W4153 – Cloud Computing

Lecture 2: IaaS, REST, BLOBs, Design Patterns

© Donald F. Ferguson, 2024

W4153 – Cloud Computing

Lecture 2: IaaS, REST, BLOBs, Design Patterns

We will start in a few minutes.

If you ask me about the waitlist,
I will become very grumpy.

© Donald F. Ferguson, 2024

Contents

© Donald F. Ferguson, 2024

Contents

- Infrastructure-as-a-Service
  - Concepts
  - AWS realization
  - Deploying first VM
- Microservices
  - Concepts and characteristics
  - Architecture best practices: 12-Factor Apps, SOLID
  - Design patterns
- REST
  - Concepts
  - A basic implementation example
- BLOBs and static content
  - Concepts
  - AWS realization
  - Deploying static content
- Next steps -- project teams, deploying on cloud, … …

© Donald F. Ferguson, 2024

Infrastructure-as-a-Service

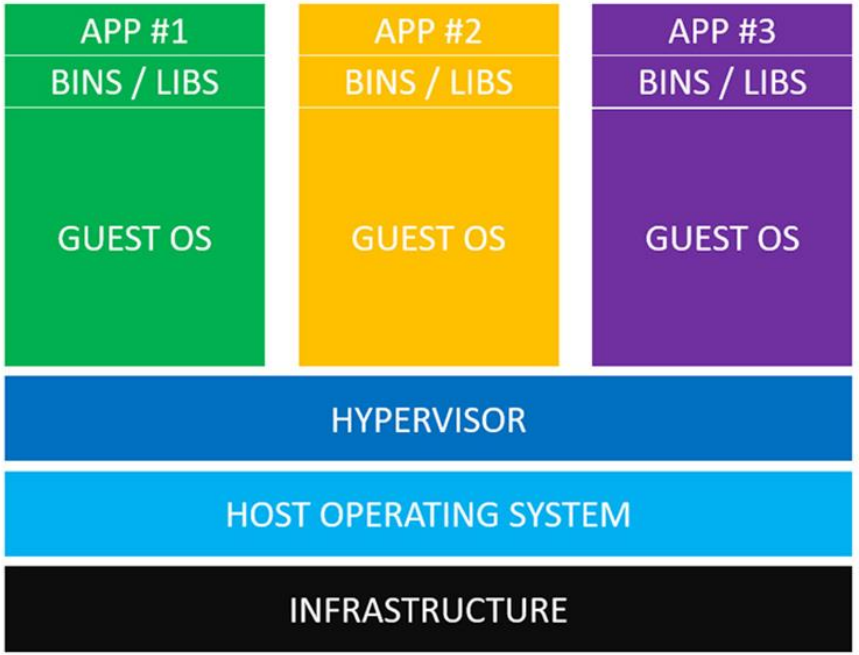
© Donald F. Ferguson, 2024

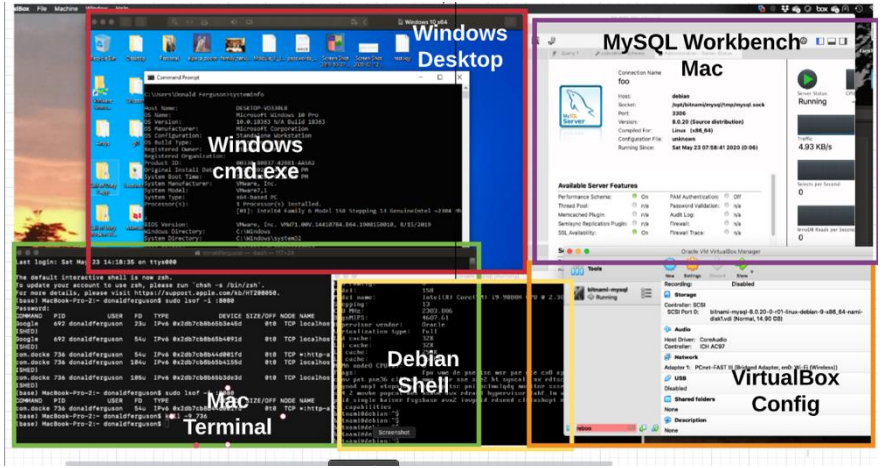
Virtualization

© Donald F. Ferguson, 2024

Virtualization, Virtual Machines

- "In computing, a virtual machine (VM) is an emulation of a computer system. Virtual machines are based on computer architectures and provide functionality of a physical computer. Their implementations may involve specialized hardware, software, or a combination." (https://en.wikipedia.org/wiki/Virtual_machine)





On my Mac laptop, I used to have:

- One host operating system and HW (Mac, MacOS)
- Two hypervisors (Virtual Box, VMWare Fusion)
- Two guest OS: Windows 10, Debian Linux

Then, …

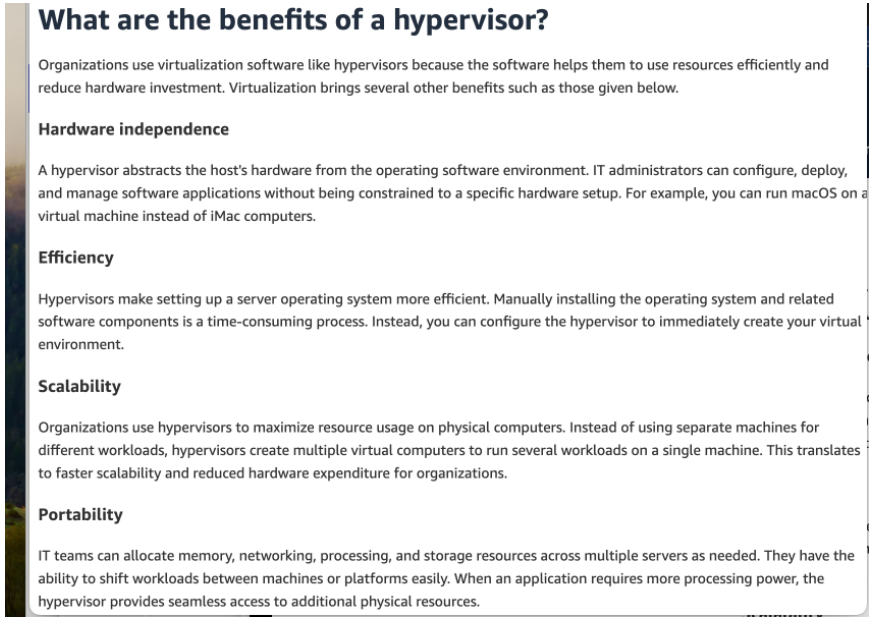
- Apple transitioned to ARM
- Broadcom bought VMWare

Then people started asking, “Why do you want to do this grandpa?”

© Donald F. Ferguson, 2024

Hypervisor

- "A hypervisor, also known as a virtual machine monitor (VMM) or virtualizer, is a type of computer software, firmware or hardware that creates and runs virtual machines. A computer on which a hypervisor runs one or more virtual machines is called a host machine, and each virtual machine is called a guest machine. The hypervisor presents the guest operating systems with a virtual operating platform and manages the execution of the guest operating systems." (https://en.wikipedia.org/wiki/Hypervisor)
- **Instructor's Note:**
  - **If I let you use Google, Stack Overflow, GitHub, … … to write code, solve problems, …**
  - **I get to use Google, Stack Overview, GitHub, … … to make my lecture slides.**



What are the benefits of a hypervisor?

Organizations use virtualization software like hypervisors because the software helps them to use resources efficiently and reduce hardware investment. Virtualization brings several other benefits such as those given below.

Hardware independence

A hypervisor abstracts the host's hardware from the operating software environment. IT administrators can configure, deploy, and manage software applications without being constrained to a specific hardware setup. For example, you can run macOS on a virtual machine instead of iMac computers.

Efficiency

Hypervisors make setting up a server operating system more efficient. Manually installing the operating system and related software components is a time-consuming process. Instead, you can configure the hypervisor to immediately create your virtual environment.

Scalability

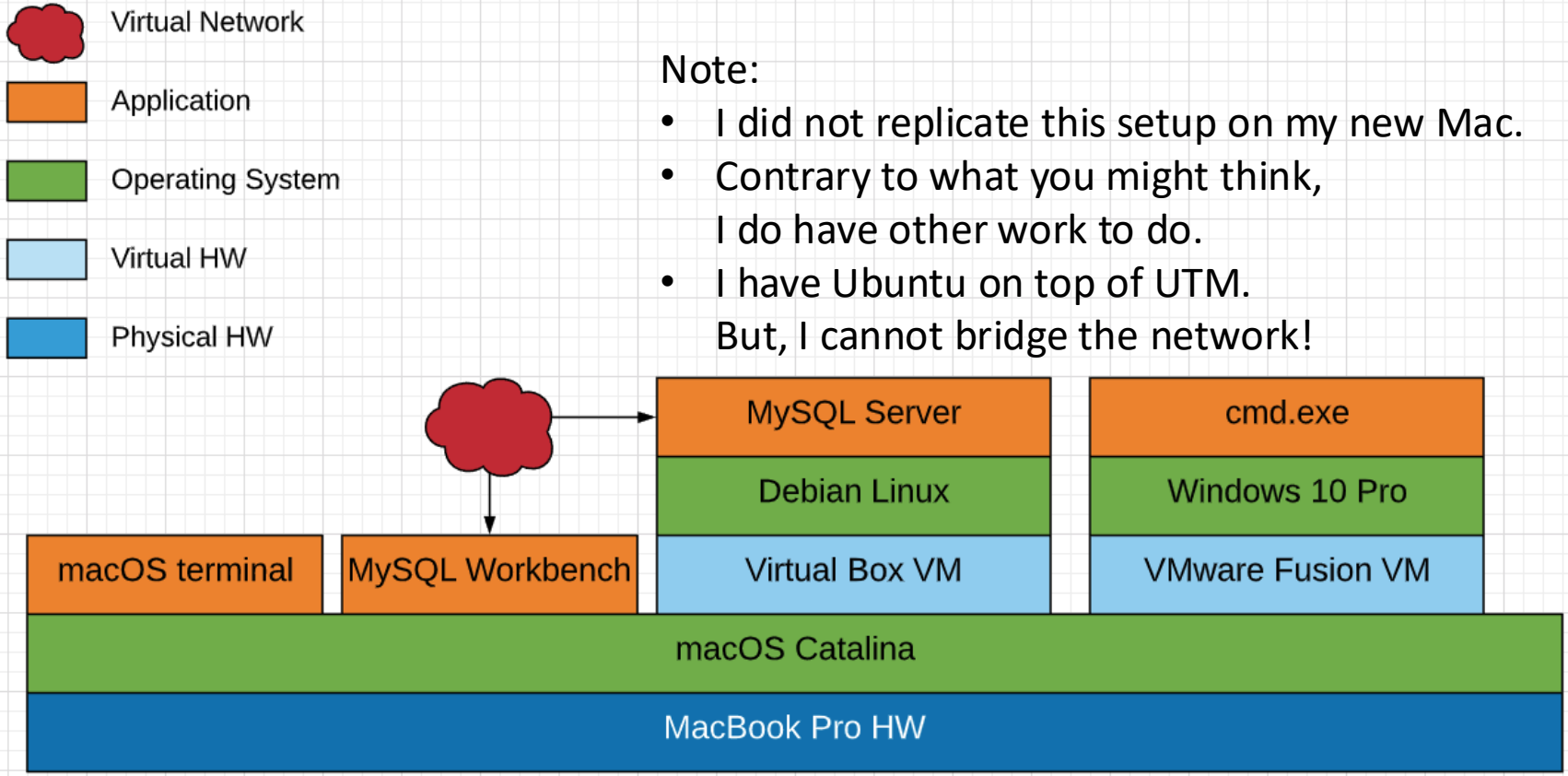
Organizations use hypervisors to maximize resource usage on physical computers. Instead of using separate machines for different workloads, hypervisors create multiple virtual computers to run several workloads on a single machine. This translates to faster scalability and reduced hardware expenditure for organizations.

