

E6156 – Topics in SW Engineering (F23)

Cloud Computing

Lecture 2a: Microservices, REST, BLOBs



E6156 – Topics in SW Engineering (F23)

Cloud Computing

Lecture 2a: Microservices, REST, BLOBs

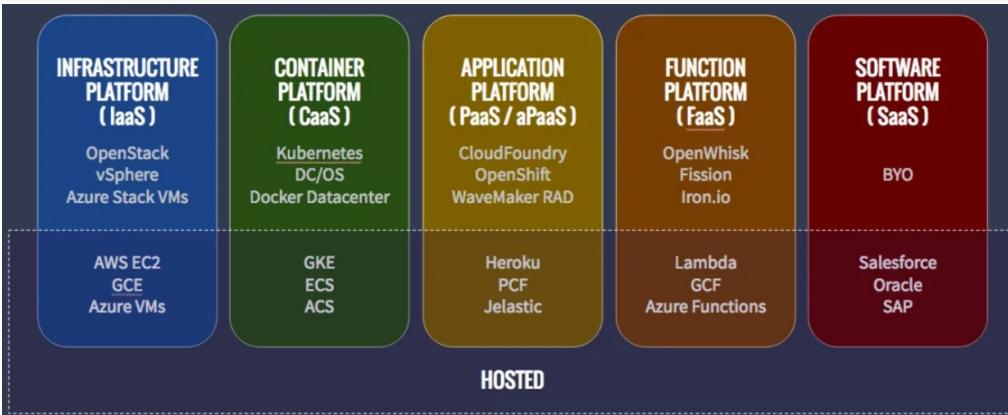
We will start in a minute or two.



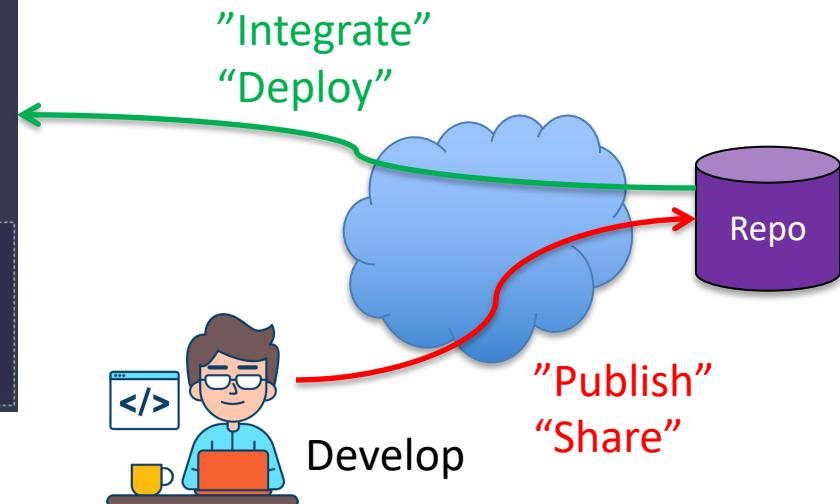
Contents

Some Concepts

Cloud Computing “Layers:” One Perspective



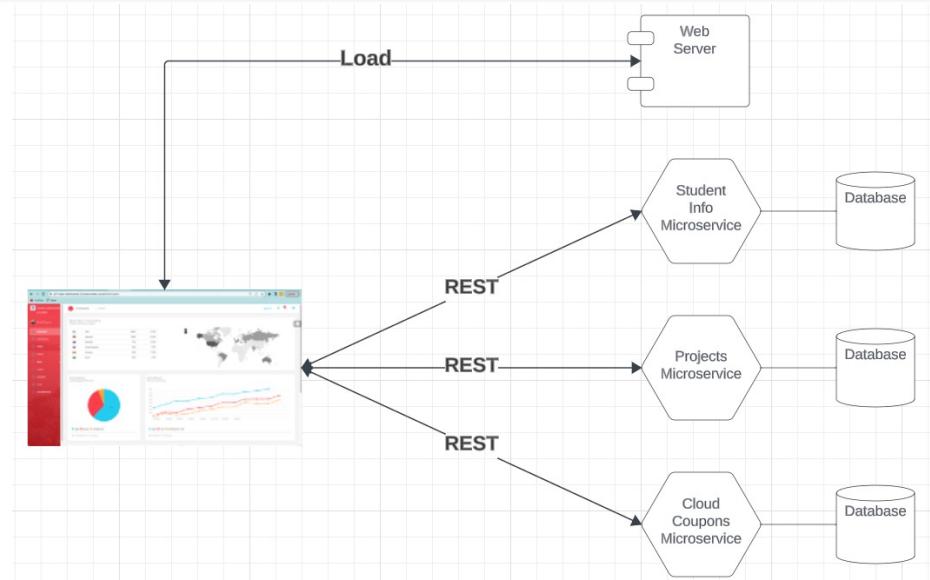
<https://medium.com/@nnilesh7756/what-are-cloud-computing-services-iaas-caas-paas-faas-saas-ac0f6022d36e>



