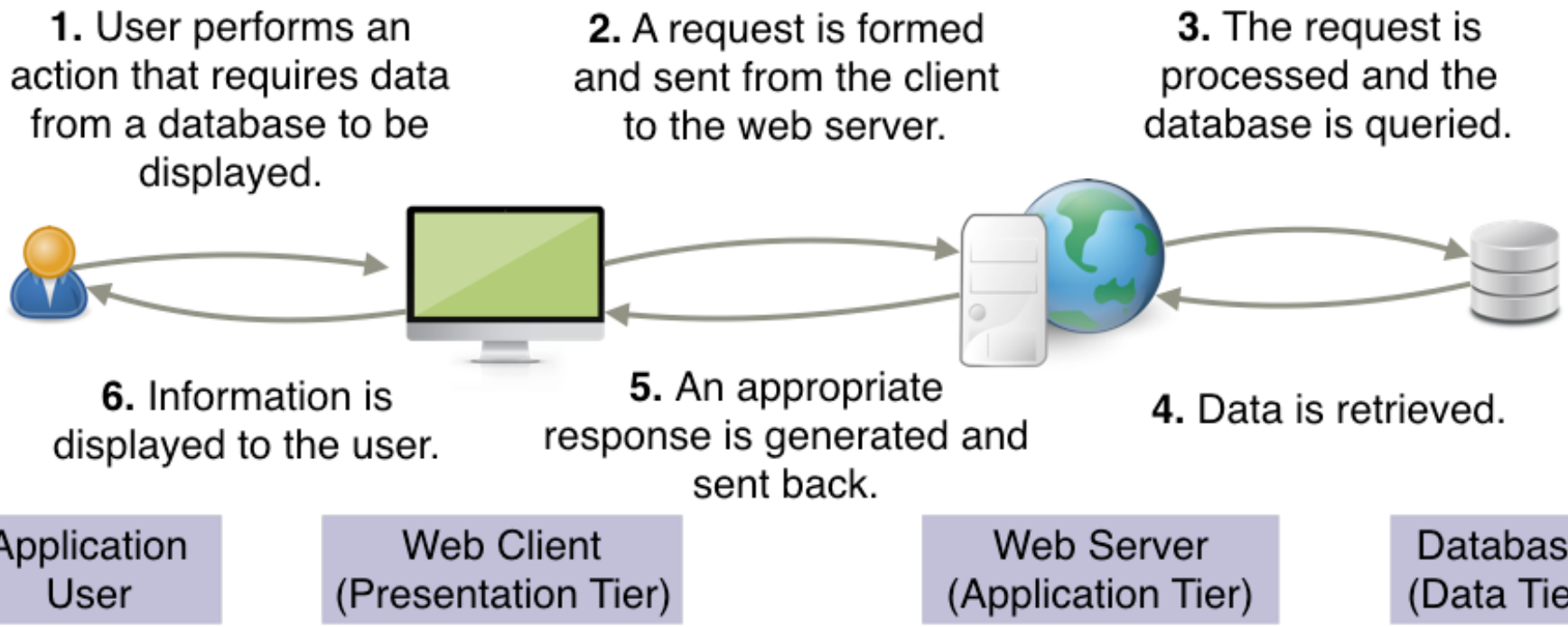


AWS Technology



- Used in alpha
  - Solution deployment.
  - Collaborative development.
  - Debug/management.
- High priority for beta.
- Some other technology
  - AngularJS, RESTAngular.
  - Java
  - XMPP
  - Cloud digital asset mgmt, transcoding and streaming.
  - Swagger