Portability

IT teams can allocate memory, networking, processing, and storage resources across multiple servers as needed. They have the ability to shift workloads between machines or platforms easily. When an application requires more processing power, the hypervisor provides seamless access to additional physical resources.

© Donald F. Ferguson, 2024

Virtual Machines on Physical Machines

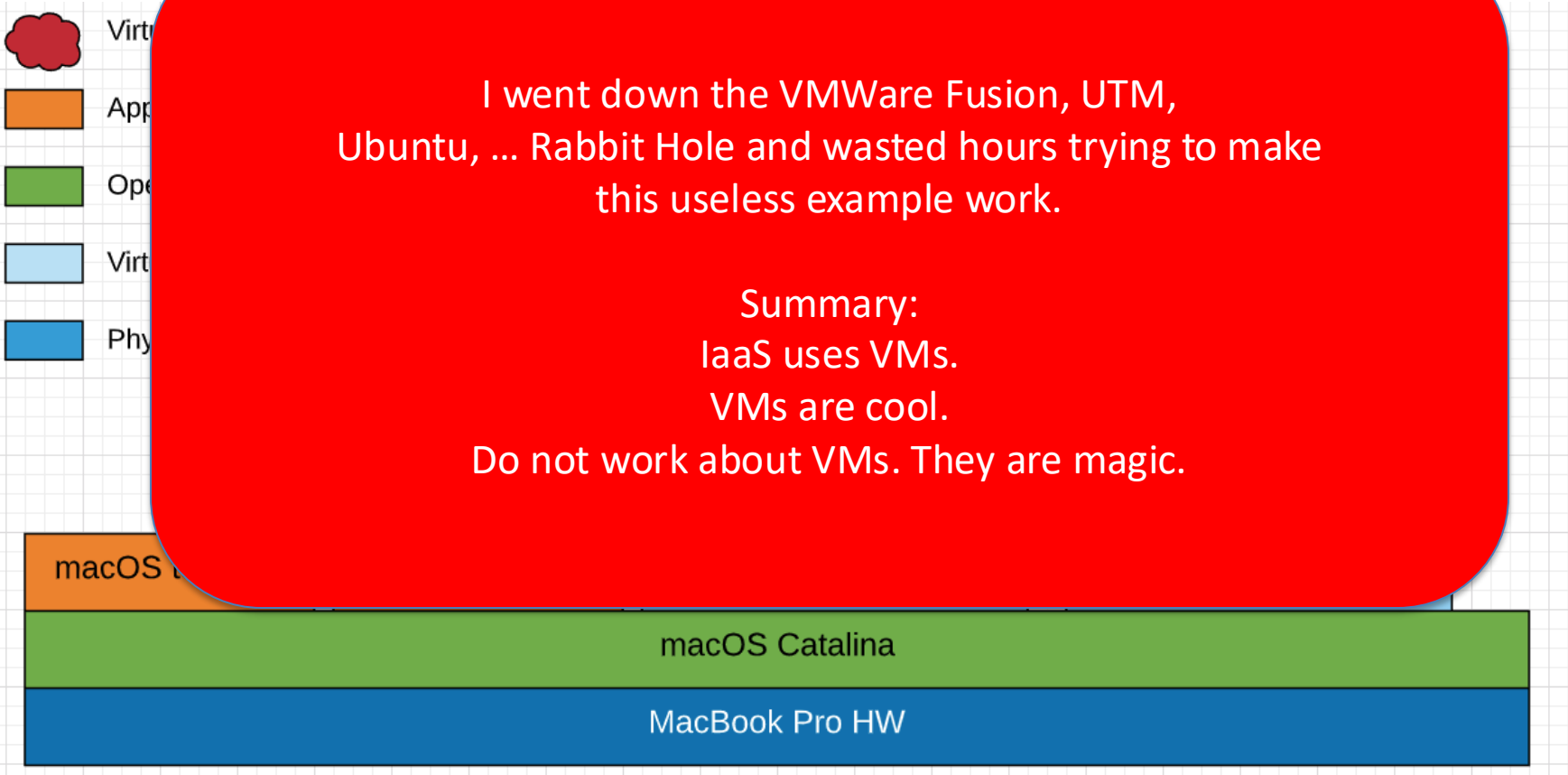


Note:

- I did not replicate this setup on my new Mac.
- Contrary to what you might think,
  I do have other work to do.
- I have Ubuntu on top of UTM.
  But, I cannot bridge the network!

© Donald F. Ferguson, 2024

Virtual Machines on Physical Machines

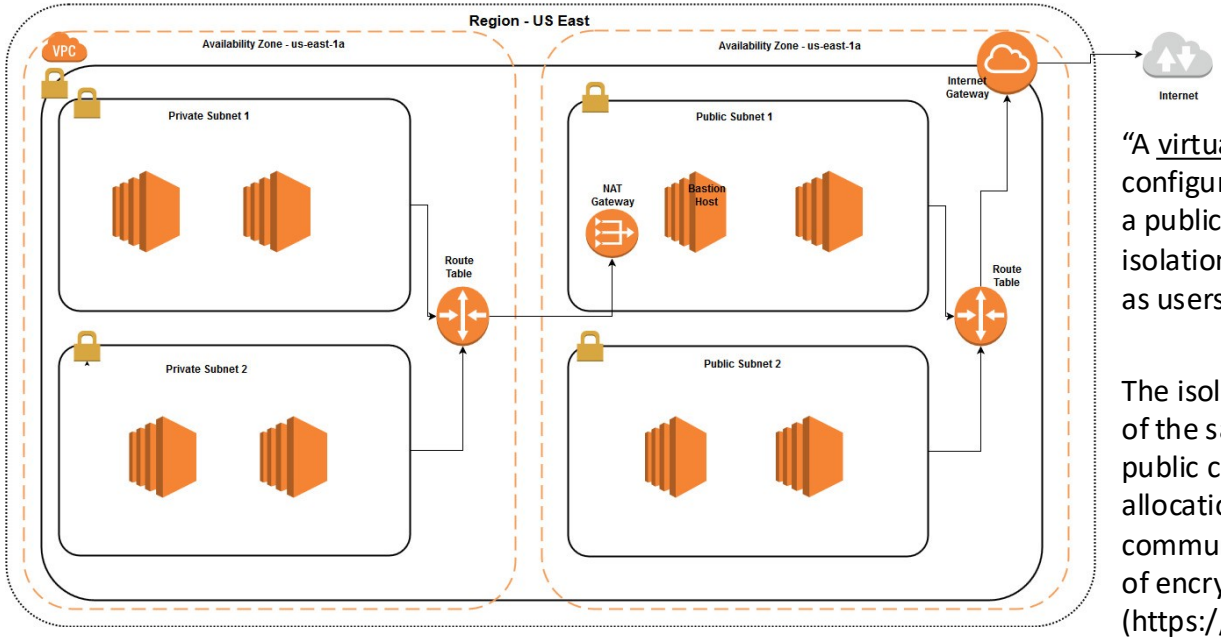


I went down the VMWare Fusion, UTM, Ubuntu, … Rabbit Hole and wasted hours trying to make this useless example work.

Summary:
IaaS uses VMs.
VMs are cool.
Do not work about VMs. They are magic.

© Donald F. Ferguson, 2024

Virtual Private Cloud



"A virtual private cloud (VPC) is an on-demand configurable pool of shared resources allocated within a public cloud environment, providing a certain level of isolation between the different organizations (denoted as users hereafter) using the resources. ... ...

The isolation between one VPC user and all other users of the same cloud (other VPC users as well as other public cloud users) is achieved normally through allocation of a private IP subnet and a virtual communication construct (such as a VLAN or a set of encrypted communication channels) per user." (https://en.wikipedia.org/wiki/Virtual_private_cloud)

- A virtual machine on my laptop has a virtual adaptor connected to a real network.
- A virtual private cloud is a set of virtual machines connected to a virtual network.
- All cloud providers support virtual private clouds with the same concepts, but with slightly different realizations and terms.

© Donald F. Ferguson, 2024

Virtual Private Cloud and VM

© Donald F. Ferguson, 2024

Some Terminology (AWS)

There are several online tutorials. Understanding some terms is useful.

- AWS has the concept of a Region, which is a physical location around the world where we cluster data centers. We call each group of logical data centers an Availability Zone. Each AWS Region consists of multiple, isolated, and physically separate AZ's within a geographic area.
- An Availability Zone (AZ) is one or more discrete data centers with redundant power, networking, and connectivity in an AWS Region. AZs give customers the ability to operate production applications and databases that are more highly available, fault tolerant, and scalable than would be possible from a single data center.
- A VPC is a virtual private cloud, which works like a private network to isolate the resources within it.
- A route table contains a set of rules, called routes, that are used to determine where network traffic is directed.
- A subnet is a defined set of network IP addresses that are used to increase the security and efficiency of network communications. You can think of them like postal codes, used for routing packages from one location to another.
- A network interface represents a virtual network card. The network interface displays its network interface ID, subnet ID, VPC ID, security group, and the Availability Zone that it exists in.
- A security group is a set of rules that controls the network access to the resources it is associated with. Access is permitted only to and from the components defined in the security group's inbound and outbound rules.
- An internet gateway is a VPC component that allows communication between instances in your VPC and the internet.
- A VPC endpoint enables you to privately connect your VPC to supported AWS services without using public IP addresses.

All clouds have similar concepts but with different names.