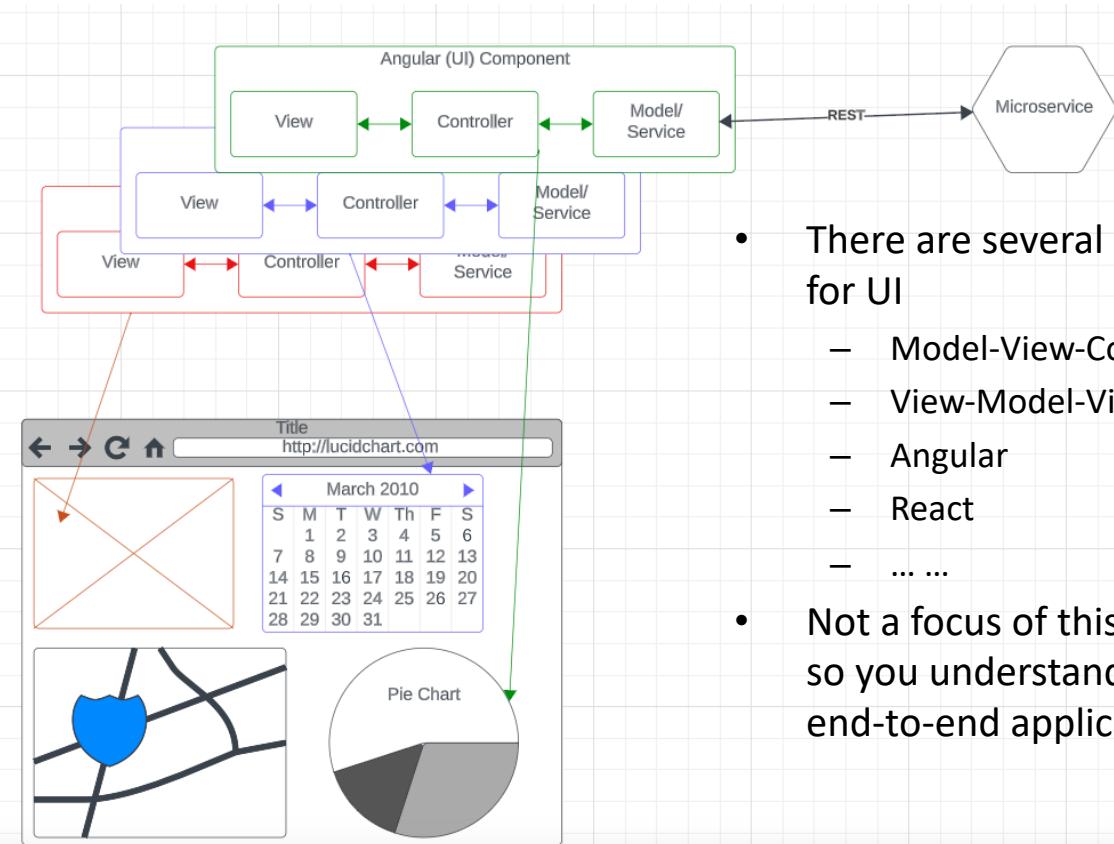
- We are seeing a recurring pattern: develop/test, publish/share, deploy.
- We have seen two examples:
 - Develop → GitHub → Deploy
 - Develop → Container Registry → Deploy
- We have been doing some things manually, e.g. provision, configure, deploy.
- We will see new models (e.g. - PaaS, DaaS, FaaS) and automation (e.g. CI/CD, infrastructure as code, GitHub Actions, ...)

Next Sprint for Professor – You Will Mirror

- Sprint 1:
 - Set up master project and sprint board.
 - Basic structure and process for 4 sub-projects.
 - Web UI Portal.
 - Student profile and information.
 - Student team, project information.
 - Cloud coupon, accounts, etc. info.
 - End-to-end flow for “Hello world” in each element.
 - 4 deployment models:
 - S3 for web UI portal.
 - PaaS (Elastic Beanstalk) for student.
 - Container for cloud coupons.
 - IaaS for student info.
- Sprint 2:
 - Add single sign on via OAuth2.
 - Add data and databases.
 - Single, consolidate API and API gateway.
 - First deployment automation.



Web UI



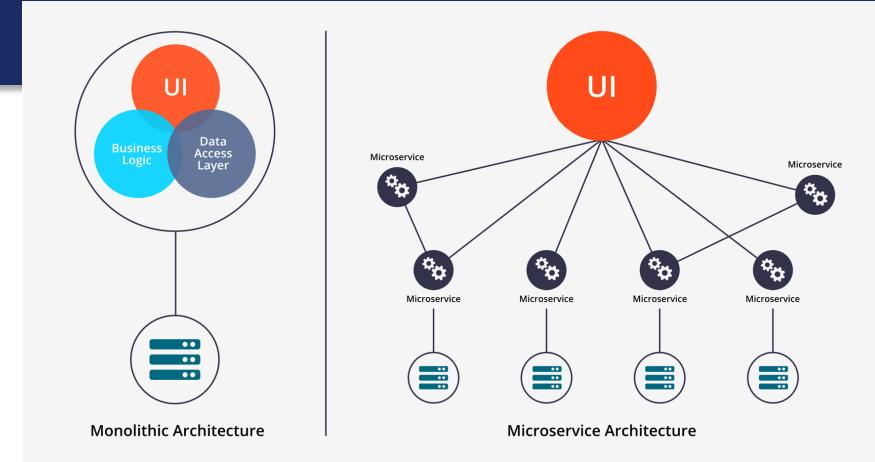
- There are several models and frameworks for UI
 - Model-View-Controller
 - View-Model-View Model
 - Angular
 - React
 -
- Not a focus of this course but will show simple examples so you understand this aspect of “full stack” and end-to-end applications.

Microservices Continued

Microservices

Historically, complex applications were implemented and deployed as “one big application.”

- **Monolith Architecture** is built in one large system and usually one code-base. A monolith is often deployed all at once, both front-end and back-end code together, regardless of what was changed.
- **Microservices Architecture** is built as a suite of small services, each with their own code-base. These services are built around specific capabilities and are usually independently deployable.



Components of Microservices architecture:

- The services are independent, small, and loosely coupled
- Encapsulates a business or customer scenario
- Every service is a different codebase
- Services can be independently deployed
- Services interact with each other using APIs

<https://medium.com/hengky-sanjaya-blog/monolith-vs-microservices-b3953650dfd>

Microservices

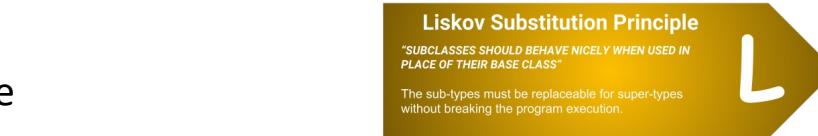
	Monolithic	Microservice
Architecture	Built as a single logical executable (typically the server-side part of a three tier client-server-database architecture)	Built as a suite of small services, each running separately and communicating with lightweight mechanisms
Modularity	Based on language features	Based on business capabilities
Agility	Changes to the system involve building and deploying a new version of the entire application	Changes can be applied to each service independently
Scaling	Entire application scaled horizontally behind a load-balancer	Each service scaled independently when needed
Implementation	Typically written in one language	Each service implemented in the language that best fits the need
Maintainability	Large code base intimidating to new developers	Smaller code base easier to manage
Transaction	ACID	BASE



- You can drive yourself crazy trying to:
 - Compare microservice architectures to other architecture patterns.
 - Define the characteristics of microservices. What makes something a microservice?
- Why do I cover microservices?
 - It does have benefits in large scale SW development projects.
 - It looks “cool” on your resume.

