COMS E6998: **Microservices and Cloud Applications**

Lecture 3: Microservices, Session State, Composition,Promises, Multitenancy

Dr. Donald F. Ferguson
dff9@columbia.edu

1st Project Reminder

1st Project – Part 1

- Implement two distinct microservices
  - Person
  - Address
- Tasks
  - Use Swagger Editor to define and document REST APIs.
  - Implement an Elastic Beanstalk application (microservice) for each resource that implements 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

Microservices

1st Project – Part 1

- Implement two distinct microservices
  - Person
  - Address
- Tasks
  - Use Swagger Editor to define and document REST APIs.
  - Implement an Elastic Beanstalk application (microservice) for each resource that implements 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

Introduction to Promises

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

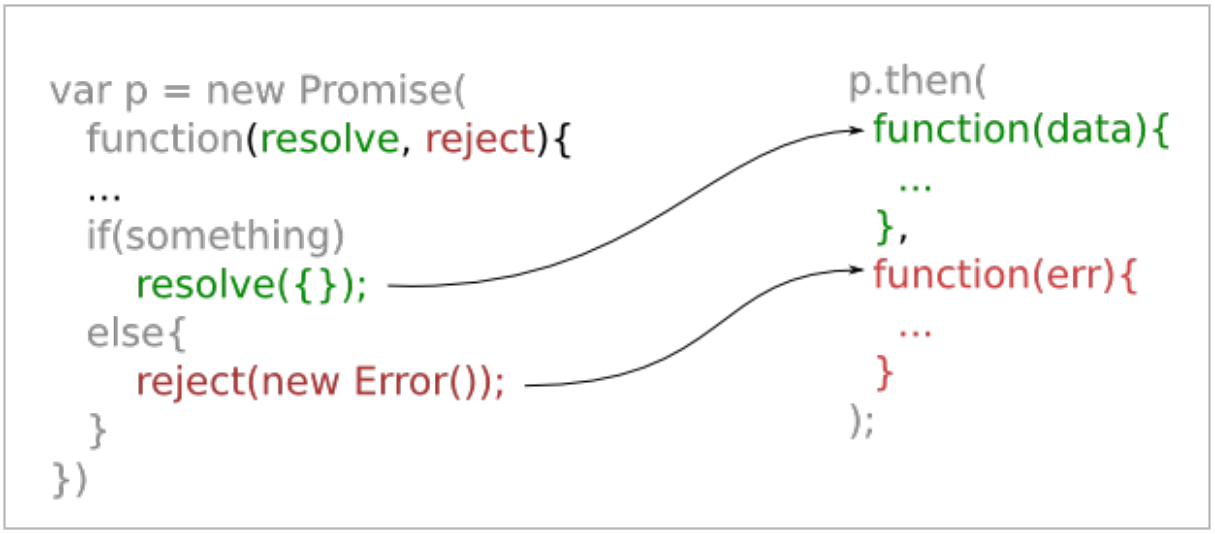
- Asynchronous requires a style of programming that
  - Is different from most applications students have written.
  - Can be counter-intuitive.
- I have seen students struggle with the transition.
- Will provide a simple overview to start, and expand through projects.

Promise Basics

Promise States

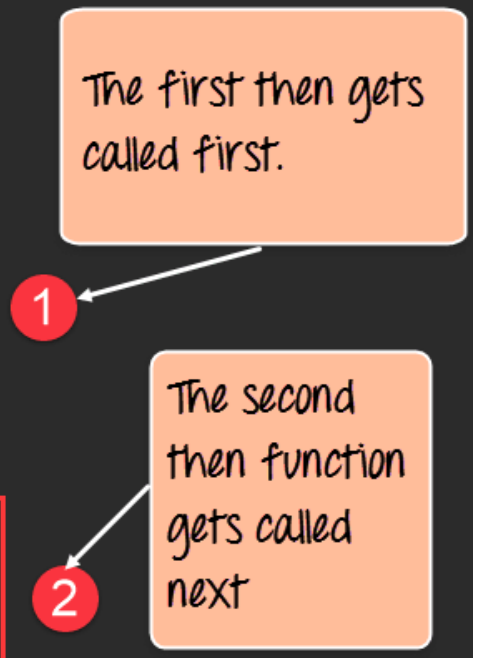
A promise can be in one of 3 states:

- Pending - the promise's outcome hasn't yet been determined, because the asynchronous operation that will produce its result hasn't completed yet.
- Fulfilled - the asynchronous operation has completed, and the promise has a value.
- Rejected - the asynchronous operation failed, and the promise will never be fulfilled. In the rejected state, a promise has a *reason* that indicates why the operation failed.



Composing Promises (https://www.guru99.com/node-js-promise-generator-event.html)

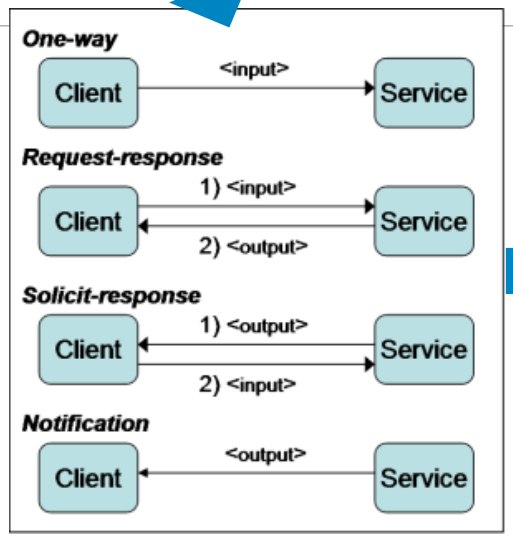
```
var Promise= require('promise');
var MongoClient = require('mongodb').MongoClient;
var url = 'mongodb://localhost/EmployeeDB';
MongoClient.connect(url)

    .then(function(db)
  {
    db.collection('Employee').insertOne({
        Employeeid :4,
        EmployeeName: "NewEmployee"})

    .then(function(db1) {
        db1.collection('Employee').insertOne({
                Employeeid: 5,
                EmployeeName: "NewEmployee1" })})});
```



Cloud/Microservice Spanning Promises

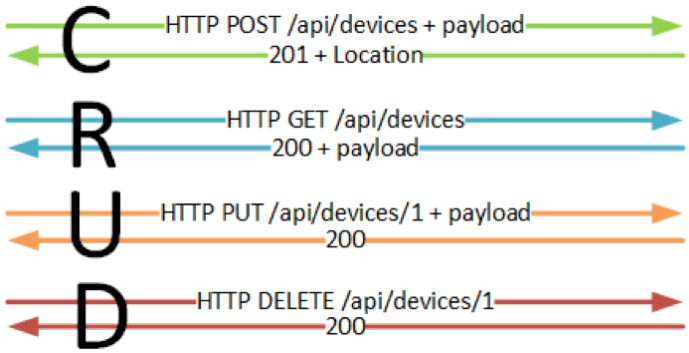
```
var p = new Promise(
  function(resolve, reject){
  ...
  if(something)
    resolve({});
  else{
    reject(new Error());
  }
})
```

```
p.then(
  function(data){
    ...
  },
  function(err){
    ...
  }
);
```

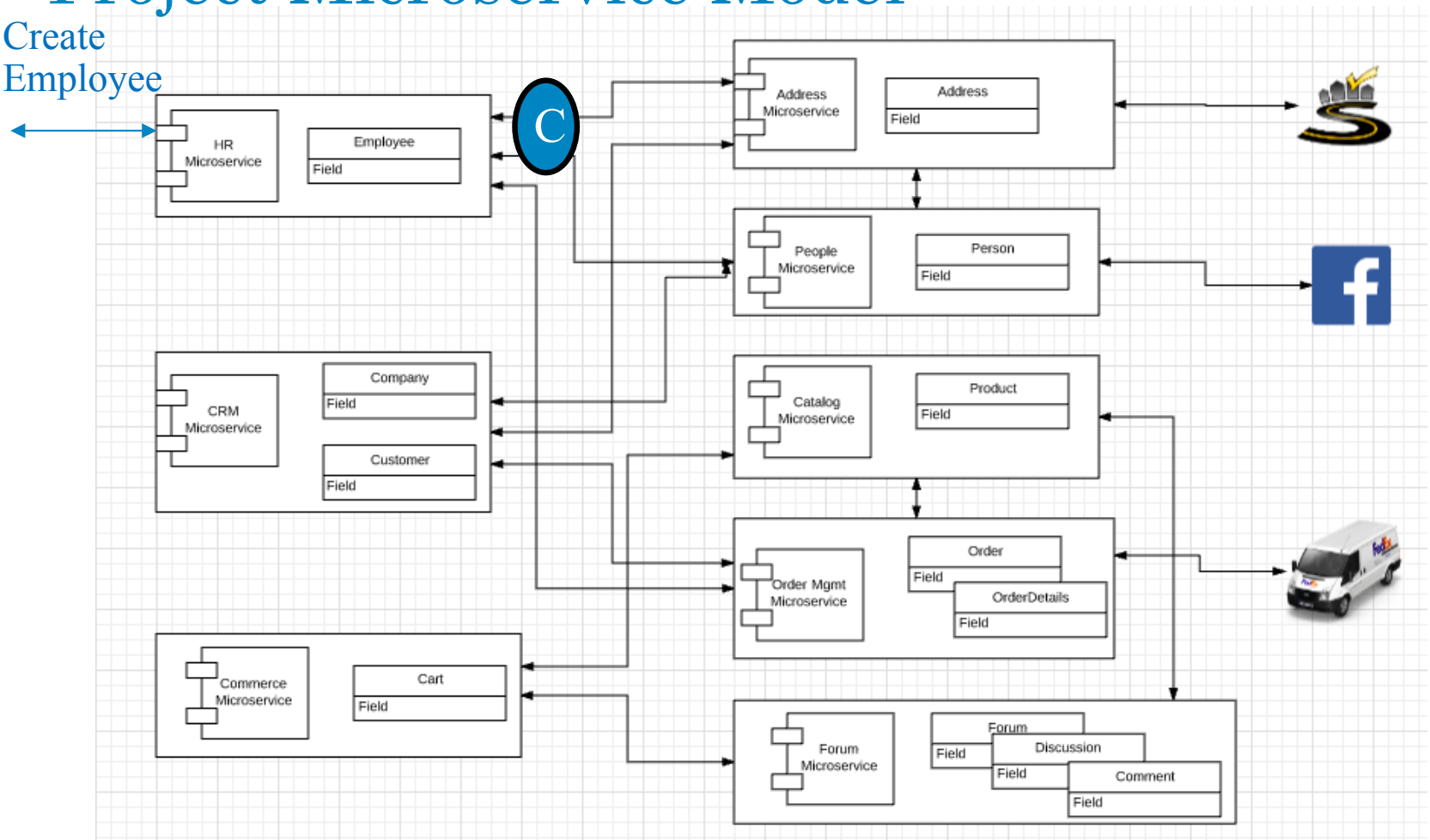


This would "seem" easy, but …

- Sometimes the language runtime "goes away."
- Multiple servers.
- etc.