Web Application Basic Concepts



Application Architecture

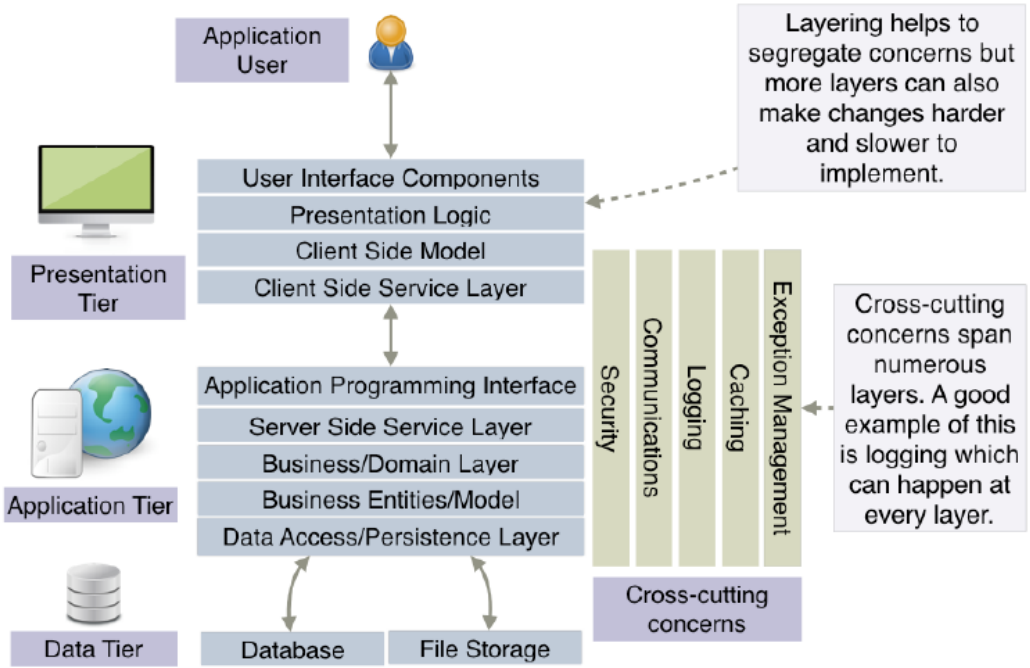
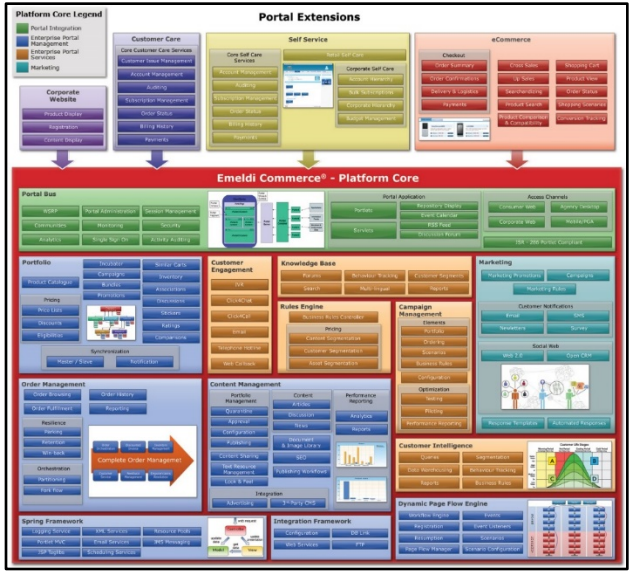
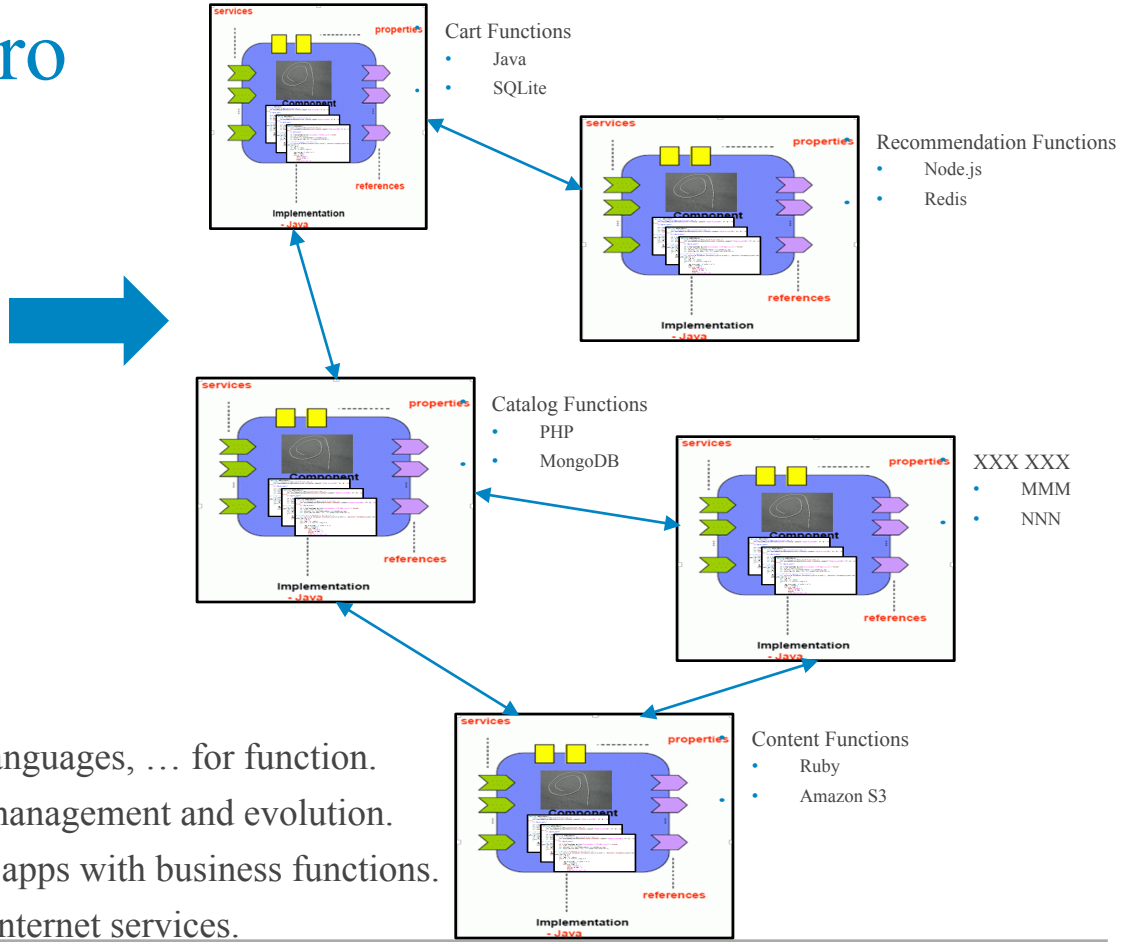