© Donald F. Ferguson, 2024

Let’s Build our First VPC and VM

- Well, actually, let’s not do that for now.
- We are just going to create our first virtual machine.
- But,
  - The virtual machine is in my “default” VPC.
  - Setting up and using the VM requires understanding some of the concepts.
- We can mostly just use the EC2 wizard for now.
- I am going to set up a VM to host our first cloud application
  - I will deploy the application and supporting SW.
  - Basic tasks:
    1. Configure and deploy VM, SSH credentials, security group rules.
    2. Install supporting software: Python, MySQL.
    3. Run “Hello World” test.
- There is an example in https://github.com/donald-f-ferguson/W4153-Hello-World-FastAPI.git

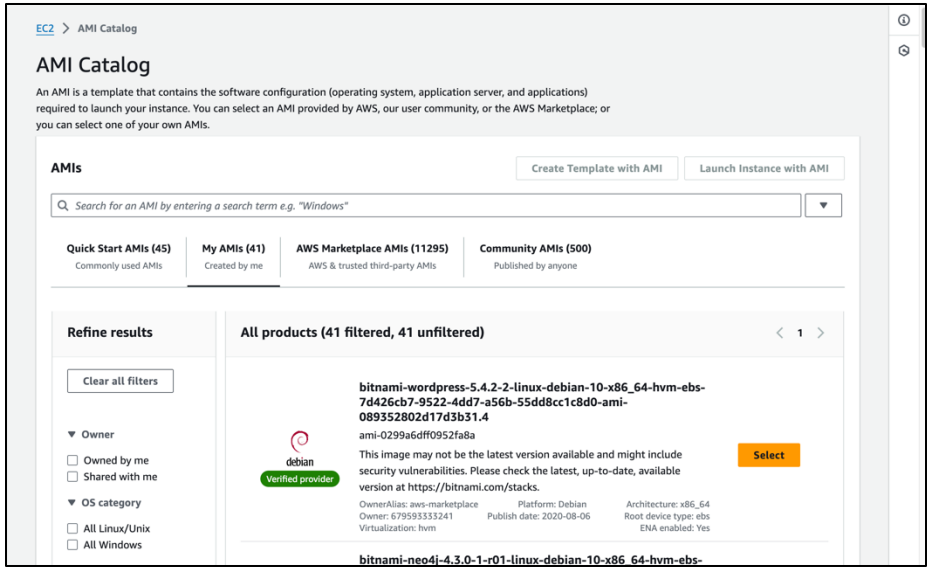
© Donald F. Ferguson, 2024

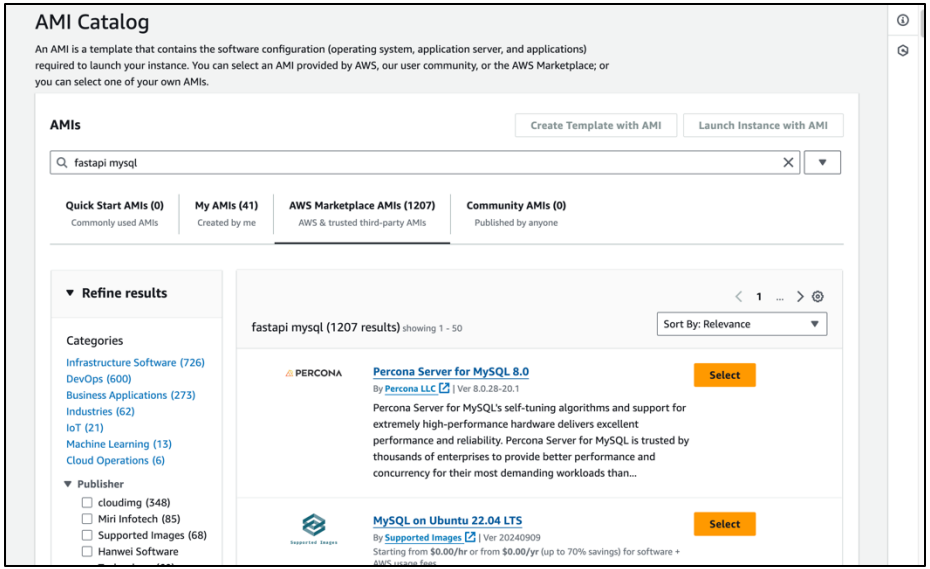
What Comes/Would Come Next?

- We did the VM work “manually,” and this was:
  - Tedious
  - Complex
- An organization typically has two “teams:”
  - Develop team
  - Infrastructure team
- The infrastructure team has a “library” of prebuilt/previously built environments. Provides automation tools to simplify iterative development and deployment.
- A development team can:
  - Create a new instance of a predefined environment.
  - Request creation or modification of an environment.
- Terms in this space are continuous integration/continuous delivery and DevOps.
- We do things manually in this course for education/understanding purposes. We will start automating in a couple of lectures.

© Donald F. Ferguson, 2024

Prebuilt Images





- There is usually an ecosystem of prebuilt versions/images that you can tailor.
- There are also often prebuilt automation scripts that you can use in deployment workflows.
- Teams/organizations build up “libraries” of “reliable, well-designed” environments.
- We will get some experience with this process.

© Donald F. Ferguson, 2024

Microservices (Continued)

© Donald F. Ferguson, 2024

Reminder

© Donald F. Ferguson, 2024

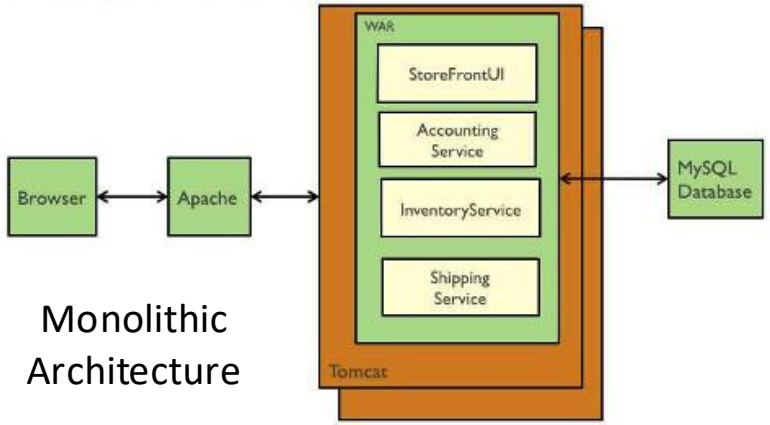
Microservices (https://microservices.io/index.html)

What are microservices?

Microservices - also known as the microservice architecture - is an architectural style that structures an application as a collection of services that are

- Highly maintainable and testable
- Loosely coupled
- Independently deployable
- Organized around business capabilities
- Owned by a small team

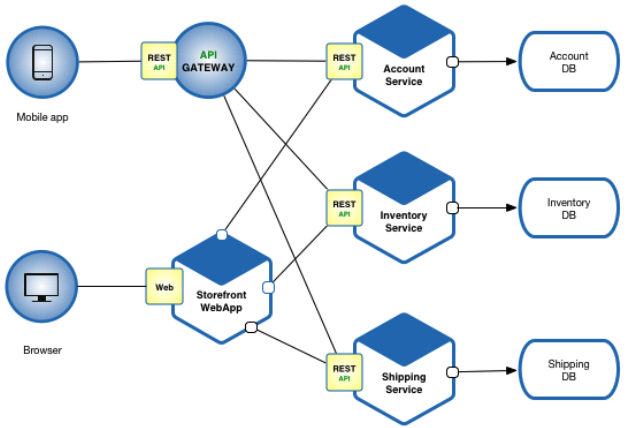
The microservice architecture enables the rapid, frequent and reliable delivery of large, complex applications. It also enables an organization to evolve its technology stack.



Monolithic Architecture

- Microservice benefits relative to monoliths are hard to understand without having developed and maintained monoliths.
- Cars are "cool." Cars are totally, super cool if you had to go into town using a horse drawn carriage.

Microservices Architecture



© Donald F. Ferguson, 2024

Microservice Characteristics

(https://dzone.com/articles/what-is-microservices-an-introduction-to-microserv)

Event based --> Loose coupled

Microservice Characteristics

- **Decoupling** - Services within a system are largely decoupled, so the application as a whole can be easily built, altered, and scaled.
- **Componentization** - Microservices are treated as independent components that can be easily replaced and upgraded.
- **Business Capabilities** - Microservices are very simple and focus on a single capability.
- **Autonomy** - Developers and teams can work independently of each other, thus increasing speed.
- **Continous Delivery** - Allows frequent releases of software through systematic automation of software creation, testing, and approval.
- **Responsibility** - Microservices do not focus on applications as projects. Instead, they treat applications as products for which they are responsible.
- **Decentralized Governance** - The focus is on using the right tool for the right job. That means there is no standardized pattern or any technology pattern. Developers have the freedom to choose the best useful tools to solve their problems.
- **Agility** - Microservices support agile development. Any new feature can be quickly developed and discarded again.

© Donald F. Ferguson, 2024

A student asked something like,
”Is there more to microservices than just breaking things into parts?”

➔

Design patterns and Design Principles

© Donald F. Ferguson, 2024

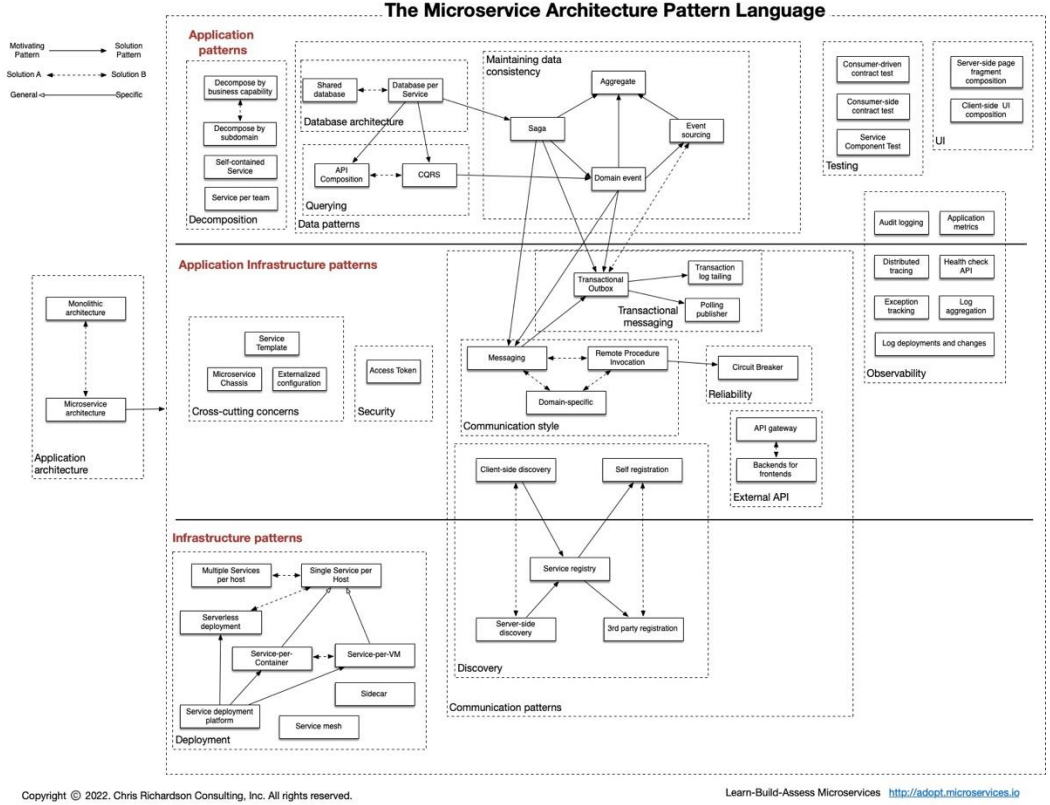
Design Patterns

- "In software engineering, a 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. Design patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system." (https://en.wikipedia.org/wiki/Software_design_pattern)