Project Microservice Model



Some Pseudo-Code

- Return a Promise
- Validate and possibly reject.
- Implementation requires calling two microservices.
- Can only process when both Promises resolve.
- Success
- May need to call other asynchronous functions to recover.

```
var     createEmployee = function(newEmployeeData) {

    return new Promise(function (resolve, reject) {

        // Do some basic validation of the input, e.g.
        // All necessary fields present.
        var customerValid = validate();

        if (customerValid === false) {
            reject(meaningfulErrorData);
        }
        else {

            p1 = AddressModule.createAddress(newEmployeeData.addressInfo);
            p2 = PeopleModule.createPerson(newEmployeeData.personInfo);

            allPromises = [];
            allPromises.push(p1);
            allPromises.push(p2);

            Promise.all(allPromises).then(
                function (result) {
                    // Some post processing
                    // ..
                    resolve(result)
                },
                function (error) {
                    // Some post processing
                    // including deleting anything that got created.
                    // Which will be other promises
                    reject(error);
                }
            );
        }
    });
};
```

Some Pseudo-Code

```
var     createEmployee = function(newEmployeeData) {
return new Promise(function (resolve, reject) {
```

- Retu
- Va
- Im
  cal
- Ca
  Pr
- Su
- Ma
  asy
  to r

- Implementing microservice code is less about traditional flow of control constructs, e.g.
  - Loops
  - Function calls
  - If {} then {}
- And more about
  - Returning a Promise
  - Composed of sub-Promises

Beyond Simple Promises (http://bluebirdjs.com/docs/api-reference.html)

API Reference

- Core
  - new Promise
  - .then
  - .spread
  - .catch
  - .error
  - .finally
  - .bind
  - Promise.join
  - Promise.try
  - Promise.method
  - Promise.resolve
  - Promise.reject
- Synchronous inspection
  - PromiseInspection
  - .isFulfilled
  - .isRejected
  - .isPending
  - .isCancelled
  - .value
  - .reason
  - Promise.using
  - .disposer
- Promisification
  - Promise.promisify
  - Promise.promisifyAll
  - Promise.fromCallback
  - .asCallback
- Timers
  - .delay
  - .timeout
- Cancellation
  - .cancel
- Generators
  - Promise.coroutine
  - Promise.coroutine.addYieldHan
- Utility
  - .tap
  - .call
  - .get
  - .return
  - .throw
  - .reason
- Collections
  - Promise.all
  - Promise.props
  - Promise.any
  - Promise.some
  - Promise.map
  - Promise.reduce
  - Promise.filter
  - Promise.each
  - Promise.mapSeries
  - Promise.race
  - .all
  - .props
  - .any
  - .some
  - .map
  - .reduce
  - .filter
  - .each
  - .mapSeries
- Resource management
  - .throw
  - .catchReturn
  - .catchThrow
  - .reflect
  - Promise.getNewLibraryCopy
  - Promise.noConflict
  - Promise.setScheduler
- Built-in error types
  - OperationalError
  - TimeoutError
  - CancellationError
  - AggregateError
- Configuration
  - Global rejection events
  - Local rejection events
  - Promise.config
  - .suppressUnhandledRejections
  - .done
- Progression migration
- Deferred migration
- Environment variables

There are a couple of problems with this section.

1. Promise.all() is parallel.
2. But, createAddress returns the URL of the created resource, which is needed to create the customer.

So, the code must use

1. Promise.each() and
2. Figure out how to pass data from the resolved 1st promise to the second function.

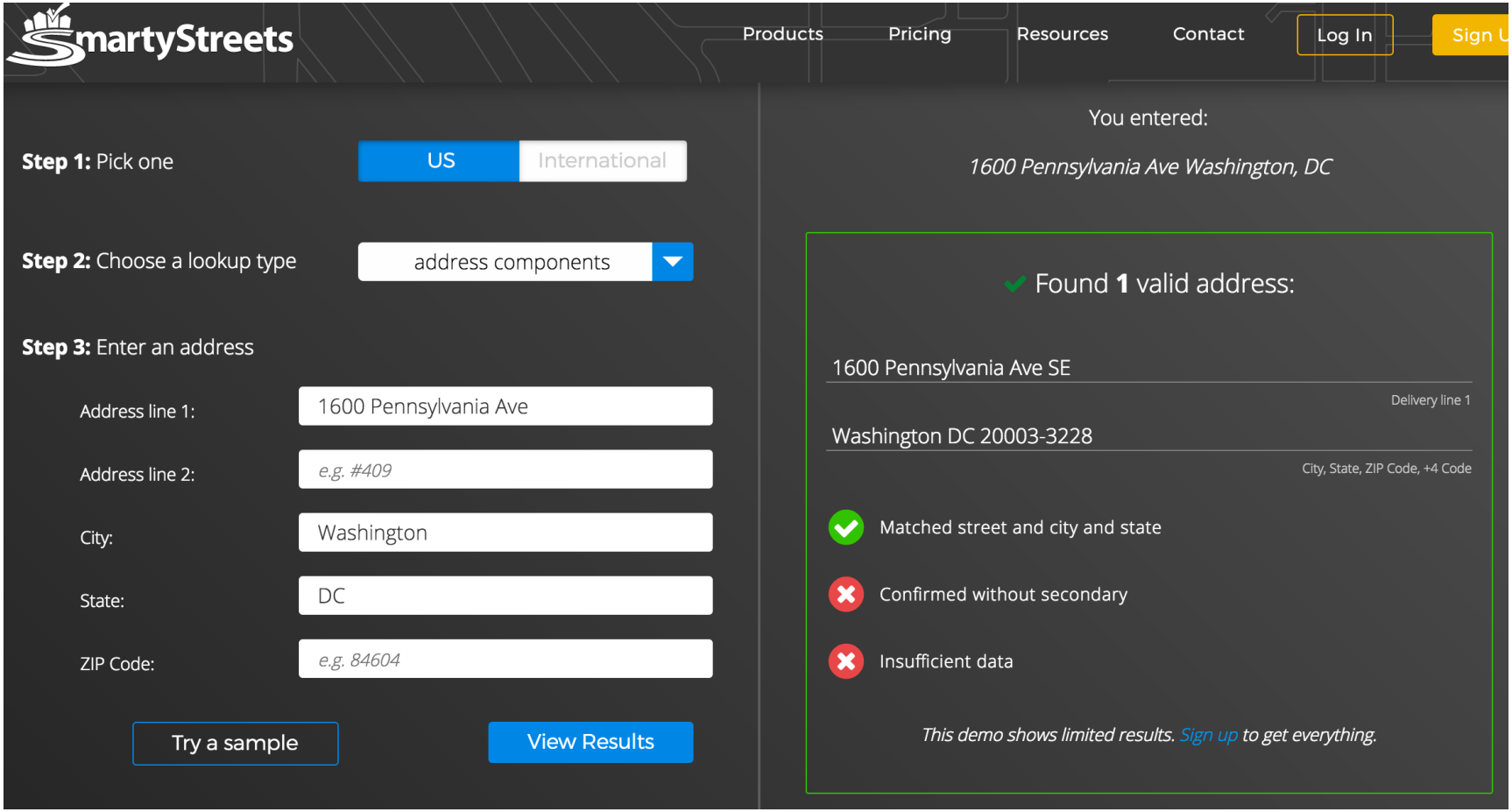
```
var     createEmployee = function(newEmployeeData) {

return new Promise(function (resolve, reject) {

    // Do some basic validation of the input, e.g.
    // All necessary fields present.
    var customerValid = validate();

    if (customerValid === false) {
        reject(meaningfulErrorData);
    }
    else {

        p1 = AddressModule.createAddress(newEmployeeData.addressInfo);
        p2 = PeopleModule.createPerson(newEmployeeData.personInfo);

        allPromises = [];
        allPromises.push(p1);
        allPromises.push(p2);

        Promise.all(allPromises).then(
            function (result) {
                // Some post processing
                // ..
                resolve(result)
            },
            function (error) {
                // Some post processing
                // including deleting anything that got created.
                // Which will be other promises
                reject(error);
            }
        );
    }
);
```

Better Pseudo-Code?

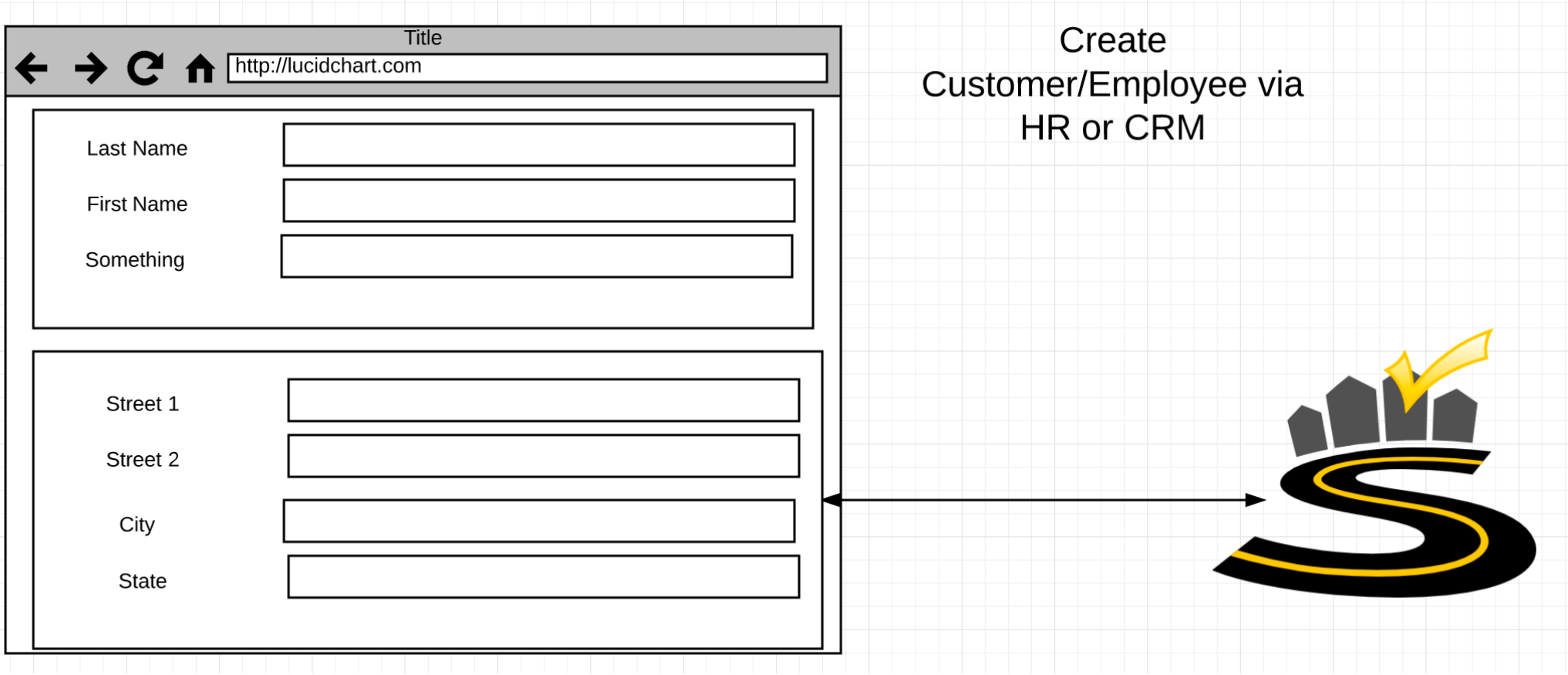
```
AddressModule.createAddress(newEmployeeData.addressInfo).then(
    function (result) {
        newEmployeeData.personInfo.addressUrl = result.url;
        PeopleModule.createPerson(newEmployeeData.personInfo).then(
            function (result) {
                resolve(result)
            },
            function (error) {
                reject(error);
            }
        );
    },
    function (error) {
        reject(error)
    });
```

- This code
  - Is relatively clear but
  - Would turn into nesting spaghetti for more than two related objects.
- This is an example of the *orchestrator pattern,* which can be done with
  - Spaghetti code
  - "Algebraically" in code with use of any(), map(), each(), series(), etc.
  - Or done with an orchestration services like workflow or step functions.