Figure 1.2. A typical 3-tier application is usually made up of a presentation, application, and data tiers. In a tier there may be multiple layers that have specific responsibilities. A developer can choose how layers will interact with one another. This can be strictly top-down or in a loose way where layers can bypass their immediate neighbors to talk to other layers.

Monolithic to Micro





Motivations

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

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

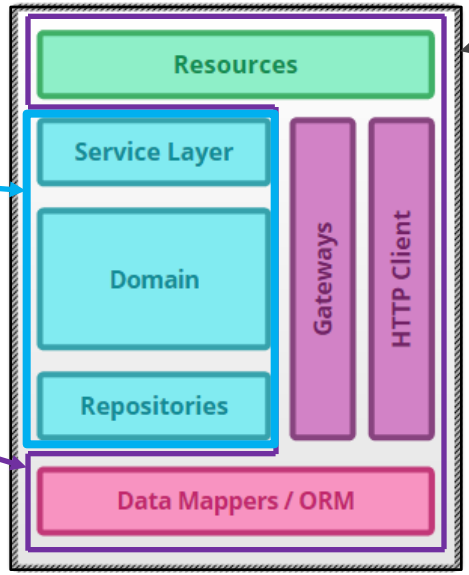
MARTIN FOWLER 4/25

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.

Inject application implementation into reusable SW containers

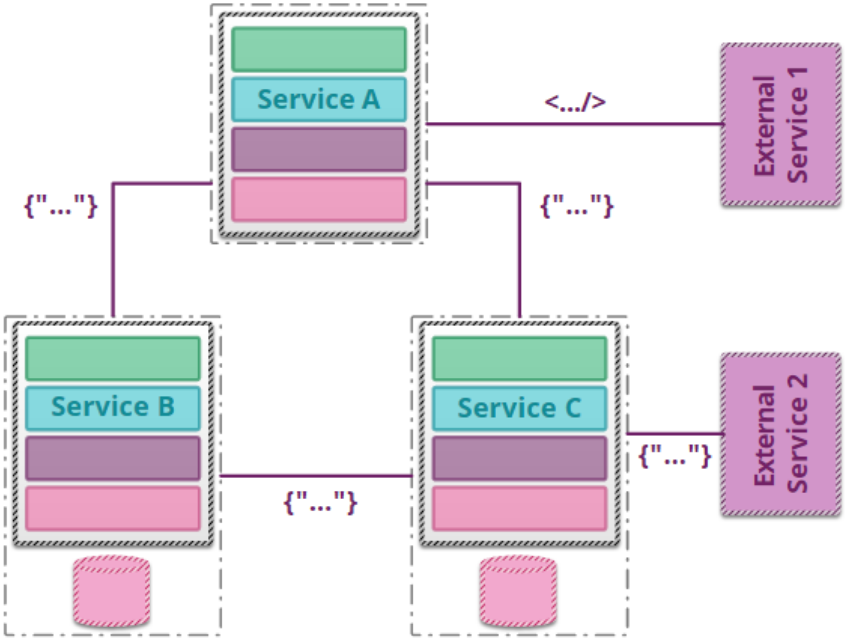
Reusable SW containers but with core technology and frameworks



Reusable infrastructure containers

Network Boundary
Protocol
Domain
External
Persistence

Multiple services work together as a system...
...to provide business valuable features



In larger systems, there are often multiple teams each with responsibility for different bounded contexts.