SOLID Principles

- “In software engineering, SOLID is a mnemonic acronym for five design principles intended to make software designs more understandable, flexible, and maintainable.” (<https://en.wikipedia.org/wiki/SOLID>)
- The concept started with OO but applies to microservices.
- We will focus on the “S” for now.
- For my project purposes, a thing can be:
 - A user profile service
 - Or
 - A commerce catalog service
 - Or
 - ...But not both.



12 Factor Applications

<https://dzone.com/articles/12-factor-app-principles-and-cloud-native-microser>



12 Factor App Principles

Codebase One codebase tracked in revision control, many deploys	Port Binding Export services via port binding
Dependencies Explicitly declare and isolate the dependencies	Concurrency Scale-out via the process model
Config Store configurations in an environment	Disposability Maximize the robustness with fast startup and graceful shutdown
Backing Services Treat backing resources as attached resources	Dev/prod parity Keep development, staging, and production as similar as possible
Build, release, and, Run Strictly separate build and run stages	Logs Treat logs as event streams
Processes Execute the app as one or more stateless processes	Admin processes Run admin/management tasks as one-off processes

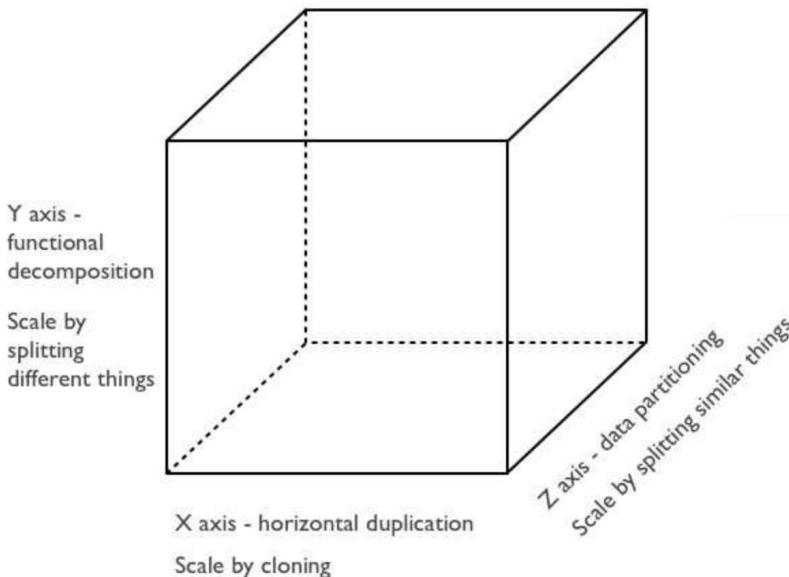
Microservices Scalability Cube

The Scale Cube

<https://microservices.io/articles/scalcube.html>

The book, [The Art of Scalability](#), describes a really useful, three dimension scalability model: the [scale cube](#).

3 dimensions to scaling



- X-axis scaling consists of running multiple copies of an application behind a load balancer. If there are N copies then each copy handles 1/N of the load. This is a simple, commonly used approach of scaling an application.
- Y-axis axis scaling splits the application into multiple, different services. Each service is responsible for one or more closely related functions.
- When using Z-axis scaling each server runs an identical copy of the code. In this respect, it's similar to X-axis scaling. The big difference is that each server is responsible for only a subset of the data. Some component of the system is responsible for routing each request to the appropriate server