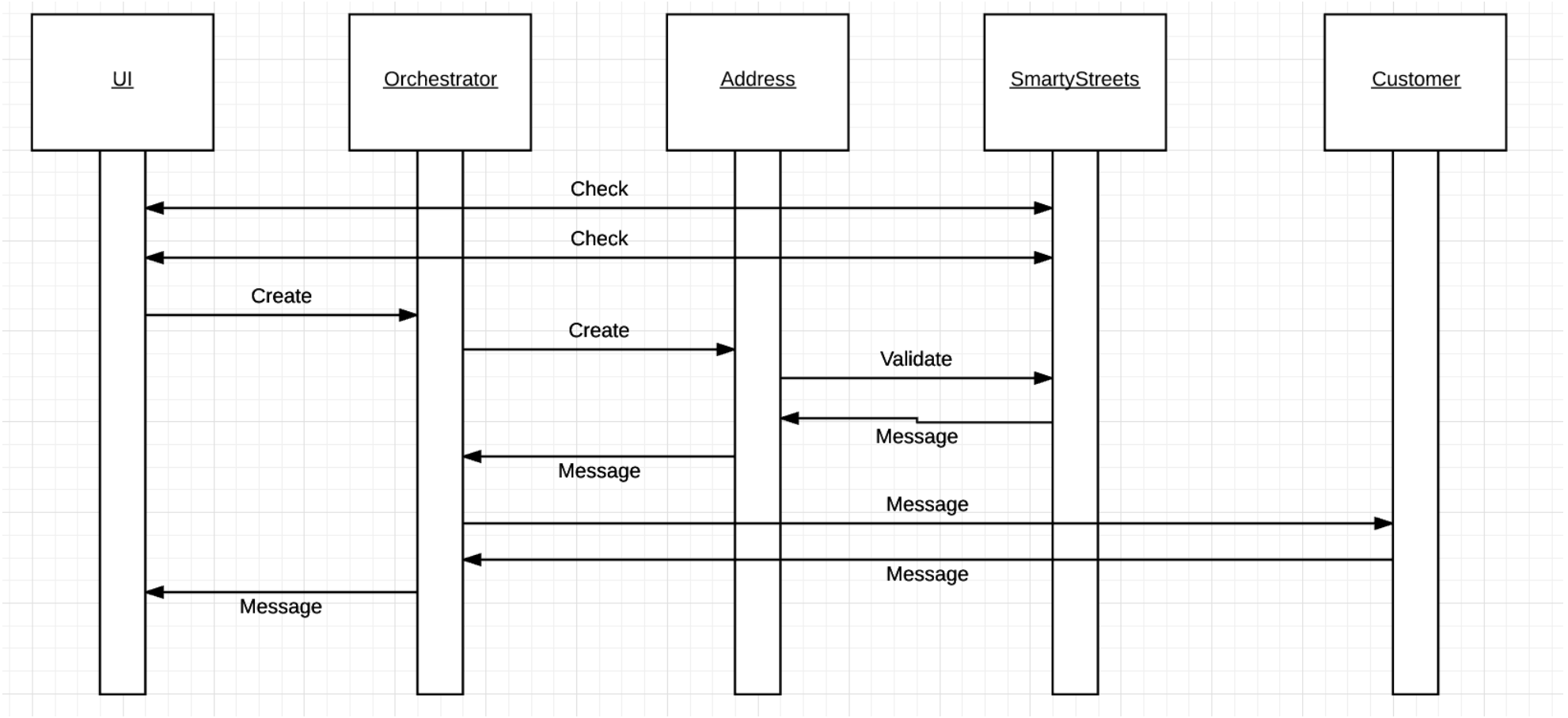
Smarty Streets (Demo)



UI

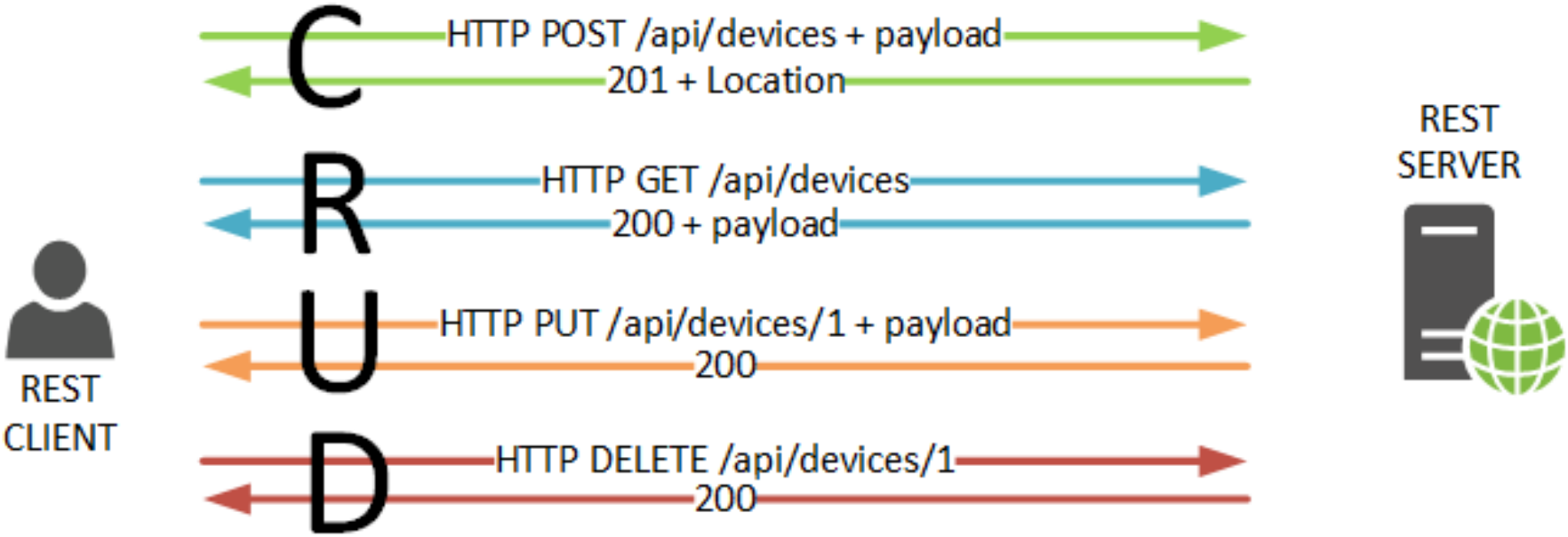


Successful Execution



Message Exchange Patterns

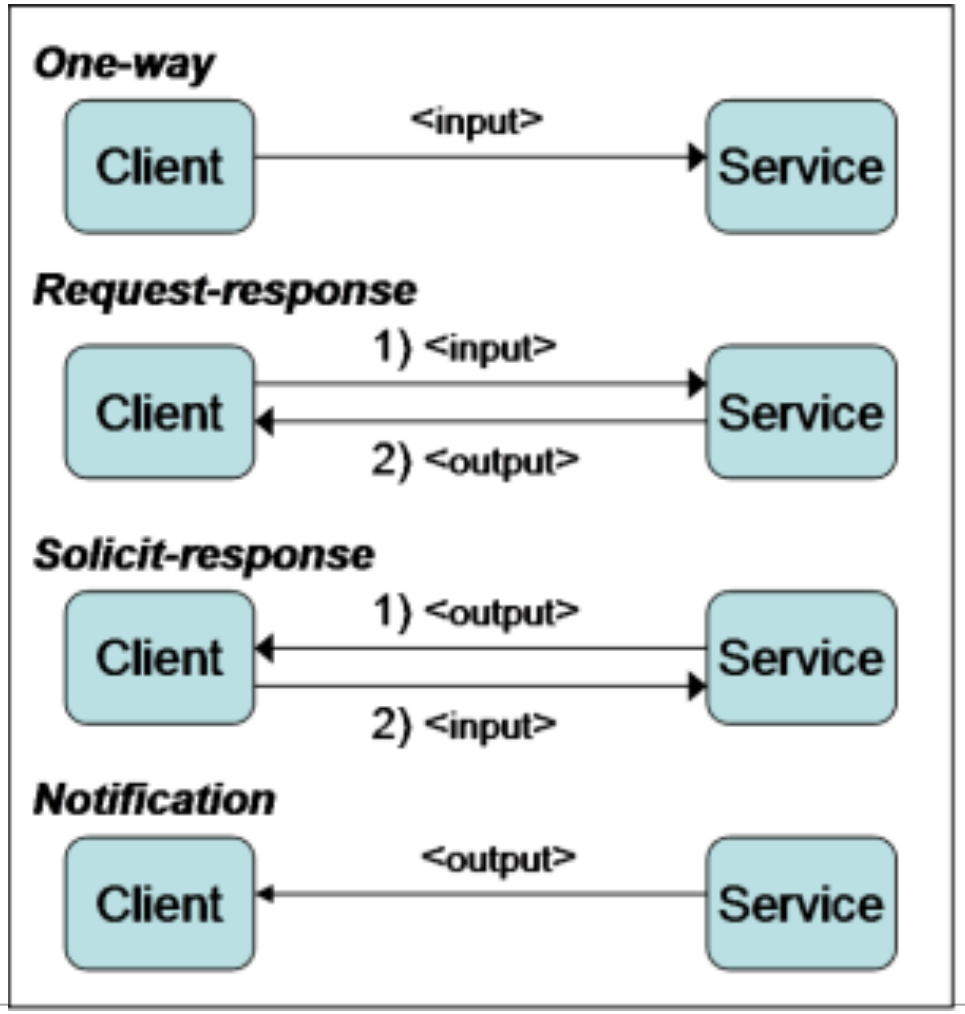
REST Messages and Verbs



- Simple examples just showing success
- There are, of course, many other return codes and patterns.