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

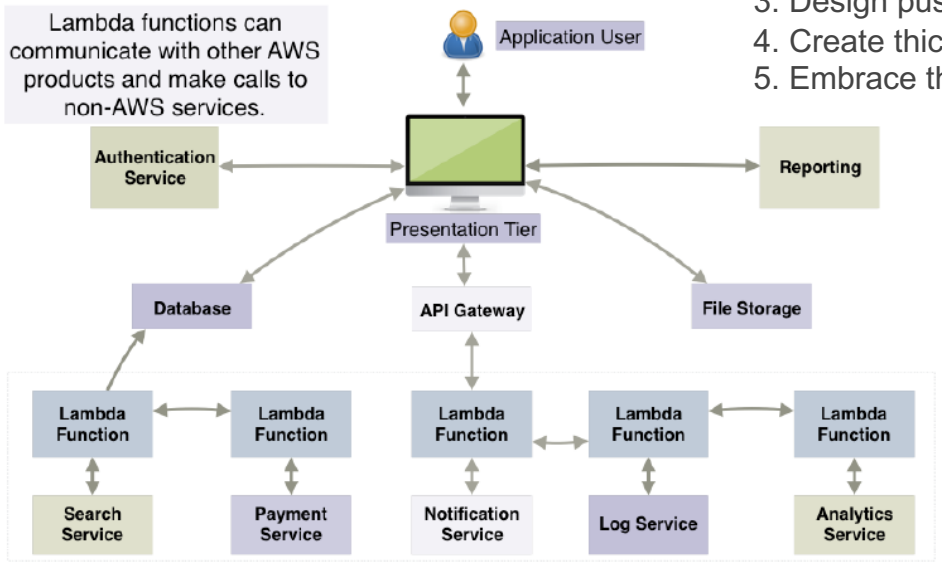
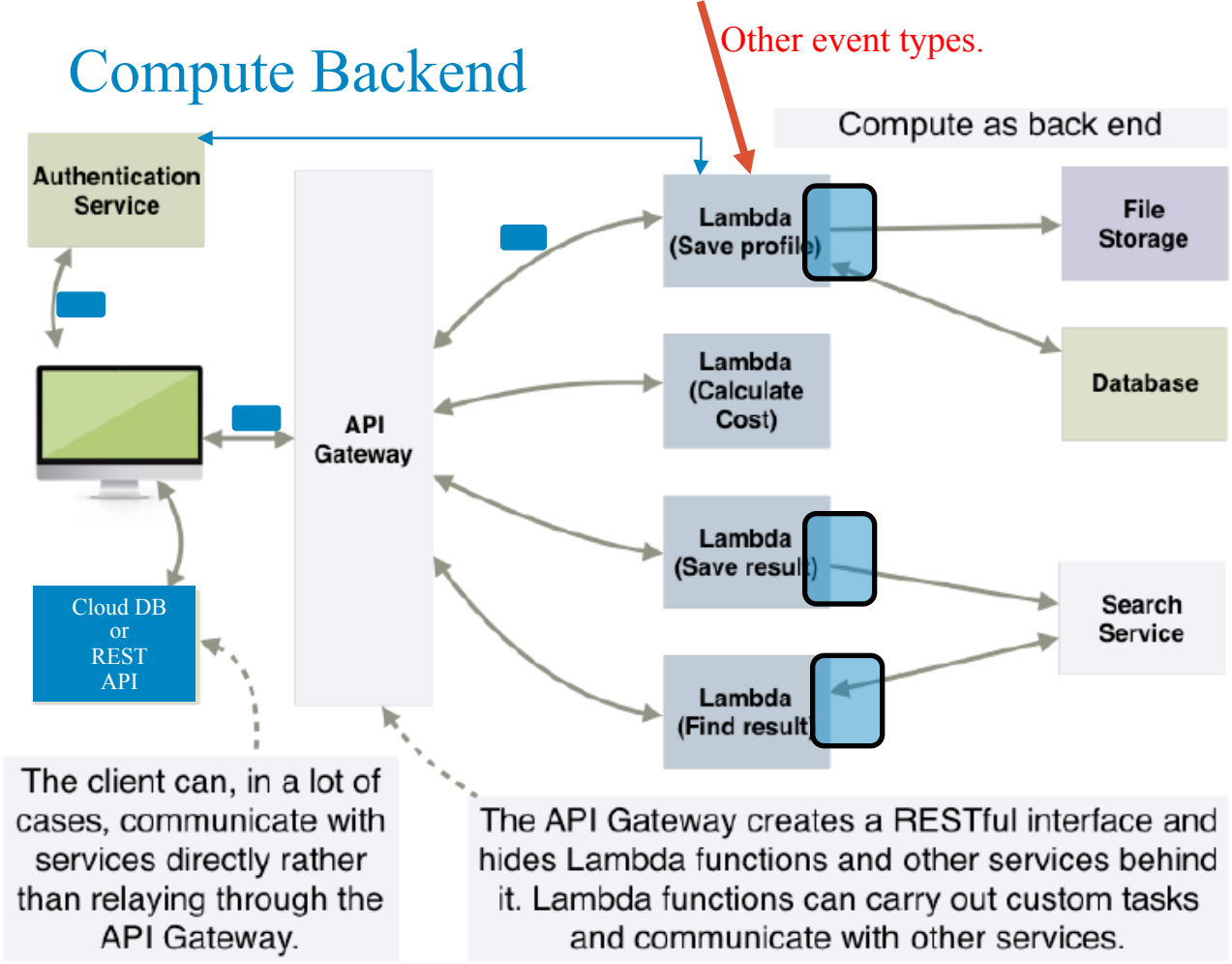