- We will cover design patterns in several aspects:
  - SW patterns for building code "in microservices." How do you write the code?
  - Architecture patterns for microservices and composition into complex solutions.
  - Cloud architecture patterns: how do you solve problems with cloud technology.
  - Data modeling and data access patters.
  - … …

© Donald F. Ferguson, 2024

Microservice Pattern Language (One Perspective)

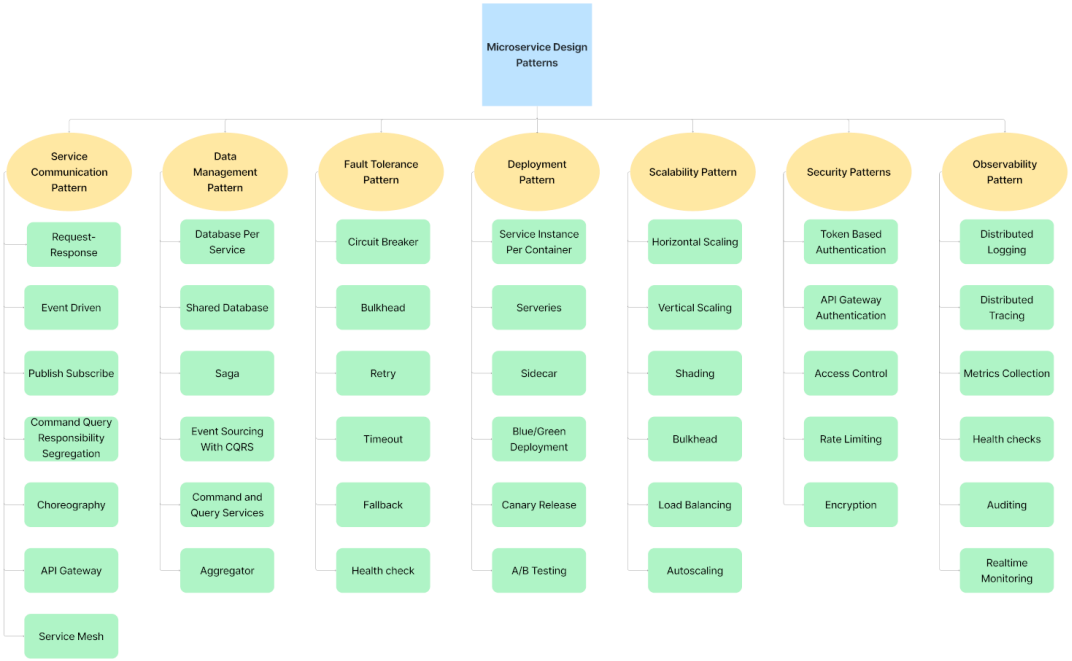


- If you are trying to solve a problem or implement something,
- You are probably not the first person to do so.
- Probably a lot of other people have tried, failed and then learned.
- Ask Google or ChatGPT what others do.
- A knowledge of basic design patterns is useful to guide architecture/implementation decisions.
- We will cover when encountering tasks and challenges.

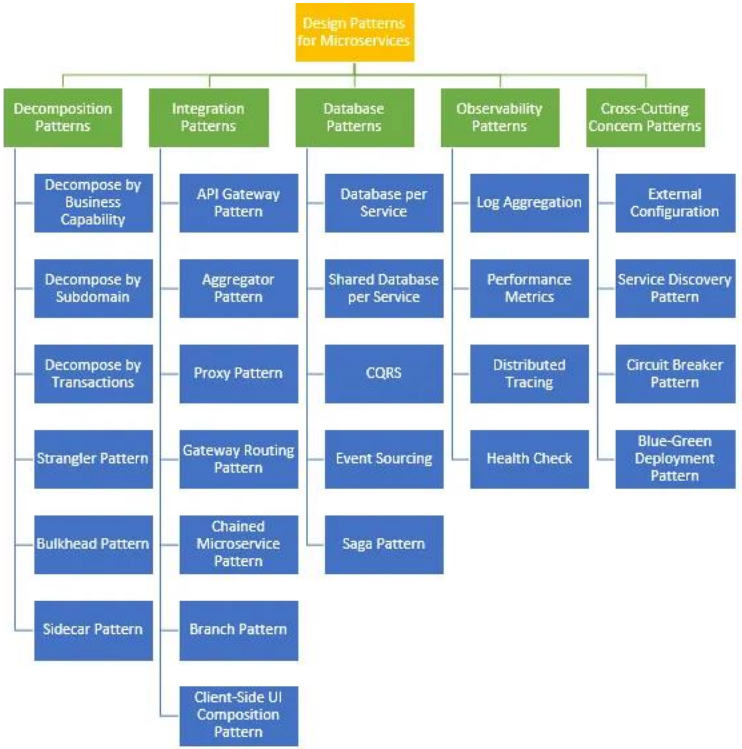
https://microservices.io/patterns/index.html

© Donald F. Ferguson, 2024

There are a Lot of Design Patterns



https://blog.beehyv.com/design-patterns-in-microservices-enhancing-scalability-flexibility-and-maintainability-197e499a575b



https://medium.com/@madhukaudantha/microservice-architecture-and-design-patterns-for-microservices-e0e5013fd58a

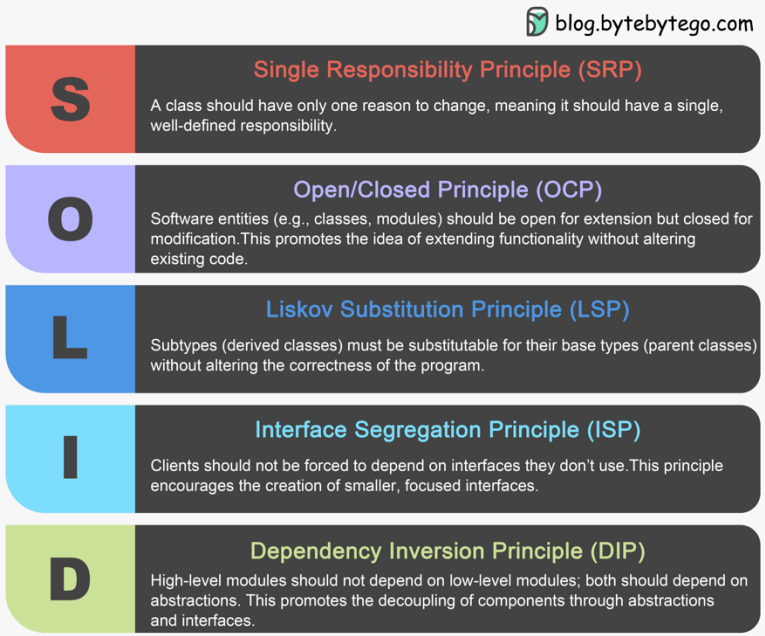
© Donald F. Ferguson, 2024

12 Factor Applications

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

© Donald F. Ferguson, 2024

SOLID Principles



Microservices Guidelines	SOLID Principles
Loosely coupled	Interface Segregation + Dependency Inversion
Testable	Single Responsibility + Dependency Inversion.
Composable	Single Responsibility + Open/close
Highly maintainable	Single Responsibility + Liskov Substitution + Interface Segregation
Independently deployable	Single Responsibility + Interface Segregation + Dependency Inversion
Capable of being developed by a small team	Single Responsibility + Interface Segregation

https://medium.com/@saurabh.engg.it/microservices-designing-effective-microservices-by-following-solid-design-principles-a995c3a033a0

This is a good overview article, and we will see patterns in later lectures.

- This slide and the preceding slides could have been a 60 slide presentation.
- If you put 3 SW architects into a room, you will get 15 design principles and patterns.
- Use common sense.