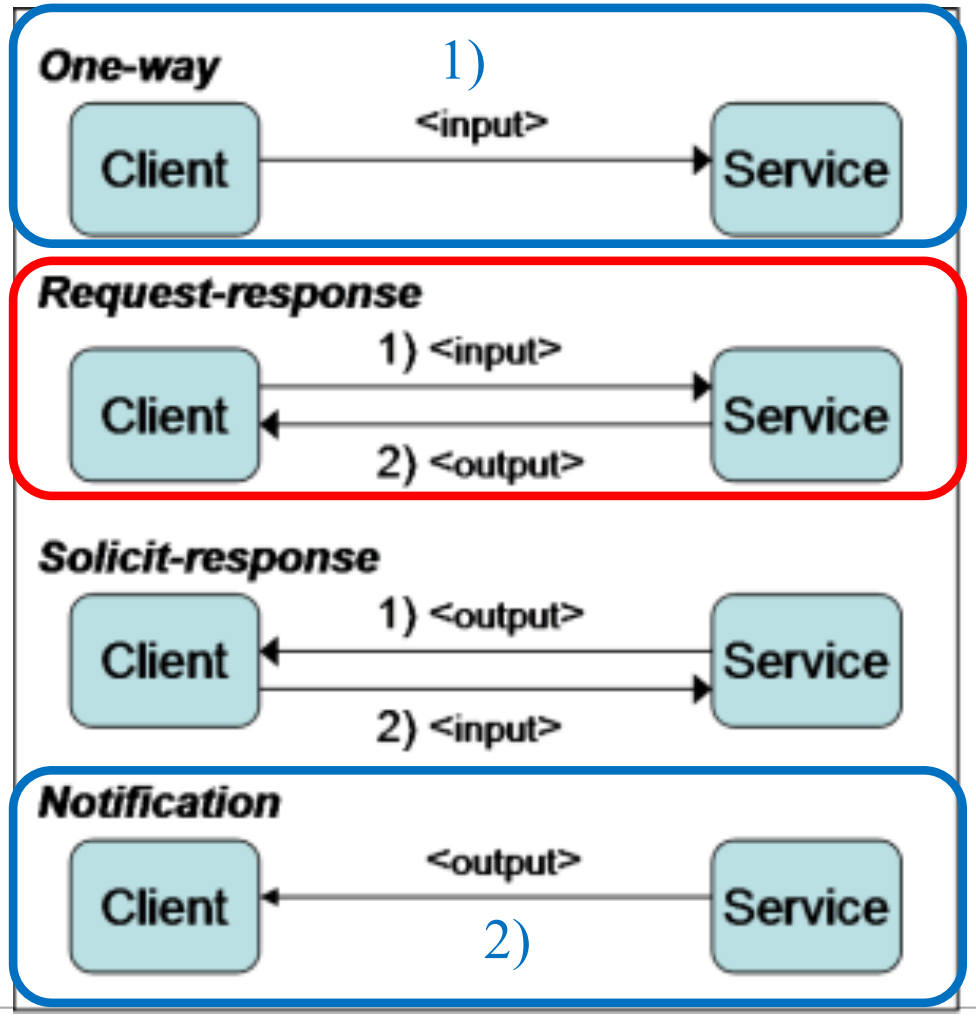
Message Patterns

- At the REST/HTTP level, each of the individual arrows may be
  - HTTP Verb.
  - HTTP Response.
- But from the application level, it is more complex.
  - PUT or POST response could mean
    - I created/updated the resource, or
    - I am working on creating/updating, and will notify you.
  - GET response could mean
    - Here is the data, or
    - In the future look here for the data.
  - A one-way input
    - Could be creating something (e.g. webhook)
    - That will generate many one-ways back.

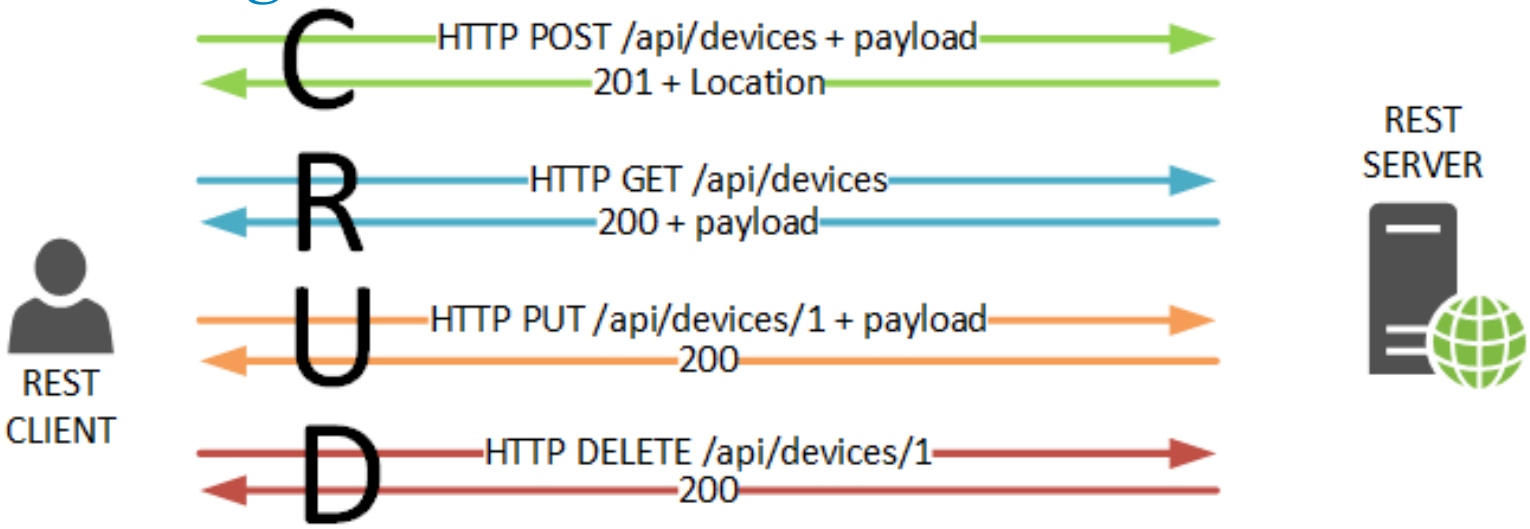


Synchronous/ Asynchronous

- "Synchronous"
  - Request is HTTP Verb
  - Response is application data in the HTTP response.
- "Asynchronous"
  - Request 1 is HTTP Verb.
    - Input is data and "callback URL."
    - HTTP Response is "Accepted."
  - Response is HTTP Verb
    - To callback URL with app rsp data.
    - HTTP response is "OK."

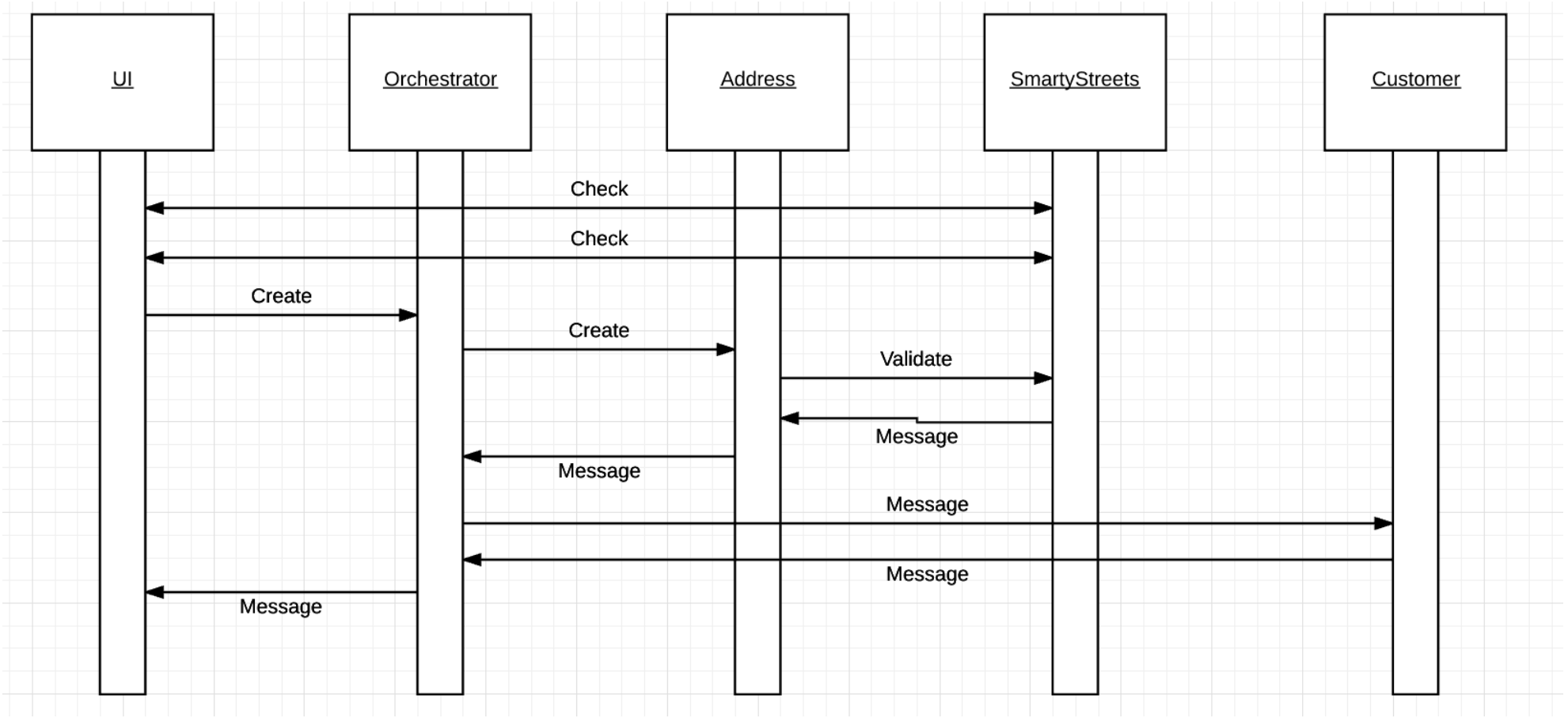


REST Messages and Verbs



- Calling an API via REST
  - Is logically similar to calling a local function in your code.
  - But,
    - The implementation is somewhere else.
    - Takes several orders of magnitude more time to complete.
- What does the calling runtime environment do? Wait? What about all the other things the runtime is doing?
  - Some runtimes, like Java model the calls as synchronous on separate threads.
  - Others (browsers, Node) are inherently, single threaded and the call inevitably goes asynchronous.