Figure 1.3: In a serverless architecture there is no single traditional back end. The front end of the application communicates directly with services, the database, or compute functions via an API gateway. Some services, however, must be hidden behind compute service functions where additional security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

Compute Backend



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.

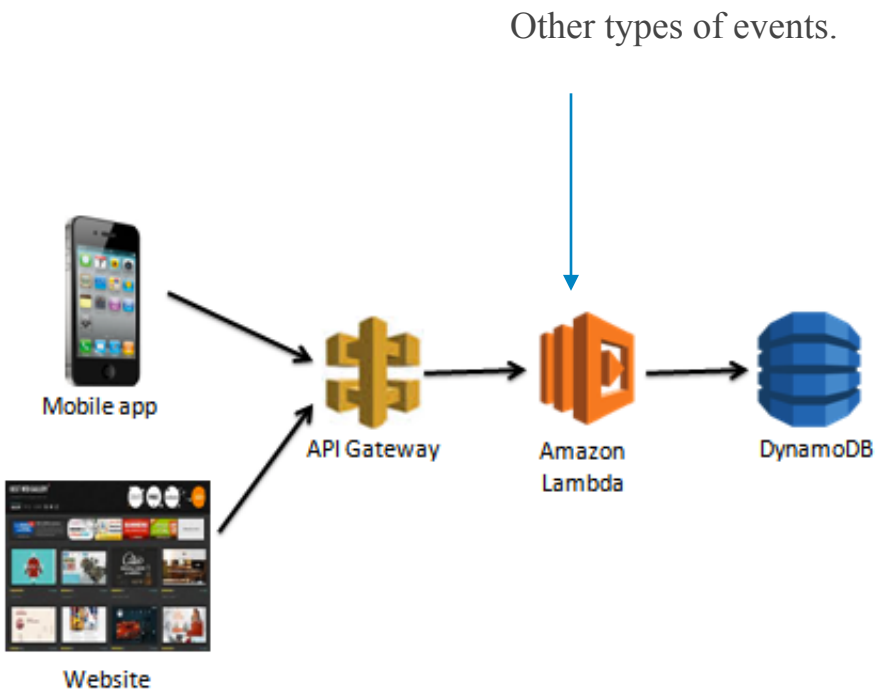
Let's Build a Lambda Function

The World's Simplest Compute Backend

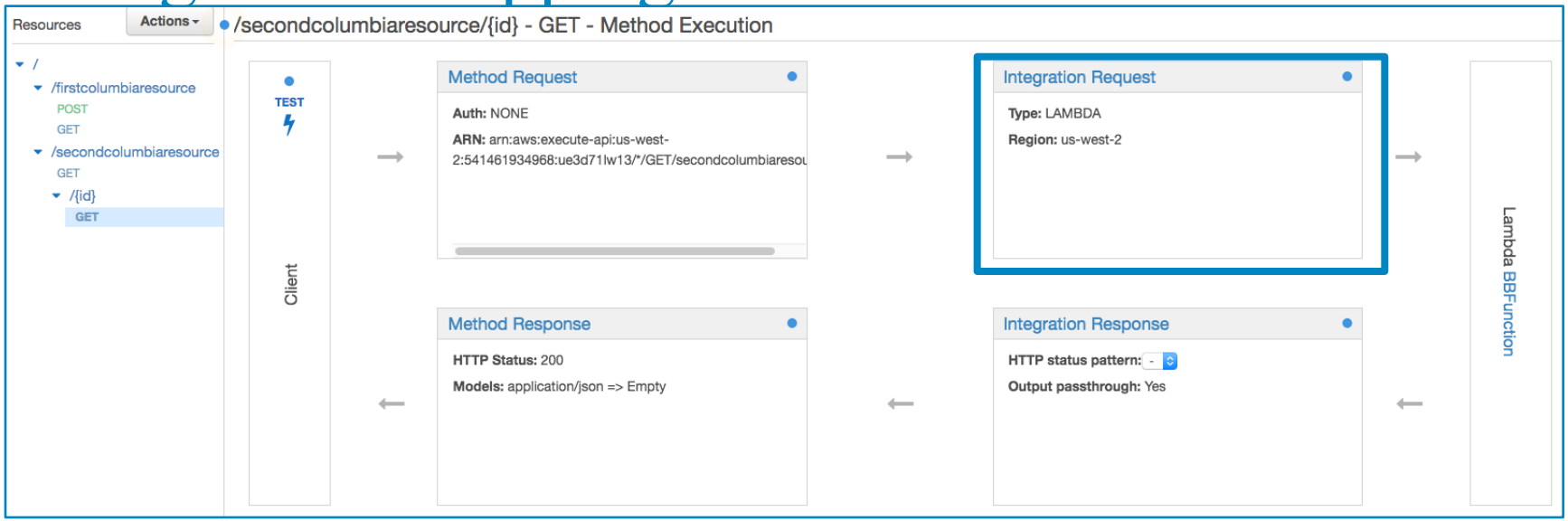
Recorded videos:

- Simple Lambda function in Eclipse.
- AWS Console, API Gateway, Postman
- Simple JavaScript example, and request/response integration.

API Gateway REST/Web enables Lambda



Integration Mapping



The Lambda functions can be triggered by multiple kinds of "event" and just receives the "event."

Something has to turn a REST/web request into a single "event" that contains

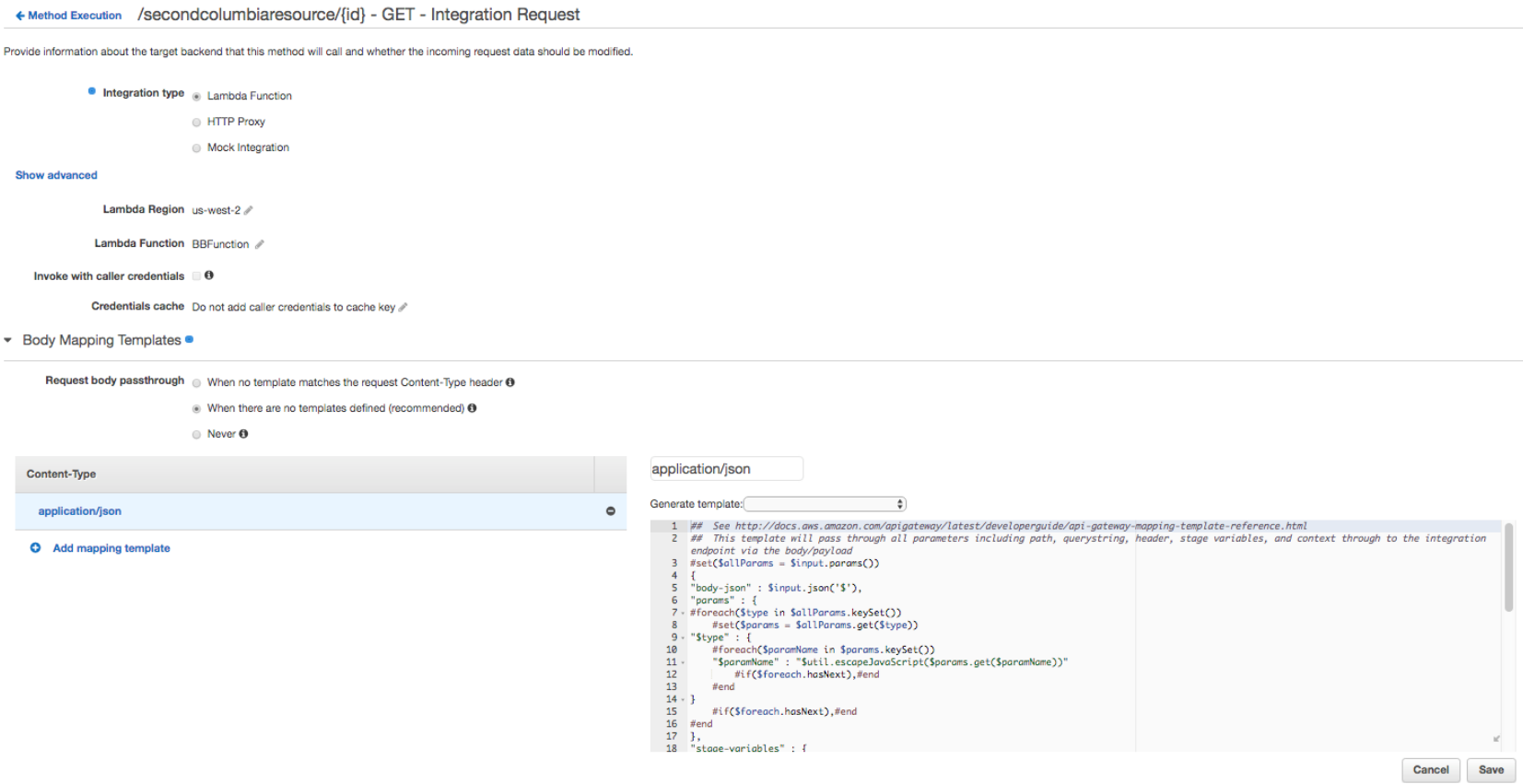
- Body
- Query parameters
- Path
- Etc.