XYZ – Scaling

X-AXIS SCALING

Network name: Horizontal scaling, scale out



Y-AXIS SCALING

Network name: Layer 7 Load Balancing, Content switching, HTTP Message Steering



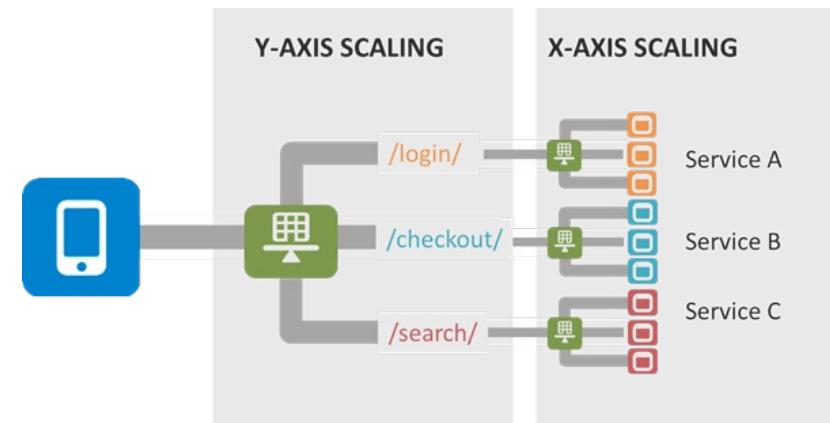
Z-AXIS SCALING

Network name: Layer 7 Load Balancing, Content switching, HTTP Message Steering



Y-AXIS SCALING

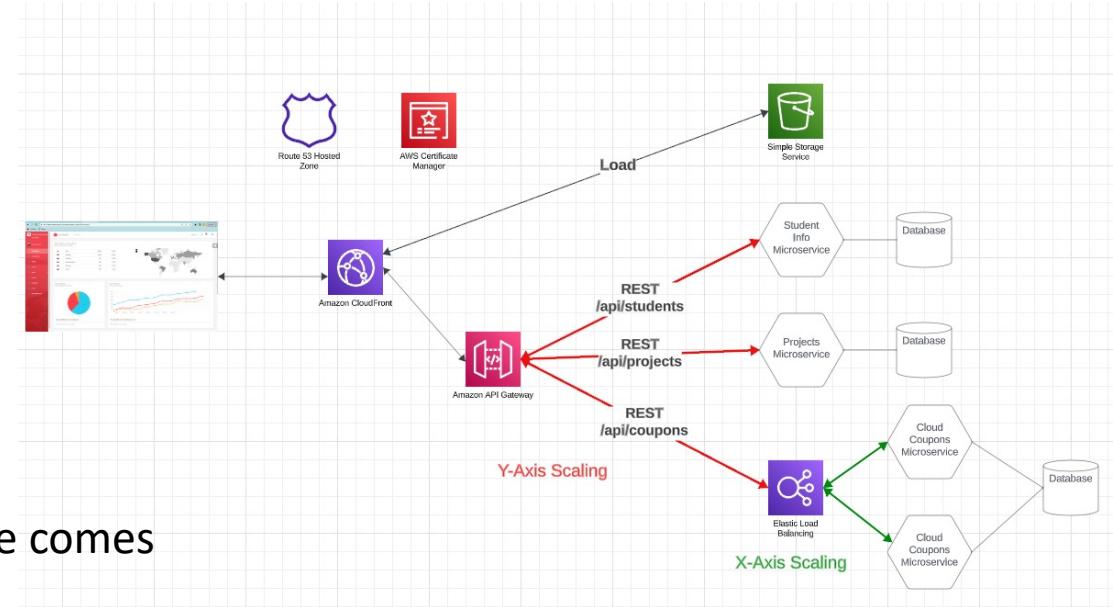
X-AXIS SCALING



- There are three dimensions.
- A complete solution/environment typical is a mix and composition of patterns.
- Application design and data access determines options.

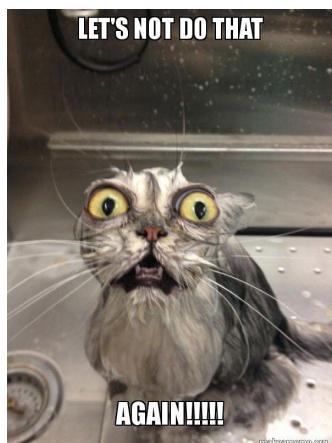
How Will We Apply Scaling Patterns

- API GW routing and multiple microservice implement Y-Axis Scaling
- I will show how to use Elastic Load Balancer and Target Groups for X-Axis Scaling but you will not have to do it.
- Just understand Z-Axis Scaling.
- Inherent scaling and performance comes from S3, CloudFront,
- Will show some of the networking to get better URLs, e.g. Route 53, Certificate Manager

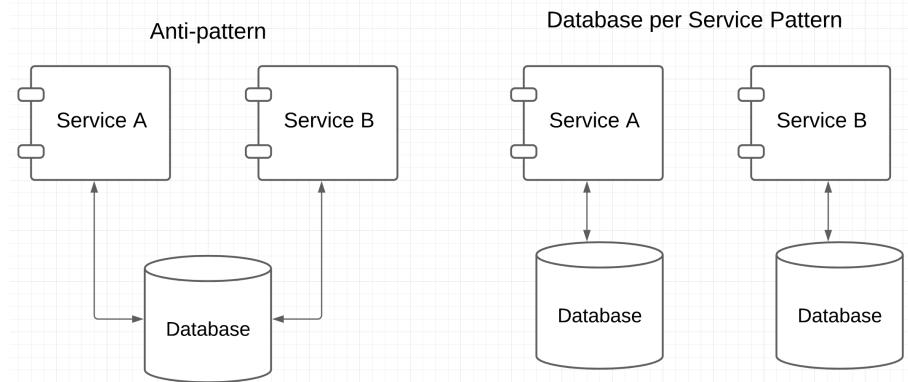


Patterns and Anti-patterns

- “software design pattern is a general, reusable solution to a commonly occurring problem within a given context in software design. It is not a finished design that can be transformed directly into source or machine code. Rather, it is a description or template for how to solve a problem that can be used in many different situations.” (https://en.wikipedia.org/wiki/Software_design_pattern)
- “An anti-pattern is a common response to a recurring problem that is usually ineffective and risks being highly counterproductive.”
(<https://en.wikipedia.org/wiki/Anti-pattern>)



<https://microservices.io/patterns/data/database-per-service.html>



Prevents independent evolution by creating dependency.

Patterns and Anti-patterns

- “software design pattern is a general, reusable solution to a commonly occurring problem within a given context in software design. It is not a finished design that can be transformed directly
- The question of “why?” has arisen. There are online discussions of pros and cons.
 - <https://docs.aws.amazon.com/prescriptive-guidance/latest/modernization-data-persistence/database-per-service.html>
 - <https://microservices.io/patterns/data/database-per-service.html>
 - https://www.tutorialspoint.com/microservices_design_patterns/microservices_design_patterns_database_per_service.htm



Prevents independent evolution by creating dependency.

REST, Part I

Concepts

REST (<https://restfulapi.net/>)

What is REST

- REST is acronym for REpresentational State Transfer. It is architectural style for **distributed hypermedia systems** and was first presented by Roy Fielding in 2000 in his famous [dissertation](#).
- Like any other architectural style, REST also does have its own [6 guiding constraints](#) which must be satisfied if an interface needs to be referred as **RESTful**. These principles are listed below.

Guiding Principles of REST

- **Client–server** – By separating the user interface concerns from the data storage concerns, we improve the portability of the user interface across multiple platforms and improve scalability by simplifying the server components.
- **Stateless** – Each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server. Session state is therefore kept entirely on the client.