Which Patterns for Which Interactions?



Success Response Codes

Operation	HTTP Request	HTTP Response Codes Supported
READ	GET	200 - OK with message body 204 - OK no message body 206 - OK with partial message body
CREATE	POST	201 - Resource created (Operation Complete) 202 - Resource accepted (Operation Pending)
UPDATE	PUT	202 - Accepted (Operation Pending) 204 - Success (Operation Complete)
DELETE	DELETE	202 - Accepted (Operation Pending) 204 - Success (Operation Complete)

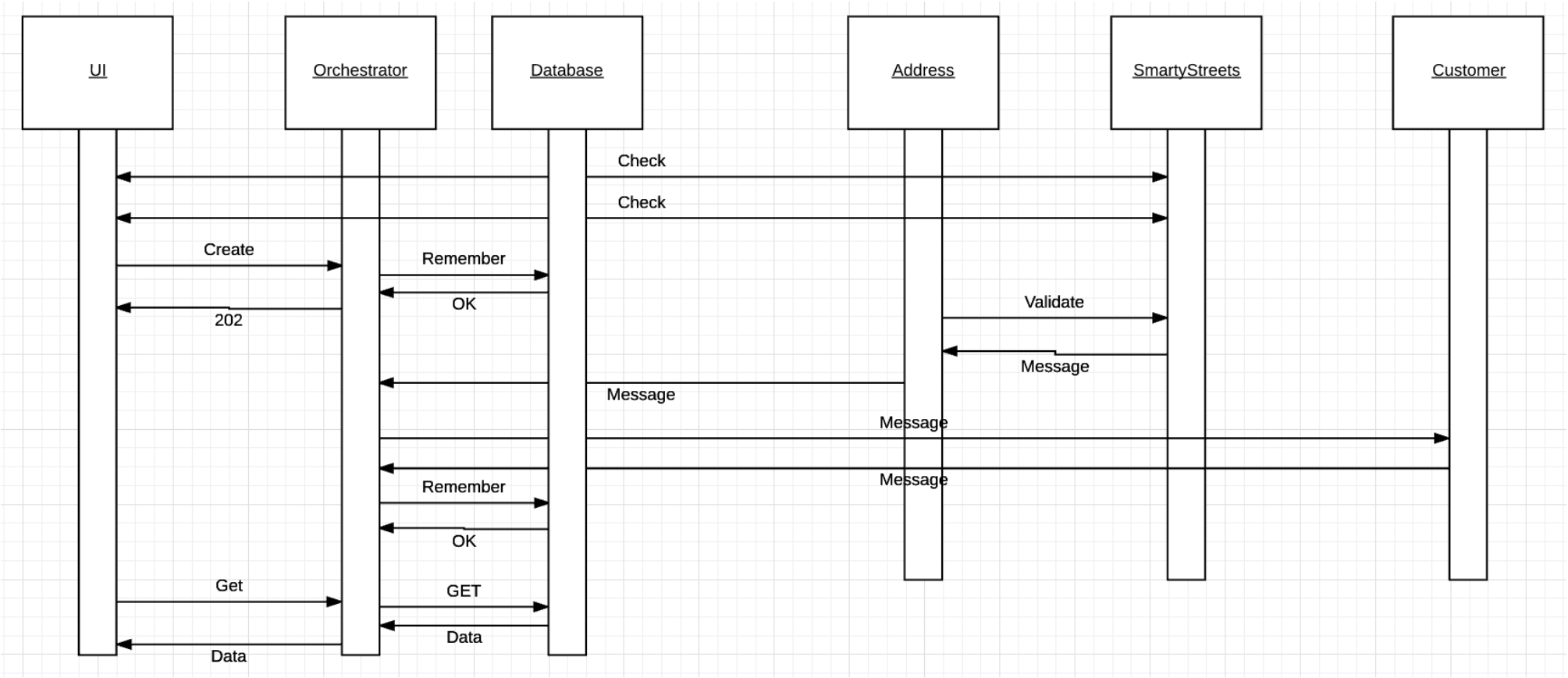
202 means

- Your request went asynch.
- The HTTP header Link is where to poll for rsp.
- We will cover later.

Examples of Link Headers in HTTP response:

```
Link: <http://api/jobs/j1>;rel=monitor;title="update profile"
Link: <http://api/reports/r1>;rel=summary;title="access report"
```

One Possibility



Enable User Interface

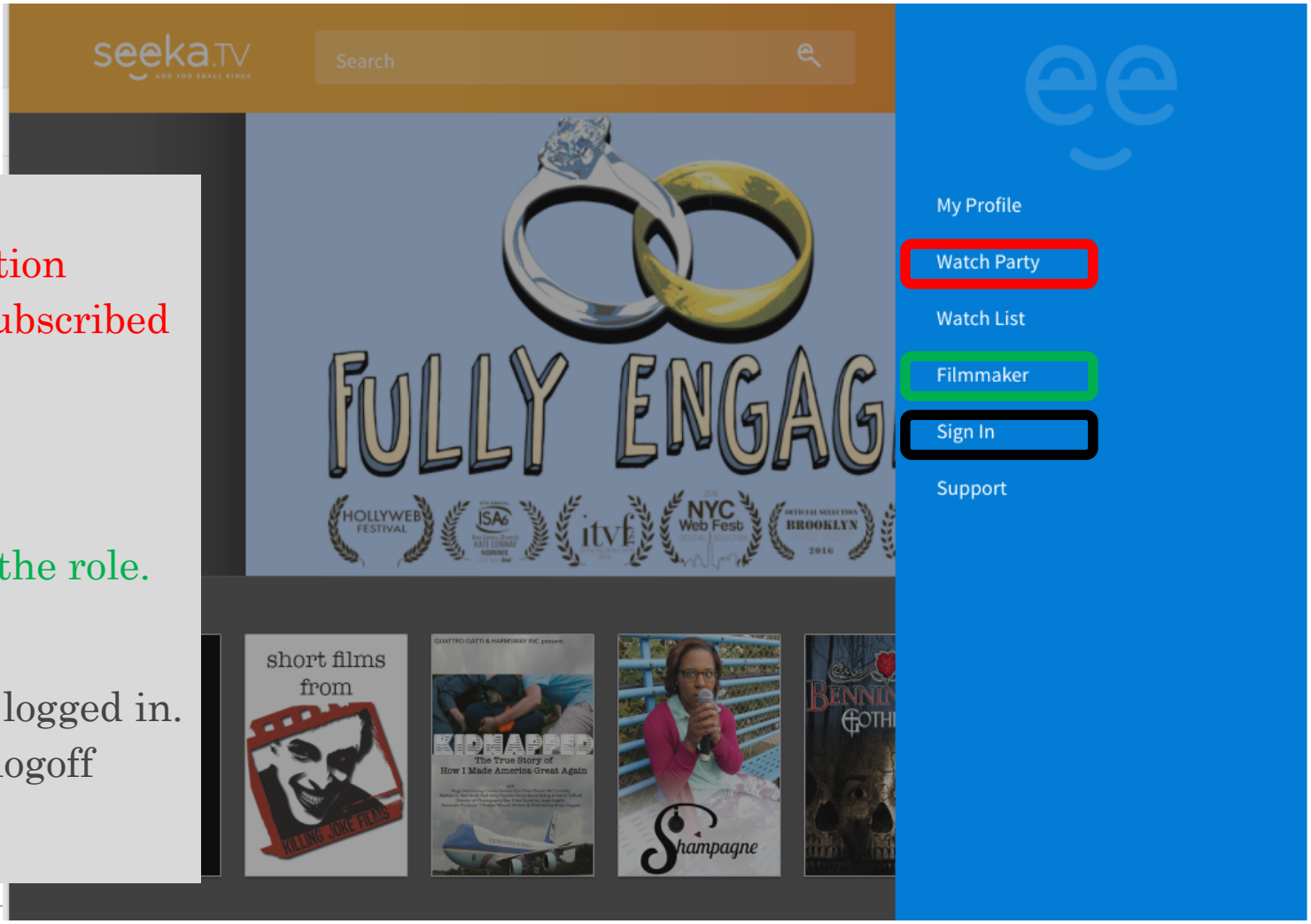
1st Project – Part 1

- Implement two distinct microservices
  - Person
  - Address
- Tasks
  - Use Swagger Editor to define and document REST APIs.
  - Implement an Elastic Beanstalk application (microservice) for each resource that implements 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

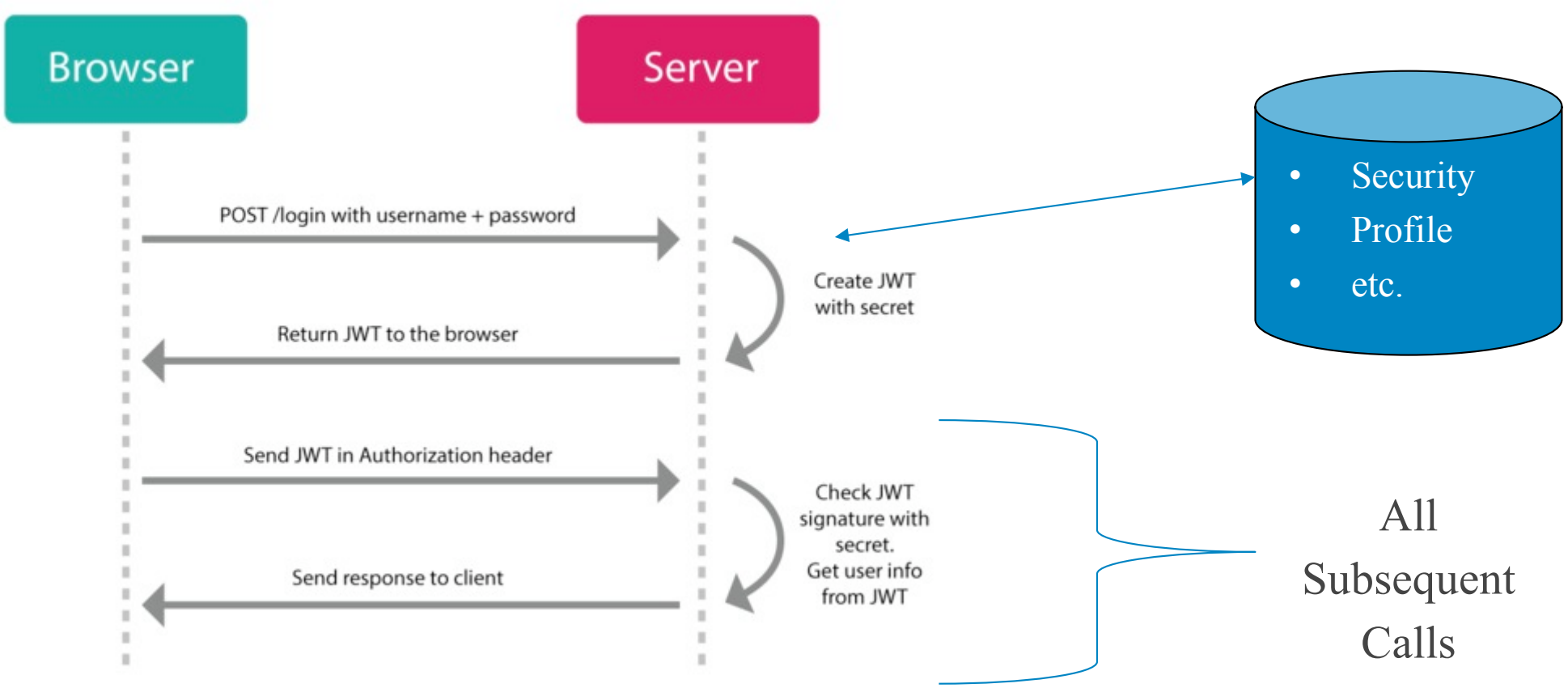
Session State

Session

- Watch Party
  - Only show the option
  - If customer has subscribed to the capability.
- Filmmaker
  - Only show option
  - If the person has the role.
- Signin
  - Only show if now logged in.
  - Otherwise, show logoff



Session State



JSON Web Token (https://jwt.io/introduction/)

What is JSON Web Token?

JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret (with the **HMAC** algorithm) or a public/private key pair using **RSA**.

Let's explain some concepts of this definition further.

- **Compact**: Because of their smaller size, JWTs can be sent through a URL, POST parameter, or inside an HTTP header. Additionally, the smaller size means transmission is fast.
- **Self-contained**: The payload contains all the required information about the user, avoiding the need to query the database more than once.

When should you use JSON Web Tokens?

Here are some scenarios where JSON Web Tokens are useful:

- **Authentication**: This is the most common scenario for using JWT. Once the user is logged in, each subsequent request will include the JWT, allowing the user to access routes, services, and resources that are permitted with that token. Single Sign On is a feature that widely uses JWT nowadays, because of its small overhead and its ability to be easily used across different domains.
- **Information Exchange**: JSON Web Tokens are a good way of securely transmitting information between parties, because as they can be signed, for example using public/private key pairs, you can be sure that the senders are who they say they are. Additionally, as the signature is calculated using the header and the payload, you can also verify that the content hasn't been tampered with.

JWT/Session State

Two core use cases:

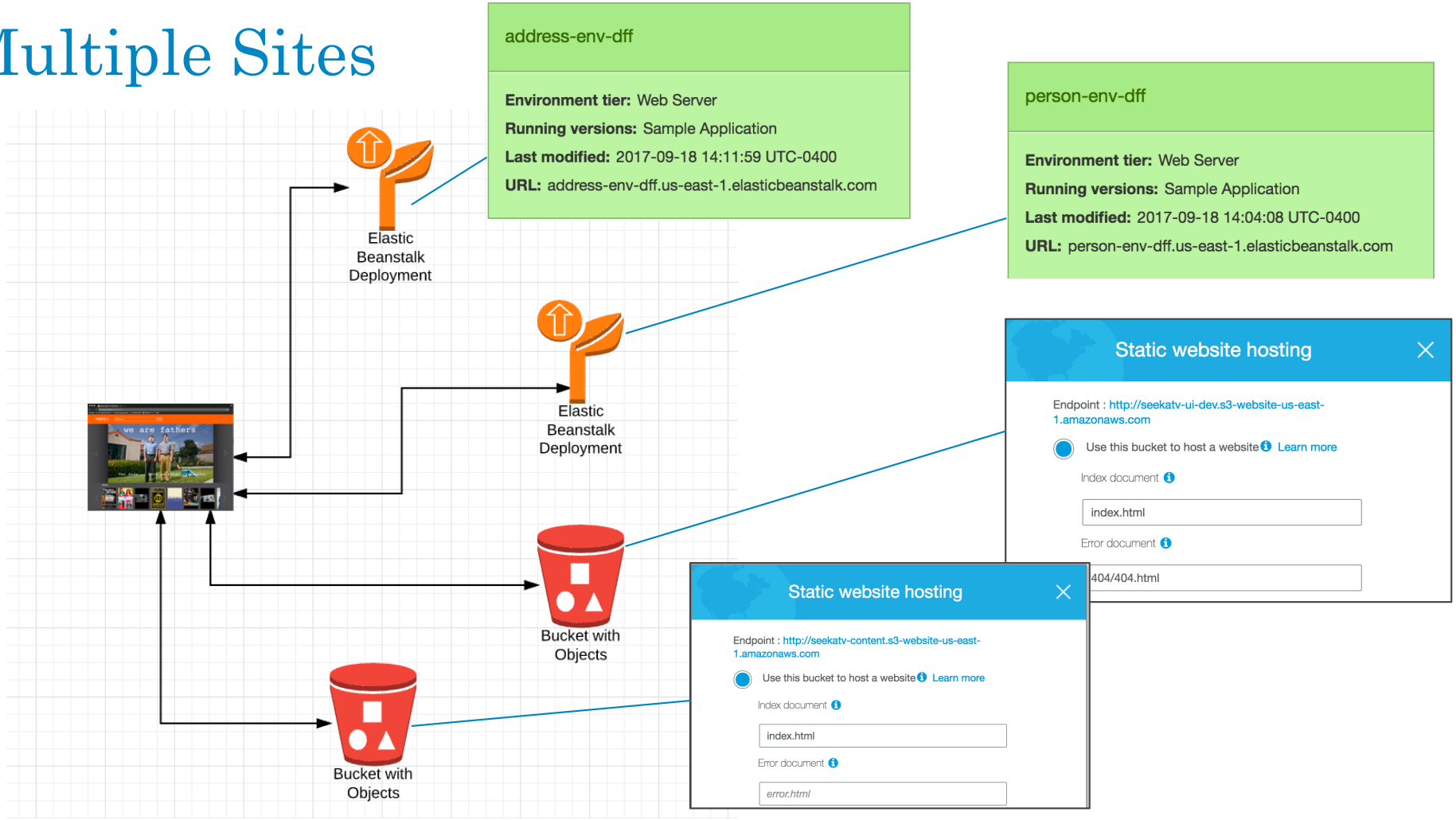
1. Correlation (security is a special case)
    1. A gives B something on a request.
    2. That B returns at a potentially much later date.
    3. So that A can correlate the response with the original request processing.
2. Propagation
    1. A send something to B.
    2. That B must pass onto C (e.g. A asked has authorized me to do something on its behalf)
    3. C must be able to verify that A originated the request.

The concepts are:

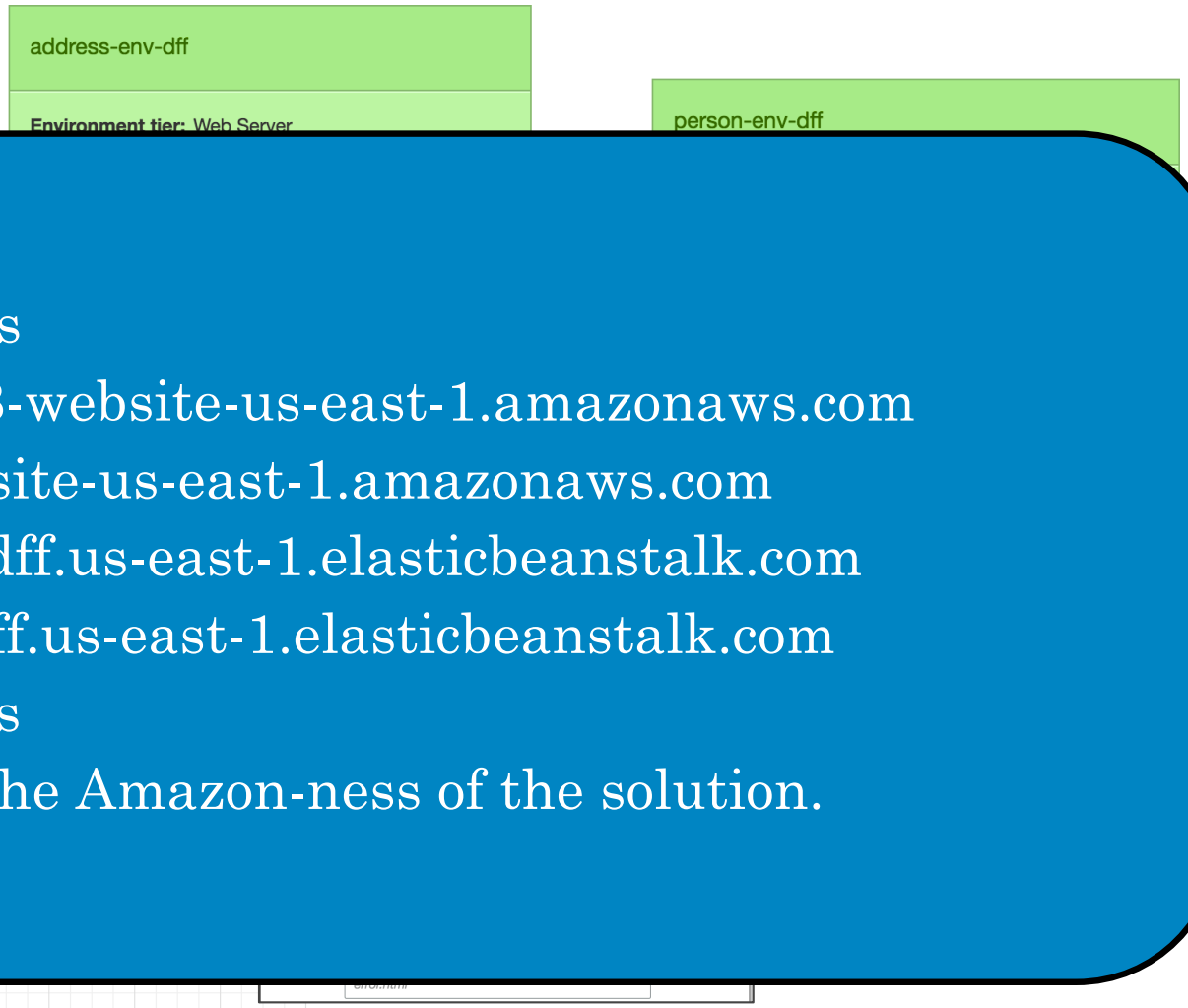
- Fundamental to complex, cloud spanning as-a-Service applications.
- Typically not something you learn in other CS courses.