Integration Mapping

```
Wed Sep 07 19:56:53 UTC 2016 : Endpoint request body after transformations: {
"body-json" : {},
"params" : {
"path" : {
        "id" : "101"
            }
    ,"querystring" : {
    }
    ,"header" : {
    }
    },
"stage-variables" : {
},
"context" : {
    "account-id" : "541461934968",
    "api-id" : "ue3d71lw13",
    "api-key" : "test-invoke-api-key",
    "authorizer-principal-id" : "",
    "caller" : "AIDAJ5OOT4SPFST7ZJDN4",
    "cognito-authentication-provider" : "",
    "cognito-authentication-type" : "",
    "cognito-identity-id" : "",
    "cognito-identity-pool-id" : "",
    "http-method" : "GET",
    "stage" : "test-invoke-stage",
    "source-ip" : "test-invoke-source-ip",
    "user" : "AIDAJ5OOT4SPFST7ZJDN4",
    "user-agent" : "Apache-HttpClient/4.5.x (Java/1.8.0_102)",
    "user-arn" : "arn:aws:iam::541461934968:user/donald.f.ferguson@gmail.com",
    "request-id" : "test-invoke-request",
    "resource-id" : "ouhzad",
    "resource-path" : "/secondcolumbiaresource/{id}"
    }
}
```

```
1  'use strict';
2  console.log('Loading function');
3
4  exports.handler = (event, context, callback) => {
5      //console.log('Received event:', JSON.stringify(event, null, 2));
6      console.log("Event = " + JSON.stringify(event));
7
8      var result = {};
9
10     if (event.params) {
11        if (event.params.path) {
12            result.message = "I am going to get a path parameter - id";
13            if (event.params.path.id) {
14                result.id = event.params.path.id;
15            }
16            else {
17                result.id = "No ID found.";
18            }
19        }
20        else {
21            result.message = "No path parameters.";
22        }
23        if (event.params.querystring) {
24            result.message2 = "I am going to get a query parameter - lastname";
25            if (event.params.querystring.lastname) {
26                result.lastname = event.params.querystring.lastname;
27            }
28            else {
29                result.lastname = "No last name?";
30            }
31        }
32        else {
33            result.message2 = "No query parameters";
34        }
35     }
36     else {
37         result.message = "No parameters at all.";
38     }
39
40     // We are going to assume that we got
41     callback(null, result);  // Echo back the first key value
42     // callback('Something went wrong');
43 };
```

Integration



Passthru Mapping

```
##  See http://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-mapping-template-reference.html
##  This template will pass through all parameters including path, querystring, header, stage variables, and context through to the integration endpoint via
#set($allParams = $input.params())
{
"body-json" : $input.json('$'),
"params" : {
#foreach($type in $allParams.keySet())
    #set($params = $allParams.get($type))
"$type" : {
    #foreach($paramName in $params.keySet())
    "$paramName" : "$util.escapeJavaScript($params.get($paramName))"
        #if($foreach.hasNext),#end
    #end
}
    #if($foreach.hasNext),#end
#end
},
"stage-variables" : {
#foreach($key in $stageVariables.keySet())
"$key" : "$util.escapeJavaScript($stageVariables.get($key))"
    #if($foreach.hasNext),#end
#end
},
"context" : {
    "account-id" : "$context.identity.accountId",
    "api-id" : "$context.apiId",
    "api-key" : "$context.identity.apiKey",
    "authorizer-principal-id" : "$context.authorizer.principalId",
    "caller" : "$context.identity.caller",
    "cognito-authentication-provider" : "$context.identity.cognitoAuthenticationProvider",
    "cognito-authentication-type" : "$context.identity.cognitoAuthenticationType",
    "cognito-identity-id" : "$context.identity.cognitoIdentityId",
    "cognito-identity-pool-id" : "$context.identity.cognitoIdentityPoolId",
    "http-method" : "$context.httpMethod",
    "stage" : "$context.stage",
    "source-ip" : "$context.identity.sourceIp",
    "user" : "$context.identity.user",
    "user-agent" : "$context.identity.userAgent",
    "user-arn" : "$context.identity.userArn",
    "request-id" : "$context.requestId",
    "resource-id" : "$context.resourceId",
    "resource-path" : "$context.resourcePath"
    }
}