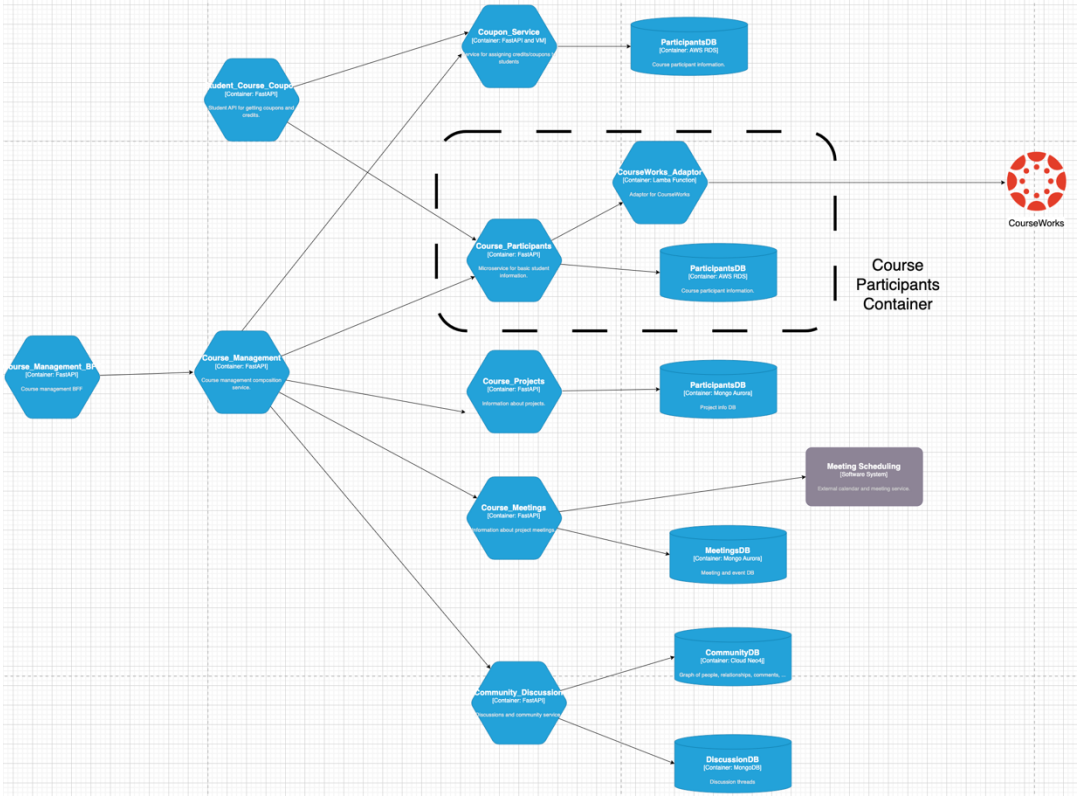
© Donald F. Ferguson, 2024

My First Steps

© Donald F. Ferguson, 2024

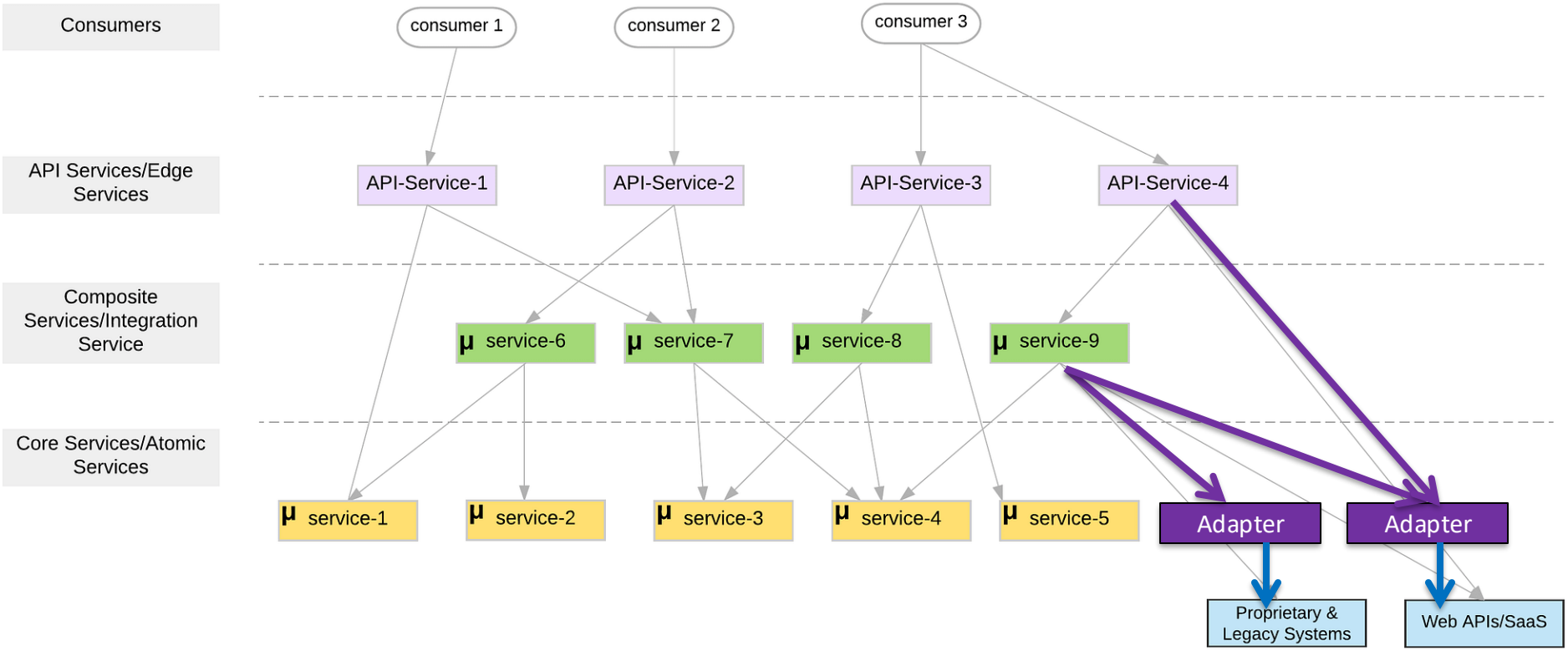
(Partial) Logical View – Interactive System

- Some of the distinctions are arbitrary or in the ”eye of the beholder.”
  - System vs Container
  - Container vs Component
  - etc.
- A container is not synonymous with a “Docker container.”
- I have been in SW for 40 years and we still cannot define terms like “component.”
- Just use common sense.
- In general, containers in the cloud are *microservices.”*



© Donald F. Ferguson, 2024

Our First Design Patterns: Adapter, Anti-Corruption



© Donald F. Ferguson, 2024

Adapter and Anti-Corruption

Adapter Microservice

You are building an application that is following a Microservices architecture. The team is not operating in a complete greenfield - there are existing sources of functionality or data that must be reused in order to complete the application on time and within budget. However,

This existing functionality cannot be migrated to a microservice architecture. We need a way to be able to take advantage of this functionality without abandoning the microservices approach.

There are many different types of existing functionality that may requires you to reuse existing code. It may be that there are specific enterprise data sources that can only be accessed through an existing API. Or it may be that you want to take advantage of a cloud-based (SaaS) API from a cloud provider that provides some or most of the functionality that you would otherwise have to build.

Therefore,

Wrap the functionality in an adapter. The adapter will (like the adapter pattern from Gamma) convert from the existing interface to a new interface consistent with microservice principles (e.g. REST or messaging).

Perhaps the hardest aspect to determining if you need to use an Adapter Microservice, particularly if the legacy service or Saas service already implements a RESTful interface, is whether or not to build an Adapter Microservice or simply use the existing REST API directly from your SPA or Mobile Application clients or your BFF's. There are several reasons why you may want to build an Adapter Microservice of one type or another in order to improve the maintainability of your code.

https://kgb1001001.github.io/cloudadoptionpatterns/

Anti-corruption Layer pattern

Azure | Azure Logic Apps

Implement a façade or adapter layer between different subsystems that don't share the same semantics. This layer translates requests that one subsystem makes to the other subsystem. Use this pattern to ensure that an application's design is not limited by dependencies on outside subsystems. This pattern was first described by Eric Evans in *Domain-Driven Design*.

Context and problem

Most applications rely on other systems for some data or functionality. For example, when a legacy application is migrated to a modern system, it may still need existing legacy resources. New features must be able to call the legacy system. This is especially true of gradual migrations, where different features of a larger application are moved to a modern system over time.

Often these legacy systems suffer from quality issues such as convoluted data schemas or obsolete APIs. The features and technologies used in legacy systems can vary widely from more modern systems. To interoperate with the legacy system, the new application may need to support outdated infrastructure, protocols, data models, APIs, or other features that you wouldn't otherwise put into a modern application.

Maintaining access between new and legacy systems can force the new system to adhere to at least some of the legacy system's APIs or other semantics. When these legacy features have quality issues, supporting them "corrupts" what might otherwise be a cleanly designed modern application.

Similar issues can arise with any external system that your development team doesn't control, not just legacy systems.

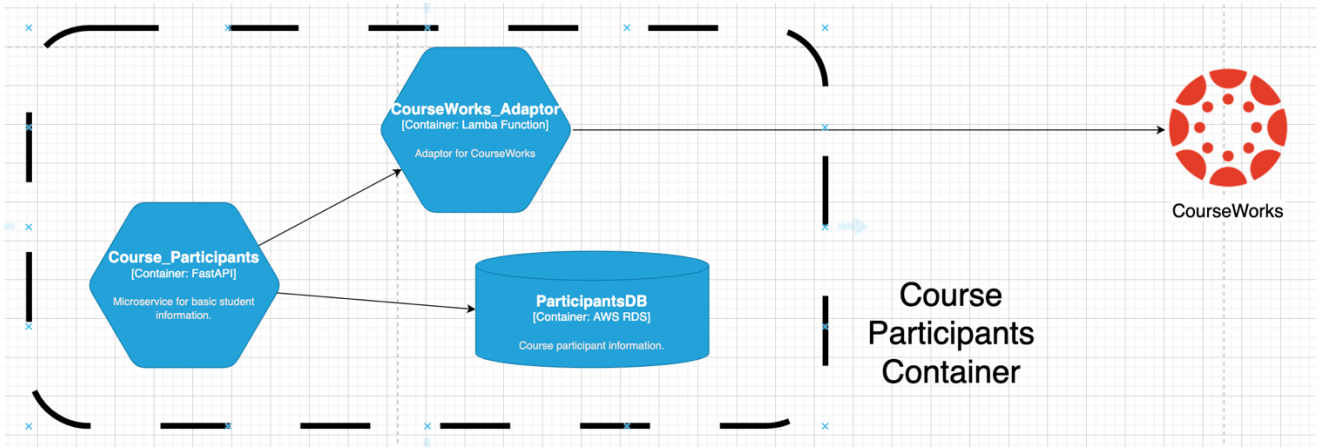
Solution

Isolate the different subsystems by placing an anti-corruption layer between them. This layer translates communications between the two systems, allowing one system to remain unchanged while the other can avoid compromising its design and technological approach.

https://learn.microsoft.com/en-us/azure/architecture/patterns/anti-corruption-layer