- **Cacheable** – Cache constraints require that the data within a response to a request be implicitly or explicitly labeled as cacheable or non-cacheable. If a response is cacheable, then a client cache is given the right to reuse that response data for later, equivalent requests.
- **Uniform interface** – By applying the software engineering principle of generality to the component interface, the overall system architecture is simplified and the visibility of interactions is improved. In order to obtain a uniform interface, multiple architectural constraints are needed to guide the behavior of components. REST is defined by four interface constraints: identification of resources; manipulation of resources through representations; self-descriptive messages; and, hypermedia as the engine of application state.
- **Layered system** – The layered system style allows an architecture to be composed of hierarchical layers by constraining component behavior such that each component cannot “see” beyond the immediate layer with which they are interacting.
- **Code on demand (optional)** – REST allows client functionality to be extended by downloading and executing code in the form of applets or scripts. This simplifies clients by reducing the number of features required to be pre-implemented.

Resources

Resources are an abstraction. The application maps to create things and actions.

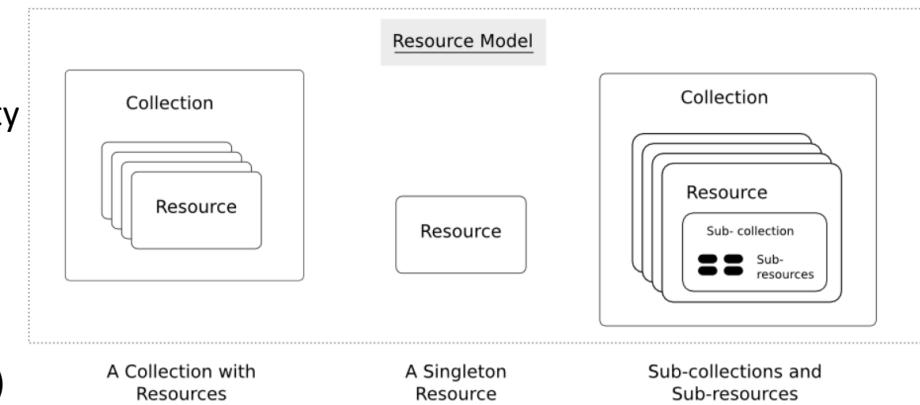
“A resource-oriented API is generally modeled as a resource hierarchy, where each node is either a *simple resource* or a *collection resource*. For convenience, they are often called a resource and a collection, respectively.

- A collection contains a list of resources of **the same type**. For example, a user has a collection of contacts.
- A resource has some state and zero or more sub-resources. Each sub-resource can be either a simple resource or a collection resource.

For example, Gmail API has a collection of users, each user has a collection of messages, a collection of threads, a collection of labels, a profile resource, and several setting resources.

While there is some conceptual alignment between storage systems and REST APIs, a service with a resource-oriented API is not necessarily a database, and has enormous flexibility in how it interprets resources and methods. For example, creating a calendar event (resource) may create additional events for attendees, send email invitations to attendees, reserve conference rooms, and update video conference schedules. (Emphasis added)

(<https://cloud.google.com/apis/design/resources#resources>)



<https://restful-api-design.readthedocs.io/en/latest/resources.html>

REST – Resource Oriented

- When writing applications, we are used to writing functions or methods:
 - `openAccount(last_name, first_name, tax_payer_id)`
 - `account.deposit(deposit_amount)`
 - `account.close()`

We can create and implement whatever functions we need.

- REST only allows four methods:

- POST: Create a resource
- GET: Retrieve a resource
- PUT: Update a resource
- DELETE: Delete a resource

That's it. That's all you get.

"The key characteristic of a resource-oriented API is that it emphasizes resources (data model) over the methods performed on the resources (functionality). A typical resource-oriented API exposes a large number of resources with a small number of methods."
(<https://cloud.google.com/apis/design/resources>)

- A REST client needs no prior knowledge about how to interact with any particular application or server beyond a generic understanding of hypermedia.

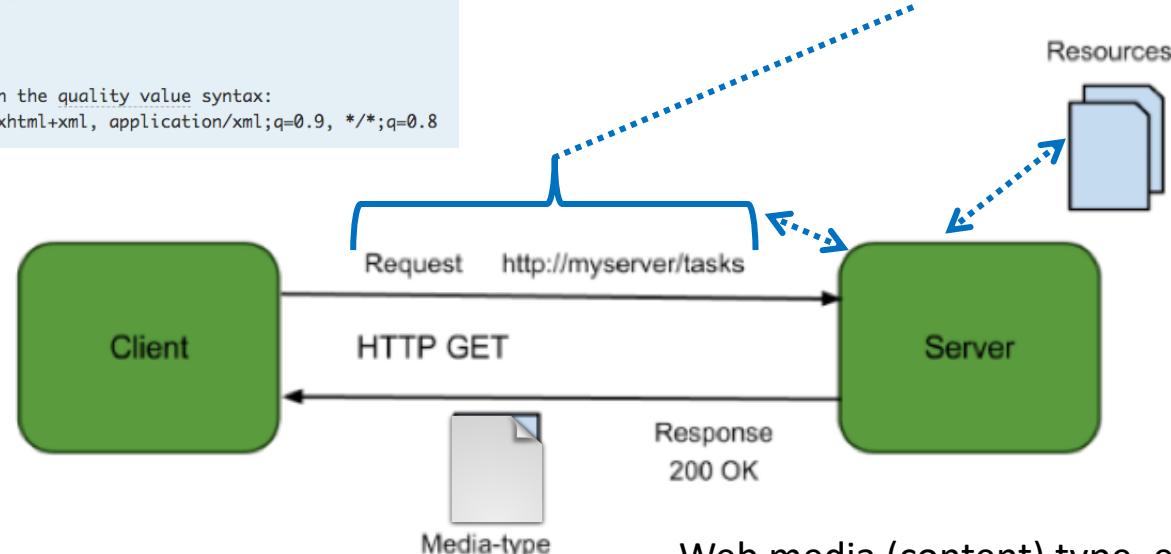
Resources, URLs, Content Types

Accept type in headers.

```
Accept: <MIME_type>/<MIME_subtype>
Accept: <MIME_type>/*
Accept: */*
```

```
// Multiple types, weighted with the quality value syntax:
Accept: text/html, application/xhtml+xml, application/xml;q=0.9, */*;q=0.8
```

- Relative URL identifies “resource” on the server.
- Server implementation maps abstract resource to tangible “thing,” file, DB row, ... and any application logic.