```

Observations

- There are some powerful concepts
  - Does for web applications what wikis did for web content.
    - Define a URL.
    - Put some code there.
    - Click done and you updated the web.
  - You can mix and match languages, DBs, …
    - For any URL in your application.
    - And for different verbs on a resource, e.g. GET in JavaScript and POST in Java.
- But,
  - You need the API Gateway to map a generic Lambda function to REST/HTTP.
  - Requests are completely stateless. No “memory” between calls.
    All state has to go into a “database” or in ”client context.”

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

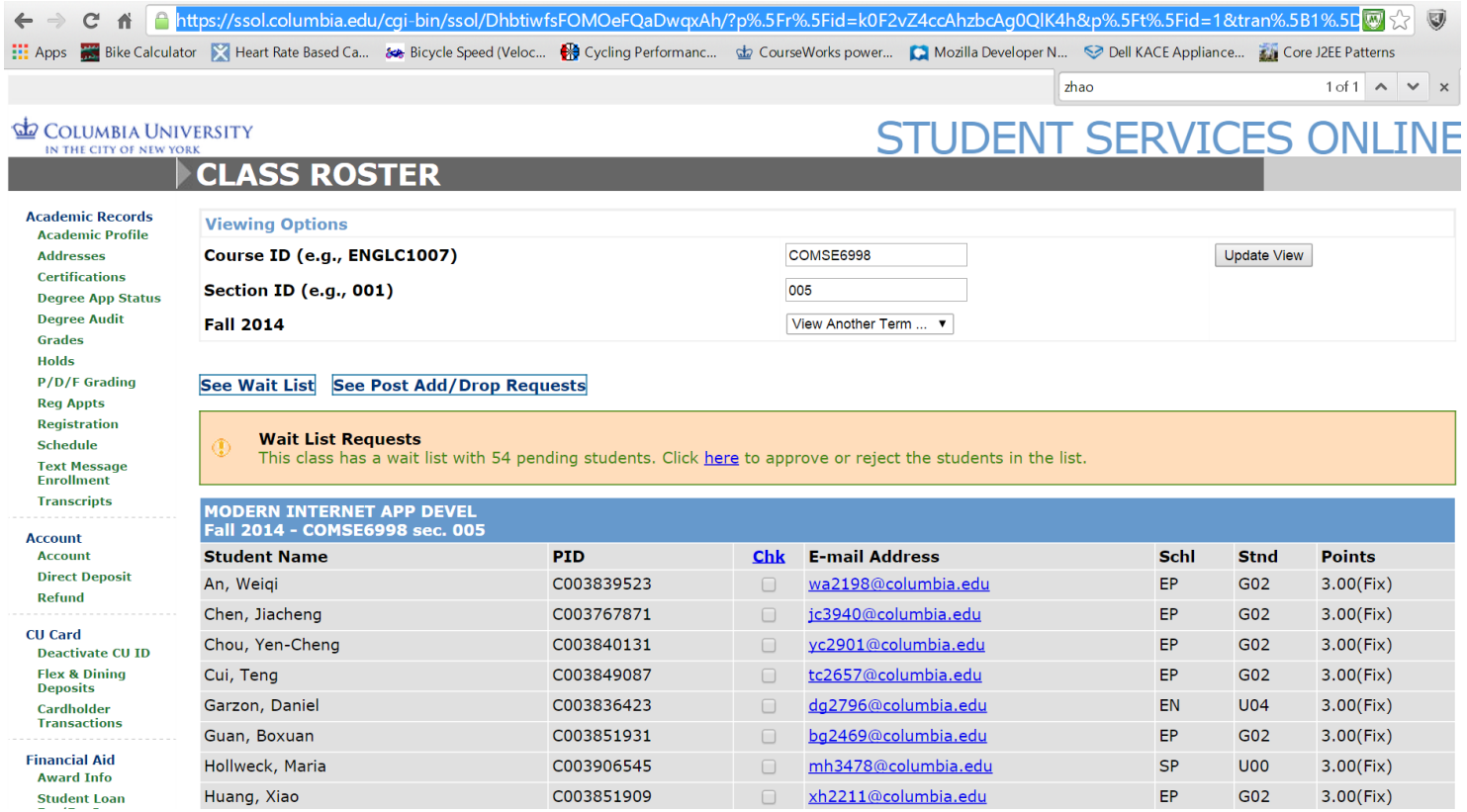
REST Tenets

- 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



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.