© Donald F. Ferguson, 2024

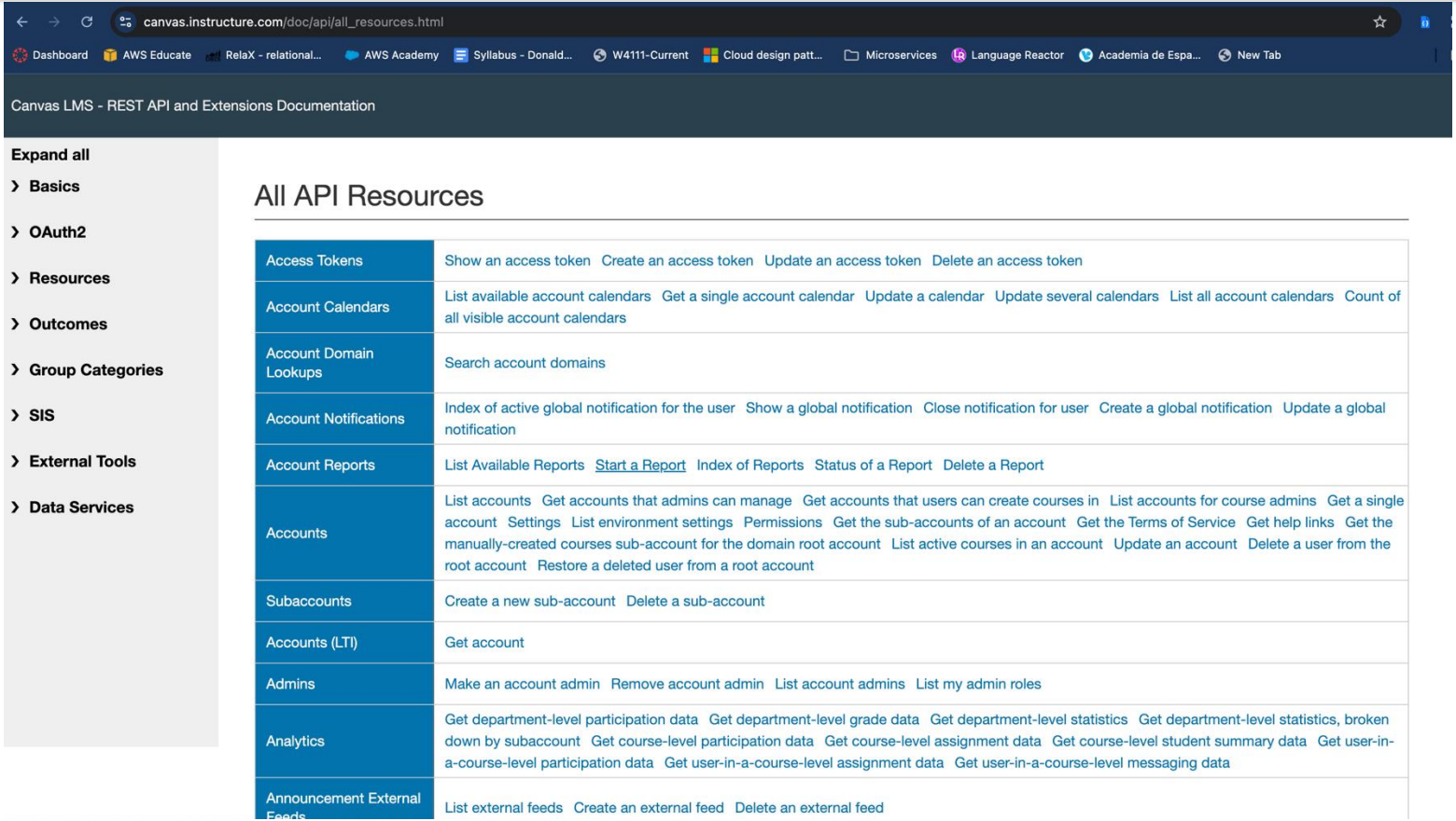
CourseWorks Adaptor



- CourseWorks has an API. It is an HTTP based API. And is sort of “REST” like, but is kind of wonky. (https://canvas.instructure.com/doc/api/all_resources.html).
- I want to define a preferred, canonical model for courses, sections, participants for my solution.
- And map the model to CourseWorks, and have the option for changing.

© Donald F. Ferguson, 2024

CourseWorks/Canvas LMS API



© Donald F. Ferguson, 2024

Applying Some Patterns/Principles

My first “real” microservice will initially demonstrate some concepts:

- 12 Factor App
  - Declare and isolate dependencies → requirements.txt
  - Store config in the environment → environment variables for API key, URLs
- SOLID Principles
  - Single Responsibility → a microservice just focused on CourseWorks.
  - Interface Segregation → the service only exposes the APIs necessary for scenario.
  - Liskov Substitution → the service will have an OpenAPI definition, and the implementation can be substituted over time.
- Design Patterns:
  - Adaptor
  - Anti-Corruption

But, my microservice exposes a REST API and I need to explain that first.

© Donald F. Ferguson, 2024

REST

© Donald F. Ferguson, 2024

Concepts
(Part 1)

© Donald F. Ferguson, 2024

REST (https://restfulapi.net/)

What is REST

- REST is acronym for **RE**presentational **S**tate **T**ransfer. 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 it's 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.

© Donald F. Ferguson, 2024

Resources

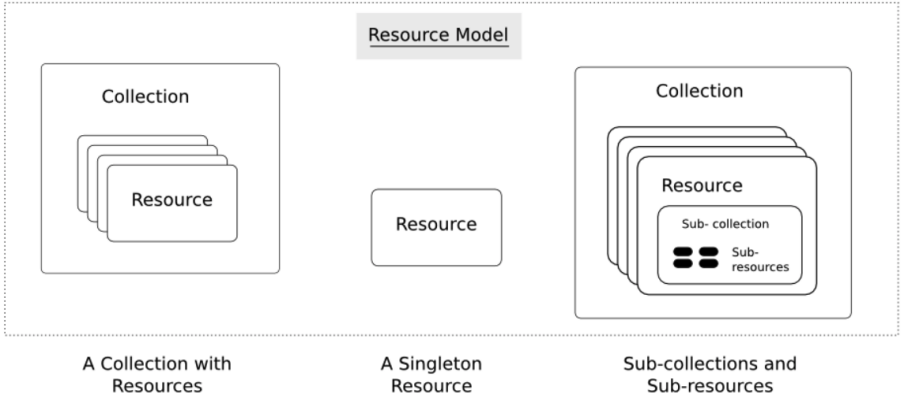
Resources are an abstraction. The application maps to create things and actions.

"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

© Donald F. Ferguson, 2024

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.

© Donald F. Ferguson, 2024

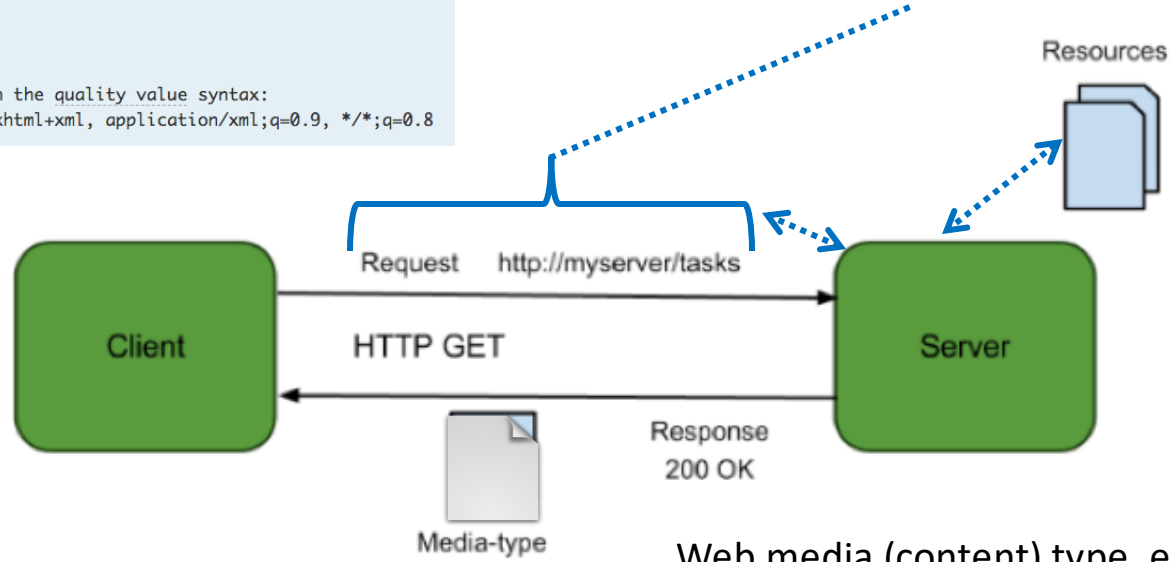
Resources, URLs, Content Types

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

© Donald F. Ferguson, 2024

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, ...
- We will also go into HATEOAS, GraphQL, … …

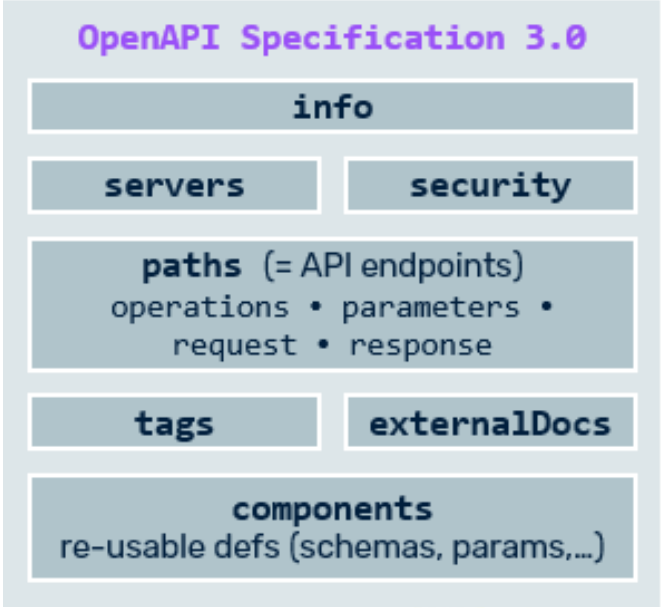
© Donald F. Ferguson, 2024

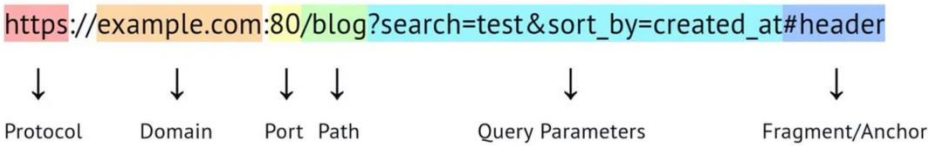
Uniform Interface
Resource Oriented
(Also, Part 1)