Client may be
Browser
Mobile device
Other REST Service
... ...

- Web media (content) type, e.g.
- `text/html`
- `application/json`

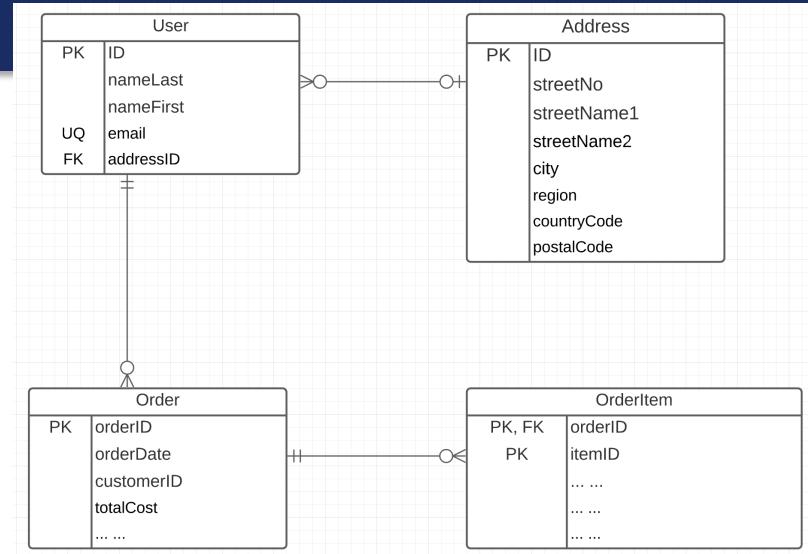
REST Principles

- What about all of those principles?
 - Client/Server
 - Stateless
 - Cacheable
 - Uniform Interface
 - Layered System
 - Code on demand
- Some of these principles
 - Are simple and obvious.
 - Are subtle and have major implications.
 - Stateless can be baffling, but we will cover later.
- We will go into the principles in later lectures, ... but first linked resources

Linked Data

Linked Data

- Consider two microservices:
 - UserService handles two resources:
 - User
 - Address
 - OrderService handles two resources:
 - Order
 - OrderItem
- Associations/Link/Relationships can be surprisingly complicated. There are many considerations. (See <https://www.uml-diagrams.org/association.html>)
- In this example, we have
 - Association (User-Address, User-Order)
 - Composition (Order-OrderItem)



Resource Paths

The simple object model might/could/would/should have the following:

- /users
- /users/<userID>
- /users/<userID>/address
- /users/<userID>/orders
- /addresses
- /addresses/<addressID>/users
- /orders/<orderID>
- /orders/<orderID>/orderitems
- /orders<orderID>/orderitems/<itemID>
-

Composition



Association



- How do you “point backwards” in a “foreign key relationship,” e.g. Address → User
 - “href” : “/users?addressID=201”
 - This means you store the addressID in the DB but return a link on REST.

Example

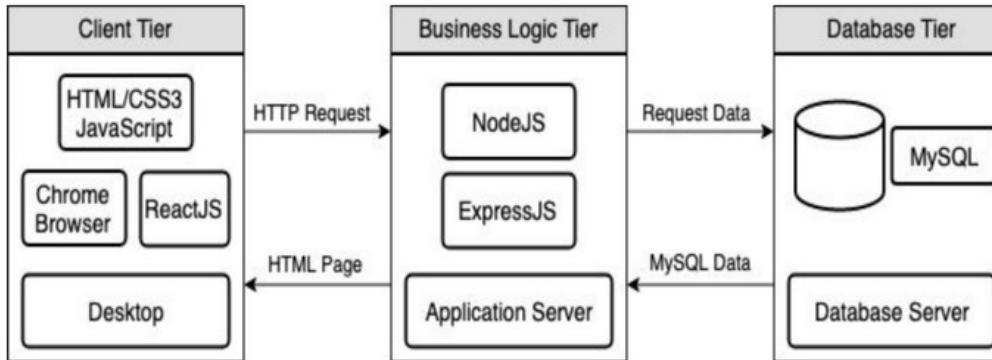
Examples

- SwaggerHub
- FastAPI and Pydantic examples.
- /Users/donaldferguson/Dropbox/000-NewProjects/W4111-FastAPI-IMDB-Solution