Single Site Image

Multiple Sites



Multiple Sites

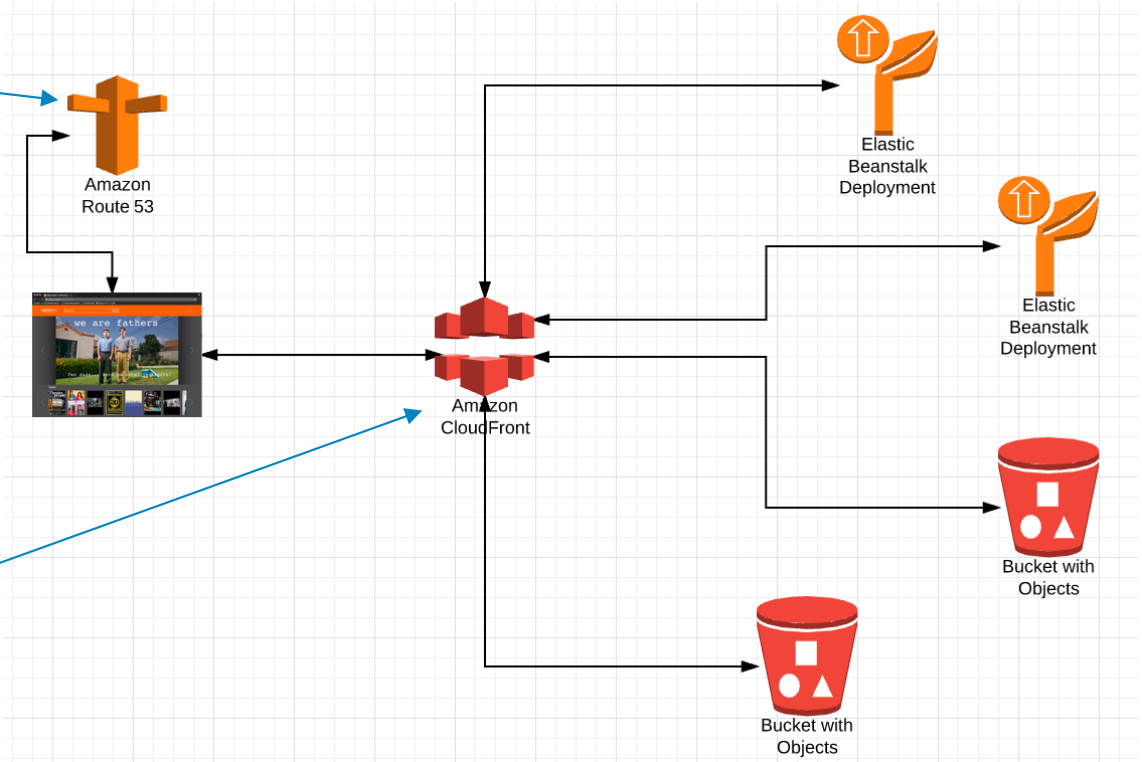


- Four IP addresses
  - dff-content.s3-website-us-east-1.amazonaws.com
  - dff-ui.s3-website-us-east-1.amazonaws.com
  - address-env-dff.us-east-1.elasticbeanstalk.com
  - person-env-dff.us-east-1.elasticbeanstalk.com
- Over two domains
- And all exposed the Amazon-ness of the solution.

Single Site

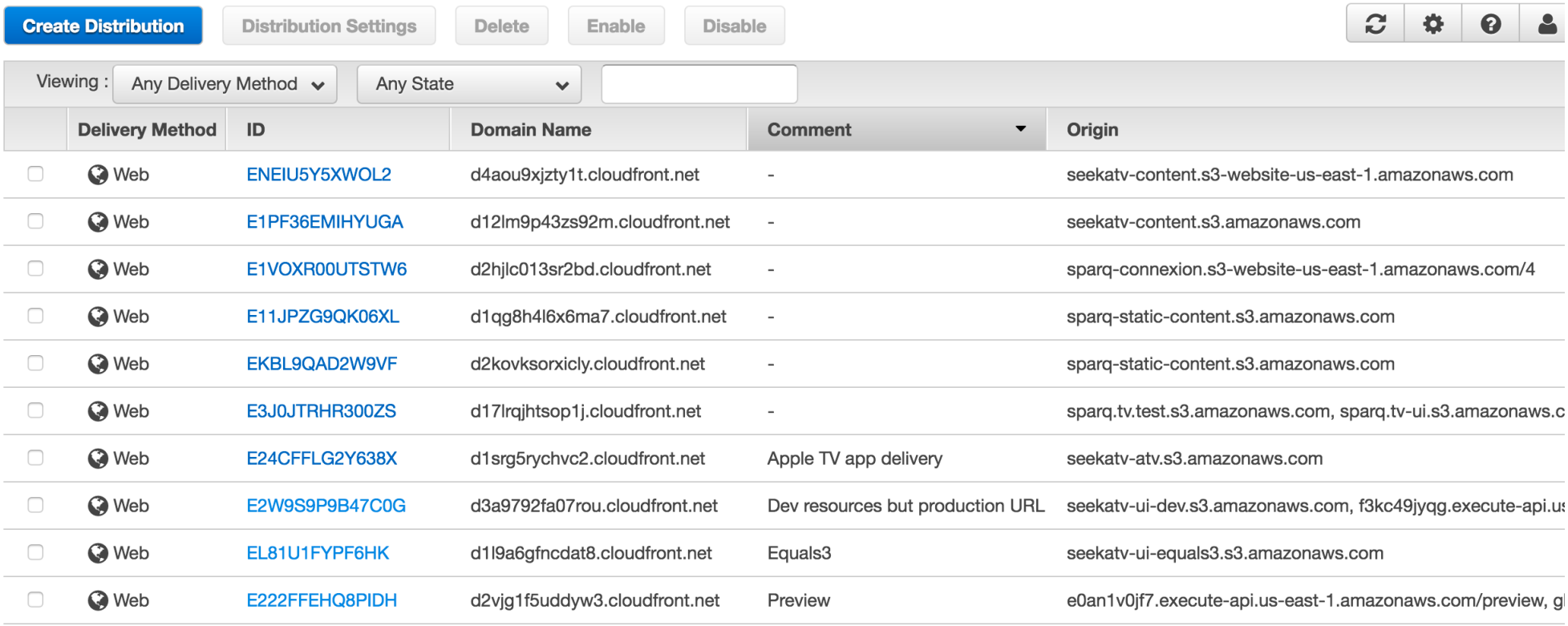
- DNS
  - Resolve dff-company.com to
  - Something.amazon.com
  - Under the covers
- Map
  - /api/person
  - /api/address
  - /app
    - /js
    - /views
    - /app-content
  - /digital-assets
    - /images
    - /videos

To correct IP addresses and sub-paths



CloudFront Distributions

CloudFront Distributions



Create Distribution | Distribution Settings | Delete | Enable | Disable

Viewing : Any Delivery Method | Any State

Delivery Method	ID	Domain Name	Comment	Origin
Web	ENEIU5Y5XWOL2	d4aou9xjzty1t.cloudfront.net	-	seekatv-content.s3-website-us-east-1.amazonaws.com
Web	E1PF36EMIHYUGA	d12lm9p43zs92m.cloudfront.net	-	seekatv-content.s3.amazonaws.com
Web	E1VOXR00UTSTW6	d2hjlc013sr2bd.cloudfront.net	-	sparq-connexion.s3-website-us-east-1.amazonaws.com/4
Web	E11JPZG9QK06XL	d1qg8h4l6x6ma7.cloudfront.net	-	sparq-static-content.s3.amazonaws.com
Web	EKBL9QAD2W9VF	d2kovksorxicly.cloudfront.net	-	sparq-static-content.s3.amazonaws.com
Web	E3J0JTRHR300ZS	d17lrqjhtsop1j.cloudfront.net	-	sparq.tv.test.s3.amazonaws.com, sparq.tv-ui.s3.amazonaws.c
Web	E24CFFLG2Y638X	d1srg5rychvc2.cloudfront.net	Apple TV app delivery	seekatv-atv.s3.amazonaws.com
Web	E2W9S9P9B47C0G	d3a9792fa07rou.cloudfront.net	Dev resources but production URL	seekatv-ui-dev.s3.amazonaws.com, f3kc49jyqg.execute-api.u
Web	EL81U1FYPF6HK	d1l9a6gfncdat8.cloudfront.net	Equals3	seekatv-ui-equals3.s3.amazonaws.com
Web	E222FFEHQ8PIDH	d2vjg1f5uddyw3.cloudfront.net	Preview	e0an1v0jf7.execute-api.us-east-1.amazonaws.com/preview, g

Distributions and Origins

CloudFront Distributions > E2W9S9P9B47C0G

General | **Origins** | Behaviors | Error Pages | Restrictions | Invalidations | Tags

Create Origin | Edit | Delete

Origin Domain Name and Path	Origin ID	Origin Type	Origin Access Identity	Origin Protocol Policy
[redacted]mazonaws.com	seekatv-ui-dev	S3 Origin	-	-
[redacted]-api.us-east-1.amazonaws.com/dev	SeekaTVServicesAPIDev	Custom Origin	-	HTTPS Only
[redacted]te-api.us-east-1.amazonaws.com/dev	SeekaTVSocialNetworksAuthAPIDev	Custom Origin	-	HTTPS Only
[redacted]-api.us-east-1.amazonaws.com/dev	SeekaTVAssetManagerAPIDev	Custom Origin	-	HTTPS Only
[redacted]-api.us-east-1.amazonaws.com/dev	SeekaTVReportAPIDev	Custom Origin	-	HTTPS Only
[redacted]te-api.us-east-1.amazonaws.com/dev	SeekaTVSearchAPIDev	Custom Origin	-	HTTPS Only
[redacted]e-api.us-east-1.amazonaws.com/dev	SeekaTVCatalogInfoAPIDev	Custom Origin	-	HTTPS Only
[redacted]098.us-east-1.elb.amazonaws.com	ELB-seekatv-lb-801820098	Custom Origin	-	Match Viewer
[redacted]te-api.us-east-1.amazonaws.com/dev	SeekaTVWatchPartyAPIDev	Custom Origin	-	HTTPS Only
[redacted]e-api.us-east-1.amazonaws.com/dev	SeekaTVCustomerInfoAPIDev	Custom Origin	-	HTTPS Only
[redacted]-api.us-east-1.amazonaws.com/dev	SeekaTVBasicAPIDev	Custom Origin	-	HTTPS Only

Behaviors

CloudFront Distributions > E2W9S9P9B47C0G

General | Origins | **Behaviors** | Error Pages | Restrictions | Invalidations | Tags

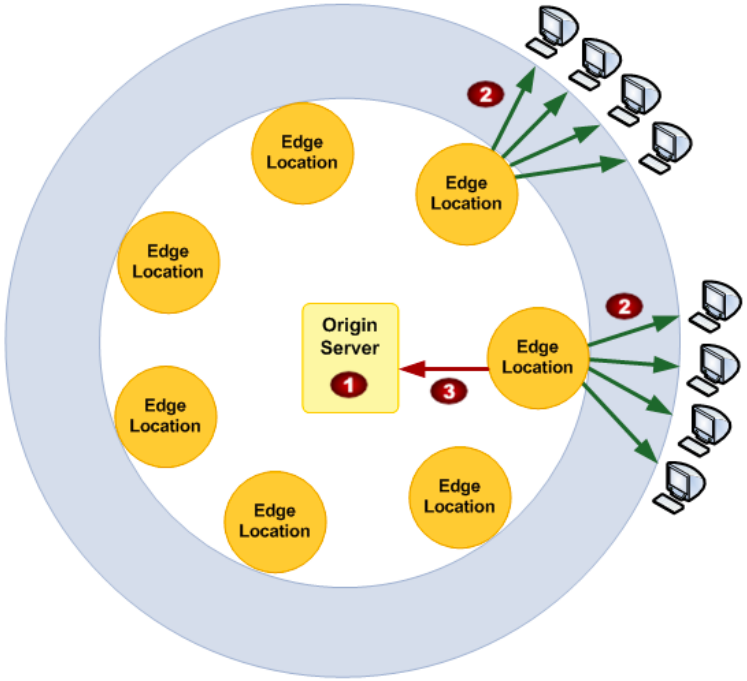
CloudFront compares a request for an object with the path patterns in your cache behaviors based on the order of the cache behaviors in your distribution. Arrange cache behaviors in the order in which you want CloudFront to evaluate them.