© Donald F. Ferguson, 2024

OpenAPI 3.0

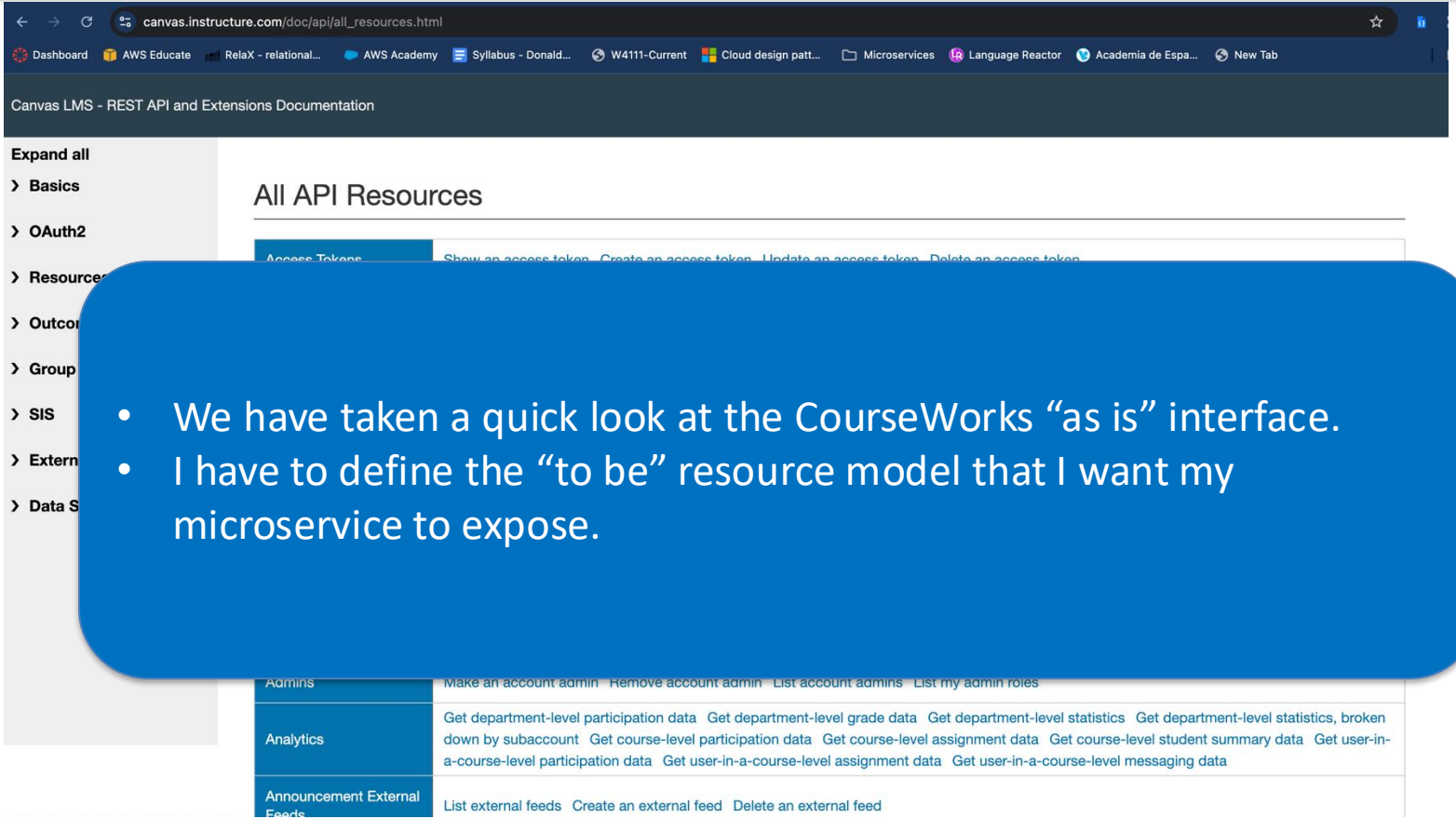
- OpenAPI is a standard for documenting REST APIs.
- An HTTP request/response has several elements:
  - Protocol
  - Server address and port
  - Path
  - Query parameters
  - Fragment
  - Headers, Cookies
  - Body content and content type
  - Method
- OpenAPI and web application frameworks introduce:
  - Path parameters
  - Models to represent bodies/content
  - Security information
  - Descriptive metadata





© Donald F. Ferguson, 2024

CourseWorks/Canvas LMS API

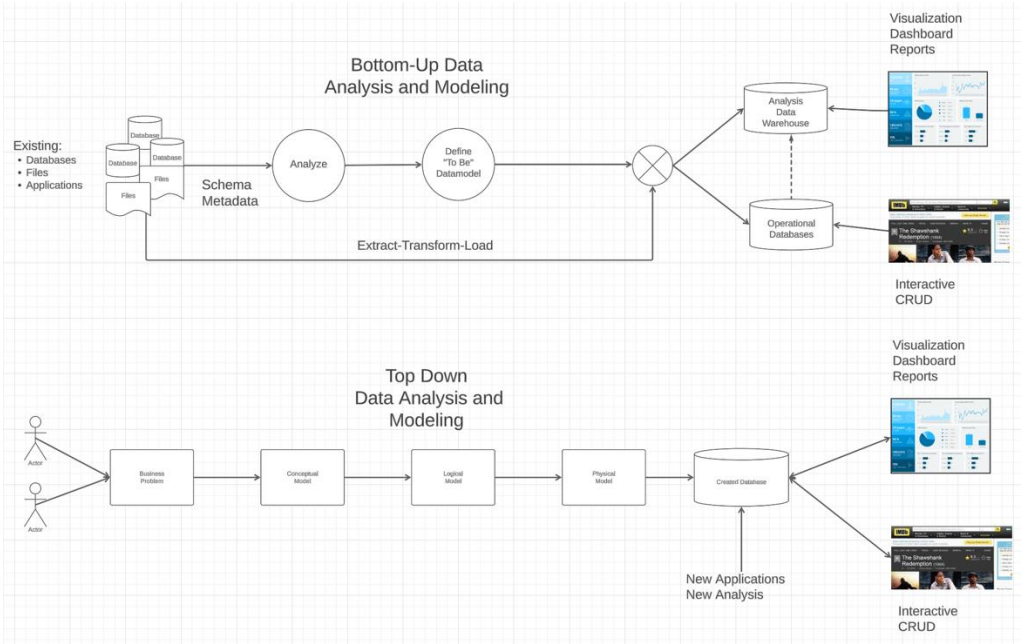


- We have taken a quick look at the CourseWorks “as is” interface.
- I have to define the “to be” resource model that I want my microservice to expose.

© Donald F. Ferguson, 2024

Top-Down, Bottom-Up, Meet-in-the-Middle

- In my DB class, I cover:
  - Top-down data modeling
  - Bottom-up data modeling
- An adaptor implements meet-in-the-middle and coverts
  - What exists to
  - What my application wants
- REST talks about “resources.”
- API design is similar to data modeling. Many concepts have analogs in REST.



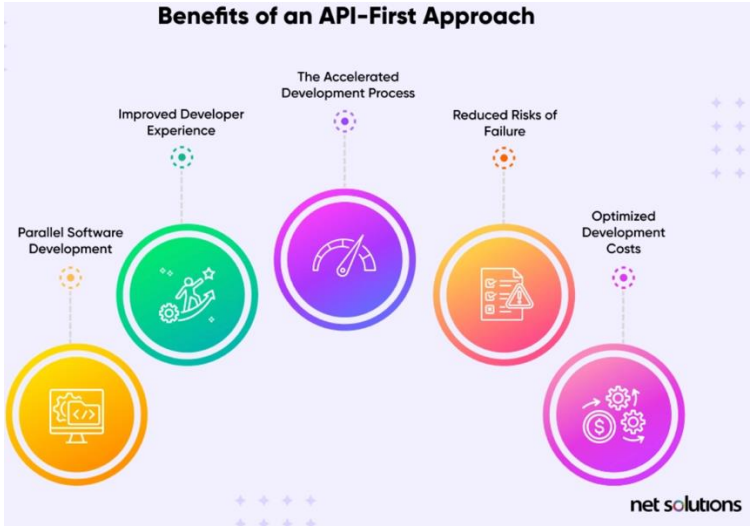
© Donald F. Ferguson, 2024

API First

- "As the connective tissue linking ecosystems of technologies and organizations, APIs allow businesses to monetize data, forge profitable partnerships, and open new pathways for innovation and growth." (McKinsey)
- "API-first, also called the API-first approach, prioritizes APIs at the beginning of the software development process, positioning APIs as the building blocks of software. API-first organizations develop APIs before writing other code, instead of treating them as afterthoughts. This lets teams construct applications with internal and external services that are delivered through APIs." (https://www.netsolutions.com/hub/mach-architecture/api-first)

Why API-First Approach?

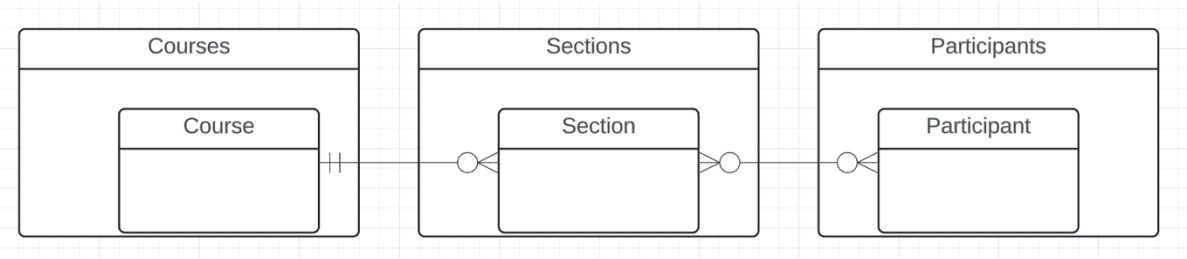
- By defining the APIs first, there is a modular separation between the services in the system. This increases the reusability of API endpoints across different parts of the system and to external entities as well (Interoperability).
- The API-first approach promotes collaboration as API documentation and contract definitions serve as a common language for stakeholders such as different teams in development (e.g. frontend and backend), V&V teams for test scenarios, etc.
- Creating the API contracts as part of the API-first design approach can be shared with frontend teams where the team can start building user interfaces parallelly with mocked data as per contract models, tested for the user experience. Also, codegen can be used to generate client mocks which can be directly used by client apps such as iOS/Android, etc.
- It gives the flexibility for tech-stack choices. Different services or applications in the system can be in different languages as long as they adhere to API contracts. It can help in identifying the best suitable tools/options in terms of cost and performance.
- With the API-First approach many concerns such as security measures, versioning, request traffic and data volume, and Integration with different services/applications within the system.



https://medium.com/@rohitranjan.pandey/api-first-design-approach-swagger-a1423db48f76