BLOBs, AWS S3, Web Server

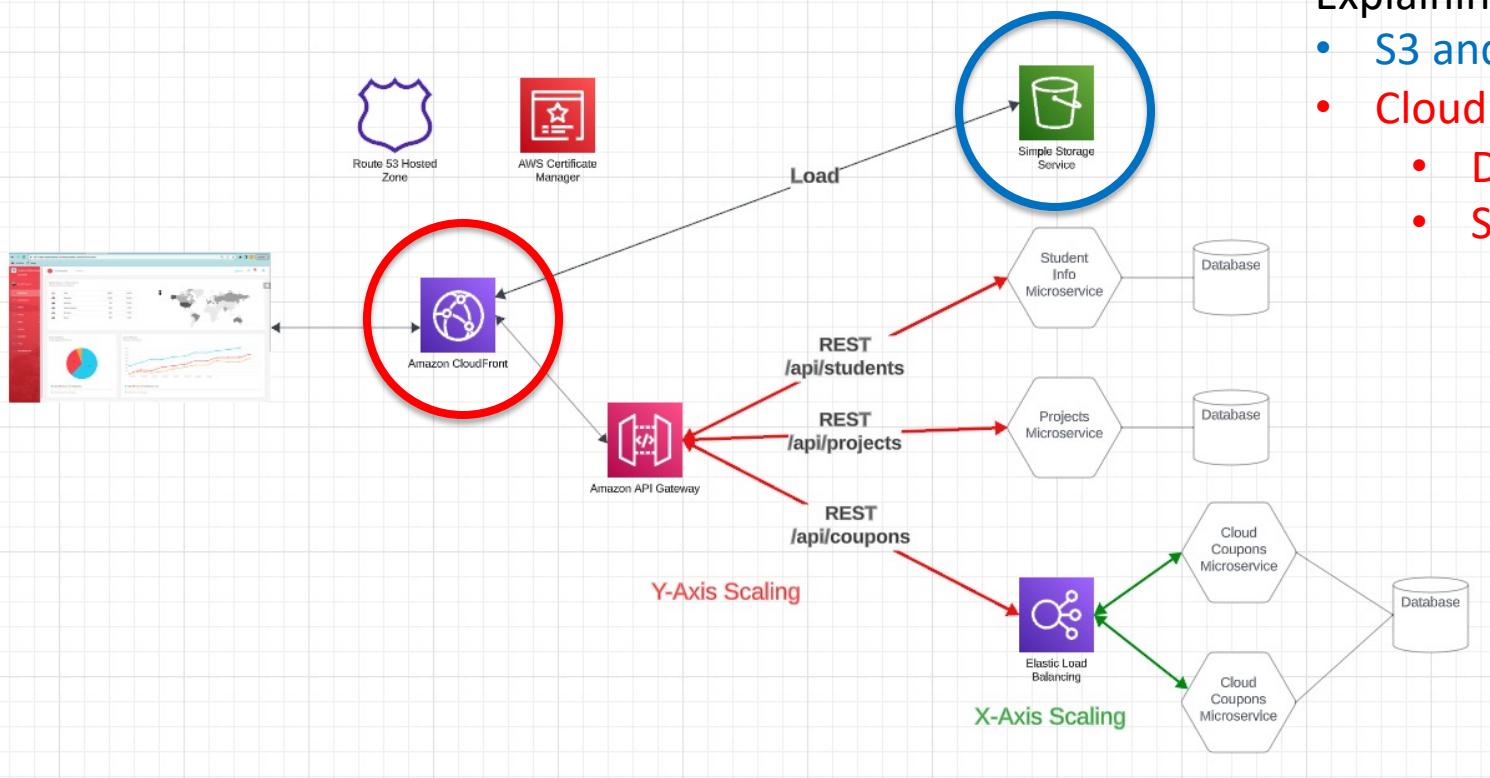
Full Stack Web Application – Reminder

<https://levelup.gitconnected.com/a-complete-guide-build-a-scalable-3-tier-architecture-with-mern-stack-es6-ca129d7df805>



- We have seen how to use Flask (or FastAPI) to implement an endpoint (a set of paths):
 - Some of the paths can run application logic and return the result data.
 - Some can return pages that a browser can render. Basically, these paths are returning the “client application” to the browser to execute, and may have API calls in JavaScript/TypeScript.
 - Do not worry about this now.
- But, running application logic and simply delivering files are two different “concerns.”
 - Having one runtime implement both would violate the SOLID principle.
 - The design points are different: “You can be a floor wax or a mouth wash, but not both.”
- So, we need a separate “web server.”

Reminder and Positioning



Explaining:

- S3 and web hosting.
- CloudFront:
 - Domain
 - Single site

BLOB: Binary Large Object

- Definitions:
 - “A binary large object (BLOB or blob) is a collection of binary data stored as a single entity. Blobs are typically images, audio or other multimedia objects, though sometimes binary executable code is stored as a blob. They can exist as persistent values inside some databases or version control system, or exist at runtime as program variables in some programming languages.”
(https://en.wikipedia.org/wiki/Binary_large_object)
 - “Blob is the data type in MySQL that helps us store the object in the binary format. It is most typically used to store the files, images, etc media files for security reasons or some other purpose in MySQL.” (<https://www.educba.com/mysql-blob/>)
- Intent:
 - File systems are not good for many application scenarios, which is why database management systems emerged.
 - Database management systems implement structured (or semi-structured) data, e.g. tables.
 - Database BLOBs were a way to have some “big things” as table or document properties.