Create Behavior | Edit | Delete | Change Precedence: | Move Up | Move Down | Save

Viewing 1 to 16 o

Precedence	Path Pattern	Origin	Viewer Protocol Policy	Forwarded Query Strings	Trusted
0	/dev/*	Custom-llixwpq5t1.execute-api.us-east-1.amazonaws	HTTPS Only	Yes	-
1	/assetmanager/*	SeekaTVAssetManagerAPIDev	HTTPS Only	Yes	-
2	/payment/*	SeekaTVPaymentAPIDev	HTTPS Only	Yes	-
3	/reports/*	SeekaTVReportAPIDev	HTTPS Only	Yes	-
4	/search/*	SeekaTVSearchAPIDev	HTTPS Only	Yes	-
5	/customerinfo/*	SeekaTVCustomerInfoAPIDev	HTTPS Only	Yes	-
6	/socialnetworksauth/*	SeekaTVSocialNetworksAuthAPIDev	HTTPS Only	Yes	-
7	/cataloginfo/*	SeekaTVCatalogInfoAPIDev	HTTPS Only	Yes	-
8	/services/*	SeekaTVServicesAPIDev	HTTPS Only	Yes	-
9	/watch-party/*	SeekaTVWatchPartyAPIDev	HTTPS Only	Yes	-

Global Caching and Optimization



Amazon CloudFront edge locations

AWS provides full-site, or media asset, delivery via a worldwide content delivery network (CDN) called Amazon **CloudFront**.

amazon web services | Webinars

Observations

- This section has been very AWS but
  - The concepts apply to any non-trivial microservice/cloud application.
  - All enterprise deployments, large scale SaaS solutions and cloud platforms have similar concepts.
- For the purposes of the next step in the project(s)
  - Do not worry about DNS resolution; I will setup/mockup for you.
  - Setup and configure CloudFront, which will be critical for
    - Simplified single site image.
    - HTTPS
  - We will need to consider caching configuration for responses.

Multitenancy Introduction

Definition

http://www.gartner.com/it-glossary/multitenancy

Gartner IT Glossary > Multitenancy

Multitenancy

Multitenancy is a reference to the mode of operation of software where multiple independent instances of one or multiple applications operate in a shared environment. The instances (tenants) are logically isolated, but physically integrated. The degree of logical isolation must be complete, but the degree of physical integration will vary. The more physical integration, the harder it is to preserve the logical isolation. The tenants (application instances) can be representations of organizations that obtained access to the multitenant application (this is the scenario of an ISV offering services of an application to multiple customer organizations). The tenants may also be multiple applications competing for shared underlying resources (this is the scenario of a private or public cloud where multiple applications are offered in a common cloud environment).

Multitenancy

From Wikipedia, the free encyclopedia

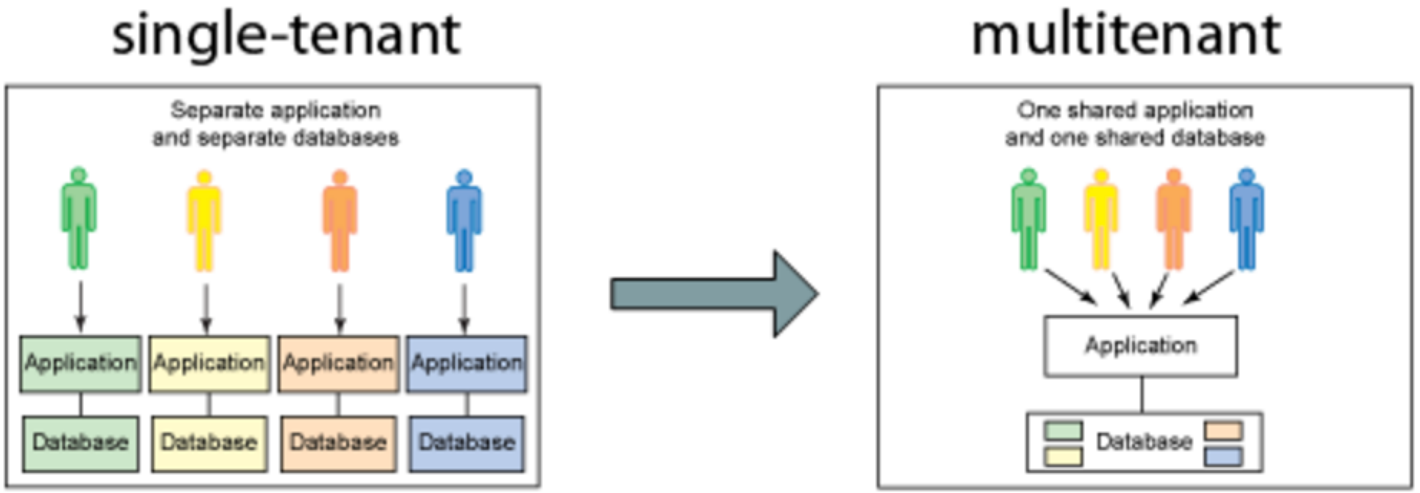
The term "**software multitenancy**" refers to a software architecture in which a single instance of software runs on a server and serves multiple tenants. A tenant is a group of users who share a common access with specific privileges to the software instance. With a multitenant architecture, a software application is designed to provide every tenant a dedicated share of the instance - including its data, configuration, user management, tenant individual functionality and non-functional properties. Multitenancy contrasts with multi-instance architectures, where separate software instances operate on behalf of different tenants. [1]

Some commentators regard multitenancy as an important feature of cloud computing.[2][3]

https://en.wikipedia.org/wiki/Multitenancy

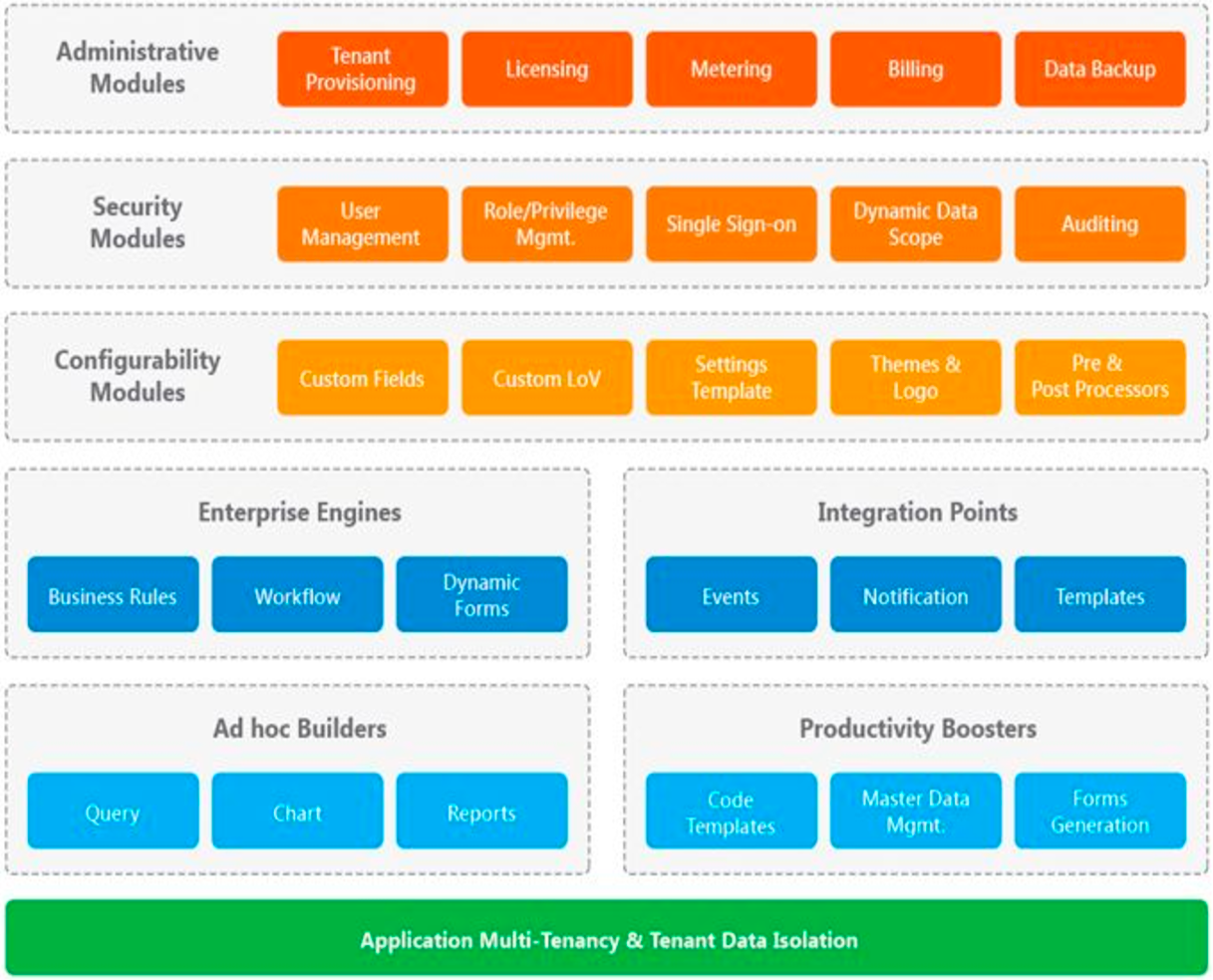
Multitenancy

http://jesusgilhernandez.com/2012/12/13/multitenancy-architecture/



- Core concepts
  - Single codebase and “database” supports multiple customers.
  - Each customer (tenant) ”thinks” that have their own, dedicated instance, including customized UI, application logic and data models.
- Provides significant cost and agility benefits to application provider, which also results in cost and agility benefits for customer/tenant.

Function Areas



https://www.techcello.com/product/architecture