© Donald F. Ferguson, 2024

Resources and APIs

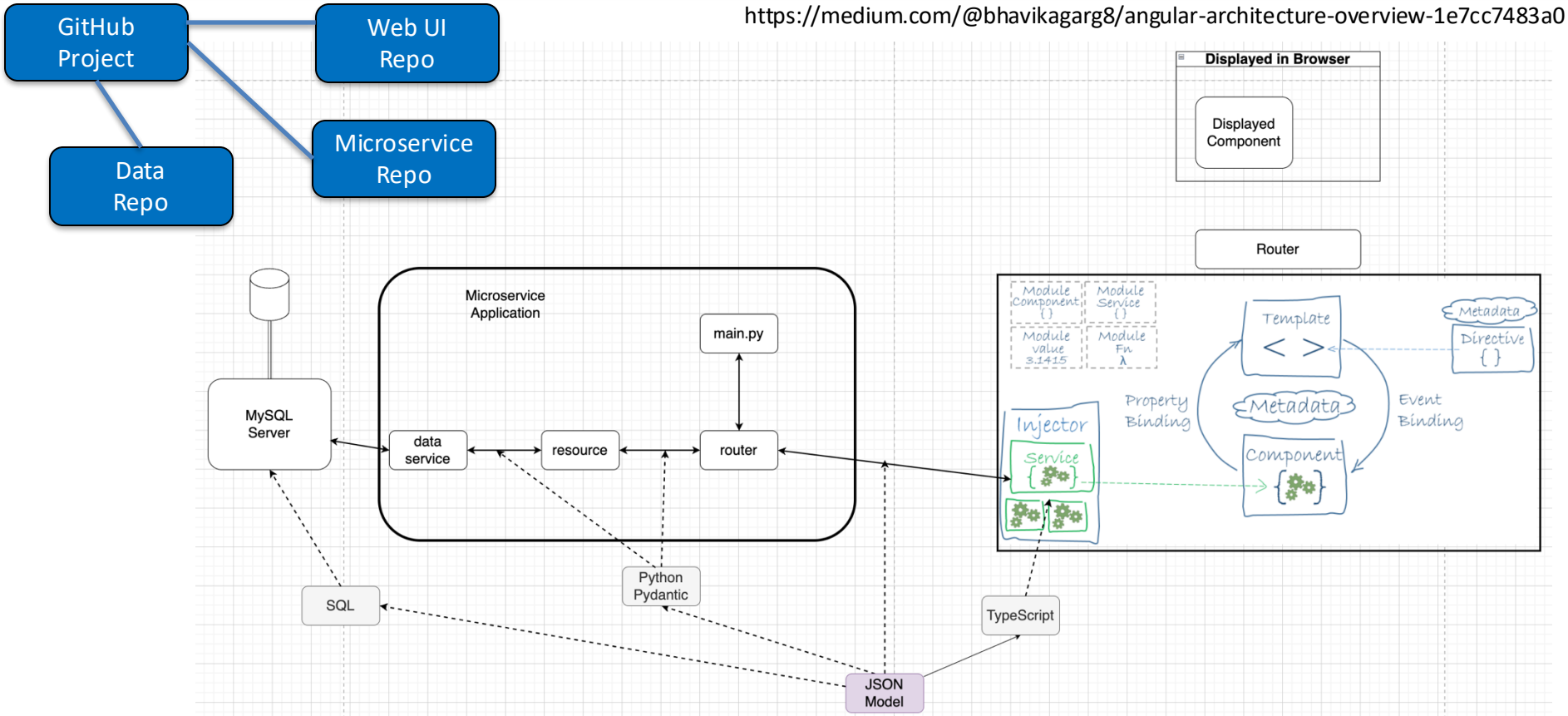


- Base resources, paths and methods:
  - /courses: GET~~, POST~~
  - /courses/<id>: GET, ~~PUT, DELETE~~
  - /sections: GET, ~~POST~~
  - /sections/<id>: GET, ~~PUT, DELETE~~
  - /participants: GET, ~~POST~~
  - /participants/<id>: GET, ~~PUT, DELETE~~
- There are relative, navigation paths:
  - /courses/<id>/sections
  - /participants/<id>/sections
  - etc.
- There are two approaches to defining API
  - Start with OpenAPI document and produce an implementation template.
  - Start with annotated code and generate API document.
- In either approach, I start with *models.*
- Also,
  - I lack the security permission to update CourseWorks.
  - I can choose to not surface the methods or raise and exception.
- GET on resources that are collections may also have query parameters.

© Donald F. Ferguson, 2024

Full Stack Application Structure

https://medium.com/@bhavikagarg8/angular-architecture-overview-1e7cc7483a0



© Donald F. Ferguson, 2024

Quick, First Code Walkthrough

© Donald F. Ferguson, 2024

First Iteration

- Core project: /Users/donald.ferguson/Dropbox/000/00-Current-Repos/student-course-application/courseworks-adaptor-service
  - This is currently some pretty awful code.
  - The pydantic stuff drives me crazy.
- My first object model is not quite correct … …
  - Participant and “Person” are not quite the same thing. A participant is a “relationship.”
  - Some of the properties are “munged together.” There really is not “course” in CourseWorks, for example. All they have are sections.
  - … …
- The way that I evolve this, or partially evolve this, is
  - Start with a simple mockup and consumers can use.
  - Continuously evolve.
- And I have to solve some other problems, like … … configuration from the environment.
- … …

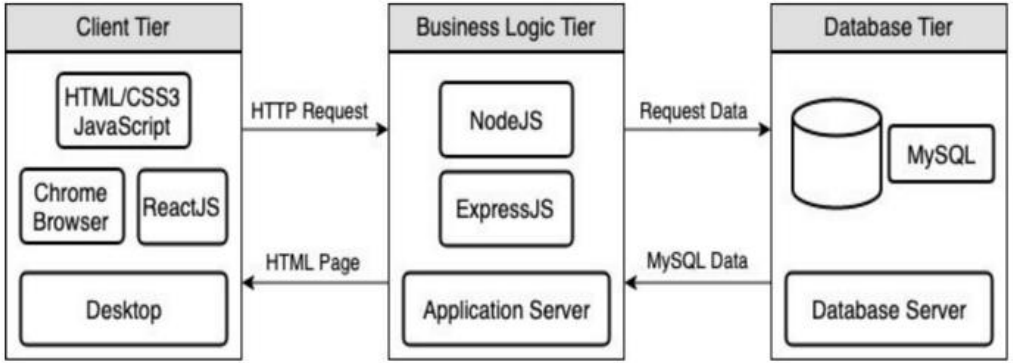
© Donald F. Ferguson, 2024

BLOBs, Static Content

© Donald F. Ferguson, 2024

Full Stack Web Application – Reminder

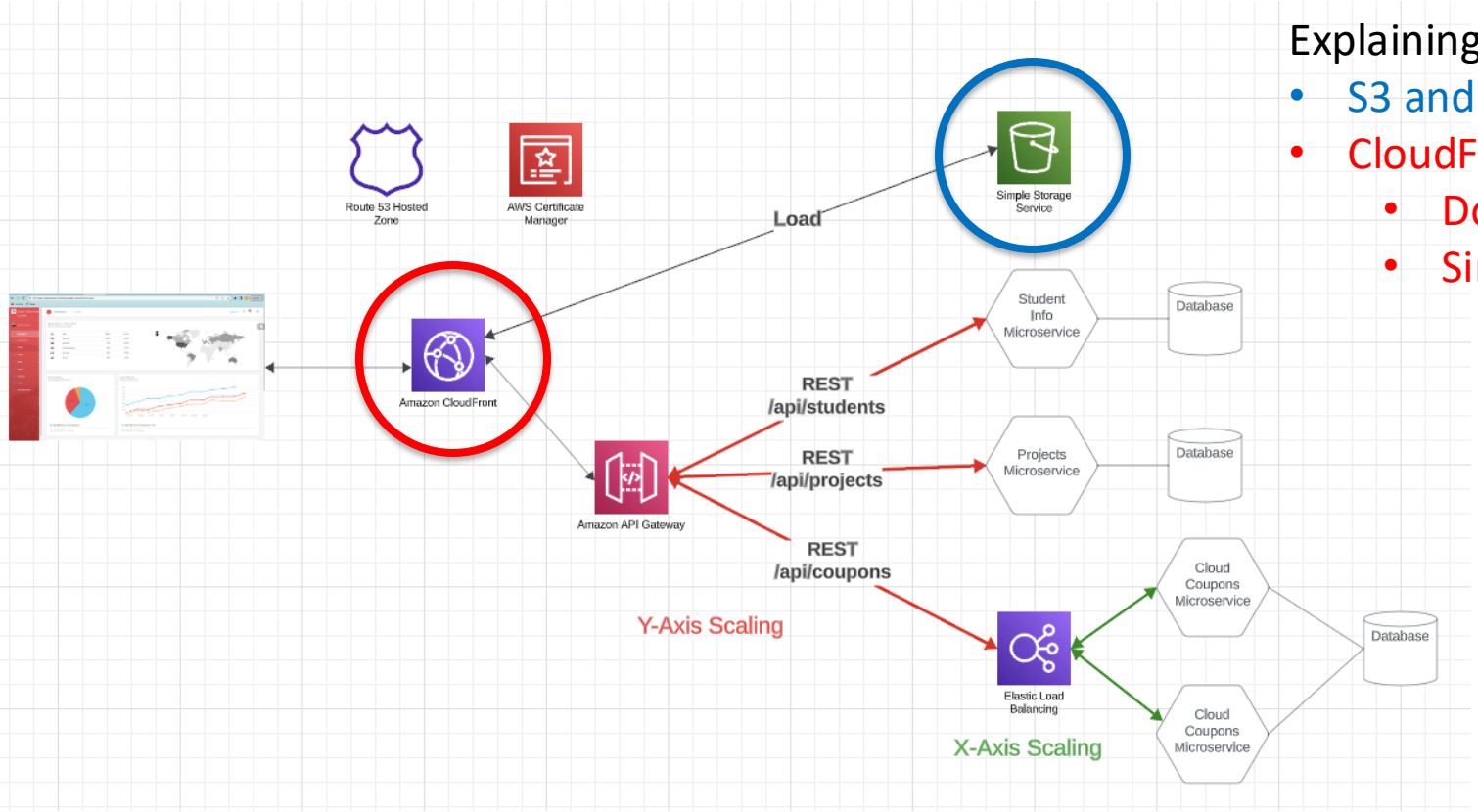
https://levelup.gitconnected.com/a-complete-guide-build-a-scalable-3-tier-architecture-with-mern-stack-es6-ca129d7df805



- 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."

© Donald F. Ferguson, 2024

Reminder and Positioning



Explaining:

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

© Donald F. Ferguson, 2024

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.

© Donald F. Ferguson, 2024

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.

© Donald F. Ferguson, 2024

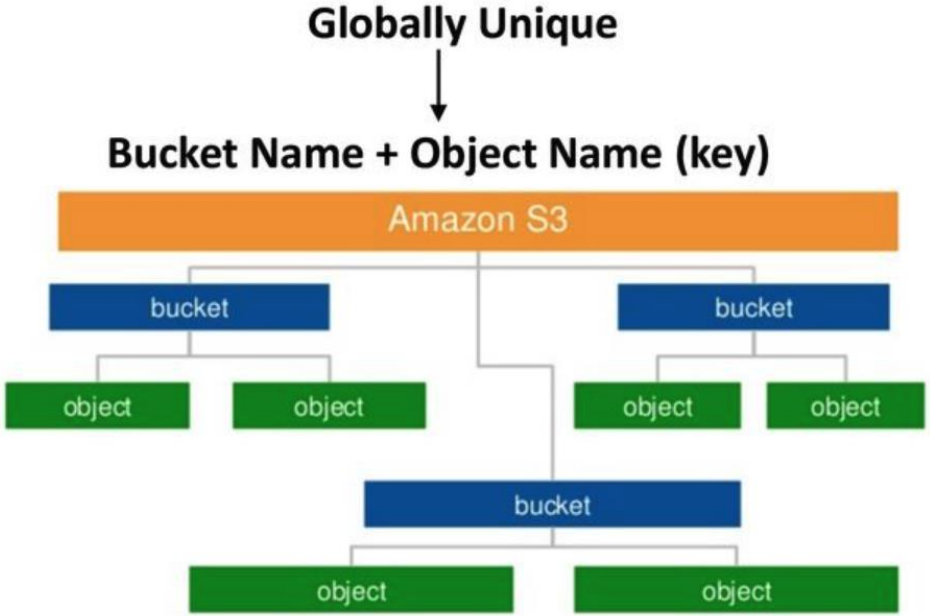
Amazon S3

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

© Donald F. Ferguson, 2024

AWS S3

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.

© Donald F. Ferguson, 2024

Simple S3 Example

```python
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"]}')
```

```python
def get_object():

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

    print("\n\n The object is ...\n", 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

© Donald F. Ferguson, 2024