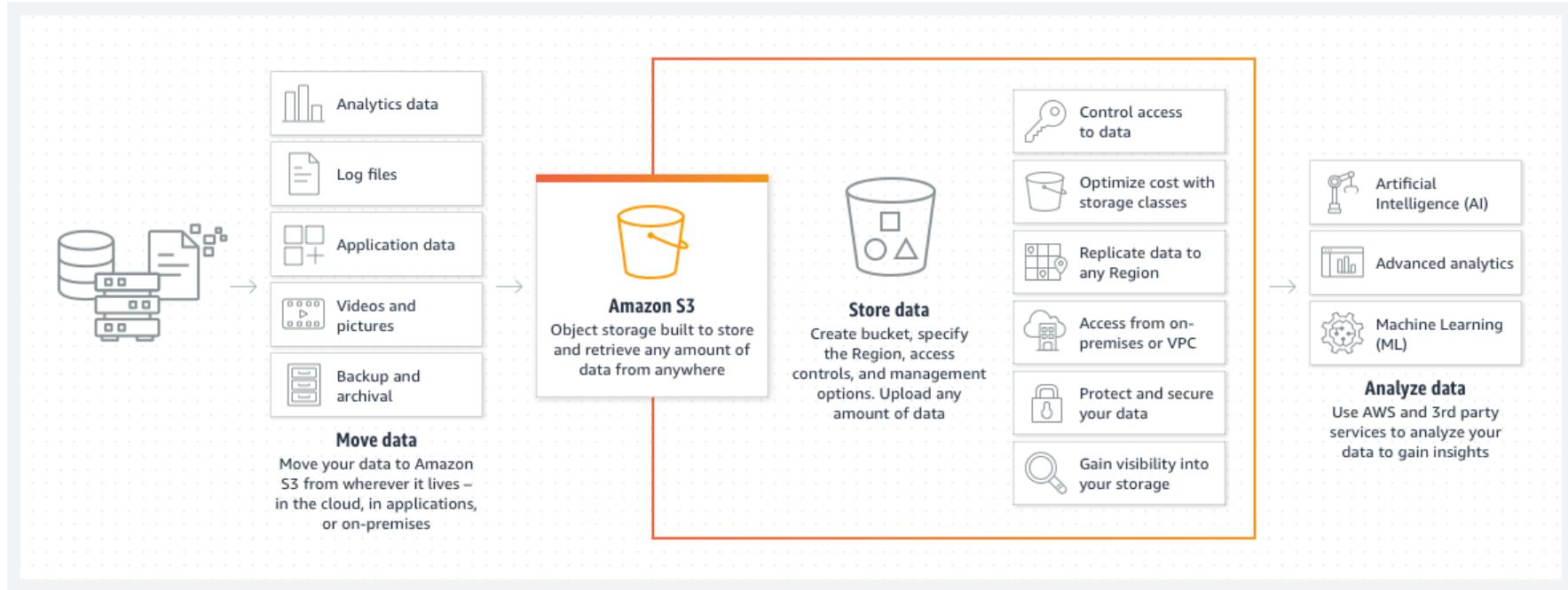
Simple Example

- `CREATE TABLE `products` (`productCode` varchar(15) NOT NULL, `productName` varchar(70) NOT NULL, `productLine` varchar(50) NOT NULL, `productScale` varchar(10) NOT NULL, `productVendor` varchar(50) NOT NULL, `productDescription` text NOT NULL, `quantityInStock` smallint NOT NULL, `buyPrice` decimal(10,2) NOT NULL, `MSRP` decimal(10,2) NOT NULL, `productImage` blob, `productManual` blob)`

- Structured or semi-structured information about an entity.
 - Queryable
 - Editable on forms
 - etc.
- Unstructured: A file-like, opaque property associated with the entity. This is a BLOB.

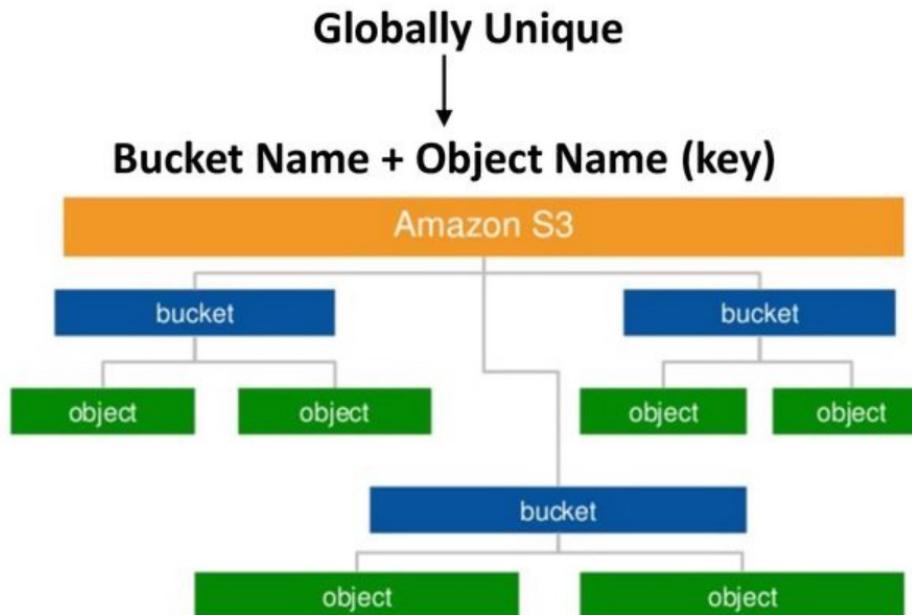
URLs have primarily replaced BLOBS.

Amazon S3



We used the equivalent capability on Azure for uploading CAD files, storing processed files and results, etc.

Concepts of S3 – Namespace



Can support a logical hierarchy with a convention on keys.

- x/y
- x/z
- x/y/z
-

Acts kind of file system like, but is really just key strings.

Simple S3 Example

```
import boto3
import json

s3_client = boto3.client('s3')

s3_resource = boto3.resource('s3')

def list_buckets():
    response = s3_client.list_buckets()

    # Output the bucket names
    print('\n\nexisting buckets:')
    for bucket in response['Buckets']:
        print(f' {bucket["Name"]}')



```

```
def get_object():

    res = s3_client.get_object(
        Bucket='e6156f20site',
        Key='assets/snuffle.png')

    print("\n\n The object is ...", json.dumps(res, indent=2, default=str))

    dd = res['Body'].read()
    print("\n\nThe first 1000 bytes of the body are:\n", dd[0:1000])

    with open("./snuffle.png", "wb") as outfile:
        outfile.write(dd)
        outfile.close()

    print("Wrote the file.")

if __name__ == "__main__":
    # list_buckets()
    get_object()
```

/Users/donaldferguson/Dropbox/00NewProjects/e6156examples/s3/s3_examples.py

Simple S3 Example

```
import boto3
import json

s3_client = boto3.client('s3')

s3_resource = boto3.resource('s3')

def list_objects(bucket):
    res = s3_client.list_objects(Bucket=bucket)
    # Objects is a list of objects in the bucket
    # We're only interested in the keys
    pri
```

```
def get_object():
    res = s3_client.get_object(
        Bucket='e6156f20site',
        Key='assets/snuffle.png')
    # The response is a dict with a 'Body' key
    # which contains the file contents
    # We're only interested in the body
    # so we'll use the default str() conversion
    # to convert the binary data to a string
    return res['Body'].read().decode('utf-8')
```

Will worry about APIs to S3 later.

/Users/donaldferguson/Dropbox/00NewProjects/e6156examples/s3/s3_examples.py

Walkthrough

- CloudFront
 - Distribution d2eakloxxxvksx.cloudfront.net
 - Origins
 - Behaviors
- Enablement:
 - Route 53 Domain and DNS
 - Certificate and HTTPS
- S3:
 - <https://s3.console.aws.amazon.com/s3/buckets/dff-class-dashboard?region=us-east-1&tab=objects>
 - Deployment and buckets.
 - Policy